GAP

Release 4.4.5
13 May 2005

Reference Manual

The GAP Group

http://www.gap-system.org

Acknowledgement

We would like to thank the many people who have made contributions of various kinds to the development of GAP since 1986, in particular:

Isabel M. Araújo, Robert Arthur, Hans Ulrich Besche, Thomas Bischops, Oliver Bonten, Thomas Breuer, Frank Celler, Gene Cooperman, Bettina Eick, Volkmar Felsch, Franz Gähler, Greg Gamble, Willem de Graaf, Burkhard Höfling, Jens Hollmann, Derek Holt, Erzsébet Horváth, Alexander Hulpke, Ansgar Kaup, Susanne Keitemeier, Steve Linton, Frank Lübeck, Bohdan Majewski, Johannes Meier, Thomas Merkwitz, Wolfgang Merkwitz, Jürgen Mnich, Robert F. Morse, Scott Murray, Joachim Neubüser, Max Neunhöffer, Werner Nickel, Alice Niemeyer, Dima Pasechnik, Götz Pfeiffer, Udo Polis, Ferenc Rákóczi, Sarah Rees, Edmund Robertson, Ute Schiffer, Martin Schönert, Ákos Seress, Andrew Solomon, Heiko Theißen, Rob Wainwright, Alex Wegner, Chris Wensley and Charles Wright.

The following list gives the authors, indicated by `A`, who designed the code in the first place as well as the current maintainers, indicated by `M` of the various modules of which GAP is composed.

Since the process of modularization was started only recently, there might be omissions both in scope and in contributors. The compilers of the manual apologize for any such errors and promise to rectify them in future editions.

Kernel
: Frank Celler (A), Steve Linton (AM), Frank Lübeck (AM), Werner Nickel (AM), Martin Schönert (A)

Automorphism groups of finite pc groups
: Bettina Eick (AM)

Binary Relations
: Robert Morse (AM), Andrew Solomon (A)

Characters and Character Degrees of certain solvable groups
: Hans Ulrich Besche (A), Thomas Breuer (AM)

Classes in nonsolvable groups
: Alexander Hulpke (AM)

Classical Groups
: Thomas Breuer (AM), Frank Celler (A), Stefan Kohl (AM), Frank Lübeck (AM), Heiko Theißen (A)

Congruences of magmas, semigroups and monoids
Robert Morse (AM), Andrew Solomon (A)

Cosets and Double Cosets
Alexander Hulpke (AM)

Cyclotomics
Thomas Breuer (AM)

Dixon-Schneider Algorithm
Alexander Hulpke (AM)

Documentation Utilities
Frank Celler (A), Heiko Theißen (A), Alexander Hulpke (A), Willem de Graaf (A), Steve Linton (A), Werner Nickel (A), Greg Gamble (AM)

Factor groups
Alexander Hulpke (AM)

Finitely presented groups
Volkmar Felsch (AM), Alexander Hulpke (AM), Martin Schoenert (A)

Finitely presented monoids and semigroups
Isabel Araújo (AM), Derek Holt (A), Alexander Hulpke (A), Götz Pfeiffer (A), Andrew Solomon (AM)

GAP for MacOS
Burkhard Höfling (AM)

Group actions
Heiko Theißen (A) and Alexander Hulpke (AM)

Homomorphism search
Alexander Hulpke (AM)

Homomorphisms for finitely presented groups
Alexander Hulpke (AM)

Identification of Galois groups
Alexander Hulpke (AM)

Intersection of subgroups of finite pc groups
Frank Celler (A), Bettina Eick (AM)

Irreducible Modules over finite fields for finite pc groups
Bettina Eick (AM)

Isomorphism testing with random methods
Hans Ulrich Besche (AM), Bettina Eick (AM)

Lie algebras
Thomas Breuer (A), Craig Struble (A), Juergen Wisliceny (A), Willem A. de Graaf (AM)

Monomiality Questions
Thomas Breuer (AM), Erzsébet Horváth (A)

Multiplier and Schur cover
Werner Nickel (AM), Alexander Hulpke (AM)

One-Cohomology and Complements
Frank Celler (A) and Alexander Hulpke (AM)

Partition Backtrack algorithm
Heiko Theißen (A), Alexander Hulpke (M)

Permutation group composition series
Ákos Seress (AM)

Permutation group homomorphisms
Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Permutation Group Pcgs
Heiko Theißen (A), Alexander Hulpke (M)

Possible Permutation Characters
Thomas Breuer (AM), Götz Pfeiffer (A)

Possible Class Fusions, Possible Power Maps Thomas Breuer (AM)

Primitive groups library
Heiko Theißen (A), Alexander Hulpke (M)

Properties and attributes of finite pc groups
Frank Celler (A), Bettina Eick (AM)

Random Schreier-Sims
Ákos Seress (AM)

Rational Functions
Frank Celler (A) and Alexander Hulpke (AM)

Semigroup relations
Isabel Araujo (A), Robert F. Morse (AM), Andrew Solomon (A)

Special Pcgs for finite pc groups
Bettina Eick (AM)

Stabilizer Chains
Ákos Seress (AM), Heiko Theißen (A), Alexander Hulpke (M)

Strings and Characters
Martin Schönert (A), Frank Celler (A), Thomas Breuer (A), Frank Lübeck (AM)

Structure Descriptions for Finite Groups
Stefan Kohl (AM), Markus Püschel(A), Sebastian Egner(A)

Subgroup lattice
Martin Schönert (A), Alexander Hulpke (AM)

Subgroup lattice for solvable groups
Alexander Hulpke (AM)

Subgroup presentations
Volkmar Felsch (AM)

The Help System
Frank Celler (A), Frank Lübeck (AM)

Tietze transformations
Volkmar Felsch (AM)

Transformation semigroups
Isabel Araujo (A), Robert Arthur (A), Robert F. Morse (AM), Andrew Solomon (A)

Transitive groups library
Alexander Hulpke (AM)

Two-cohomology and extensions of finite pc groups
Bettina Eick (AM)

Contents

Copyright Notice **19**

1 About the GAP Reference Manual **20**

1.1 Manual Conventions 20

1.2 Credit 21

2 The Help System **22**

2.1 Invoking the Help 22

2.2 Browsing through the Sections . . 22

2.3 Changing the Help Viewer 23

2.4 The Pager Command 25

3 Running GAP **27**

3.1 Command Line Options 27

3.2 Advanced Features of GAP . . . 29

3.3 Running GAP under MacOS . . . 31

3.4 The .gaprc file 33

3.5 Completion Files 34

3.6 Testing for the System Architecture 34

3.7 The Compiler 35

3.8 Suitability for Compilation . . . 35

3.9 Compiling Library Code 36

3.10 CRC Numbers 36

3.11 Saving and Loading a Workspace . 37

3.12 Coloring the Prompt and Input . . 38

4 The Programming Language **39**

4.1 Language Overview 39

4.2 Lexical Structure 40

4.3 Symbols 40

4.4 Whitespaces 41

4.5 Keywords 41

4.6 Identifiers 42

4.7 Expressions 42

4.8 Variables 43

4.9 More About Global Variables . . 44

4.10 Function Calls 46

4.11 Comparisons 47

4.12 Arithmetic Operators 48

4.13 Statements 49

4.14 Assignments 50

4.15 Procedure Calls 50

4.16 If 51

4.17 While 52

4.18 Repeat 52

4.19 For 53

4.20 Break 55

4.21 Continue 55

4.22 Function 55

4.23 Return 58

4.24 The Syntax in BNF 59

5 Functions **61**

5.1 Information about a function . . . 61

5.2 Calling a function with a list argument that is interpreted as several arguments 62

5.3 Functions that do nothing 63
5.4 Function Types 63
6 Main Loop and Break Loop 64
6.1 Main Loop 64
6.2 View and Print 65
6.3 Break Loops 67
6.4 Variable Access in a Break Loop . 71
6.5 Error 72
6.6 ErrorCount 72
6.7 Leaving GAP 73
6.8 Line Editing 73
6.9 Editing Files 74
6.10 Editor Support 75
6.11 SizeScreen 75
7 Debugging and Profiling Facilities 76
7.1 Recovery from NoMethodFound-Errors 76
7.2 ApplicableMethod 77
7.3 Tracing Methods 78
7.4 Info Functions 79
7.5 Assertions 80
7.6 Timing 81
7.7 Profiling 81
7.8 Information about the version used 84
7.9 Test Files 84
7.10 Debugging Recursion 84
7.11 Global Memory Information . . . 86
8 Options Stack 88
9 Files and Filenames 90
9.1 Portability 90
9.2 GAP Root Directory 90
9.3 Directories 91
9.4 Filename 92
9.5 Special Filenames 93
9.6 File Access 93
9.7 File Operations 94
10 Streams 97
10.1 Categories for Streams and the StreamsFamily 97
10.2 Operations applicable to All Streams 98
10.3 Operations for Input Streams . . 99
10.4 Operations for Output Streams . . 101
10.5 File Streams 103
10.6 User Streams 104
10.7 String Streams 104
10.8 Input-Output Streams 105
10.9 Dummy Streams 106
10.10 Handling of Streams in the Background 106
11 Processes 108
11.1 Process 108
11.2 Exec 109
12 Objects and Elements 110
12.1 Objects 110
12.2 Elements as equivalence classes . . 110
12.3 Sets 111
12.4 Domains 111
12.5 Identical Objects 111
12.6 Mutability and Copyability . . . 112
12.7 Duplication of Objects 114
12.8 Other Operations Applicable to any Object 115
13 Types of Objects 116
13.1 Families 116
13.2 Filters 117
13.3 Categories 118
13.4 Representation 120

13.5 Attributes 121
13.6 Setter and Tester for Attributes . . 122
13.7 Properties 124
13.8 Other Filters 125
13.9 Types 125
14 Integers **126**
14.1 Elementary Operations for Integers 127
14.2 Quotients and Remainders . . . 129
14.3 Prime Integers and Factorization . 131
14.4 Residue Class Rings 134
15 Number Theory **136**
15.1 Prime Residues 136
15.2 Primitive Roots and Discrete Logarithms 137
15.3 Roots Modulo Integers 138
15.4 Multiplicative Arithmetic Functions 140
15.5 Continued Fractions 141
15.6 Miscellaneous 142
16 Rational Numbers **143**
16.1 Elementary Operations for Rationals 143
17 Combinatorics **145**
17.1 Combinatorial Numbers 145
17.2 Combinations, Arrangements and Tuples 147
17.3 Fibonacci and Lucas Sequences . . 153
17.4 Permanent of a Matrix 154
18 Cyclotomic Numbers **155**
18.1 Operations for Cyclotomics . . . 155
18.2 Infinity 158
18.3 Comparisons of Cyclotomics . . . 159
18.4 ATLAS Irrationalities 159
18.5 Galois Conjugacy of Cyclotomics . 162
18.6 Internally Represented Cyclotomics 164
19 Unknowns **165**
20 Booleans **167**
20.1 Fail 167
20.2 Comparisons of Booleans 167
20.3 Operations for Booleans 168
21 Lists **170**
21.1 List Categories 170
21.2 Basic Operations for Lists 172
21.3 List Elements 172
21.4 List Assignment 174
21.5 IsBound and Unbind for Lists . . 176
21.6 Identical Lists 176
21.7 Duplication of Lists 178
21.8 Membership Test for Lists 179
21.9 Enlarging Internally Represented Lists 179
21.10 Comparisons of Lists 180
21.11 Arithmetic for Lists 180
21.12 Filters Controlling the Arithmetic Behaviour of Lists 181
21.13 Additive Arithmetic for Lists . . . 182
21.14 Multiplicative Arithmetic for Lists . 184
21.15 Mutability Status and List Arithmetic 186
21.16 Finding Positions in Lists 187
21.17 Properties and Attributes for Lists . 190
21.18 Sorting Lists 191
21.19 Sorted Lists and Sets 193
21.20 Operations for Lists 195
21.21 Advanced List Manipulations . . 202
21.22 Ranges 203
21.23 Enumerators 205
22 Boolean Lists **206**
22.1 Boolean Lists Representing Subsets 206
22.2 Set Operations via Boolean Lists . 207

22.3 Function that Modify Boolean Lists 208
22.4 More about Boolean Lists 209
23 Row Vectors 210
23.1 Operators for Row Vectors . . . 210
23.2 Row Vectors over Finite Fields . . 212
23.3 Coefficient List Arithmetic . . . 213
23.4 Shifting and Trimming Coefficient Lists 214
23.5 Functions for Coding Theory . . . 215
23.6 Vectors as coefficients of polynomials 215
24 Matrices 218
24.1 Categories of Matrices 218
24.2 Operators for Matrices 219
24.3 Properties and Attributes of Matrices 221
24.4 Matrix Constructions 223
24.5 Random Matrices 225
24.6 Matrices Representing Linear Equations and the Gaussian Algorithm . . . 225
24.7 Eigenvectors and eigenvalues . . . 226
24.8 Elementary Divisors 227
24.9 Echelonized Matrices 227
24.10 Matrices as Basis of a Row Space . 229
24.11 Triangular Matrices 230
24.12 Matrices as Linear Mappings . . . 230
24.13 Matrices over Finite Fields . . . 231
24.14 Special Multiplication Algorithms for Matrices over GF(2) 234
24.15 Block Matrices 235
25 Integral matrices and lattices 236
25.1 Linear equations over the integers and Integral Matrices 236
25.2 Normal Forms over the Integers . . 237
25.3 Determinant of an integer matrix . 240
25.4 Decompositions 240
25.5 Lattice Reduction 241
25.6 Orthogonal Embeddings 243
26 Strings and Characters 245
26.1 Special Characters 247
26.2 Internally Represented Strings . . 248
26.3 Recognizing Characters 249
26.4 Comparisons of Strings 249
26.5 Operations to Produce or Manipulate Strings 250
26.6 Character Conversion 252
26.7 Operations to Evaluate Strings . . 253
26.8 Calendar Arithmetic 254
27 Records 257
27.1 Accessing Record Elements . . . 258
27.2 Record Assignment 258
27.3 Identical Records 259
27.4 Comparisons of Records 260
27.5 IsBound and Unbind for Records . 261
27.6 Record Access Operations 262
28 Collections 263
28.1 Collection Families 263
28.2 Lists and Collections 264
28.3 Attributes and Properties for Collections 268
28.4 Operations for Collections 270
28.5 Membership Test for Collections . 272
28.6 Random Elements 272
28.7 Iterators 273
29 Orderings 276
29.1 Building new orderings 276
29.2 Properties and basic functionality . 277
29.3 Orderings on families of associative words 278

30 Domains and their Elements 282
30.1 Operational Structure of Domains . 282
30.2 Equality and Comparison of Domains 283
30.3 Constructing Domains 283
30.4 Changing the Structure 284
30.5 Changing the Representation . . . 285
30.6 Domain Categories 285
30.7 Parents 286
30.8 Constructing Subdomains 287
30.9 Operations for Domains 287
30.10 Attributes and Properties of Elements 288
30.11 Comparison Operations for Elements 291
30.12 Arithmetic Operations for Elements 292
30.13 Relations Between Domains . . . 293
30.14 Useful Categories of Elements . . 295
30.15 Useful Categories for all Elements of a Family 297

31 Mappings 299
31.1 Creating Mappings 300
31.2 Properties and Attributes of (General) Mappings 301
31.3 Images under Mappings 302
31.4 Preimages under Mappings . . . 303
31.5 Arithmetic Operations for General Mappings 305
31.6 Mappings which are Compatible with Algebraic Structures 305
31.7 Magma Homomorphisms 305
31.8 Mappings that Respect Multiplication 306
31.9 Mappings that Respect Addition . 307
31.10 Linear Mappings 307
31.11 Ring Homomorphisms 308
31.12 General Mappings 308
31.13 Technical Matters Concerning General Mappings 308

32 Relations 310
32.1 General Binary Relations 310
32.2 Properties and Attributes of Binary Relations 310
32.3 Binary Relations on Points . . . 312
32.4 Closure Operations and Other Constructors 312
32.5 Equivalence Relations 313
32.6 Attributes of and Operations on Equivalence Relations 313
32.7 Equivalence Classes 314

33 Magmas 315
33.1 Magma Categories 315
33.2 Magma Generation 316
33.3 Magmas Defined by Multiplication Tables 317
33.4 Attributes and Properties for Magmas 318

34 Words 321
34.1 Categories of Words and Nonassociative Words 321
34.2 Comparison of Words 323
34.3 Operations for Words 323
34.4 Free Magmas 324
34.5 External Representation for Nonassociative Words 325

35 Associative Words 326
35.1 Categories of Associative Words . . 326
35.2 Free Groups, Monoids and Semigroups 327
35.3 Comparison of Associative Words . 329
35.4 Operations for Associative Words . 330
35.5 Operations for Associative Words by their Syllables 331
35.6 Representations for Associative Words 332

35.7 The External Representation for Associative Words 334
35.8 Straight Line Programs 334
35.9 Straight Line Program Elements . 338
36 Rewriting Systems 339
36.1 Operations on rewriting systems . 339
36.2 Operations on elements of the algebra 340
36.3 Properties of rewriting systems . . 341
36.4 Rewriting in Groups and Monoids . 341
36.5 Developing rewriting systems . . . 342
37 Groups 344
37.1 Group Elements 344
37.2 Creating Groups 344
37.3 Subgroups 346
37.4 Closures of (Sub)groups 348
37.5 Expressing Group Elements as Words in Generators 348
37.6 Structure Descriptions 349
37.7 Cosets 351
37.8 Transversals 352
37.9 Double Cosets 353
37.10 Conjugacy Classes 354
37.11 Normal Structure 356
37.12 Specific and Parametrized Subgroups 357
37.13 Sylow Subgroups and Hall Subgroups 359
37.14 Subgroups characterized by prime powers 360
37.15 Group Properties 361
37.16 Numerical Group Attributes . . . 363
37.17 Subgroup Series 364
37.18 Factor Groups 367
37.19 Sets of Subgroups 368
37.20 Subgroup Lattice 369
37.21 Specific Methods for Subgroup Lattice Computations 371
37.22 Special Generating Sets 374
37.23 1-Cohomology 374
37.24 Schur Covers and Multipliers . . . 377
37.25 Tests for the Availability of Methods 377
38 Group Homomorphisms 379
38.1 Creating Group Homomorphisms . 379
38.2 Operations for Group Homomorphisms 381
38.3 Efficiency of Homomorphisms . . 382
38.4 Homomorphism for very large groups 383
38.5 Nice Monomorphisms 384
38.6 Group Automorphisms 384
38.7 Groups of Automorphisms 386
38.8 Calculating with Group Automorphisms 387
38.9 Searching for Homomorphisms . . 388
38.10 Representations for Group Homomorphisms 390
39 Group Actions 391
39.1 About Group Actions 391
39.2 Basic Actions 392
39.3 Orbits 394
39.4 Stabilizers 396
39.5 Elements with Prescribed Images . 397
39.6 The Permutation Image of an Action 397
39.7 Action of a group on itself 399
39.8 Permutations Induced by Elements and Cycles 400
39.9 Tests for Actions 401
39.10 Block Systems 402
39.11 External Sets 403
40 Permutations 406
40.1 Comparison of Permutations . . . 407

40.2 Moved Points of Permutations . . 407
40.3 Sign and Cycle Structure 408
40.4 Creating Permutations 409
41 Permutation Groups 410
41.1 The Natural Action 410
41.2 Computing a Permutation Representation 411
41.3 Symmetric and Alternating Groups 411
41.4 Primitive Groups 412
41.5 Stabilizer Chains 413
41.6 Randomized Methods for Permutation Groups 414
41.7 Construction of Stabilizer Chains . 416
41.8 Stabilizer Chain Records 417
41.9 Operations for Stabilizer Chains . 418
41.10 Low Level Routines to Modify and Create Stabilizer Chains 420
41.11 Backtrack 421
41.12 Working with large degree permutation groups 422
42 Matrix Groups 424
42.1 Attributes and Properties for Matrix Groups 424
42.2 Actions of Matrix Groups 425
42.3 GL and SL 425
42.4 Invariant Forms 426
42.5 Matrix Groups in Characteristic 0 . 427
42.6 Acting OnRight and OnLeft . . . 429
43 Polycyclic Groups 430
43.1 Polycyclic Generating Systems . . 430
43.2 Computing a Pcgs 431
43.3 Defining a Pcgs Yourself 431
43.4 Elementary Operations for a Pcgs . 432
43.5 Elementary Operations for a Pcgs and an Element 433
43.6 Exponents of Special Products . . 434
43.7 Subgroups of Polycyclic Groups - Induced Pcgs 435
43.8 Subgroups of Polycyclic Groups - Canonical Pcgs 436
43.9 Factor Groups of Polycyclic Groups - Modulo Pcgs 437
43.10 Factor Groups of Polycyclic Groups in their Own Representation 438
43.11 Pcgs and Normal Series 439
43.12 Sum and Intersection of Pcgs . . . 441
43.13 Special Pcgs 442
43.14 Action on Subfactors Defined by a Pcgs 444
43.15 Orbit Stabilizer Methods for Polycyclic Groups 445
43.16 Operations which have Special Methods for Groups with Pcgs 445
43.17 Conjugacy Classes in Solvable Groups 445
44 Pc Groups 447
44.1 The family pcgs 448
44.2 Elements of pc groups 448
44.3 Pc groups versus fp groups . . . 449
44.4 Constructing Pc Groups 449
44.5 Computing Pc Groups 451
44.6 Saving a Pc Group 452
44.7 Operations for Pc Groups 452
44.8 2-Cohomology and Extensions . . 453
44.9 Coding a Pc Presentation 456
44.10 Random Isomorphism Testing . . 456
45 Finitely Presented Groups 457
45.1 Creating Finitely Presented Groups 458
45.2 Comparison of Elements of Finitely Presented Groups 459

45.3 Preimages in the Free Group . . . 459
45.4 Operations for Finitely Presented Groups 460
45.5 Coset Tables and Coset Enumeration 461
45.6 Standardization of coset tables . . 464
45.7 Coset tables for subgroups in the whole group 465
45.8 Augmented Coset Tables and Rewriting 465
45.9 Low Index Subgroups 466
45.10 Converting Groups to Finitely Presented Groups 467
45.11 New Presentations and Presentations for Subgroups 469
45.12 Preimages under Homomorphisms from an FpGroup 470
45.13 Quotient Methods 471
45.14 Abelian Invariants for Subgroups . 472
45.15 Testing Finiteness of Finitely Presented Groups 473

46 Presentations and Tietze Transformations 475

46.1 Creating Presentations 475
46.2 SimplifiedFpGroup 477
46.3 Subgroup Presentations 478
46.4 Relators in a Presentation 481
46.5 Printing Presentations 481
46.6 Changing Presentations 483
46.7 Tietze Transformations 483
46.8 Elementary Tietze Transformations 486
46.9 Tietze Transformations that introduce new Generators 488
46.10 Tracing generator images through Tietze transformations 491
46.11 DecodeTree 493
46.12 Tietze Options 496

47 Group Products 498

47.1 Direct Products 498
47.2 Semidirect Products 499
47.3 Subdirect Products 500
47.4 Wreath Products 501
47.5 Embeddings and Projections for Group Products 502

48 Group Libraries 503

48.1 Basic Groups 503
48.2 Classical Groups 505
48.3 Conjugacy Classes in Classical Groups 509
48.4 Constructors for Basic Groups . . 509
48.5 Selection Functions 510
48.6 Transitive Permutation Groups . . 511
48.7 Small Groups 512
48.8 Finite Perfect Groups 515
48.9 Primitive Permutation Groups . . 519
48.10 Index numbers of primitive groups . 521
48.11 Irreducible Solvable Matrix Groups 522
48.12 Irreducible Maximal Finite Integral Matrix Groups 523

49 Semigroups 531

49.1 Making transformation semigroups 533
49.2 Ideals of semigroups 533
49.3 Congruences for semigroups . . . 534
49.4 Quotients 534
49.5 Green's Relations 534
49.6 Rees Matrix Semigroups 536

50 Monoids 538

51 Finitely Presented Semigroups and Monoids 540

51.1 Creating Finitely Presented Semigroups 542
51.2 Comparison of Elements of Finitely Presented Semigroups 543

51.3 Preimages in the Free Semigroup . 543
51.4 Finitely presented monoids . . . 544
51.5 Rewriting Systems and the Knuth-Bendix Procedure 545
51.6 Todd-Coxeter Procedure 546
52 Transformations 547
53 Additive Magmas (preliminary) 550
53.1 (Near-)Additive Magma Categories 550
53.2 (Near-)Additive Magma Generation 551
53.3 Attributes and Properties for (Near-)Additive Magmas 552
53.4 Operations for (Near-)Additive Magmas 553
54 Rings 554
54.1 Generating Rings 554
54.2 Ideals in Rings 556
54.3 Rings With One 558
54.4 Properties of Rings 559
54.5 Units and Factorizations 560
54.6 Euclidean Rings 562
54.7 Gcd and Lcm 563
55 Modules (preliminary) 566
55.1 Generating modules 566
55.2 Submodules 567
55.3 Free Modules 568
56 Fields and Division Rings 570
56.1 Generating Fields 570
56.2 Subfields of Fields 572
56.3 Galois Action 572
57 Finite Fields 576
57.1 Finite Field Elements 576
57.2 Operations for Finite Field Elements 577
57.3 Creating Finite Fields 579
57.4 FrobeniusAutomorphism 580
57.5 Conway Polynomials 580
58 Abelian Number Fields 582
58.1 Construction of Abelian Number Fields 582
58.2 Operations for Abelian Number Fields 583
58.3 Integral Bases of Abelian Number Fields 585
58.4 Galois Groups of Abelian Number Fields 587
58.5 Gaussians 588
59 Vector Spaces 589
59.1 Constructing Vector Spaces . . . 589
59.2 Operations and Attributes for Vector Spaces 590
59.3 Domains of Subspaces of Vector Spaces 591
59.4 Bases of Vector Spaces 591
59.5 Operations for Vector Space Bases . 593
59.6 Operations for Special Kinds of Bases 595
59.7 Mutable Bases 596
59.8 Row and Matrix Spaces 597
59.9 Vector Space Homomorphisms . . 600
59.10 Vector Spaces Handled By Nice Bases 602
59.11 How to Implement New Kinds of Vector Spaces 603
60 Algebras 605
60.1 Constructing Algebras by Generators 605
60.2 Constructing Algebras as Free Algebras 606
60.3 Constructing Algebras by Structure Constants 607
60.4 Some Special Algebras 609
60.5 Subalgebras 610
60.6 Ideals 611
60.7 Categories and Properties of Algebras 612
60.8 Attributes and Operations for Algebras 613

60.9 Homomorphisms of Algebras . . . 619
60.10 Representations of Algebras . . . 622
61 Lie Algebras 630
61.1 Lie objects 630
61.2 Constructing Lie algebras 631
61.3 Distinguished Subalgebras 633
61.4 Series of Ideals 634
61.5 Properties of a Lie Algebra . . . 635
61.6 Direct Sum Decompositions . . . 636
61.7 Semisimple Lie Algebras and Root Systems 636
61.8 Restricted Lie algebras 640
61.9 The Adjoint Representation . . . 642
61.10 Universal Enveloping Algebras . . 643
61.11 Finitely Presented Lie Algebras . . 643
61.12 Modules over Lie Algebras and Their Cohomology 645
61.13 Modules over Semisimple Lie Algebras 647
61.14 Tensor Products and Exterior and Symmetric Powers 650
62 Finitely Presented Algebras 652
63 Magma Rings 653
63.1 Free Magma Rings 654
63.2 Elements of Free Magma Rings . . 655
63.3 Natural Embeddings related to Magma Rings 655
63.4 Magma Rings modulo Relations . . 656
63.5 Magma Rings modulo the Span of a Zero Element 657
63.6 Technical Details about the Implementation of Magma Rings . 657
64 Polynomials and Rational Functions 659
64.1 Indeterminates 659
64.2 Operations for Rational Functions . 661
64.3 Comparison of Rational Functions . 661
64.4 Properties and Attributes of Rational Functions 662
64.5 Univariate Polynomials 664
64.6 Polynomials as Univariate Polynomials in one Indeterminate 665
64.7 Multivariate Polynomials 666
64.8 Minimal Polynomials 667
64.9 Cyclotomic Polynomials 667
64.10 Polynomial Factorization 667
64.11 Polynomials over the Rationals . . 668
64.12 Laurent Polynomials 669
64.13 Univariate Rational Functions . . 670
64.14 Polynomial Rings 670
64.15 Univariate Polynomial Rings . . . 672
64.16 Monomial Orderings 673
64.17 Groebner Bases 676
64.18 Rational Function Families . . . 677
64.19 The Representations of Rational Functions 678
64.20 The Defining Attributes of Rational Functions 678
64.21 Creation of Rational Functions . . 679
64.22 Arithmetic for External Representations of Polynomials 680
64.23 Cancellation Tests for Rational Functions 681
65 Algebraic extensions of fields 682
65.1 Creation of Algebraic Extensions . 682
65.2 Elements in Algebraic Extensions . 682
66 p-adic Numbers (preliminary) 684
66.1 Pure p-adic Numbers 684
66.2 Extensions of the p-adic Numbers . 685

67 The MeatAxe 687
67.1 MeatAxe Modules 687
67.2 Module Constructions 687
67.3 Selecting a Different MeatAxe . . 688
67.4 Accessing a Module 688
67.5 Irreducibility Tests 688
67.6 Finding Submodules 688
67.7 Induced Actions 689
67.8 Module Homomorphisms 690
67.9 Invariant Forms 690
67.10 The Smash MeatAxe 691
67.11 Smash MeatAxe Flags 692
68 Tables of Marks 693
68.1 More about Tables of Marks . . . 693
68.2 Table of Marks Objects in GAP . . 694
68.3 Constructing Tables of Marks . . 694
68.4 Printing Tables of Marks 696
68.5 Sorting Tables of Marks 697
68.6 Technical Details about Tables of Marks 698
68.7 Attributes of Tables of Marks . . 699
68.8 Properties of Tables of Marks . . 702
68.9 Other Operations for Tables of Marks 703
68.10 Standard Generators of Groups . . 706
68.11 Accessing Subgroups via Tables of Marks 709
68.12 The Interface between Tables of Marks and Character Tables 711
68.13 Generic Construction of Tables of Marks 713
68.14 The Library of Tables of Marks . . 714
69 Character Tables 715
69.1 Some Remarks about Character Theory in GAP 715
69.2 History of Character Theory Stuff in GAP 716
69.3 Creating Character Tables 717
69.4 Character Table Categories . . . 720
69.5 Conventions for Character Tables . 721
69.6 The Interface between Character Tables and Groups 721
69.7 Operators for Character Tables . . 724
69.8 Attributes and Properties of Character Tables 724
69.9 Operations Concerning Blocks . . 731
69.10 Other Operations for Character Tables 734
69.11 Printing Character Tables 737
69.12 Computing the Irreducible Characters of a Group 740
69.13 Representations given by modules . 742
69.14 The Dixon-Schneider Algorithm . . 743
69.15 Advanced Methods for Dixon-Schneider Calculations 743
69.16 Components of a Dixon Record . . 744
69.17 An Example of Advanced Dixon-Schneider Calculations . . 745
69.18 Constructing Character Tables from Others 746
69.19 Sorted Character Tables 749
69.20 Automorphisms and Equivalence of Character Tables 752
69.21 Storing Normal Subgroup Information 754
70 Class Functions 756
70.1 Why Class Functions? 756
70.2 Basic Operations for Class Functions 758
70.3 Comparison of Class Functions . . 759
70.4 Arithmetic Operations for Class Functions 760
70.5 Printing Class Functions 762

70.6 Creating Class Functions from Values Lists 763
70.7 Creating Class Functions using Groups 764
70.8 Operations for Class Functions . . 765
70.9 Restricted and Induced Class Functions 769
70.10 Reducing Virtual Characters . . . 771
70.11 Symmetrizations of Class Functions 777
70.12 Molien Series 779
70.13 Possible Permutation Characters . 781
70.14 Computing Possible Permutation Characters 784
70.15 Operations for Brauer Characters . 788
70.16 Domains Generated by Class Functions 789

71 Maps Concerning Character Tables 790

71.1 Power Maps 790
71.2 Class Fusions between Character Tables 794
71.3 Parametrized Maps 799
71.4 Subroutines for the Construction of Power Maps 807
71.5 Subroutines for the Construction of Class Fusions 809

72 Monomiality Questions 812

72.1 Character Degrees and Derived Length 813
72.2 Primitivity of Characters 813
72.3 Testing Monomiality 815
72.4 Minimal Nonmonomial Groups . . 818

73 Installing GAP 819

73.1 Installation Overview 819
73.2 Get the Archives 820
73.3 Unpacking 820
73.4 Compilation 821
73.5 Test of the installation 822
73.6 Packages 823
73.7 Finish Installation and Cleanup . . 823
73.8 The Documentation 824
73.9 If Things Go Wrong 825
73.10 Known Problems of the Configure Process 826
73.11 Problems on Particular Systems . . 827
73.12 Optimization and Compiler Options 827
73.13 Porting GAP 828
73.14 GAP for Macintosh OS X 829
73.15 GAP for MacOS 830
73.16 Installation of GAP for MacOS . . 830
73.17 Expert Windows installation . . . 832
73.18 Copyrights 833

74 GAP Packages 834

74.1 Installing a GAP Package 834
74.2 Loading a GAP Package 834
74.3 Functions for GAP Packages . . . 835

75 Replaced and Removed Command Names 838

75.1 Group Actions - Name Changes . . 838
75.2 Package Interface - Obsolete Functions and Name Changes 838
75.3 Normal Forms of Integer Matrices - Name Changes 839
75.4 Miscellaneous Name Changes or Removed Names 839

Bibliography 841

Index 848

Copyright Notice

Copyright © (1987–2004) by the GAP Group,

incorporating the Copyright © 1999, 2000 by School of Mathematical and Computational Sciences, University of St Andrews, North Haugh, St Andrews, Fife KY16 9SS, Scotland

being the Copyright © 1992 by Lehrstuhl D für Mathematik, RWTH, 52056 Aachen, Germany, transferred to St Andrews on July 21st, 1997.

except for files in the distribution, which have an explicit different copyright statement. In particular, the copyright of packages distributed with GAP is usually with the package authors or their institutions.

GAP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version. For details, see the file `GPL` in the `etc` directory of the GAP distribution or see

`http://www.gnu.org/licenses/gpl.html`

If you obtain GAP please send us a short notice to that effect, e.g., an e-mail message to the address `support@gap-system.org`, containing your full name and address. This allows us to keep track of the number of GAP users.

If you publish a mathematical result that was partly obtained using GAP, please cite GAP, just as you would cite another paper that you used (see below for sample citation). Also we would appreciate if you could inform us about such a paper.

Specifically, please refer to

```
[GAP] The GAP Group, GAP --- Groups, Algorithms, and Programming,
      Version 4.4.5; 2005
      (http://www.gap-system.org)
```

GAP is distributed by us without any warranty, to the extent permitted by applicable state law. We distribute GAP **as is** without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

The entire risk as to the quality and performance of the program is with you. Should GAP prove defective, you assume the cost of all necessary servicing, repair or correction.

In no case unless required by applicable law will we, and/or any other party who may modify and redistribute GAP as permitted above, be liable to you for damages, including lost profits, lost monies or other special, incidental or consequential damages arising out of the use or inability to use GAP.

You are permitted to modify and redistribute GAP, but you are not allowed to restrict further redistribution. That is to say proprietary modifications will not be allowed. We want all versions of GAP to remain free.

If you modify any part of GAP and redistribute it, you must supply a `README` document. This should specify what modifications you made in which files. We do not want to take credit or be blamed for your modifications.

Of course we are interested in all of your modifications. In particular we would like to see bug-fixes, improvements and new functions. So again we would appreciate it if you would inform us about all modifications you make.

1 About the GAP Reference Manual

This is one of four parts of the GAP documentation, the others being the GAP **Tutorial**, a beginner's introduction to GAP, **Programming in** GAP and **Extending** GAP, which provide information for those who want to write their own GAP extensions.

This manual, the GAP **reference manual** contains the official definitions of GAP functions. It should give all information to someone who wants to use GAP as it is. It is not intended to be read cover-to-cover.

This manual is divided into chapters. Each chapter is divided into sections and, within each section, important definitions are numbered. References are therefore triples.

Chapter 2 describes the **help system**, which provides online access to the information of the manual. Chapter 3 gives technical advice for **running** GAP. Chapter 4 introduces the GAP language, while the next chapters deal with the **environment** provided by GAP for the user. These are followed by the main bulk of chapters which is devoted to various mathematical structures that GAP can handle.

Pages are numbered consecutively in each of the four manuals.

1.1 Manual Conventions

The printed manual uses different text styles for several purposes. Note that the online help may use other symbols to express the meanings listed below.

`*text*`

Text printed in boldface is used to emphasize single words or phrases.

text

Text printed in italics is used for arguments in the descriptions of functions and for other place holders. It means that you should not actually enter this text into GAP but replace it by appropriate text depending on what you want to do. For example when we write that you should enter `?`*section* to see the section with the name *section*, *section* serves as a place holder, indicating that you can enter the name of the section that you want to see at this place.

`text`

Text printed in a monospaced (all characters have the same width) typewriter font is used for names of variables and functions and other text that you may actually enter into your computer and see on your screen. Such text may contain place holders printed in italics as described above. For example when the information for `IsPrime` says that the form of the call is `IsPrime(` *n* `)` this means that you should actually enter the strings "IsPrime(" and ")", without the quotes, but replace the *n* with the number (or expression) that you want to test.

`Oper(` *arg1*`,` *arg2*`[,` *opt*`] ) F`

starts a subsection on the command `Oper` that takes two arguments *arg1* and *arg2* and an optional third argument *opt*. As in the above example, the letter `F` at the end of a line that starts with a little black triangle in the left margin indicates that the command is a simple function. Other possible letters at the

end of such a line are `A`, `P`, `O`, `C`, `R`, and `V`; they indicate "Attribute", "Property", "Operation", "Category", "Representation" (see Chapter 13), or "Variable", respectively.

In the printed manual, **mathematical formulas** are typeset in italics (actually math italics), and subscripts and superscripts are actually lowered and raised.

Longer **examples** are usually paragraphs of their own. Everything on the lines with the prompts `gap>` and `>`, except the prompts themselves of course, is the input you have to type; everything else is GAP's response. In the printed manual, examples are printed in a monospaced typewriter font.

1.2 Credit

The manual tries to give credit to designers and implementors of major parts of GAP. For many parts of the GAP code it is impossible to give detailed credit, because over the time of its development many persons have contributed from first ideas, even in prerunners of GAP such as CAS or SOGOS, via first implementations, improvements, and even total reimplementations. The documentation of the code gives further details, but again, it suffers from the same problem. We have attempted to give attributions with the different chapters of the manual where this seemed to be possible, but we apologise for all (unavoidable) shortcomings of this attempt.

2 The Help System

This chapter describes the GAP help system. The help system lets you read the documentation interactively.

2.1 Invoking the Help

The basic command to read GAP's documentation from within a GAP session is as follows.

1 ► `?[`*book*`:][?]`*topic*

For an explanation and some examples see 2.8.

Note that the first question mark must appear in the **first position** after the `gap>` prompt. The search strings *book* and *topic* are normalized in a certain way (see the end of this section for details) before the search starts. This makes the search case insensitive and there can be arbitrary white space after the first question mark.

When there are several manual sections that match the query a numbered list of topics is displayed. These matches can be accessed with `?`*number*.

There are some further specially handled commands which start with a question mark. They are explained in section 2.2.

As default GAP shows the help sections as text in the terminal (window), page by page if the shown text does not fit on the screen. But there are several other choices to read (other formats of) the documents: via a viewer for `dvi`-files (produced by TEX) or files in Acrobat's `pdf`-format or via a Web-browser. This is explained in section 2.3.

Details of the string normalization process

Here now is precisely how the search strings *book* and *topic* are normalized before a search starts: backslashes and double or single quotes are removed, parentheses and braces are substituted by blanks, non-ASCII characters are considered as ISO-latin1 characters and the accented letters are substituted by their non-accented counterpart. Finally white space is normalized.

2.2 Browsing through the Sections

Help books for GAP are organized in chapters, sections and subsections. There are a few special commands starting with a question mark (in the first position after the `gap>` prompt) which allow browsing a book section or chapter wise.

1 ► `?>`
► `?<`

The two help commands `?<` and `?>` allow to browse through a whole help book. `?<` displays the section preceding the previously shown section, and `?>` takes you to the section following the previously shown one.

2 ► `?>>`
► `?<<`

`?<<` takes you back to the first section of the current chapter, which gives an overview of the sections described in this chapter. If you are already in this section `?<<` takes you to the first section of the previous chapter. `?>>` takes you to the first section of the next chapter.

3 ► `?-`
► `?+`

GAP remembers the last few sections that you have read. `?-` takes you to the one that you have read before the current one, and displays it again. Further applications of `?-` take you further back in this history. `?+` reverses this process, i.e., it takes you back to the section that you have read after the current one. It is important to note that `?-` and `?+` do not alter the history like the other help commands.

4 ► `?books`

This command shows a list of books which are currently known to the help system. For each book there is a short name which is used with the *book* part of the basic help query and there is a long name which hopefully tells you what this book is about.

A short name which ends in `(not loaded)` refers to a GAP package whose documentation is loaded but which needs a call of `LoadPackage` (see 74.2.1) before you can use the described functions.

5 ► `?[`*book*`:]sections`
► `?[`*book*`:][chapters]`

These commands show tables of content for all available, respectively the matching books.

6 ► `?`
► `?&`

These commands redisplay the last shown help section. In the form `?&` the next preferred help viewer is used for the display (provided one has chosen several viewers), see 2.3.1 below.

2.3 Changing the Help Viewer

Books of the GAP help system can be available in several formats. Currently the following formats occur (not all of them may be available for all books):

text
: This is used for display in the terminal window in which GAP is running. Complicated mathematical expressions may not be well readable in this format.

dvi
: The standard output format of TEX. Only useful if TEX is installed on your system. Can be used for printing a help book and onscreen reading. Some books include hyperlink information in this format which can be useful for onscreen reading.

ps
: Postscript format. Can be printed on most systems and also be used with an onscreen viewer.

pdf
: Adobe's `pdf`-format. Can also be used for printing and onscreen reading on most current systems (with freely available software). Some books have hyperlink information included in this format.

HTML
: The format of Web-pages. Can be used with any Web-browser. There may be hyperlink information available which allows a convenient browsing through the book via cross-references. This format also has the problem that complicated formulae may be not well readable since there is no syntax

for formulae in HTML. Some books use special symbol fonts for formulae and need an appropriate Web-browser for correct display.

Depending on your operating system and available additional software you can use several of these formats with GAP's online help. This is configured with the following command.

1 ▶ `SetHelpViewer(` *viewer1*`,` *viewer2*`, ... )`

This command takes an arbitrary number of arguments which must be strings describing a viewer. The recognized viewer are explained below. A call with no arguments shows the current setting.

The first given arguments are those with higher priority. So, if a help section is available in the format needed by *viewer1*, this viewer is used. If not, availability of the format for *viewer2* is checked and so on. Recall that the command `?&` displays the last seen section again but with the next possible viewer in your list, see 2.2.6.

The viewer `"screen"` (see below) is always silently appended since we assume that each help book is available in text format.

If you want to change the default setting you will probably put a call of `SetHelpViewer` into your `.gaprc` file (see 3.4).

`"screen"`
: This is the default setting. The help is shown in text-format using the `Pager` command explained in the next section 2.4.1. (Hint: Some formatting procedures assume that your terminal displays at least 80 characters per line, if this is not the case some sections may look very bad. Furthermore the terminal (window) should use a fixed width font and we suggest to take one with `ISO-8859-1` (also called `latin1`) encoding.

`"firefox"`, `"mozilla"`, `"netscape"`, `"konqueror"`
: If a book is available in HTML-format this is shown using the corresponding web-browser. How well this works, for example by using a running instance of this browser, depends on your particular start script of this browser. Note, that for some books the browser must be configured to use symbol fonts.

`"w3m"`, `"lynx"`
: If a book is available in HTML-format this is shown using the text based `w3m` or `lynx` web-browser inside the terminal running GAP. Formulae which use symbol fonts may be unreadable.

`"mac default browser"`, `"safari"`
: (for Apple Macintosh) If a book is available in HTML-format this is shown in a web-browser. The web browser used is the program set to handle the `file` protocol in the `Internet` control panel (System 9 and System X). For some browsers (e.g., Internet Explorer), you may have to enter the GAP command `HELP_MAC_PROTOCOL := "file:/";` for this to work correctly. If you wish to use the online html version of the manual, you may use `HELP_EXTERNAL_URL := "http://www.gap-system.org/";`. Note that `HELP_EXTERNAL_URL := "";` switches back to the local html files. It may be a good idea to put the relevant line in the `gap.rc` file (see 3.4).

`"xdvi"`
: (on X-windows systems) If a book is available in dvi-format it is shown with the onscreen viewer program `xdvi`. (Of course, `xdvi` and TEX must be installed on your system.) This program doesn't allow remote commands, so usually for each shown topic a new `xdvi` is launched. You can try to compile the program `GAPPATH/etc/xrmtcmd.c` and to put the executable `xrmtcmd` into your `PATH`. Then this viewer tries to reuse one running `xdvi` for each help book.

`"xpdf"`
: (on X-windows systems) If a book is available in pdf-format it is shown with the onscreen viewer program `xpdf` (which must be installed on your system). This is a nice program, once it is running it is reused by GAP for the next displays of help sections. (Hint: On many systems `xpdf` shows a very

bad display quality, this is due to a wrong or missing font configuration. One needs to set certain X-resources; for more details follow the `Problems` link at

`http://www.foolabs.com/xpdf/`

`"acroread"`
: If a book is available in pdf-format it is shown with the onscreen viewer program `acroread` (which must be available on your system). This program doesn't allow remote commands or startup with a given page. Therefore the page numbers you have to visit are just printed on the screen. When you are looking at several sections of the same book, this viewer assumes that the acroread window still exists. When you go to another book a new acroread window is launched.

`"less"` or `"more"`
: This is the same as `"screen"` but additionally the `PAGER` and `PAGER_OPTIONS` variables are set, see the next section 2.4 for more details.

Please, send ideas for further viewer commands to `support@gap-system.org`.

2.4 The Pager Command

GAP contains a builtin pager which shows a text string which doesn't fit on the screen page by page. Its functionality is very rudimentary and self-explaining. This is because (at least under UNIX) there are powerful external standard programs which do this job.

1 ▶ `Pager(` *lines* `)`

This function can be used to display a text on screen using a pager, i.e., the text is shown page by page.

There is a default builtin pager in GAP which has very limited capabilities but should work on any system.

At least on a UNIX system one should use an external pager program like `less` or `more`. GAP assumes that this program has a command line option `+nr` which starts the display of the text with line number `nr`.

Which pager is used can be controlled by setting the variable `PAGER`. The default setting is `PAGER := "builtin";` which means that the internal pager is used.

On UNIX systems you probably want to set `PAGER := "less";` or `PAGER := "more";`, you can do this for example in your `.gaprc` file. In that case you can also tell GAP a list of standard options for the external pager. These are specified as list of strings in the variable `PAGER_OPTIONS`.

Example:

```
PAGER := "less";
PAGER_OPTIONS := ["-f", "-r", "-a", "-i", "-M", "-j2"];
```

The argument *lines* can have one of the following forms:

(1) a string (i.e., lines are separated by newline characters)

(2) a list of strings (without newline characters) which are interpreted as lines of the text to be shown

(3) a record with component `.lines` as in (1) or (2) and optional further components

In case (3) currently the following additional components are used:

`.formatted`
: can be `false` or `true`. If set to `true` the builtin pager tries to show the text exactly as it is given (avoiding GAPs automatic line breaking)

`.start`
: must be a positive integer. This is interpreted as the number of the first line shown by the pager (one may see the beginning of the text via back scrolling).

The `Pager` command is used by GAP's help system for displaying help sections in text-format. But, of course, it may be used for other purposes as well.

```
gap> s6 := SymmetricGroup(6);;
gap> words := ["This", "is", "a", "very", "stupid", "example"];;
gap> l := List(s6, p-> Permuted(words, p));;
gap> Pager(List(l, a-> JoinStringsWithSeparator(a," ")));;
```

3 Running GAP

This chapter informs about command line options for GAP under UNIX and OS X (see 3.1, 3.2), and features of GAP on the Macintosh (see 3.3), the `.gaprc` file (see 3.4), completion files (see 3.5), the GAP compiler (see 3.7, 3.8, 3.9), and how to save and load a GAP workspace (see 3.11).

3.1 Command Line Options

When you start GAP under UNIX, you may specify a number of options on the command-line to change the default behaviour of GAP. All these options start with a hyphen `-`, followed by a single letter. Options must not be grouped, e.g., `gap -gq` is illegal, use `gap -g -q` instead. Some options require an argument, this must follow the option and must be separated by a *space*, e.g., `gap -m 256k`, it is not correct to say `gap -m256k` instead.

GAP for UNIX will distinguish between upper and lower case options.

As is described in Chapter 73 (see 73), usually you will not execute GAP directly. Instead you will call a shell script, with the name `gap`, which in turn executes GAP. This shell script sets some options which are necessary to make GAP work on your system. This means that the default settings mentioned below may not be what you experience when you execute GAP on your system.

`-h`

tells GAP to print a summary of all available options (`-h` is mnemonic for "help"). GAP exits after printing the summary, all other options are ignored.

`-b`

tells GAP to suppress the banner. That means that GAP immediately prints the prompt. This is useful when, after a while, you get tired of the banner.

`-q`

tells GAP to be quiet. This means that GAP displays neither the banner nor the prompt `gap>`. This is useful if you want to run GAP as a filter with input and output redirection and want to avoid the banner and the prompts appearing in the output file.

`-e`

tells GAP not to quit when receiving a *ctr*`-D` on an empty input line (see 6.3.1). This option should not be used when the input is a file or pipe.

`-f`

tells GAP to enable the line editing and history (see 6.8).
In general line editing will be enabled if the input is connected to a terminal. There are rare circumstances, for example when using a remote session with a corrupted telnet implementation, when this detection fails. Try using `-f` in this case to enable line editing.

`-n`

tells GAP to disable the line editing and history (see 6.8).
You may want to do this if the command line editing is incompatible with another program that is used to run GAP. For example if GAP is run from inside a GNU Emacs shell window, `-n` should be

used since otherwise every input line will be echoed twice, once by Emacs and once by GAP.

`-x` *length*
: With this option you can tell GAP how long lines are. GAP uses this value to decide when to split long lines. After starting GAP you may use `SizeScreen` (see 6.11.1) to alter the line length.
The default value is 80, unless another value can be obtained from the Operating System, which is the right value if you have a standard ASCII terminal. If you have a larger monitor, or use a smaller font, or redirect the output to a printer, you may want to increase this value.

`-y` *length*
: With this option you can tell GAP how many lines your screen has. GAP uses this value to decide after how many lines of on-line help it should wait. After starting GAP you may use `SizeScreen` (see 6.11.1) to alter the number of lines.
The default value is 24, unless another value can be obtained from the Operating System, which is the right value if you have a standard ASCII terminal. If you have a larger monitor, or use a smaller font, or redirect the output to a printer, you may want to increase this value.

`-g`
: tells GAP to print a information message every time a full garbage collection is performed.

```
#G  FULL 44580/2479kb live   57304/4392kb dead   734/4096kb free
```

For example, this tells you that there are 44580 live objects that survived a full garbage collection, that 57304 unused objects were reclaimed by it, and that 734 KBytes from a total allocated memory of 4096 KBytes are available afterwards.

`-g -g`
: If you give the option `-g` twice, GAP prints a information message every time a partial or full garbage collection is performed. The message,

```
#G  PART 9405/961kb+live   7525/1324kb+dead   2541/4096kb free
```

for example, tells you that 9405 objects survived the partial garbage collection and 7525 objects were reclaimed, and that 2541 KBytes from a total allocated memory of 4096 KBytes are available afterwards.

`-m` *memory*
: tells GAP to allocate *memory* bytes at startup time. If the last character of *memory* is `k` or `K` it is taken as KBytes, if the last character is `m` or `M` *memory* is taken as MBytes and if it is 'g' or 'G' it is taken as Gigabytes.
Under UNIX the default amount of memory allocated by GAP is 24 MBytes. The amount of memory should be large enough so that computations do not require too many garbage collections. On the other hand, if GAP allocates more virtual memory than is physically available, it will spend most of the time paging.

`-o` *memory*
: tells GAP to allocate at most *memory* bytes. If the last character of *memory* is `k` or `K` it is taken as KBytes, if the last character is `m` or `M` *memory* is taken as MBytes and if it is 'g' or 'G' it is taken as Gigabytes.
Under UNIX the default amount is 256 MBytes. If more than this amount is required during the GAP session, GAP prints an error messages and enters a break loop.

`-K` *memory*
: is like `-o` above. But while the latter actually allocates more memory if the system allows it and then prints a warning inside a break loop the `-K` options tells GAP not even to try to allocate more

memory. Instead GAP just exits with an appropriate message. The default is that this feature is switched off. You have to set it explicitly when you want to enable it.

`-l` *path_list*

can be used to modify GAP's list of root directories (see 9.2). Before the option `-l` is used for the first time, the only root directory is `./`, i.e., GAP has only one root directory which is the current directory. Usually this option is used inside a startup script to specify where GAP is installed on the system. The `-l` option can also be used by individual users to tell GAP about privately installed modifications of the library, additional GAP packages and so on. Section 9.2 explains how several root paths can be used to do this.

path_list should be a list of directories separated by semicolons. No whitespace is permitted before or after a semicolon. Each directory name should end with a pathname separator, i.e., `/`, but GAP will silently add one if it is missing. If *path_list* does not start or end with a semicolon, then *path_list* replaces the existing list of root directories. If *path_list* starts with a semicolon, then *path_list* is appended to the existing list of root directories. If *path_list* ends with a semicolon (and does not start with one), then the new list of root directories is the concatenation of *path_list* and the existing list of root directories. After GAP has completed its startup procedure and displays the prompt, the list of root directories can be viewed in the variable `GAPInfo.RootPaths`.

GAP will attempt to read the file *root_dir*`/lib/init.g` during startup where *root_dir* is one of the directories in its list of root directories. If GAP cannot find `init.g` it will print the following warning

```
gap: hmm, I cannot find 'lib/init.g' maybe use option '-l <gaproot>'?
```

It is not possible to use GAP without the library files, so you must not ignore this warning. You should leave GAP and start it again, specifying the correct root path using the `-l` option.

`-r`

The option `-r` tells GAP not to read the user supplied `~/.gaprc` files.

`-L` *filename*

The option `-L` tells GAP to load a saved workspace. See section 3.11.

`-R`

The option `-R` tells GAP not to load a saved workspace previously specified via the `-L` option.

filename `...`

Further arguments are taken as filenames of files that are read by GAP during startup, after the system and private init files are read, but before the first prompt is printed. The files are read in the order in which they appear on the command line. GAP only accepts 14 filenames on the command line. If a file cannot be opened GAP will print an error message and will abort.

3.2 Advanced Features of GAP

The following options are in general not needed for the normal operation of GAP. They are mostly used for debugging.

`-a` *memory*

GASMAN, the storage manager of GAP uses `sbrk` to get blocks of memory from (certain) operating systems and it is required that subsequent calls to `sbrk` produce adjacent blocks of memory in this case because GAP only wants to deal with one large block of memory. If the C function `malloc` is called for whatever reason, it is likely that `sbrk` will no longer produce adjacent blocks, therefore GAP does not use `malloc` itself.

However some operating systems insist on calling `malloc` to create a buffer when a file is opened, or for some other reason. In order to catch these cases GAP preallocates a block of memory with `malloc` which is immediately freed. The amount preallocated can be controlled with the `-a` option.

If the last character of *memory* is `k` or `K` it is taken as KBytes and if the last character is `m` or `M` *memory* is taken as MBytes.

`-A`

By default, some GAP packages (see 74) are loaded, if present, into the GAP session when it starts. This option disables (actually toggles) this behaviour, which can be useful for debugging or testing.

`-B` *architecture*

Executable binary files that form part of GAP or of a GAP package are kept in a subdirectory of the `bin` directory with in the GAP or package root directory. The subdirectory name is determined from the operating system, processor and compiler details when GAP (resp. the package) is installed. Under rare circumstances, it may be necessary to override this name, and this can be done using the `-B` option.

`-D`

The `-D` option tells GAP to print short messages when it is reading or completing files or loading modules. The message,

```
#I  READ_GAP_ROOT: loading 'lib/kernel.g' as GAP file
```

tells you that GAP has started to read the library file `lib/kernel.g`.

```
#I  READ_GAP_ROOT: loading 'lib/kernel.g' statically
```

tells you that GAP has used the compiled version of the library file `lib/kernel.g`. This compiled module was statically linked to the GAP kernel at the time the kernel was created.

```
#I  READ_GAP_ROOT: loading 'lib/kernel.g' dynamically
```

tells you that GAP has loaded the compiled version of the library file `lib/kernel.g`. This compiled module was dynamically loaded to the GAP kernel at runtime from a corresponding `.so` file.

```
#I  completing 'lib/domain.gd'
```

tells you that GAP has completed the file `lib/domain.gd`. See 3.5 for more information about completion of files.

`-M`

tells GAP not to check for, nor to use, compiled versions of library files.

`-N`

tells GAP not to check for, nor to use, completion files, see 3.5.

`-O`

enables a GAP 3 compatibility mode, in which (for instance) the values `false` and `fail` are identified. Use of this mode is not recommended other than as a transitional step in porting GAP 3 code to GAP 4, because the GAP 4 library may not work reliably in this mode.

`-T`

suppresses the usual break loop behaviour of GAP. With this option GAP behaves as if the user `quit` immediately from every break loop. This is intended for automated testing of GAP.

`-X`

tells GAP to do a consistency check of the library file and the corresponding completion file when reading the completion file.

`-Y`

tells GAP to do a consistency check of the library file and the corresponding completion file when completing the library file.

`-i` *filename*

changes the name of the init file from the default `init.g` to *filename*.

Additional options, `-C`, `-U`, `-P`, `-W` and `-z` are used internally in the GAP compiler and/or on specific operating systems.

3.3 Running GAP under MacOS

This sections describes the features of GAP for MacOS that differ from those described earlier in this chapter.

Since you cannot enter command line options directly when you launch the GAP application on a Macintosh, another mechanism is being used: Hold down any of the command (apple), option, control or shift keys or space bar when launching the GAP application, e.g., by double-clicking on its icon. Please note that some keys have side effects (e.g., pressing the option key usually closes Findeer windows), and that System X behaves slightly differently from other systems.

A dialog box will open, into which you can enter the desired GAP command line options. as described in 3.1. For example, if you want GAP to start with a workspace of 32 megabytes, the dialog box should contain the following text:

```
-m 32m
```

Note that the dialog box may already contain settings which you have previously saved. The `OK` button accepts the command line for the current GAP session, and the `Save` button can be used to save these options for subsequent GAP sessions. The command line options will be saved in a text file called `GAP options` in the `Preferences` folder in the system folder. You may also modify the file `GAP options` directly; note that changes only take effect the next time you launch GAP.

There are three additional command line option on the Mac.

`-z` *n*
: sets the time between checks for events (keystrokes, mouse clicks etc.) to $n/60$ second. Lower values make GAP more responsive but computations are somewhat slower. A value greater than 60 is not recommended, the default value for n is 6.

`-P` *m*
: sets the amount of memory required for printing. The reason is that printer drivers may require quite a bit of memory, and may even crash if not enough is found. To prevent this, GAP will not print unless at least the specified amount of memory is available. The default value is 64 Kilobytes, which is enough for the Apple LaserWriter printer driver. Setting the printing memory to 0 disables printing altogether.

`-W` *m*
: sets the size of the log window to m bytes. This means that if the text in the log window exceeds this amount, then lines at the beginning of the log are deleted. The default value is 32 Kilobytes.

The following command line options work differently on the Mac.

`-a`
: On the Mac, the `-a` option has a different meaning from the one described in 3.2. On the Mac, it must be used to reserve memory for loading dynamic libraries into GAP. See 3.7 for details about dynamic libraries (and note that the PPC version of GAP for MacOS **can** use dynamic libraries).

`-f`, `-n`
: The `-f` and `-n` command line options do not have any effect on the Mac.

`-e`
: The `-e` command line option enables *ctr*`-D`.

`-o`
: The `-o` command line option should not normally be used on the Mac. The value set by the `-o` option is only used if it is lower than the size of the workspace that would normally be available for GAP.

The file called `.gaprc` on UNIX systems (see 3.4) is called `gap.rc` on the Mac; it must be in the same folder as the GAP application.

All interaction between GAP and you takes place via the `GAP log` window: this is where GAP prints its messages and waits for your input. The amount of text in this window is limited (see the `-W` command line option above), so don't be surprised if old GAP messages are deleted from the beginning of the text when this limit is reached. The reason for deleting old lines is that otherwise GAP may run out of memory just because of the messages it has printed.

GAP for the Mac now remembers the font and text size (which can be set choosing `Format...` in the `Edit` menu) as well as the window position of the GAP log window from one session to the next.

Almost all of the GAP editing keys described in Section 6.8 work on the Mac. In addition, GAP for MacOS also supports the usual editing keys on the Mac, such as Copy and Paste, Undo, arrow keys (also with *shift*, *option* and *command*. Note that you can also move forward and backward in the command line history by pressing *ctrl-arrow down* and *ctrl-arrow up*.

Note that `Quit` in GAP's file menu works differently from the `quit` GAP command (see 6.3.1): `Quit` in the file menu always quits the GAP application, it cannot be used to quit from a break loop.

GAP for MacOS also contains a simple built-in text editor, which is mainly intended to create GAP files. `New`, `Open...`, `Save` and `Close` from the `File` menu work in the usual way.

The `Read...` and `LogTo` commands in the `File` menu work basically like the corresponding GAP commands (see 9.7). The only difference is that GAP will prompt you for the file with a standard Mac file opening dialog, so you do not have to enter the path name yourself. (You will see the file's path name in the log window afterwards). Note that if a file you want to read is open in GAP's built-in editor, then GAP will read the file from the edit window, not from the disk.

If you press the shift key while choosing `Read...` from the `File` menu, the menu item will change to `Reread...` which will then use the GAP command `Reread` (see 9.7.13) to read the chosen file.

The `Read...` command in the `File` menu changes to `Read` if the front window belongs to a file in GAP's built-in editor – choosing `Read` then makes GAP read that file – and while the file is being read, the `File` menu item changes to `Abort Read`. You cannot close the file's window while it is being read by GAP – choose `Abort Read` first.

Garbage collection messages, which are switched on and off by the `-g` command line option (see 3.1) can also be switched on and off by choosing `Show garbage collections` and `Show partial collections` from the `Window` menu.

If `Always scroll to printout` is selected in the `Window` menu, GAP will always scroll the `GAP log` window so that you can see what GAP is currently printing. Otherwise, the GAP log window is only scrolled to the current print position when GAP prints its prompt and waits for you to enter a command. Note that you may see text lines disappear even if `Always scroll to printout` is off – this happens if you are viewing the text at the beginning of the log window and some lines are just being deleted from the log because it has exceeded its 32000 character limit.

The contents of the `Help` menu should be quite self-explanatory. Note that, unlike in GAP 3 for the Mac, the online help is not displayed in a separate window, nor is the online help available while GAP is computing.

Holding down the Command (Apple) key while selecting text does the same as selecting the text and choosing `Find selection in table of contents` from the `Help` menu, holding down both Command and Option keys while selecting tries to find the selection in the index.

When you want to refer to files or folders in GAP (for example in the `Read`, `PrintTo`, `AppendTo`, `LogTo` commands), or have to specify files or folders for a command line option, these files must be identified by UNIX style path names. (Presently, GAP for MacOS also supports Mac path names, but this may change in the future.)

Users who are familiar with UNIX path names may skip the rest of this section, noting that the working directory (i.e., folder) is the one in which the GAP application resides, and that file names on the Mac are **not** case sensitive.

Paths are strings used to describe where a file is stored on a hard disk. There are two ways for specifying UNIX path names: absolute and relative paths. An absolute path starts with a `/`, then the name of the disk where the file is located, another `/`, then a list of folders, each containing the next one, separated by `/`, and finally the name of the file, which resides in the last folder in the list. For instance, if your hard disk is called `My HD`, and your file `program.g` resides (or should be created) in the folder `programs` in the folder `documents` on `My HD`, the absolute path name to that file is

```
/My HD/documents/programs/program.g
```

Relative path names work similarly, except that the starting point is not a disk but the folder in which the GAP application program resides. Relative path names are formed like absolute ones, except that they do not start with a `/`. Thus, if you want to access the file `temp.g` in the folder `tmp` in the GAP folder, you may use the following path name: `tmp/temp.g`. It is also possible to move upward to a parent folder: suppose that the folder containing GAP is called `applications`, which contains a folder `editor` which in turn contains the file 'program.g', then you could access this file by the path `../editor/program.g`. The path `./` refers to the GAP folder itself, and `../` refers to "the folder above".

Note also that GAP for the Mac follows (resolves) aliases to folders and files.

3.4 The .gaprc file

When you start GAP, it looks for the file with the name `.gaprc` in your home directory (on UNIX systems). On a Macintosh or a Windows system the equivalent to the `.gaprc` file is `gap.rc`, and for it to be read it must be in the same folder as the GAP application. (Note that the file must be called `gap.rc`. If you use a Windows text editor, in particular if your default is not to show file suffixes, you might accidentaly create a file `gap.rc.txt` or `gap.rc.doc` which GAP will not recognize.)

If such a file is found it is read after *libname*`/init.g`, but before any of the files mentioned on the command line are read. You can use this file for your private customizations. For example, if you have a file containing functions or data that you always need, you could read this from `.gaprc`. Or if you find some of the names in the library too long, you could define abbreviations for those names in `.gaprc`. The following sample `.gaprc` file does both.

```
Read("/usr/you/dat/mygroups.grp");
Ac := Action;
AcHom := ActionHomomorphism;
RepAc := RepresentativeAction;
```

If you have already a `.gaprc` file for GAP 3, its settings might not be compatible with GAP 4. In this case it has to be removed. On UNIX Systems the following `.gaprc` file can be used to load alternatively a `.gap3rc` or a `.gap4rc` file from your home directory.

```
if IsBound(Permutations) then
  # GAP 3
  Exec("echo \"READ(\\\"`pwd ~`/.gap3rc\\\");\" > /tmp/jJj");
else
  # GAP 4
  Exec("echo \"READ(\\\"`pwd ~`/.gap4rc\\\");\" > /tmp/jJj");
fi;
Read("/tmp/jJj");
```

3.5 Completion Files

The standard distribution of GAP already contains completion files so in general **you do not need to create these files by yourself**.

When starting, GAP reads in the whole library. As this takes some time, library files are normally condensed into completion files. These completion files contain the basic skeleton of the library but not the function bodies. When a function body is required, for example because you want to execute the corresponding function, the library file containing the function body is completed.

Completion files reduce the startup time of GAP drastically. However, this technique also means that the information stored in the completion files and the library must be consistent. If you change a library file without recreating the completion files disaster is bound to happen.

Bugfixes distributed for GAP will also update the completion files. Therefore you only need to update them if you have changed the library by yourself.

However, if you are modifying a library file a more convenient way is to use the `-X` option (see 3.1) that allows you (in most cases) to use the completion files for the unchanged parts of library files and avoids using the completion files for the changed parts. After you have finished modifying the library files you can recreate the completion files using:

1 ▶ `CreateCompletionFiles( )` F
▶ `CreateCompletionFiles(` *path* `)` F

To create completion files you must have write permissions to *path*, which defaults to the first root directory. Start GAP with the `-N` option (to suppress the reading of any existing completion files), then execute the command `CreateCompletionFiles(` *path* `);`, where *path* is a string giving a path to the home directory of GAP (the directory containing the `lib` directory).

This produces, in addition to lots of informational output, the completion files.

```
$ gap4 -N
gap> CreateCompletionFiles();
#I  converting "gap4/lib/read2.g" to "gap4/lib/read2.co"
#I    parsing "gap4/lib/process.gd"
#I    parsing "gap4/lib/listcoef.gi"
...
```

3.6 Testing for the System Architecture

1 ▶ `ARCH_IS_UNIX( )` F

tests whether GAP is running on a UNIX system.

2 ▶ `ARCH_IS_MAC( )` F

tests whether GAP is running on a Macintosh under MacOS

3 ▶ `ARCH_IS_WINDOWS( )` F

tests whether GAP is running on a Windows system.

3.7 The Compiler

The GAP compiler GAC creates C code from GAP code and then calls the system's C compiler to produce machine code from it. This can result in a speedup (see section 3.8 for more details).

To use the compiler to produce dynamically loadable modules, call it with the `-d` option:

```
M193 /home/ahulpke > gap4/bin/i386-ibm-linux-gcc2/gac -d test.g
gap4/bin/i386-ibm-linux-gcc2/gap -C /tmp/5827_test.c test.g Init_Dynamic
gcc -fpic -ansi -Wall -O2 -o /tmp/5827_test.o -I
gap4/bin/i386-ibm-linux-gcc2/../../src -c /tmp/5827_test.c
ld -Bshareable -x -o test.so /tmp/5827_test.o
rm -f /tmp/5827_test.o
rm -f /tmp/5827_test.c
```

This produces a file *file*`.so`.

1 ▶ `LoadDynamicModule(` *filename* `)` F
▶ `LoadDynamicModule(` *filename*`,` *crc* `)` F

To load a compiled file, the command `LoadDynamicModule` is used. This command loads *filename* as module. If given, the CRC checksum *crc* must match the value of the module (see 3.10).

```
gap> LoadDynamicModule("./test.so");
gap> CrcFile("test.g");
2906458206
gap> LoadDynamicModule("./test.so",1);
Error, <crc> mismatch (or no support for dynamic loading) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap> LoadDynamicModule("./test.so",2906458206);
```

If you want to see or modify the intermediate C code, you can also instruct the compiler to produce only the C files by using the option `-C` instead of `-d`.

There are some known problems with C code produced with the GAP compiler on 32 bit architectures and used on 64 bit architectures (and vice versa).

On some operating systems, once you have loaded a dynamic module with a certain filename, loading another with the same filename will have no effect, even if the file on disk has changed.

3.8 Suitability for Compilation

Typically algorithms spend large parts of their runtime only in small parts of the code. The design of GAP reflects this situation with kernel methods for many time critical calculations such as matrix or permutation arithmetic.

Compiling an algorithm whose time critical parts are already in the kernel of course will give disappointing results: Compilation will only speed up the parts that are not already in the kernel and if they make us a small part of the runtime, the overall gain is small.

Routines that benefit from compilation are those which do extensive operations with basic data types, such as lists or small integers.

3.9 Compiling Library Code

The most tempting code to compile is probably the library. This section describes the mechanism used to make GAP recognize compiled versions of library files. Note however that there is no point in compiling the whole library as typically only few functions benefit from compilation as described in Section 3.8.

All files that come with GAP are read using the internal function `READ_GAP_ROOT`. This function then checks whether a compiled version of the file exists and if its CRC number (see 3.10) matches the file. If it does, the compiled version is loaded. Otherwise the file is read. You can start GAP with the `-D -N` option to see information printed about this process.

To make GAP find the compiled versions, they must be put in the `bin/`*systemname*`/compiled` directory (*systemname* is the name you gave for compilation, for example `i386-ibm-linux-gcc2`). They have to be called according to the following scheme: Suppose the file is `humpty/dumpty.gi` in the GAP home directory. Then the compiled version will be `bin/`*systemname*`/compiled/humpty/gi/dumpty.so`. That is, the directory hierarchy is mirrored under the `compiled` directory. A further directory level is added for the suffix of the file, and the suffix of the compiled version of the file is set to `.so` (as produced by the compiler).

For example we show how to compile the `combinat.gi` file on a Linux machine. Suppose we are in the home directory of the gap distribution.

```
bin/gac -d lib/combinat.gi
```

creates a file `combinat.so`. We now put it in the right place, creating also the necessary directories:

```
mkdir bin/i386-ibm-linux-gcc2/compiled
mkdir bin/i386-ibm-linux-gcc2/compiled/lib
mkdir bin/i386-ibm-linux-gcc2/compiled/lib/gi
mv combinat.so bin/i386-ibm-linux-gcc2/compiled/lib/gi
```

If you now start GAP and look, for example, at the function `Binomial`, defined in `combinat.gi`, you see it is indeed compiled:

```
gap> Print(Binomial);
function ( <<arg-1>>, <<arg-2>> )
    <<compiled code>>
end
```

The command line option `-M` disables the loading of compiled modules and always reads code from the library.

3.10 CRC Numbers

CRC (cyclic redundancy check) numbers provide a certain method of doing checksums. They are used by GAP to check whether files have changed. Whenever files are "condensed" – for example for completion files (see Section 3.5) or when compiling files (see Section 3.7) – such a checksum is computed implicitly and stored within the condensed file.

When reading a condensed version of the file instead of the original one, the CRC checksum, which is computed via `CrcFile` (see 9.7.11), can be used to check whether the original has been changed in the meantime, e.g.

```
gap> CrcFile("lib/morpheus.gi");
2705743645
```

3.11 Saving and Loading a Workspace

1 ▶ `SaveWorkspace(` *filename* `)` F

will save a "snapshot" image of the current GAP workspace in the file *filename*. This image then can be loaded by another copy of GAP which then will behave as at the point when `SaveWorkspace` was called.

```
gap> a:=1;
gap> SaveWorkspace("savefile");
true
gap> quit;
```

`SaveWorkspace` can only be used at the main `gap>` prompt. It cannot be included in the body of a loop or function, or called from a break loop.

2 ▶ `-L` *filename*

A saved workspace can be loaded by starting GAP with the option `-L` (see 3.1). This will start GAP and load the workspace.

```
you@unix> gap -L savefile
gap> a;
1
```

Please note that paths to workspaces have to be given in full, expansion of the tilde to denote a home directory will **not** work.

Under UNIX, it is possible to compress savefiles using `gzip`. Compression typically reduces the size of a workspace by a factor 3 or 4. If GAP is started with a compressed savefile (omit the `.gz` ending), it will try to locate `gzip` on the system and uncompress the file automatically while reading it.

```
you@unix> gzip -9 savefile
you@unix> gap -L savefile
gap> a;
1
```

We cannot guarantee that saved workspaces are portable between different system architectures or over different versions of GAP or its library.

If compiled modules had been loaded into GAP before the workspace was saved, they will be loaded into the new GAP session during the workspace loading process. If they are not available then the load will fail. Additional compiled modules will **not** be used, even if they are available, although they may be loaded later using `Reread` (see 9.7.13). `SaveWorkspace` may sometimes produce warning messages, as in

```
gap> SaveWorkspace("b5");
#W bad bag id 4 found, 0 saved
#W bad bag id 20 found, 0 saved
true
```

A small number of such messages can probably be ignored (they arise because the garbage collector may not always collect all dead objects, and dead objects may contain data that `SaveWorkspace` does not know how to process).

GAP packages which had been loaded before the workspace was saved are loaded also when the workspace is loaded. Packages which had been available but not loaded before the workspace was saved are available also when the workspace is loaded, provided these packages have not been upgraded. Packages which have been newly installed **after** the workspace was saved are **not** available when the workspace is loaded.

3.12 Coloring the Prompt and Input

GAP provides hooks for functions which are called when the prompt is to be printed and when an input line is finished.

An example of using this feature is the following function.

1 ▶ `ColorPrompt( bool )` F

With `ColorPrompt(true);` GAP changes its user interface: The prompts and the user input are displayed in different colors. It also sets the variable `ANSI_COLORS` to `true` (which has the side effect that some help pages are also displayed with color markup. Switch the colored prompts off with `ColorPrompt(false);`.

Note that this will only work if your terminal emulation in which you run GAP understands the so called ANSI color escape sequences - almost all terminal emulations on current UNIX/Linux (`xterm`, `rxvt`, `konsole`, ...) systems do so.

The colors shown depend on the terminal configuration and cannot be forced from an application. If your terminal follows the ANSI conventions you see the standard prompt in bold blue and the break loop prompt in bold red, as well as your input in red.

If it works for you and you like it, put the line `ColorPrompt(true);` in your `.gaprc` file (see 3.4).

4 The Programming Language

This chapter describes the GAP programming language. It should allow you in principle to predict the result of each and every input. In order to know what we are talking about, we first have to look more closely at the process of interpretation and the various representations of data involved.

4.1 Language Overview

First we have the input to GAP, given as a string of characters. How those characters enter GAP is operating system dependent, e.g., they might be entered at a terminal, pasted with a mouse into a window, or read from a file. The mechanism does not matter. This representation of expressions by characters is called the **external representation** of the expression. Every expression has at least one external representation that can be entered to get exactly this expression.

The input, i.e., the external representation, is transformed in a process called **reading** to an internal representation. At this point the input is analyzed and inputs that are not legal external representations, according to the rules given below, are rejected as errors. Those rules are usually called the **syntax** of a programming language.

The internal representation created by reading is called either an **expression** or a **statement**. Later we will distinguish between those two terms. However for now we will use them interchangeably. The exact form of the internal representation does not matter. It could be a string of characters equal to the external representation, in which case the reading would only need to check for errors. It could be a series of machine instructions for the processor on which GAP is running, in which case the reading would more appropriately be called compilation. It is in fact a tree-like structure.

After the input has been read it is again transformed in a process called **evaluation** or **execution**. Later we will distinguish between those two terms too, but for the moment we will use them interchangeably. The name hints at the nature of this process, it replaces an expression with the value of the expression. This works recursively, i.e., to evaluate an expression first the subexpressions are evaluated and then the value of the expression is computed from those values according to rules given below. Those rules are usually called the **semantics** of a programming language.

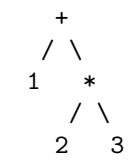
The result of the evaluation is, not surprisingly, called a **value**. Again the form in which such a value is represented internally does not matter. It is in fact a tree-like structure again.

The last process is called **printing**. It takes the value produced by the evaluation and creates an external representation, i.e., a string of characters again. What you do with this external representation is up to you. You can look at it, paste it with the mouse into another window, or write it to a file.

Lets look at an example to make this more clear. Suppose you type in the following string of 8 characters

```
1 + 2 * 3;
```

GAP takes this external representation and creates a tree-like internal representation, which we can picture as follows



This expression is then evaluated. To do this GAP first evaluates the right subexpression `2*3`. Again, to do this GAP first evaluates its subexpressions 2 and 3. However they are so simple that they are their own value, we say that they are self-evaluating. After this has been done, the rule for `*` tells us that the value is the product of the values of the two subexpressions, which in this case is clearly 6. Combining this with the value of the left operand of the `+`, which is self-evaluating, too, gives us the value of the whole expression 7. This is then printed, i.e., converted into the external representation consisting of the single character `7`.

In this fashion we can predict the result of every input when we know the syntactic rules that govern the process of reading and the semantic rules that tell us for every expression how its value is computed in terms of the values of the subexpressions. The syntactic rules are given in sections 4.2, 4.3, 4.4, 4.5, 4.6, and 4.24, the semantic rules are given in sections 4.7, 4.8, 4.10, 4.11, 4.12, 4.13, 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.22.1, and the chapters describing the individual data types.

4.2 Lexical Structure

The input of GAP consists of sequences of the following characters.

Digits, uppercase and lowercase letters, *space*, *tab*, *newline*, and the special characters

```
"    `    (    )    *    +    ,    _    #
.    /    :    ;    <    =    >    ~    &
[    \    ]    ^    _    {    }    !
```

Other characters will be signalled as illegal. Inside strings (see section 4.3 and chapter 26) and comments (see 4.4) the full character set supported by the computer is allowed.

4.3 Symbols

The process of reading, i.e., of assembling the input into expressions, has a subprocess, called **scanning**, that assembles the characters into symbols. A **symbol** is a sequence of characters that form a lexical unit. The set of symbols consists of keywords, identifiers, strings, integers, and operator and delimiter symbols.

A **keyword** is a reserved word consisting entirely of lowercase letters (see 4.5). An **identifier** is a sequence of letters and digits that contains at least one letter and is not a keyword (see 4.6). An integer is a sequence of digits (see 14). A **string** is a sequence of arbitrary characters enclosed in double quotes (see 26).

Operator and delimiter symbols are

```
+    -    *    /    ^    ~    !.
=    <>   <    <=   >    >=   ![
:=   .    ..   ->   ,    ;    !{
[    ]    {    }    (    )     :
```

Note also that during the process of scanning all whitespace is removed (see 4.4).

4.4 Whitespaces

The characters *space*, *tab*, *newline*, and *return* are called **whitespace characters**. Whitespace is used as necessary to separate lexical symbols, such as integers, identifiers, or keywords. For example `Thorondor` is a single identifier, while `Th or ondor` is the keyword `or` between the two identifiers `Th` and `ondor`. Whitespace may occur between any two symbols, but not within a symbol. Two or more adjacent whitespace characters are equivalent to a single whitespace. Apart from the role as separator of symbols, whitespace characters are otherwise insignificant. Whitespace characters may also occur inside a string, where they are significant. Whitespace characters should also be used freely for improved readability.

A **comment** starts with the character `#`, which is sometimes called sharp or hatch, and continues to the end of the line on which the comment character appears. The whole comment, including `#` and the *newline* character is treated as a single whitespace. Inside a string, the comment character `#` loses its role and is just an ordinary character.

For example, the following statement

```
if i<0 then a:=-i;else a:=i;fi;
```

is equivalent to

```
if i < 0 then   # if i is negative
  a := -i;      #   take its additive inverse
else            # otherwise
  a := i;       #   take itself
fi;
```

(which by the way shows that it is possible to write superfluous comments). However the first statement is **not** equivalent to

```
ifi<0thena:=-i;elsea:=i;fi;
```

since the keyword `if` must be separated from the identifier `i` by a whitespace, and similarly `then` and `a`, and `else` and `a` must be separated.

4.5 Keywords

Keywords are reserved words that are used to denote special operations or are part of statements. They must not be used as identifiers. The keywords are

```
and     do       elif   else    end     fi
for     function if     in      local   mod
not     od       or     repeat  return  then
until   while    quit   QUIT    break   rec
continue
```

Note that (almost) all keywords are written in lowercase and that they are case sensitive. For example only `else` is a keyword; `Else`, `eLsE`, `ELSE` and so forth are ordinary identifiers. Keywords must not contain whitespace, for example `el if` is not the same as `elif`.

Note: A number of tokens that appear to be normal identifiers representing functions or literals of various kinds are actually implemented as keywords for technical reasons. The only consequence of this is that those identifiers cannot be re-assigned, and do not actually have function objects bound to them, which could be assigned to other variables or passed to functions. These keywords are:

```
false  true IsBound Unbind TryNextMethod
Info Assert SaveWorkspace   fail
```

4.6 Identifiers

An **identifier** is used to refer to a variable (see 4.8). An identifier consists of letters, digits, and underscores _, and must contain at least one letter or underscore. An identifier is terminated by the first character not in this class. Examples of valid identifiers are

```
a           foo         aLongIdentifier
hello       Hello       HELLO
x100        100x       _100
some_people_prefer_underscores_to_separate_words
WePreferMixedCaseToSeparateWords
```

Note that case is significant, so the three identifiers in the second line are distinguished.

The backslash \ can be used to include other characters in identifiers; a backslash followed by a character is equivalent to the character, except that this escape sequence is considered to be an ordinary letter. For example

```
G\(2\,5\)
```

is an identifier, not a call to a function `G`.

An identifier that starts with a backslash is never a keyword, so for example `\*` and `\mod` are identifiers.

The length of identifiers is not limited, however only the first 1023 characters are significant. The escape sequence *newline* is ignored, making it possible to split long identifiers over multiple lines.

1 ▶ `IsValidIdentifier(` *str* `)` F

returns `true` if the string *str* would form a valid identifier consisting of letters, digits and underscores; otherwise it returns `false`. It does not check whether *str* contains characters escaped by a backslash \.

4.7 Expressions

An **expression** is a construct that evaluates to a value. Syntactic constructs that are executed to produce a side effect and return no value are called **statements** (see 4.13). Expressions appear as right hand sides of assignments (see 4.14), as actual arguments in function calls (see 4.10), and in statements.

Note that an expression is not the same as a value. For example `1 + 11` is an expression, whose value is the integer 12. The external representation of this integer is the character sequence `12`, i.e., this sequence is output if the integer is printed. This sequence is another expression whose value is the integer 12. The process of finding the value of an expression is done by the interpreter and is called the **evaluation** of the expression.

Variables, function calls, and integer, permutation, string, function, list, and record literals (see 4.8, 4.10, 14, 40, 26, 4.22.1s, 21, 27), are the simplest cases of expressions.

Expressions, for example the simple expressions mentioned above, can be combined with the operators to form more complex expressions. Of course those expressions can then be combined further with the operators to form even more complex expressions. The **operators** fall into three classes. The **comparisons** are `=`, `<>`, `<`, `<=`, `>`, `>=`, and `in` (see 4.11 and 28.5). The **arithmetic operators** are `+`, `-`, `*`, `/`, `mod`, and `^` (see 4.12). The **logical operators** are `not`, `and`, and `or` (see 20.3).

The following example shows a very simple expression with value 4 and a more complex expression.

```
gap> 2 * 2;
4
gap> 2 * 2 + 9 = Fibonacci(7) and Fibonacci(13) in Primes;
true
```

For the precedence of operators, see 4.11.

4.8 Variables

A **variable** is a location in a GAP program that points to a value. We say the variable is **bound** to this value. If a variable is evaluated it evaluates to this value.

Initially an ordinary variable is not bound to any value. The variable can be bound to a value by **assigning** this value to the variable (see 4.14). Because of this we sometimes say that a variable that is not bound to any value has no assigned value. Assignment is in fact the only way by which a variable, which is not an argument of a function, can be bound to a value. After a variable has been bound to a value an assignment can also be used to bind the variable to another value.

A special class of variables is the class of **arguments** of functions. They behave similarly to other variables, except they are bound to the value of the actual arguments upon a function call (see 4.10).

Each variable has a name that is also called its **identifier**. This is because in a given scope an identifier identifies a unique variable (see 4.6). A **scope** is a lexical part of a program text. There is the **global scope** that encloses the entire program text, and there are local scopes that range from the `function` keyword, denoting the beginning of a function definition, to the corresponding `end` keyword. A **local scope** introduces new variables, whose identifiers are given in the formal argument list and the `local` declaration of the function (see 4.22.1). Usage of an identifier in a program text refers to the variable in the innermost scope that has this identifier as its name. Because this mapping from identifiers to variables is done when the program is read, not when it is executed, GAP is said to have **lexical scoping**. The following example shows how one identifier refers to different variables at different points in the program text.

```
g := 0;      # global variable g
x := function ( a, b, c )
  local  y;
  g := c;     # c refers to argument c of function x
  y := function ( y )
    local d, e, f;
    d := y;   # y refers to argument y of function y
    e := b;   # b refers to argument b of function x
    f := g;   # g refers to global variable g
    return d + e + f;
  end;
  return y( a ); # y refers to local y of function x
end;
```

It is important to note that the concept of a variable in GAP is quite different from the concept of a variable in programming languages like PASCAL.

In those languages a variable denotes a block of memory. The value of the variable is stored in this block. So in those languages two variables can have the same value, but they can never have identical values, because they denote different blocks of memory. Note that PASCAL has the concept of a reference argument. It seems as if such an argument and the variable used in the actual function call have the same value, since changing the argument's value also changes the value of the variable used in the actual function call. But this is not so; the reference argument is actually a pointer to the variable used in the actual function call, and it is the compiler that inserts enough magic to make the pointer invisible. In order for this to work the compiler needs enough information to compute the amount of memory needed for each variable in a program, which is readily available in the declarations PASCAL requires for every variable.

In GAP on the other hand each variable just points to a value, and different variables can share the same value.

1 ▶ `Unbind( ` *ident* ` )` F

deletes the identifier *ident*. If there is no other variable pointing to the same value as *ident* was, this value will be removed by the next garbage collection. Therefore `Unbind` can be used to get rid of unwanted large objects.

For records and lists `Unbind` can be used to delete components or entries, respectively (see Chapters 27 and 21).

4.9 More About Global Variables

The vast majority of variables in GAP are defined at the outer level (the global scope). They are used to access functions and other objects created either in the GAP library or in the user's code. Certain special facilities are provided for manipulating these variables which are not available for other types of variable (such as local variables or function arguments).

First, such variables may be marked **read-only**. In which case attempts to change them will fail. Most of the global variables defined in the GAP library are so marked.

1 ▶ `IsReadOnlyGlobal( ` *name* ` )` F

returns `true` if the global variable named by the string *name* is read-only and `false` otherwise (the default).

2 ▶ `MakeReadOnlyGlobal( ` *name* ` )` F

marks the global variable named by the string *name* as read-only.

A warning is given if *name* has no value bound to it or if it is already read-only.

3 ▶ `MakeReadWriteGlobal( ` *name* ` )` F

marks the global variable named by the string *name* as read-write.

A warning is given if *name* is already read-write.

```
gap> xx := 17;
17
gap> IsReadOnlyGlobal("xx");
false
gap> xx := 15;
15
gap> MakeReadOnlyGlobal("xx");
gap> xx := 16;
Variable: 'xx' is read only
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' after making it writable to continue
brk> quit;
gap> IsReadOnlyGlobal("xx");
true
gap> MakeReadWriteGlobal("xx");
gap> xx := 16;
16
gap> IsReadOnlyGlobal("xx");
false
```

A group of functions are also supplied for accessing and altering the values assigned to global variables. Use of these functions differs from the use of assignment, `Unbind` and `IsBound` statements, in two ways. First,

these functions always affect global variables, even if local variables of the same names exist. Second, the variable names are passed as strings, rather than being written directly into the statements.

4 ▶ `ValueGlobal(` *name* `)` F

returns the value currently bound to the global variable named by the string *name*. An error is raised if no value is currently bound.

5 ▶ `IsBoundGlobal(` *name* `)` F

returns `true` if a value currently bound to the global variable named by the string *name* and `false` otherwise.

6 ▶ `UnbindGlobal(` *name* `)` F

removes any value currently bound to the global variable named by the string *name*. Nothing is returned.

A warning is given if *name* was not bound. The global variable named by *name* must be writable, otherwise an error is raised.

7 ▶ `BindGlobal(` *name*`,` *val* `)` F

sets the global variable named by the string *name* to the value *val*, provided it is writable, and makes it read-only. If *name* already has a value, a warning message is printed.

This is intended to be the normal way to create and set "official" global variables (such as Operations and Categories).

Caution should be exercised in using these functions, especially `BindGlobal` and `UnbindGlobal` as unexpected changes in global variables can be very confusing for the user.

```
gap> xx := 16;
16
gap> IsReadOnlyGlobal("xx");
false
gap> ValueGlobal("xx");
16
gap> IsBoundGlobal("xx");
true
gap> BindGlobal("xx",17);
#W BIND_GLOBAL: variable ‘xx’ already has a value
gap> xx;
17
gap> IsReadOnlyGlobal("xx");
true
```

Finally, there are a group of functions dealing with the **global namespace**.

8 ▶ `NamesGVars()` F

This function returns an immutable (see 12.6) sorted (see 21.19) list of all the global variable names known to the system. This includes names of variables which were bound but have now been unbound and some other names which have never been bound but have become known to the system by various routes.

9 ▶ `NamesSystemGVars()` F

This function returns an immutable sorted list of all the global variable names created by the GAP library when GAP was started.

10 ▶ `NamesUserGVars()` F

This function returns an immutable sorted list of the global variable names created since the library was read, to which a value is currently bound.

11 ▶ `TemporaryGlobalVarName( [`*prefix*`] )` F

returns a string that can be used as the name of a global variable that is not bound at the time when `TemporaryGlobalVarName()` is called. The optional argument *prefix* can specify a string with which the name of the global variable starts.

4.10 Function Calls

1 ▶ *function-var*`()`
▶ *function-var*`(` *arg-expr*`[,` *arg-expr*`, ...] )`

The function call has the effect of calling the function *function-var*. The precise semantics are as follows.

First GAP evaluates the *function-var*. Usually *function-var* is a variable, and GAP does nothing more than taking the value of this variable. It is allowed though that *function-var* is a more complex expression, such as a reference to an element of a list (see Chapter 21) *list-var*`[`*int-expr*`]`, or to a component of a record (see Chapter 27) *record-var*`.`*ident*. In any case GAP tests whether the value is a function. If it is not, GAP signals an error.

Next GAP checks that the number of actual arguments *arg-expr*s agrees with the number of **formal arguments** as given in the function definition. If they do not agree GAP signals an error. An exception is the case when there is exactly one formal argument with the name `arg`, in which case any number of actual arguments is allowed (see 4.22.1 for examples).

Now GAP allocates for each formal argument and for each **formal local** (that is, the identifiers in the `local` declaration) a new variable. Remember that a variable is a location in a GAP program that points to a value. Thus for each formal argument and for each formal local such a location is allocated.

Next the arguments *arg-expr*s are evaluated, and the values are assigned to the newly created variables corresponding to the formal arguments. Of course the first value is assigned to the new variable corresponding to the first formal argument, the second value is assigned to the new variable corresponding to the second formal argument, and so on. However, GAP does not make any guarantee about the order in which the arguments are evaluated. They might be evaluated left to right, right to left, or in any other order, but each argument is evaluated once. An exception again occurs if the function has only one formal argument with the name `arg`. In this case the values of all the actual arguments are stored in a list and this list is assigned to the new variable corresponding to the formal argument `arg`.

The new variables corresponding to the formal locals are initially not bound to any value. So trying to evaluate those variables before something has been assigned to them will signal an error.

Now the body of the function, which is a statement, is executed. If the identifier of one of the formal arguments or formal locals appears in the body of the function it refers to the new variable that was allocated for this formal argument or formal local, and evaluates to the value of this variable.

If during the execution of the body of the function a `return` statement with an expression (see 4.23) is executed, execution of the body is terminated and the value of the function call is the value of the expression of the `return`. If during the execution of the body a `return` statement without an expression is executed, execution of the body is terminated and the function call does not produce a value, in which case we call this call a procedure call (see 4.15). If the execution of the body completes without execution of a `return` statement, the function call again produces no value, and again we talk about a procedure call.

```
gap> Fibonacci( 11 );
89
```

The above example shows a call to the function `Fibonacci` with actual argument `11`, the following one shows a call to the operation `RightCosets` where the second actual argument is another function call.

```
gap> RightCosets( G, Intersection( U, V ) );;
```

2 ► *function-var*`(` *arg-expr*`[,` *arg-expr*`, ...][ : [` *option-expr* `[,`*option-expr*`, ....]]])`

As well as passing arguments to a function, providing the mathematical input to its calculation, it is sometimes useful to supply "hints" suggesting to GAP how the desired result may be computed more quickly, or specifying a level of tolerance for random errors in a Monte Carlo algorithm.

Such hints may be supplied to a function-call **and to all subsidiary functions called from that call** using the options mechanism. Options are separated from the actual arguments by a colon `:` and have much the same syntax as the components of a record expression. The one exception to this is that a component name may appear without a value, in which case the value `true` is silently inserted.

The following example shows a call to `Size` passing the options `hard` (with the value `true`) and `tcselection` (with the string "external" as value).

```
gap> Size( fpgrp : hard, tcselection := "external" );
```

Options supplied with function calls in this way are passed down using the global options stack described in chapter 8, so that the call above is exactly equivalent to

```
gap> PushOptions( rec( hard := true, tcselection := "external") );
gap> Size( fpgrp );
gap> PopOptions( );
```

Note that any option may be passed with any function, whether or not it has any actual meaning for that function, or any function called by it. The system provides no safeguard against misspelled option names.

4.11 Comparisons

1 ► *left-expr* `=` *right-expr*
► *left-expr* `<>` *right-expr*

The operator `=` tests for equality of its two operands and evaluates to `true` if they are equal and to `false` otherwise. Likewise `<>` tests for inequality of its two operands. Note that any two objects can be compared, i.e., `=` and `<>` will never signal an error. For each type of objects the definition of equality is given in the respective chapter. Objects in different families (see 13.1) are never equal, i.e., `=` evaluates in this case to `false`, and `<>` evaluates to `true`.

2 ► *left-expr* `<` *right-expr*
► *left-expr* `>` *right-expr*
► *left-expr* `<=` *right-expr*
► *left-expr* `>=` *right-expr*

`<` denotes less than, `<=` less than or equal, `>` greater than, and `>=` greater than or equal of its two operands. For each kind of objects the definition of the ordering is given in the respective chapter.

Only for the following kinds of objects, an ordering via `<` of objects in **different** families (see 13.1) is supported. Rationals (see 16.1.1) are smallest, next are cyclotomics (see 18.1.3), followed by finite field elements (see 57.1.1); finite field elements in different characteristics are compared via their characteristics, next are permutations (see 40), followed by the boolean values `true`, `false`, and `fail` (see 20), characters (such as `'a'`, see 26), and lists (see 21.1.1) are largest; note that two lists can be compared with `<` if and only if their elements are again objects that can be compared with `<`.

For other objects, GAP does **not** provide an ordering via `<`. The reason for this is that a total ordering of all GAP objects would be hard to maintain when new kinds of objects are introduced, and such a total ordering is hardly used in its full generality.

However, for objects in the filters listed above, the ordering via `<` has turned out to be useful. For example, one can form **sorted lists** containing integers and nested lists of integers, and then search in them using `PositionSorted` (see 21.16).

Of course it would in principle be possible to define an ordering via `<` also for certain other objects, by installing appropriate methods for the operation `\<`. But this may lead to problems at least as soon as one loads GAP code in which the same is done, under the assumption that one is completely free to define an ordering via `<` for other objects than the ones for which the "official" GAP provides already an ordering via `<`.

Comparison operators, including the operator `in` (see 21.8), are not associative, Hence it is not allowed to write *a* `=` *b* `<>` *c* `=` *d*, you must use `(`*a* `=` *b*`) <> (`*c* `=` *d*`)` instead. The comparison operators have higher precedence than the logical operators (see 20.3), but lower precedence than the arithmetic operators (see 4.12). Thus, for instance, *a* `*` *b* `=` *c* `and` *d* is interpreted as `((`*a* `*` *b*`) =` *c*`) and` *d*`)`.

The following example shows a comparison where the left operand is an expression.

```
gap> 2 * 2 + 9 = Fibonacci(7);
true
```

For the underlying operations of the operators introduced above, see 30.11.

4.12 Arithmetic Operators

1 ▶ `+` *right-expr*
▶ `-` *right-expr*
▶ *left-expr* `+` *right-expr*
▶ *left-expr* `-` *right-expr*
▶ *left-expr* `*` *right-expr*
▶ *left-expr* `/` *right-expr*
▶ *left-expr* `mod` *right-expr*
▶ *left-expr* `^` *right-expr*

The arithmetic operators are `+`, `-`, `*`, `/`, `mod`, and `^`. The meanings (semantics) of those operators generally depend on the types of the operands involved, and, except for `mod`, they are defined in the various chapters describing the types. However basically the meanings are as follows.

a `+` *b* denotes the addition of additive elements *a* and *b*.

a `-` *b* denotes the addition of *a* and the additive inverse of *b*.

a `*` *b* denotes the multiplication of multiplicative elements *a* and *b*.

a `/` *b* denotes the multiplication of *a* with the multiplicative inverse of *b*.

a `mod` *b*, for integer or rational left operand *a* and for non-zero integer right operand *b*, is defined as follows. If *a* and *b* are both integers, *a* `mod` *b* is the integer *r* in the integer range `0 .. |`*b*`| - 1` satisfying *a* `=` *r* `+` *bq*, for some integer *q* (where the operations occurring have their usual meaning over the integers, of course).

If *a* is a rational number and *b* is a non-zero integer, and *a* `=` *m* `/` *n* where *m* and *n* are coprime integers with *n* positive, then *a* `mod` *b* is the integer *r* in the integer range `0 .. |`*b*`| - 1` such that *m* is congruent to *rn* modulo *b*, and *r* is called the "modular remainder" of *a* modulo *b*. Also, `1 /` *n* `mod` *b* is called the "modular inverse" of *n* modulo *b*. (A pair of integers is said to be **coprime** (or **relatively prime**) if their gcd is 1.)

With the above definition, `4 / 6 mod 32` equals `2 / 3 mod 32` and hence exists (and is equal to 22), despite the fact that 6 has no inverse modulo 32.

Note. For rational a, a `mod` b could have been defined to be the non-negative rational c less than `|`b`|` such that a `-` c is a multiple of b. However this definition is seldom useful and **not** the one chosen for GAP.

`+` and `-` can also be used as unary operations. The unary `+` is ignored. The unary `-` returns the additive inverse of its operand; over the integers it is equivalent to multiplication by `-1`.

`^` denotes powering of a multiplicative element if the right operand is an integer, and is also used to denote the action of a group element on a point of a set if the right operand is a group element.

The **precedence** of those operators is as follows. The powering operator `^` has the highest precedence, followed by the unary operators `+` and `-`, which are followed by the multiplicative operators `*`, `/`, and `mod`, and the additive binary operators `+` and `-` have the lowest precedence. That means that the expression `-2 ^ -2 * 3 + 1` is interpreted as `(-(2 ^ (-2)) * 3) + 1`. If in doubt use parentheses to clarify your intention.

The **associativity** of the arithmetic operators is as follows. `^` is not associative, i.e., it is illegal to write `2^3^4`, use parentheses to clarify whether you mean `(2^3)^4` or `2^(3^4)`. The unary operators `+` and `-` are right associative, because they are written to the left of their operands. `*`, `/`, `mod`, `+`, and `-` are all left associative, i.e., `1-2-3` is interpreted as `(1-2)-3` not as `1-(2-3)`. Again, if in doubt use parentheses to clarify your intentions.

The arithmetic operators have higher precedence than the comparison operators (see 4.11 and 28.5) and the logical operators (see 20.3). Thus, for example, a `*` b `=` c `and` d is interpreted, `((`a `*` b`) =` c`) and` d.

```
gap> 2 * 2 + 9;  # a very simple arithmetic expression
13
```

For other arithmetic operations, and for the underlying operations of the operators introduced above, see 30.12.

4.13 Statements

Assignments (see 4.14), Procedure calls (see 4.15), `if` statements (see 4.16), `while` (see 4.17), `repeat` (see 4.18) and `for` loops (see 4.19), and the `return` statement (see 4.23) are called **statements**. They can be entered interactively or be part of a function definition. Every statement must be terminated by a semicolon.

Statements, unlike expressions, have no value. They are executed only to produce an effect. For example an assignment has the effect of assigning a value to a variable, a `for` loop has the effect of executing a statement sequence for all elements in a list and so on. We will talk about **evaluation** of expressions but about **execution** of statements to emphasize this difference.

Using expressions as statements is treated as syntax error.

```
gap> i := 7;;
gap> if i <> 0 then k = 16/i; fi;
Syntax error: := expected
if i <> 0 then k = 16/i; fi;
                 ^
gap>
```

As you can see from the example this warning does in particular address those users who are used to languages where `=` instead of `:=` denotes assignment.

Empty statements are permitted and have no effect.

A sequence of one or more statements is a **statement sequence**, and may occur everywhere instead of a single statement. There is nothing like PASCAL's BEGIN-END, instead each construct is terminated by a keyword. The simplest statement sequence is a single semicolon, which can be used as an empty statement sequence. In fact an empty statement sequence as in `for i in [1..2] do od` is also permitted and is silently translated into the sequence containing just a semicolon.

4.14 Assignments

1 ► *var* := *expr*;

The **assignment** has the effect of assigning the value of the expressions *expr* to the variable *var*.

The variable *var* may be an ordinary variable (see 4.8), a list element selection *list-var*[*int-expr*] (see 21.4) or a record component selection *record-var*.*ident* (see 27.2). Since a list element or a record component may itself be a list or a record the left hand side of an assignment may be arbitrarily complex.

Note that variables do not have a type. Thus any value may be assigned to any variable. For example a variable with an integer value may be assigned a permutation or a list or anything else.

```
gap> data:= rec( numbers:= [ 1, 2, 3 ] );
rec( numbers := [ 1, 2, 3 ] )
gap> data.string:= "string";; data;
rec( numbers := [ 1, 2, 3 ], string := "string" )
gap> data.numbers[2]:= 4;; data;
rec( numbers := [ 1, 4, 3 ], string := "string" )
```

If the expression *expr* is a function call then this function must return a value. If the function does not return a value an error is signalled and you enter a break loop (see 6.3). As usual you can leave the break loop with `quit;`. If you enter `return` *return-expr*`;` the value of the expression *return-expr* is assigned to the variable, and execution continues after the assignment.

```
gap> f1:= function( x ) Print( "value: ", x, "\n" ); end;;
gap> f2:= function( x ) return f1( x ); end;;
gap> f2( 4 );
value: 4
Function Calls: <func> must return a value at
return f1( x );
 called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can supply one by 'return <value>;' to continue
brk> return "hello";
"hello"
```

In the above example, the function `f2` calls `f1` with argument `4`, and since `f1` does not return a value (but only prints a line "`value:` *x*"), the `return` statement of `f2` cannot be executed. The error message says that it is possible to return an appropriate value, and the returned string `"hello"` is used by `f2` instead of the missing return value of `f1`.

4.15 Procedure Calls

1 ► *procedure-var*`();`
► *procedure-var*`(` *arg-expr* `[,`*arg-expr*`, ...] );`

The **procedure call** has the effect of calling the procedure *procedure-var*. A procedure call is done exactly like a function call (see 4.10). The distinction between functions and procedures is only for the sake of the discussion, GAP does not distinguish between them. So we state the following conventions.

A **function** does return a value but does not produce a side effect. As a convention the name of a function is a noun, denoting what the function returns, e.g., `Length`, `Concatenation` and `Order`.

A **procedure** is a function that does not return a value but produces some effect. Procedures are called only for this effect. As a convention the name of a procedure is a verb, denoting what the procedure does, e.g., `Print`, `Append` and `Sort`.

```
gap> Read( "myfile.g" );   # a call to the procedure Read
gap> l := [ 1, 2 ];;
gap> Append( l, [3,4,5] );  # a call to the procedure Append
```

There are a few exceptions of GAP functions that do both return a value and produce some effect. An example is `Sortex` which sorts a list and returns the corresponding permutation of the entries (see 21.18.3).

4.16 If

1 ▶ `if` *bool-expr1* `then` *statements1* `{ elif` *bool-expr2* `then` *statements2* `}[ else` *statements3* `] fi;`

The `if` statement allows one to execute statements depending on the value of some boolean expression. The execution is done as follows.

First the expression *bool-expr1* following the `if` is evaluated. If it evaluates to `true` the statement sequence *statements1* after the first `then` is executed, and the execution of the `if` statement is complete.

Otherwise the expressions *bool-expr2* following the `elif` are evaluated in turn. There may be any number of `elif` parts, possibly none at all. As soon as an expression evaluates to `true` the corresponding statement sequence *statements2* is executed and execution of the `if` statement is complete.

If the `if` expression and all, if any, `elif` expressions evaluate to `false` and there is an `else` part, which is optional, its statement sequence *statements3* is executed and the execution of the `if` statement is complete. If there is no `else` part the `if` statement is complete without executing any statement sequence.

Since the `if` statement is terminated by the `fi` keyword there is no question where an `else` part belongs, i.e., GAP has no "dangling else". In

`if` *expr1* `then if` *expr2* `then` *stats1* `else` *stats2* `fi; fi;`

the `else` part belongs to the second `if` statement, whereas in

`if` *expr1* `then if` *expr2* `then` *stats1* `fi; else` *stats2* `fi;`

the `else` part belongs to the first `if` statement.

Since an `if` statement is not an expression it is not possible to write

```
abs := if x > 0 then x; else -x; fi;
```

which would, even if legal syntax, be meaningless, since the `if` statement does not produce a value that could be assigned to `abs`.

If one of the expressions *bool-expr1*, *bool-expr2* is evaluated and its value is neither `true` nor `false` an error is signalled and a break loop (see 6.3) is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `true`. Likewise, if you enter `return false;`, execution of the `if` statement continues as if the expression whose evaluation failed had evaluated to `false`.

```
gap> i := 10;;
gap> if 0 < i then
>    s := 1;
>  elif i < 0 then
>    s := -1;
>  else
>    s := 0;
>  fi;
gap> s;  # the sign of i
1
```

4.17 While

1 ▶ `while` *bool-expr* `do` *statements* `od;`

The `while` loop executes the statement sequence *statements* while the condition *bool-expr* evaluates to `true`.

First *bool-expr* is evaluated. If it evaluates to `false` execution of the `while` loop terminates and the statement immediately following the `while` loop is executed next. Otherwise if it evaluates to `true` the *statements* are executed and the whole process begins again.

The difference between the `while` loop and the `repeat until` loop (see 4.18) is that the *statements* in the `repeat until` loop are executed at least once, while the *statements* in the `while` loop are not executed at all if *bool-expr* is `false` at the first iteration.

If *bool-expr* does not evaluate to `true` or `false` an error is signalled and a break loop (see 6.3) is entered. As usual you can leave the break loop with `quit;`. If you enter `return false;`, execution continues with the next statement immediately following the `while` loop. If you enter `return true;`, execution continues at *statements*, after which the next evaluation of *bool-expr* may cause another error.

The following example shows a `while` loop that sums up the squares $1^2, 2^2, \ldots$ until the sum exceeds 200.

```
gap> i := 0;; s := 0;;
gap> while s <= 200 do
>    i := i + 1; s := s + i^2;
>  od;
gap> s;
204
```

A `while` loop may be left prematurely using `break`, see 4.20.

4.18 Repeat

1 ▶ `repeat` *statements* `until` *bool-expr*`;`

The `repeat` loop executes the statement sequence *statements* until the condition *bool-expr* evaluates to `true`.

First *statements* are executed. Then *bool-expr* is evaluated. If it evaluates to `true` the `repeat` loop terminates and the statement immediately following the `repeat` loop is executed next. Otherwise if it evaluates to `false` the whole process begins again with the execution of the *statements*.

The difference between the `while` loop (see 4.17) and the `repeat until` loop is that the *statements* in the `repeat until` loop are executed at least once, while the *statements* in the `while` loop are not executed at all if *bool-expr* is `false` at the first iteration.

If *bool-expr* does not evaluate to `true` or `false` an error is signalled and a break loop (see 6.3) is entered. As usual you can leave the break loop with `quit;`. If you enter `return true;`, execution continues with the next statement immediately following the `repeat` loop. If you enter `return false;`, execution continues at *statements*, after which the next evaluation of *bool-expr* may cause another error.

The `repeat` loop in the following example has the same purpose as the `while` loop in the preceding example, namely to sum up the squares $1^2, 2^2, \ldots$ until the sum exceeds 200.

```
gap> i := 0;; s := 0;;
gap> repeat
>    i := i + 1; s := s + i^2;
>  until s > 200;
gap> s;
204
```

A `repeat` loop may be left prematurely using `break`, see 4.20.

4.19 For

1 ▶ `for` *simple-var* `in` *list-expr* `do` *statements* `od;`

The `for` loop executes the statement sequence *statements* for every element of the list *list-expr*.

The statement sequence *statements* is first executed with *simple-var* bound to the first element of the list *list-expr*, then with *simple-var* bound to the second element of *list-expr* and so on. *simple-var* must be a simple variable, it must not be a list element selection *list-var*`[`*int-expr*`]` or a record component selection *record-var*`.`*ident*.

The execution of the `for` loop over a list is exactly equivalent to the following `while` loop.

```
loop-list := list;
loop-index := 1;
while loop-index <= Length(loop-list) do
 variable := loop-list[loop-index];
 statements
 loop-index := loop-index + 1;
od;
```

with the exception that *loop-list* and *loop-index* are different variables for each `for` loop, i.e., these variables of different `for` loops do not interfere with each other.

The list *list-expr* is very often a range (see 21.22).

2 ▶ `for` *variable* `in [`*from*`..`*to*`] do` *statements* `od;`

corresponds to the more common

`for` *variable* `from` *from* `to` *to* `do` *statements* `od;`

in other programming languages.

```
gap> s := 0;;
gap> for i in [1..100] do
>    s := s + i;
> od;
gap> s;
5050
```

Note in the following example how the modification of the **list** in the loop body causes the loop body also to be executed for the new values.

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i in l do
>    Print( i, " " );
>    if i mod 2 = 0 then Add( l, 3 * i / 2 ); fi;
> od; Print( "\n" );
1 2 3 4 5 6 3 6 9 9
gap> l;
[ 1, 2, 3, 4, 5, 6, 3, 6, 9, 9 ]
```

Note in the following example that the modification of the **variable** that holds the list has no influence on the loop.

```
gap> l := [ 1, 2, 3, 4, 5, 6 ];;
gap> for i in l do
>    Print( i, " " );
>    l := [];
> od; Print( "\n" );
1 2 3 4 5 6
gap> l;
[  ]
```

3► `for` *variable* `in` *iterator* `do` *statements* `od;`

It is also possible to have a `for`-loop run over an iterator (see 28.7). In this case the `for`-loop is equivalent to

```
while not IsDoneIterator(iterator) do
 variable := NextIterator(iterator)
 statements
od;
```

4► `for` *variable* `in` *object* `do` *statements* `od;`

Finally, if an object *object* which is not a list or an iterator appears in a `for`-loop, then GAP will attempt to evaluate the function call `Iterator(`*object*`)`. If this is successful then the loop is taken to run over the iterator returned.

```
gap> g := Group((1,2,3,4,5),(1,2)(3,4)(5,6));
Group([ (1,2,3,4,5), (1,2)(3,4)(5,6) ])
gap> count := 0;; sumord := 0;;
gap> for x in g do
> count := count + 1; sumord := sumord + Order(x); od;
gap> count;
120
gap> sumord;
471
```

The effect of

```
for variable in domain do
```

should thus normally be the same as

```
for variable in AsList(domain) do
```

but may use much less storage, as the iterator may be more compact than a list of all the elements.

See 28.7 for details about iterators.

A `for` loop may be left prematurely using `break`, see 4.20. This combines especially well with a loop over an iterator, as a way of searching through a domain for an element with some useful property.

4.20 Break

1 ► `break;`

The statement `break;` causes an immediate exit from the innermost loop enclosing it. It is an error to use this statement other than inside a loop.

```
gap> g := Group((1,2,3,4,5),(1,2)(3,4)(5,6));
Group([ (1,2,3,4,5), (1,2)(3,4)(5,6) ])
gap> for x in g do
> if Order(x) = 3 then
> break;
> fi; od;
gap> x;
(1,4,3)(2,6,5)
gap> break;
A break statement can only appear inside a loop
```

4.21 Continue

1 ► `continue;`

The statement `continue;` causes the rest of the current iteration of the innermost loop enclosing it to be skipped. It is an error to use this statement other than inside a loop.

```
gap> g := Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> for x in g do
> if Order(x) = 3 then
> continue;
> fi; Print(x,"\n"); od;
()
(2,3)
(1,3)
(1,2)
gap> continue;
A continue statement can only appear inside a loop
```

4.22 Function

1 ► `function( [` *arg-ident* `{, ` *arg-ident*`} ] )`
`[local` *loc-ident* `{, ` *loc-ident*`} ; ]`
statements
`end`

A function is in fact a literal and not a statement. Such a function literal can be assigned to a variable or to a list element or a record component. Later this function can be called as described in 4.10.

The following is an example of a function definition. It is a function to compute values of the Fibonacci sequence (see 17.3.1).

```
gap> fib := function ( n )
>     local f1, f2, f3, i;
>     f1 := 1; f2 := 1;
>     for i in [3..n] do
>       f3 := f1 + f2;
>       f1 := f2;
>       f2 := f3;
>     od;
>     return f2;
>   end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Because for each of the formal arguments *arg-ident* and for each of the formal locals *loc-ident* a new variable is allocated when the function is called (see 4.10), it is possible that a function calls itself. This is usually called **recursion**. The following is a recursive function that computes values of the Fibonacci sequence

```
gap> fib := function ( n )
>     if n < 3 then
>       return 1;
>     else
>       return fib(n-1) + fib(n-2);
>     fi;
>   end;;
gap> List( [1..10], fib );
[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
```

Note that the recursive version needs `2 * fib(`n`)-1` steps to compute `fib(`n`)`, while the iterative version of `fib` needs only n`-2` steps. Both are not optimal however, the library function `Fibonacci` only needs about `Log(`n`)` steps.

As noted in Section 4.10, the case where a function is defined with exactly one formal argument with the name `arg`, is special. It provides a way of defining a function with a variable number of arguments; the values of all the actual arguments are stored in a list and this list is assigned to the new variable corresponding to the formal argument `arg`. There are two typical scenarios for wanting such a possibility: having optional arguments and having any number of arguments.

The following example shows one way that the function `Position` (see 21.16.1) might be encoded and demonstrates the "optional argument" scenario.

```
gap> position := function ( arg )
>     local list, obj, pos;
>     list := arg[1];
>     obj  := arg[2];
>     if 2 = Length(arg) then
>       pos := 0;
>     else
>       pos := arg[3];
>     fi;
>     repeat
>       pos := pos + 1;
>       if pos > Length(list) then
>         return fail;
>       fi;
```

```
>     until list[pos] = obj;
>     return pos;
>    end;
function( arg ) ... end
gap> position([1, 4, 2], 4);
2
gap> position([1, 4, 2], 3);
fail
gap> position([1, 4, 2], 4, 2);
fail
```

The following example demonstrates the "any number of arguments" scenario.

```
gap> sum := function ( arg )
>     local total, x;
>     total := 0;
>     for x in arg do
>       total := total + x;
>     od;
>     return total;
>    end;
function( arg ) ... end
gap> sum(1, 2, 3);
6
gap> sum(1, 2, 3, 4);
10
gap> sum();
0
```

The user should compare the above with the GAP function `Sum` (see 21.20.24) which, for example, may take a list argument and optionally an initial element (which zero should the sum of an empty list return?).

Note that if a function f is defined as above with the single formal argument `arg` then `NumberArguments-Function(`f`)` returns -1 (see 5.1.2).

The argument `arg` when used as the single argument name of some function f tells GAP that when it encounters f that it should form a list out of the arguments of f. What if one wishes to do the "opposite": tell GAP that a list should be "unwrapped" and passed as several arguments to a function. The function `CallFuncList` (see 5.2.1) is provided for this purpose.

Also see Chapter 5.

2 ► *arg-ident* `->` *expr*

This is a shorthand for

`function (` *arg-ident* `) return` *expr*`; end.`

arg-ident must be a single identifier, i.e., it is not possible to write functions of several arguments this way. Also `arg` is not treated specially, so it is also impossible to write functions that take a variable number of arguments this way.

The following is an example of a typical use of such a function

```
gap> Sum( List( [1..100], x -> x^2 ) );
338350
```

When the definition of a function *fun1* is evaluated inside another function *fun2*, GAP binds all the identifiers inside the function *fun1* that are identifiers of an argument or a local of *fun2* to the corresponding variable.

This set of bindings is called the environment of the function *fun1*. When *fun1* is called, its body is executed in this environment. The following implementation of a simple stack uses this. Values can be pushed onto the stack and then later be popped off again. The interesting thing here is that the functions `push` and `pop` in the record returned by `Stack` access the local variable `stack` of `Stack`. When `Stack` is called, a new variable for the identifier `stack` is created. When the function definitions of `push` and `pop` are then evaluated (as part of the `return` statement) each reference to `stack` is bound to this new variable. Note also that the two stacks `A` and `B` do not interfere, because each call of `Stack` creates a new variable for `stack`.

```
gap> Stack := function ()
>     local  stack;
>     stack := [];
>     return rec(
>       push := function ( value )
>         Add( stack, value );
>       end,
>       pop := function ()
>         local  value;
>         value := stack[Length(stack)];
>         Unbind( stack[Length(stack)] );
>         return value;
>       end
>     );
>  end;;
gap> A := Stack();;
gap> B := Stack();;
gap> A.push( 1 ); A.push( 2 ); A.push( 3 );
gap> B.push( 4 ); B.push( 5 ); B.push( 6 );
gap> A.pop(); A.pop(); A.pop();
3
2
1
gap> B.pop(); B.pop(); B.pop();
6
5
4
```

This feature should be used rarely, since its implementation in GAP is not very efficient.

4.23 Return

1 ▶ `return;`

In this form `return` terminates the call of the innermost function that is currently executing, and control returns to the calling function. An error is signalled if no function is currently executing. No value is returned by the function.

2 ▶ `return` *expr*`;`

In this form `return` terminates the call of the innermost function that is currently executing, and returns the value of the expression *expr*. Control returns to the calling function. An error is signalled if no function is currently executing.

Both statements can also be used in break loops (see 6.3). `return;` has the effect that the computation continues where it was interrupted by an error or the user hitting *ctr*-`C`. `return` *expr*`;` can be used to continue execution after an error. What happens with the value *expr* depends on the particular error.

For examples of `return` statements, see the functions `fib` and `Stack` in Chapter 5.

4.24 The Syntax in BNF

This section contains the definition of the GAP syntax in Backus-Naur form. A few recent additions to the syntax may be missing from this definition. Also, the actual rules for identifier names implemented by the system, are somewhat more permissive than those given below (see section 4.6).

A BNF is a set of rules, whose left side is the name of a syntactical construct. Those names are enclosed in angle brackets and written in *italics*. The right side of each rule contains a possible form for that syntactic construct. Each right side may contain names of other syntactic constructs, again enclosed in angle brackets and written in *italics*, or character sequences that must occur literally; they are written in `typewriter style`.

Furthermore each righthand side can contain the following metasymbols written in **boldface**. If the right hand side contains forms separated by a pipe symbol (|) this means that one of the possible forms can occur. If a part of a form is enclosed in square brackets ([ ]) this means that this part is optional, i.e. might be present or missing. If part of the form is enclosed in curly braces (`{ }`) this means that the part may occur arbitrarily often, or possibly be missing.

```
Ident       := a|...|z|A|...|Z|_ {a|...|z|A|...|Z|0|...|9|_}
Var         := Ident
             | Var . Ident
             | Var . ( Expr )
             | Var [ Expr ]
             | Var { Expr }
             | Var ( [ Expr { ,Expr } ] )
             | Var !. Ident
             | Var !. ( Expr )
             | Var ![ Expr ]
List        := [ [ Expr ] {, [ Expr ] } ]
             | [ Expr [, Expr ] .. Expr ]
             | List ' List `
Record      := rec( [ Ident := Expr {, Ident := Expr } ] )
Permutation := ( Expr {, Expr } ) { ( Expr {, Expr } ) }
Function    := function ( [ Ident {, Ident } ] )
               [ local Ident {, Ident } ; ]
               Statements
               end
             | Ident -> Expr
Char        := 'any character '
String      := " { any character } "
Int         := 0|1|...|9 {0|1|...|9}
Atom        := Int
             | Var
             | ( Expr )
             | Permutation
             | Char
             | String
             | Function
             | List
             | Record
             | { not } true
             | { not } false
Factor      := {+|-} Atom [ ^ {+|-} Atom ]
Term        := Factor { *|/|mod Factor }
Arith       := Term { +|- Term }
Rel         := { not } Arith [ =|<>|<|>|<=|>=|in Arith ]
And         := Rel { and Rel }
Logical     := And { or And }
Expr        := Logical
             | Var
Statement   := Expr
             | Var := Expr
             | if Expr then Statements
               { elif Expr then Statements }
               [ else           Statements ] fi
             | for Var in Expr do Statements od
             | while Expr do Statements od
             | repeat Statements until Expr
             | return [ Expr ]
             | break
             | quit
             | QUIT
             |
Statements  := { Statement ; }
             | ;
             |
```

5 Functions

The section 4.22.1 describes how to define a function. In this chapter we describe functions that give information about functions, and various utility functions used either when defining functions or calling functions.

5.1 Information about a function

1 ▶ `NameFunction( `*func*` )` F

returns the name of a function. For operations, this is the name used in their declaration. For functions, this is the variable name they were first assigned to. (For some internal functions, this might be a name **different** from the name that is documented.) If no such name exists, `"unknown"` is returned.

```
gap> NameFunction(SylowSubgroup);
"SylowSubgroup"
gap> Blubberflutsch:=x->x;;
gap> NameFunction(Blubberflutsch);
"Blubberflutsch"
gap> a:=Blubberflutsch;;
gap> NameFunction(a);
"Blubberflutsch"
gap> NameFunction(x->x);
"unknown"
gap> NameFunction(NameFunction);
"NAME_FUNC"
```

2 ▶ `NumberArgumentsFunction( `*func*` )` F

returns the number of arguments the function *func* accepts. For functions that use `arg` to take a variable number of arguments, as well as for operations, -1 is returned. For attributes, 1 is returned.

```
gap> NumberArgumentsFunction(function(a,b,c,d,e,f,g,h,i,j,k)return 1;end);
11
gap> NumberArgumentsFunction(Size);
1
gap> NumberArgumentsFunction(IsCollsCollsElms);
3
gap> NumberArgumentsFunction(Sum);
-1
```

3 ▶ `NamesLocalVariablesFunction( `*func*` )` F

returns a mutable list of strings; the first entries are the names of the arguments of the function *func*, in the same order as they were entered in the definition of *func*, and the remaining ones are the local variables as given in the `local` statement in *func*. (The number of arguments can be computed with `NumberArgumentsFunction`.)

```
gap> NamesLocalVariablesFunction( function( a, b ) local c; return 1; end );
[ "a", "b", "c" ]
gap> NamesLocalVariablesFunction( function( arg ) local a; return 1; end );
[ "arg", "a" ]
gap> NamesLocalVariablesFunction( Size );
fail
```

5.2 Calling a function with a list argument that is interpreted as several arguments

1 ▶ `CallFuncList( `*func*`, `*args*` )` F

returns the result, when calling function *func* with the arguments given in the list *args*, i.e. *args* is "unwrapped" so that *args* appears as several arguments to *func*.

```
gap> CallFuncList(\+, [6, 7]);
13
gap> #is equivalent to:
gap> \+(6, 7);
13
```

A more useful application of `CallFuncList` is for a function g that is called in the body of a function f with (a sublist of) the arguments of f, where f has been defined with a single formal argument `arg` (see 4.22.1); see the following code fragment.

```
f := function ( arg )
       CallFuncList(g, arg);
       ...
     end;
```

In the body of f the several arguments passed to f become a list `arg`. If g were called instead via g`( arg )` then g would see a single list argument, so that g would, in general, have to "unwrap" the passed list. The following (not particularly useful) example demonstrates both described possibilities for the call to g.

```
gap> PrintNumberFromDigits := function ( arg )
>     CallFuncList( Print, arg );
>     Print( "\n" );
>    end;
function( arg ) ... end
gap> PrintNumberFromDigits( 1, 9, 7, 3, 2 );
19732
gap> PrintDigits := function ( arg )
>     Print( arg );
>     Print( "\n" );
>    end;
function( arg ) ... end
gap> PrintDigits( 1, 9, 7, 3, 2 );
[ 1, 9, 7, 3, 2 ]
```

5.3 Functions that do nothing

The following functions return fixed results (or just their own argument). They can be useful in places when the syntax requires a function, but actually no functionality is required. So `ReturnTrue` is often used as family predicate in `InstallMethod` (see 2.2.1 in "Programming in GAP").

1 ▶ `ReturnTrue( ... )` F

This function takes any number of arguments, and always returns `true`.

2 ▶ `ReturnFalse( ... )` F

This function takes any number of arguments, and always returns `false`.

3 ▶ `ReturnFail( ... )` F

This function takes any number of arguments, and always returns `fail`.

4 ▶ `IdFunc(` *obj* `)` F

returns *obj*.

5.4 Function Types

Functions are GAP objects and thus have categories and a family.

1 ▶ `IsFunction(` *obj* `)` C

is the category of functions.

2 ▶ `IsOperation(` *obj* `)` C

is the category of operations. Every operation is a function, but not vice versa.

3 ▶ `FunctionsFamily` V

is the family of all functions.

6 Main Loop and Break Loop

This chapter is a first of a series of chapters that describe the interactive environment in which you use GAP.

6.1 Main Loop

The normal interaction with GAP happens in the so-called **read-eval-print** loop. This means that you type an input, GAP first reads it, evaluates it, and then shows the result. Note that the term **print** may be confusing since there is a GAP function called `Print` (see 6.2) which is in fact **not** used in the read-eval-print loop, but traditions are hard to break. In the following, whenever we want to express that GAP places some characters on the standard output, we will say that GAP **shows** something.

The exact sequence in the read-eval-print loop is as follows.

To signal that it is ready to accept your input, GAP shows the **prompt** `gap>`. When you see this, you know that GAP is waiting for your input.

Note that every statement must be terminated by a semicolon. You must also enter *return* (i.e., strike the "return" key) before GAP starts to read and evaluate your input. (The "return" key may actually be marked with the word `Enter` and a returning arrow on your terminal.) Because GAP does not do anything until you enter *return*, you can edit your input to fix typos and only when everything is correct enter *return* and have GAP take a look at it (see 6.8). It is also possible to enter several statements as input on a single line. Of course each statement must be terminated by a semicolon.

It is absolutely acceptable to enter a single statement on several lines. When you have entered the beginning of a statement, but the statement is not yet complete, and you enter *return*, GAP will show the **partial prompt** `>`. When you see this, you know that GAP is waiting for the rest of the statement. This happens also when you forget the semicolon `;` that terminates every GAP statement. Note that when *return* has been entered and the current statement is not yet complete, GAP will already evaluate those parts of the input that are complete, for example function calls that appear as arguments in another function call which needs several input lines. So it may happen that one has to wait some time for the partial prompt.

When you enter *return*, GAP first checks your input to see if it is syntactically correct (see Chapter 4 for the definition of syntactically correct). If it is not, GAP prints an error message of the following form

```
gap> 1 * ;
Syntax error: expression expected
1 * ;
    ^
```

The first line tells you what is wrong about the input, in this case the `*` operator takes two expressions as operands, so obviously the right one is missing. If the input came from a file (see 9.7.1), this line will also contain the filename and the line number. The second line is a copy of the input. And the third line contains a caret pointing to the place in the previous line where GAP realized that something is wrong. This need not be the exact place where the error is, but it is usually quite close.

Sometimes, you will also see a partial prompt after you have entered an input that is syntactically incorrect. This is because GAP is so confused by your input, that it thinks that there is still something to follow. In

this case you should enter ;*return* repeatedly, ignoring further error messages, until you see the full prompt again. When you see the full prompt, you know that GAP forgave you and is now ready to accept your next – hopefully correct – input.

If your input is syntactically correct, GAP evaluates or executes it, i.e., performs the required computations (see Chapter 4 for the definition of the evaluation).

If you do not see a prompt, you know that GAP is still working on your last input. Of course, you can **type ahead**, i.e., already start entering new input, but it will not be accepted by GAP until GAP has completed the ongoing computation.

When GAP is ready it will usually show the result of the computation, i.e., the value computed. Note that not all statements produce a value, for example, if you enter a `for` loop, nothing will be printed, because the `for` loop does not produce a value that could be shown.

Also sometimes you do not want to see the result. For example if you have computed a value and now want to assign the result to a variable, you probably do not want to see the value again. You can terminate statements by **two semicolons** to suppress showing the result.

If you have entered several statements on a single line GAP will first read, evaluate, and show the first one, then read, evaluate, and show the second one, and so on. This means that the second statement will not even be checked for syntactical correctness until GAP has completed the first computation.

After the result has been shown GAP will display another prompt, and wait for your next input. And the whole process starts all over again. Note that if you have entered several statements on a single line, a new prompt will only be printed after GAP has read, evaluated, and shown the last statement.

In each statement that you enter, the result of the previous statement that produced a value is available in the variable `last`. The next to previous result is available in `last2` and the result produced before that is available in `last3`.

```
gap> 1; 2; 3;
1
2
3
gap> last3 + last2 * last;
7
```

Also in each statement the time spent by the last statement, whether it produced a value or not, is available in the variable `time`. This is an integer that holds the number of milliseconds.

6.2 View and Print

1 ▶ `View(` *obj1*`,` *obj2*`...)` F

`View` shows the objects *obj1*, *obj2*... etc. **in a short form** on the standard output. `View` is called in the read–eval–print loop, thus the output looks exactly like the representation of the objects shown by the main loop. Note that no space or newline is printed between the objects.

2 ▶ `Print(` *obj1*`,` *obj2*`...)` F

Also `Print` shows the objects *obj1*, *obj2*... etc. on the standard output. The difference compared to `View` is in general that the shown form is not required to be short, and that in many cases the form shown by `Print` is GAP readable.

```
gap> z:= Z(2);
Z(2)^0
gap> v:= [ z, z, z, z, z, z, z ];
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ]
gap> ConvertToVectorRep(v);; v;
<a GF2 vector of length 7>
gap> Print( v );
[ Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0, Z(2)^0 ]gap>
```

Another difference is that `Print` shows strings without the enclosing quotes, so `Print` can be used to produce formatted text on the standard output (see also chapter 26). Some characters preceded by a backslash, such as `\n`, are processed specially (see chapter 26.1). `PrintTo` can be used to print to a file (see 9.7.3).

```
gap> for i in [1..5] do
>      Print( i, " ", i^2, " ", i^3, "\n" );
>    od;
1 1 1
2 4 8
3 9 27
4 16 64
5 25 125

gap> g:= SmallGroup(12,5);
<pc group of size 12 with 3 generators>
gap> Print( g, "\n" );
Group( [ f1, f2, f3 ] )
gap> View( g );
<pc group of size 12 with 3 generators>gap>
```

3 ▶ `ViewObj( obj )` O
▶ `PrintObj( obj )` O

The functions `View` and `Print` actually call the operations `ViewObj` and `PrintObj`, respectively, for each argument. By installing special methods for these operations, it is possible to achieve special printing behavior for certain objects (see chapter 2 in the Programmer's Manual). The only exceptions are strings (see Chapter 26), for which the default `PrintObj` and `ViewObj` methods as well as the function `View` print also the enclosing doublequotes, whereas `Print` strips the doublequotes.

The default method for `ViewObj` is to call `PrintObj`. So it is sufficient to have a `PrintObj` method for an object in order to `View` it. If one wants to supply a "short form" for `View`, one can install additionally a method for `ViewObj`.

4 ▶ `Display( obj )` O

Displays the object *obj* in a nice, formatted way which is easy to read (but might be difficult for machines to understand). The actual format used for this depends on the type of *obj*. Each method should print a newline character as last character.

```
gap> Display( [ [ 1, 2, 3 ], [ 4, 5, 6 ] ] * Z(5) );
 2 4 1
 3 . 2
```

One can assign a string to an object that `Print` will use instead of the default used by `Print`, via `SetName` (see 12.8.1). Also, `Name` (see 12.8.2) returns the string previously assigned to the object for printing, via `SetName`. The following is an example in the context of domains.

```
gap> g:= Group( (1,2,3,4) );
Group([ (1,2,3,4) ])
gap> SetName( g, "C4" ); g;
C4
gap> Name( g );
"C4"
```

6.3 Break Loops

When an error has occurred or when you interrupt GAP (usually by hitting *ctrl*-C) GAP enters a break loop, that is in most respects like the main read eval print loop (see 6.1). That is, you can enter statements, GAP reads them, evaluates them, and shows the result if any. However those evaluations happen within the context in which the error occurred. So you can look at the arguments and local variables of the functions that were active when the error happened and even change them. The prompt is changed from `gap>` to `brk>` to indicate that you are in a break loop.

```
gap> 1/0;
Rational operations: <divisor> must not be zero
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <divisor> via 'return <divisor>;' to continue
```

If errors occur within a break loop GAP enters another break loop at a **deeper level**. This is indicated by a number appended to `brk`:

```
brk> 1/0;
Rational operations: <divisor> must not be zero
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <divisor> via 'return <divisor>;' to continue
brk_02>
```

There are two ways to leave a break loop.

1 ► `quit`

The first is to **quit** the break loop. To do this you enter `quit;` or type the *eof* (**e**nd **o**f **f**ile) character, which is usually *ctrl*-D except when using the `-e` option (see Section 3.1). Note that GAP code between `quit;` and the end of the input line is ignored.

```
brk_02> quit;
brk>
```

In this case control returns to the break loop one level above or to the main loop, respectively. So iterated break loops must be left iteratively. Note also that if you type `quit;` from a `gap>` prompt, GAP will exit (see 6.7).

Note: If you leave a break loop with `quit` without completing a command it is possible (though not very likely) that data structures will be corrupted or incomplete data have been stored in objects. Therefore no guarantee can be given that calculations afterwards will return correct results! If you have been using options `quit`ting a break loop generally leaves the options stack with options you no longer want. The

function `ResetOptionsStack` (see 8) removes all options on the options stack, and this is the sole intended purpose of this function.

2 ► `return [`*obj*`];`

The other way is to **return** from a break loop. To do this you type `return;` or `return` *expr*`;`. If the break loop was entered because you interrupted GAP, then you can continue by typing `return;`. If the break loop was entered due to an error, you may have to modify the value of a variable before typing `return;` (see the example for 21.1.2) or you may have to return a *value* (by typing: `return` *value*`;`) to continue the computation; in any case, the message printed on entering the break loop will tell you which of these alternatives is possible. For example, if the break loop was entered because a variable had no assigned value, the value to be returned is often a value that this variable should have to continue the computation.

```
brk> return 9;  # we had tried to enter the divisor 9 but typed 0 ...
1/9
gap>
```

3 ► `OnBreak` V

By default, when a break loop is entered, GAP prints a trace of the innermost 5 commands currently being executed. This behaviour can be configured by changing the value of the global variable `OnBreak`. When a break loop is entered, the value of `OnBreak` is checked. If it is a function, then it is called with no arguments. By default, the value of `OnBreak` is `Where` (see 6.3.5).

```
gap> OnBreak := function() Print("Hello\n"); end;
function(  ) ... end

gap> Error("!\n");
Error, !
Hello
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

In cases where a break loop is entered during a function that was called with options (see Chapter 8), a `quit;` will also cause the options stack to be reset and an `Info`-ed warning stating this is emitted at `InfoWarning` level 1 (see Chapter 7.4).

Note that for break loops entered by a call to `Error`, the lines after "`Entering break read-eval-print loop ...`" and before the `brk>` prompt can also be customised, namely by redefining `OnBreakMessage` (see 6.3.4).

Also, note that one can achieve the effect of changing `OnBreak` **locally**. As mentioned above, the default value of `OnBreak` is `Where`. Thus, a call to `Error` (see 6.5.1) generally gives a trace back up to five levels of calling functions. Conceivably, we might like to have a function like `Error` that does not trace back without globally changing `OnBreak`. Such a function we might call `ErrorNoTraceBack` and here is how we might define it. (Note `ErrorNoTraceBack` is **not** a GAP function.)

```
gap> ErrorNoTraceBack := function(arg) # arg is a special variable that GAP
>                                      # knows to treat as a list of arg'ts
>      local SavedOnBreak, ENTBOnBreak;
>      SavedOnBreak := OnBreak;        # save the current value of OnBreak
>
>      ENTBOnBreak := function()       # our 'local' OnBreak
>      local s;
>        for s in arg do
```

```
>          Print(s);
>        od;
>        OnBreak := SavedOnBreak;      # restore OnBreak afterwards
>      end;
>
>      OnBreak := ENTBOnBreak;
>      Error();
>    end;
function( arg ) ... end
```

Here is a somewhat trivial demonstration of the use of `ErrorNoTraceBack`.

```
gap> ErrorNoTraceBack("Gidday!", " How's", " it", " going?\n");
Error, Gidday! How's it going?
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

Now we call `Error` with the same arguments to show the difference.

```
gap> Error("Gidday!", " How's", " it", " going?\n");
Error, Gidday! How's it going?
Hello
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

Observe that the value of `OnBreak` before the `ErrorNoTraceBack` call was restored. However, we had changed `OnBreak` from its default value; to restore `OnBreak` to its default value, we should do the following.

```
gap> OnBreak := Where;;
```

4 ▶ `OnBreakMessage` V

When a break loop is entered by a call to `Error` (see 6.5.1) the message after the "`Entering break read-eval-print loop ...`" line is produced by the function `OnBreakMessage`, which just like `OnBreak` (see 6.3.3) is a user-configurable global variable that is a **function** with **no arguments**.

```
gap> OnBreakMessage(); # By default, OnBreakMessage prints the following
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
```

Perhaps you are familiar with what's possible in a break loop, and so don't need to be reminded. In this case, you might wish to do the following (the first line just makes it easy to restore the default value later).

```
gap> NormalOnBreakMessage := OnBreakMessage;; # save the default value
gap> OnBreakMessage := function() end;        # do-nothing function
function(  ) ... end
```

With `OnBreak` still set away from its default value, calling `Error` as we did above, now produces:

```
gap> Error("!\n");
Error, !
Hello
Entering break read-eval-print loop ...
brk> quit; # to get back to outer loop
```

However, suppose you are writing a function which detects an error condition and `OnBreakMessage` needs to be changed only **locally**, i.e., the instructions on how to recover from the break loop need to be specific to that function. The same idea used to define `ErrorNoTraceBack` (see 6.3.3) can be adapted to achieve this. The function `CosetTableFromGensAndRels` (see 45.5.5) is an example in the GAP code where the idea is actually used.

5 ▶ `Where( [`*nr*`] )` F

shows the last *nr* commands on the execution stack during whose execution the error occurred. If not given, *nr* defaults to 5. (Assume, for the following example, that after the last example `OnBreak` (see 6.3.3) has been set back to its default value.)

```
gap> StabChain(SymmetricGroup(100)); # After this we typed ^C
user interrupt at
bpt := S.orbit[1];
 called from
SiftedPermutation( S, (g * rep) ^ -1 ) called from
StabChainStrong( S.stabilizer, [ sch ], options ); called from
StabChainStrong( S.stabilizer, [ sch ], options ); called from
StabChainStrong( S, GeneratorsOfGroup( G ), options ); called from
StabChainOp( G, rec(
     ) ) called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> Where(2);
 called from
SiftedPermutation( S, (g * rep) ^ -1 ) called from
StabChainStrong( S.stabilizer, [ sch ], options ); called from
...
```

Note that the variables displayed even in the first line of the `Where` list (after the `called from` line) may be already one environment level higher and `DownEnv` (see 6.4.1) may be necessary to access them.

At the moment this backtrace does not work from within compiled code (this includes the method selection which by default is compiled into the kernel). If this creates problems for debugging, call GAP with the `-M` option (see 3.2) to avoid loading compiled code.

(Function calls to `Info` and methods installed for binary operations are handled in a special way. In rare circumstances it is possible therefore that they do not show up in a `Where` log but the log refers to the **last** proper function call that happened before.)

The command line option `-T` to GAP disables the break loop. This is mainly intended for testing purposes and for special applications. If this option is given then errors simply cause GAP to return to the main loop.

6.4 Variable Access in a Break Loop

In a break loop access to variables of the current break level and higher levels is possible, but if the same variable name is used for different objects or if a function calls itself recursively, of course only the variable at the lowest level can be accessed.

1 ▶ `DownEnv( [`*nr*`] )` F
▶ `UpEnv( [`*nr*`] )` F

`DownEnv` moves up *nr* steps in the environment and allows one to inspect variables on this level; if *nr* is negative it steps down in the environment again; *nr* defaults to 1 if not given. `UpEnv` acts similarly to `DownEnv` but in the reverse direction. (The names of `DownEnv` and `UpEnv` are the wrong way 'round; I guess it all depends on which direction defines is "up" – just use `DownEnv` and get used to that.)

```
gap> OnBreak := function() Where(0); end;; # eliminate back-tracing on
gap>                                       # entry to break loop
gap> test:= function( n )
>    if n > 3 then Error( "!\n" ); fi; test( n+1 ); end;;
gap> test( 1 );
Error, !
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> n;
4
brk> DownEnv();
brk> n;
3
brk> Where();
 called from
test( n + 1 ); called from
test( n + 1 ); called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( 2 );
brk> n;
1
brk> Where();
 called from
<function>( <arguments> ) called from read-eval-loop
brk> DownEnv( -2 );
brk> n;
3
brk> quit;
gap> OnBreak := Where;; # restore OnBreak to its default value
```

Note that the change of the environment caused by `DownEnv` only affects variable access in the break loop. If you use `return` to continue a calculation GAP automatically jumps to the right environment level again.

Note also that search for variables looks first in the chain of outer functions which enclosed the definition of a currently executing function, before it looks at the chain of calling functions which led to the current invocation of the function.

```
gap> foo := function()
> local x; x := 1;
> return function() local y; y := x*x; Error("!!\n"); end;
> end;
function(  ) ... end
gap> bar := foo();
function(  ) ... end
gap> fun := function() local x; x := 3; bar(); end;
function(  ) ... end
gap> fun();
Error, !!
 called from
bar(  ); called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> x;
1
brk> DownEnv(1);
brk> x;
3
```

Here the `x` of `foo` which contained the definition of `bar` is found before that of `fun` which caused its execution. Using `DownEnv` we can access the `x` from `fun`.

6.5 Error

1 ▶ `Error(` *messages*`...)` F

`Error` signals an error from within a function. First the messages *messages* are printed, this is done exactly as if `Print` (see 6.2) were called with these arguments. Then a break loop (see 6.3) is entered, unless the standard error output is not connected to a terminal. You can leave this break loop with `return;` to continue execution with the statement following the call to `Error`.

6.6 ErrorCount

1 ▶ `ErrorCount()` F

`ErrorCount` returns a count of the number of errors (including user interruptions) which have occurred in the GAP session so far. This count is reduced modulo 2^{28} on 32 bit systems, 2^{60} on 64 bit systems. The count is incremented by each error, even if GAP was started with the `-T` option to disable the break loop.

6.7 Leaving GAP

The normal way to terminate a GAP session is to enter either `quit;` (note the semicolon) or an end-of-file character (usually ctrl-D) at the `gap>` prompt in the main read eval print loop.

An emergency way to leave GAP is to enter

1 ► `QUIT`

at any `gap>` or `brk>` or `brk_`*nn*`>` prompt.

2 ► `InstallAtExit(` *func* `)` F
► `QUITTING` V

Before actually terminating, GAP will call (with no arguments) all of the functions that have been installed using `InstallAtExit`. These typically perform tasks such as cleaning up temporary files created during the session, and closing open files. If an error occurs during the execution of one of these functions, that function is simply abandoned, no break loop is entered.

```
gap> InstallAtExit(function() Print("bye\n"); end);
gap> quit;
bye
```

During execution of these functions, the global variable `QUITTING` will be set to `true` if GAP is exiting because the user typed `QUIT` and `false` otherwise. Since `QUIT` is considered as an emergency measure, different action may be appropriate.

3 ► `SaveOnExitFile` V

If, when GAP is exiting due to a `quit` or end-of-file (ie not due to a `QUIT`) the variable `SaveOnExitFile` is bound to a string value, then the system will try to save the workspace to that file.

6.8 Line Editing

GAP allows one you to edit the current input line with a number of editing commands. Those commands are accessible either as **control keys** or as **escape keys**. You enter a control key by pressing the *ctrl* key, and, while still holding the *ctrl* key down, hitting another key *key*. You enter an escape key by hitting *esc* and then hitting another key *key*. Below we denote control keys by *ctrl-key* and escape keys by *esc-key*. The case of *key* does not matter, i.e., *ctrl*-`A` and *ctrl*-`a` are equivalent.

Typing *ctrl-key* or *esc-key* for characters not mentioned below always inserts *ctrl-key* resp. *esc-key* at the current cursor position.

The first few commands allow you to move the cursor on the current line.

ctrl-`A` move the cursor to the beginning of the line.

esc-`B` move the cursor to the beginning of the previous word.

ctrl-`B` move the cursor backward one character.

ctrl-`F` move the cursor forward one character.

esc-`F` move the cursor to the end of the next word.

ctrl-`E` move the cursor to the end of the line.

The next commands delete or kill text. The last killed text can be reinserted, possibly at a different position, with the "yank" command *ctrl*-`Y`.

ctrl-`H` or *del* delete the character left of the cursor.

ctrl-`D` delete the character under the cursor.

ctrl-K kill up to the end of the line.

esc-D kill forward to the end of the next word.

esc-*del* kill backward to the beginning of the last word.

ctrl-X kill entire input line, and discard all pending input.

ctrl-Y insert (yank) a just killed text.

The next commands allow you to change the input.

ctrl-T exchange (twiddle) current and previous character.

esc-U uppercase next word.

esc-L lowercase next word.

esc-C capitalize next word.

The *tab* character, which is in fact the control key *ctrl*-I, looks at the characters before the cursor, interprets them as the beginning of an identifier and tries to complete this identifier. If there is more than one possible completion, it completes to the longest common prefix of all those completions. If the characters to the left of the cursor are already the longest common prefix of all completions hitting *tab* a second time will display all possible completions.

tab complete the identifier before the cursor.

The next commands allow you to fetch previous lines, e.g., to correct typos, etc. This history is limited to about 8000 characters.

ctrl-L insert last input line before current character.

ctrl-P redisplay the last input line, another *ctrl*-P will redisplay the line before that, etc. If the cursor is not in the first column only the lines starting with the string to the left of the cursor are taken.

ctrl-N Like *ctrl*-P but goes the other way round through the history.

esc-< goes to the beginning of the history.

esc-> goes to the end of the history.

ctrl-O accepts this line and perform a *ctrl*-N.

Finally there are a few miscellaneous commands.

ctrl-V enter next character literally, i.e., enter it even if it is one of the control keys.

ctrl-U execute the next line editing command 4 times.

esc-*num* execute the next line editing command *num* times.

esc-*ctrl*-L redisplay input line.

The four arrow keys (cursor keys) can be used instead of *ctrl*-B, *ctrl*-F, *ctrl*-P, and *ctrl*-N, respectively.

6.9 Editing Files

In most cases, it is preferable to create longer input (in particular GAP programs) separately in an editor, and to read in the result via `Read`. Note that `Read` by default reads from the directory in which GAP was started (respectively under Windows the directory containing the GAP binary), so you might hav eto give an absolute path to the file.

If you cannot create several windows, the `Edit` command may be used to leave GAP, start an editor, and read in the edited file automatically.

1 ▶ `Edit(` *filename* `)` F

`Edit` starts an editor with the file whose filename is given by the string *filename*, and reads the file back into GAP when you exit the editor again. You should set the GAP variable `EDITOR` to the name of the editor that you usually use, e.g., `/usr/ucb/vi`. This can for example be done in your `.gaprc` file (see the sections on operating system dependent features in Chapter 73).

6.10 Editor Support

In the `etc` subdirectory of the GAP installation we provide some setup files for the editors `vim` and `emacs/xemacs`.

`vim` is a powerful editor that understands the basic `vi` commands but provides much more functionality. You can find more information about it (and download it) from

`http://www.vim.org` .

To get support for GAP syntax in vim, create in your home directory a directory `.vim` and a subdirectory `.vim/indent` (If you are not using Unix, refer to the `vim` documentation on where to place syntax files). Then copy the file `etc/gap.vim` in this `.vim` directory and copy the file `etc/gap_indent.vim` to `.vim/indent/gap.vim`.

Then edit the `.vimrc` file in your home directory. Add lines as in the following example:

```
if has("syntax")
  syntax on             " Default to no syntax highlightning
endif

" For GAP files
augroup gap
  " Remove all gap autocommands
  au!
  autocmd BufRead,BufNewFile *.g,*.gi,*.gd source ~/.vim/gap.vim
autocmd BufRead,BufNewFile *.g,*.gi,*.gd set filetype=gap comments=s:##\ \ ,m:##\ \ ,e:##\ \ b:#

" I'm using the external program 'par' for formating comment lines starting
" with '##  '. Include these lines only when you have par installed.
  autocmd BufRead,BufNewFile *.g,*.gi,*.gd set formatprg="par w76p4s0j"
  autocmd BufWritePost,FileWritePost *.g,*.gi,*.gd set formatprg="par w76p0s0j"
augroup END
```

See the headers of the two mentioned files for additional comments. Adjust details according to your personal taste.

Setup files for `(x)emacs` are contained in the `etc/emacs` subdirectory.

6.11 SizeScreen

1 ▶ `SizeScreen()` F
▶ `SizeScreen( [` x`,` y `] )` F

With no arguments, `SizeScreen` returns the size of the screen as a list with two entries. The first is the length of each line, the second is the number of lines.

With one argument that is a list, `SizeScreen` sets the size of the screen; x is the length of each line, y is the number of lines. Either value x or y may be missing (i.e. left unbound), to leave this value unaffected. It returns the new values. Note that those parameters can also be set with the command line options `-x` x and `-y` y (see Section 3.1).

To check/change whether line breaking occurs for files and streams see 10.4.9 and 10.4.8.

The screen width must be between 20 and 256 characters (inclusive) and the depth at least 10 lines. Values outside this range will be adjusted to the nearest endpoint of the range.

7 Debugging and Profiling Facilities

This chapter describes some functions that are useful mainly for debugging and profiling purposes.

The sections 7.2.1 and 7.3 show how to get information about the methods chosen by the method selection mechanism (see chapter 2 in the programmer's manual).

The final sections describe functions for collecting statistics about computations (see 7.6.2, 7.7).

7.1 Recovery from NoMethodFound-Errors

When the method selection fails because there is no applicable method, an error as in the following example occurs and a break loop is entered:

```
gap> IsNormal(2,2);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 1st choice method found for 'IsNormal' on 2 arguments called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk>
```

This only says, that the method selection tried to find a method for `IsNormal` on two arguments and failed. In this situation it is crucial to find out, why this happened. Therefore there are a few functions which can display further information. Note that you can leave the break loop by the `quit` command (see 6.3.1) and that the information about the incident is no longer accessible afterwards.

1 ▶ `ShowArguments( )` F

This function is only available within a break loop caused by a "No Method Found"-error. It prints as a list the arguments of the operation call for which no method was found.

2 ▶ `ShowArgument(` *nr* `)` F

This function is only available within a break loop caused by a "No Method Found"-error. It prints the *nr*-th arguments of the operation call for which no method was found. `ShowArgument` needs exactly one argument which is an integer between 0 and the number of arguments the operation was called with.

3 ▶ `ShowDetails( )` F

This function is only available within a break loop caused by a "No Method Found"-error. It prints the details of this error: The operation, the number of arguments, a flag which indicates whether the operation is being traced, a flag which indicates whether the operation is a constructor method, and the number of methods that refused to apply by calling `TryNextMethod`. The last number is called `Choice` and is printed as an ordinal. So if exactly k methods were found but called `TryNextMethod` and there were no more methods it says `Choice:` k`th`.

4 ▶ `ShowMethods( )` F
▶ `ShowMethods(` *verbosity* `)` F

This function is only available within a break loop caused by a "No Method Found"-error. It prints an overview about the installed methods for those arguments the operation was called with (using `ApplicableMethod`, see 7.2.1). The verbosity can be controlled by the optional integer parameter *verbosity*. The default is 2, which lists all applicable methods. With verbosity 1 `ShowMethods` only shows the number of installed methods and the methods matching, which can only be those that were already called but refused to work by calling `TryNextMethod`. With verbosity 3 not only all installed methods but also the reasons why they do not match are displayed.

5 ▶ `ShowOtherMethods( )` F
▶ `ShowOtherMethods(` *verbosity* `)` F

This function is only available within a break loop caused by a "No Method Found"-error. It prints an overview about the installed methods for a different number of arguments than the number of arguments the operation was called with (using `ApplicableMethod`, see 7.2.1). The verbosity can be controlled by the optional integer parameter *verbosity*. The default is 1 which lists only the number of applicable methods. With verbosity 2 `ShowOtherMethods` lists all installed methods and with verbosity 3 also the reasons, why they are not applicable. Calling `ShowOtherMethods` with verbosity 3 in this function will normally not make any sense, because the different numbers of arguments are simulated by supplying the corresponding number of ones, for which normally no reasonable methods will be installed.

7.2 ApplicableMethod

1 ▶ `ApplicableMethod(` *opr*`,` *args* `[,` *printlevel* `] )` F
▶ `ApplicableMethod(` *opr*`,` *args*`,` *printlevel*`,` *nr* `)` F
▶ `ApplicableMethod(` *opr*`,` *args*`,` *printlevel*`, "all" )` F
▶ `ApplicableMethodTypes(` *opr*`,` *args* `[,` *printlevel* `] )` F
▶ `ApplicableMethodTypes(` *opr*`,` *args*`,` *printlevel*`,` *nr* `)` F
▶ `ApplicableMethodTypes(` *opr*`,` *args*`,` *printlevel*`, "all" )` F

In the first form, `ApplicableMethod` returns the method of highest rank that is applicable for the operation *opr* with the arguments in the list *args*. The default *printlevel* is `0`. If no method is applicable then `fail` is returned.

In the second form, if *nr* is a positive integer then `ApplicableMethod` returns the *nr*-th applicable method for the operation *opr* with the arguments in the list *args*, where the methods are ordered according to descending rank. If less than *nr* methods are applicable then `fail` is returned.

If the fourth argument is the string `"all"` then `ApplicableMethod` returns a list of all applicable methods for *opr* with arguments *args*, ordered according to descending rank.

Depending on the integer value *printlevel*, additional information is printed. Admissible values and their meaning are as follows.

0 no information,

1 information about the applicable method,

2 also information about the not applicable methods of higher rank,

3 also for each not applicable method the first reason why it is not applicable,

4 also for each not applicable method all reasons why it is not applicable.

6 also the function body of the selected method(s)

When a method returned by `ApplicableMethod` is called then it returns either the desired result or the string `TRY_NEXT_METHOD`, which corresponds to a call to `TryNextMethod` in the method and means that the method selection would call the next applicable method.

Note: The kernel provides special treatment for the infix operations `\+`, `\-`, `\*`, `\/`, `\^`, `\mod` and `\in`. For some kernel objects (notably cyclotomic numbers, finite field elements and vectors thereof) it calls kernel methods circumventing the method selection mechanism. Therefore for these operations `ApplicableMethod` may return a method which is not the kernel method actually used.

`ApplicableMethod` does not work for constructors (for example `GeneralLinearGroupCons` is a constructor).

The function `ApplicableMethodTypes` takes the **types** or **filters** of the arguments as argument (if only filters are given of course family predicates cannot be tested).

7.3 Tracing Methods

1 ▶ `TraceMethods( `*oprs*` )` F

After the call of `TraceMethods` with a list *oprs* of operations, whenever a method of one of the operations in *oprs* is called the information string used in the installation of the method is printed.

2 ▶ `UntraceMethods( `*oprs*` )` F

turns the tracing off for all operations in *oprs*.

```
gap> TraceMethods( [ Size ] );
gap> g:= Group( (1,2,3), (1,2) );;
gap> Size( g );
#I  Size: for a permutation group
#I  Setter(Size): system setter
#I  Size: system getter
#I  Size: system getter
6
gap> UntraceMethods( [ Size ] );
```

3 ▶ `TraceImmediateMethods( `*flag*` )` F

If *flag* is true, tracing for all immediate methods is turned on. If *flag* is false it is turned off. (There is no facility to trace **specific** immediate methods.)

```
gap> TraceImmediateMethods( true );
gap> g:= Group( (1,2,3), (1,2) );;
#I  immediate: Size
#I  immediate: IsCyclic
#I  immediate: IsCommutative
#I  immediate: IsTrivial
#I  immediate: IsStabChainViaChainSubgroup
#I  immediate: IsChainTypeGroup
gap> Size( g );
#I  immediate: IsNonTrivial
#I  immediate: Size
#I  immediate: IsStabChainViaChainSubgroup
#I  immediate: IsChainTypeGroup
#I  immediate: IsNonTrivial
#I  immediate: GeneralizedPcgs
#I  immediate: IsPerfectGroup
#I  immediate: IsEmpty
```

```
6
gap> TraceImmediateMethods( false );
gap> UntraceMethods( [ Size ] );
```

This example gives an explanation for the two calls of the "system getter" for `Size`. Namely, there are immediate methods that access the known size of the group. Note that the group `g` was known to be finitely generated already before the size was computed, the calls of the immediate method for `IsFinitelyGeneratedGroup` after the call of `Size` have other arguments than `g`.

7.4 Info Functions

The `Info` mechanism permits operations to display intermediate results or information about the progress of the algorithms. Information is always given according to one or more **info classes**. Each of the info classes defined in the GAP library usually covers a certain range of algorithms, so for example `InfoLattice` covers all the cyclic extension algorithms for the computation of a subgroup lattice.

The amount of information to be displayed can be specified by the user for each info class separately by a **level**, the higher the level the more information will be displayed. Ab initio all info classes have level zero except `InfoWarning` (see 7.4.6) which initially has level 1.

1 ▶ `NewInfoClass(` *name* `)` O

creates a new info class with name *name*.

2 ▶ `DeclareInfoClass(` *name* `)` F

creates a new info class with name *name* and binds it to the global variable *name*. The variable must previously be writable, and is made readonly by this function.

3 ▶ `SetInfoLevel(` *infoclass*`,` *level* `)` O

Sets the info level for *infoclass* to *level*.

4 ▶ `InfoLevel(` *infoclass* `)` O

returns the info level of *infoclass*.

5 ▶ `Info(` *infoclass*`,` *level*`,` *info* `[,`*moreinfo* `. . .] )`

If the info level of *infoclass* is at least *level* the remaining arguments (*info* and possibly *moreinfo* and so on) are evaluated and viewed, preceded by '#I ' and followed by a newline. Otherwise the third and subsequent arguments are not evaluated. (The latter can save substantial time when displaying difficult results.)

```
gap> InfoExample:=NewInfoClass("InfoExample");;
gap> Info(InfoExample,1,"one");Info(InfoExample,2,"two");
gap> SetInfoLevel(InfoExample,1);
gap> Info(InfoExample,1,"one");Info(InfoExample,2,"two");
#I  one
gap> SetInfoLevel(InfoExample,2);
gap> Info(InfoExample,1,"one");Info(InfoExample,2,"two");
#I  one
#I  two
gap> InfoLevel(InfoExample);
2
gap> Info(InfoExample,3,Length(Combinations([1..9999])));
```

Note that the last `Info` call is executed without problems, since the actual level `2` of `InfoExample` causes `Info` to ignore the last argument, which prevents `Length(Combinations([1..9999]))` from being evaluated; note that an evaluation would be impossible due to memory restrictions.

A set of info classes (called an **info selector**) may be passed to a single `Info` statement. As a shorthand, info classes and selectors may be combined with `+` rather than `Union`. In this case, the message is triggered if the level of **any** of the classes is high enough.

```
gap> InfoExample:=NewInfoClass("InfoExample");;
gap> SetInfoLevel(InfoExample,0);
gap> Info(InfoExample + InfoWarning, 1, "hello");
#I  hello
gap> Info(InfoExample + InfoWarning, 2, "hello");
gap> SetInfoLevel(InfoExample,2);
gap> Info(InfoExample + InfoWarning, 2, "hello");
#I  hello
gap> InfoLevel(InfoWarning);
1
```

6 ▶ `InfoWarning` V

is an info class to which general warnings are sent at level 1, which is its default level. More specialised warnings are `Info`-ed at `InfoWarning` level 2, e.g. information about the autoloading of GAP packages and the initial line matched when displaying an on-line help topic.

7.5 Assertions

Assertions are used to find errors in algorithms. They test whether intermediate results conform to required conditions and issue an error if not.

1 ▶ `SetAssertionLevel(` *lev* `)` F

assigns the global assertion level to *lev*. By default it is zero.

2 ▶ `AssertionLevel()` F

returns the current assertion level.

3 ▶ `Assert(` *lev*`,` *cond* `)` F
▶ `Assert(` *lev*`,` *cond*`,` *message* `)` F

With two arguments, if the global assertion level is at least *lev*, condition *cond* is tested and if it does not return `true` an error is raised. Thus `Assert(lev,` *cond*`)` is equivalent to the code

```
if AssertionLevel() >= lev and not <cond> then
  Error("Assertion failure");
fi;
```

With the *message* argument form of the `Assert` statement, if the global assertion level is at least *lev*, condition *cond* is tested and if it does not return `true` then *message* is evaluated and printed.

Assertions are used at various places in the library. Thus turning assertions on can slow code execution significantly.

7.6 Timing

1 ▶ `Runtimes()` F

`Runtimes` returns a record with components bound to integers or `fail`. Each integer is the cpu time (processor time) in milliseconds spent by GAP in a certain status:

`user_time` cpu time spent with GAP functions (without child processes).

`system_time` cpu time spent in system calls, e.g., file access (`fail` if not available).

`user_time_children` cpu time spent in child processes (`fail` if not available).

`system_time_children` cpu time spent in system calls by child processes (`fail` if not available).

Note that this function is not fully supported on all systems. Only the `user_time` component is (and may on some systems include the system time).

The following example demonstrates tasks which contribute to the different time components:

```
gap> Runtimes(); # after startup
rec( user_time := 3980, system_time := 60, user_time_children := 0,
  system_time_children := 0 )
gap> Exec("cat /usr/bin/*|wc"); # child process with a lot of file access
 893799 7551659 200928302
gap> Runtimes();
rec( user_time := 3990, system_time := 60, user_time_children := 1590,
  system_time_children := 600 )
gap> a:=0;;for i in [1..100000000] do a:=a+1; od; # GAP user time
gap> Runtimes();
rec( user_time := 12980, system_time := 70, user_time_children := 1590,
  system_time_children := 600 )
gap> ?blabla  # first call of help, a lot of file access
Help: no matching entry found
gap> Runtimes();
rec( user_time := 13500, system_time := 440, user_time_children := 1590,
  system_time_children := 600 )
```

2 ▶ `Runtime()` F

`Runtime` returns the time spent by GAP in milliseconds as an integer. It is the same as the value of the `user_time` component given by `Runtimes`, as explained above.

See `StringTime` (26.8.9) for a translation from milliseconds into hour/minute format.

3 ▶ `time;`

in the read-eval-print loop returns the time the last command took.

7.7 Profiling

Profiling of code can be used to determine in which parts of a program how much time has been spent during runtime.

1 ▶ `ProfileOperations( [`*true/false*`] )` F

When called with argument *true*, this function starts profiling of all operations. Old profiling information is cleared. When called with *false* it stops profiling of all operations. Recorded information is still kept, so you can display it even after turning the profiling off.

When called without argument, profiling information for all profiled operations is displayed (see 7.7.8).

2 ▶ `ProfileOperationsAndMethods( [`*true/false*`] )` F

When called with argument *true*, this function starts profiling of all operations and their methods. Old profiling information is cleared. When called with *false* it stops profiling of all operations and their methods. Recorded information is still kept, so you can display it even after turning the profiling off.

When called without argument, profiling information for all profiled operations and their methods is displayed (see 7.7.8).

3 ▶ `ProfileMethods(` *ops* `)` F

starts profiling of the methods for all operations in *ops*.

4 ▶ `UnprofileMethods(` *ops* `)` F

stops profiling of the methods for all operations in *ops*. Recorded information is still kept, so you can display it even after turning the profiling off.

5 ▶ `ProfileFunctions(` *funcs* `)` F

turns profiling on for all function in *funcs*. You can use `ProfileGlobalFunctions` (see 7.7.7) to turn profiling on for all globally declared functions simultaneously.

6 ▶ `UnprofileFunctions(` *funcs* `)` F

turns profiling off for all function in *funcs*. Recorded information is still kept, so you can display it even after turning the profiling off.

7 ▶ `ProfileGlobalFunctions( true )` F
▶ `ProfileGlobalFunctions( false )` F

`ProfileGlobalFunctions(true)` turns on profiling for all functions that have been declared via `DeclareGlobalFunction`. A function call with the argument `false` turns it off again.

8 ▶ `DisplayProfile( )` F
▶ `DisplayProfile(` *funcs* `)` F

In the first form, `DisplayProfile` displays the profiling information for profiled operations, methods and functions. If an argument *funcs* is given, only profiling information for the functions in *funcs* is given. The information for a profiled function is only displayed if the number of calls to the function or the total time spent in the function exceeds a given threshold (see 7.7.9).

Profiling information is displayed in a list of lines for all functions (also operations and methods) which are profiled. For each function, "count" gives the number of times the function has been called. "self" gives the time spent in the function itself, "child" the time spent in profiled functions called from within this function. The list is sorted according to the total time spent, that is the sum "self"+"child".

9 ▶ `PROFILETHRESHOLD` V

This variable is a list [*cnt*, *time*] of length two. `DisplayProfile` will only display lines for functions which are called at least *cnt* times or whose **total** time ("self"+"child") is at least *time*. The default value of `PROFILETHRESHOLD` is [10000,30].

10 ▶ `ClearProfile( )` F

clears all stored profiling information.

```
gap> ProfileOperationsAndMethods(true);
gap> ConjugacyClasses(PrimitiveGroup(24,1));;
gap> ProfileOperationsAndMethods(false);
gap> DisplayProfile();
  count  self/ms  chld/ms  function
[the following is excerpted from a much longer list]
   1620      170       90  CycleStructurePerm: default method
   1620       20      260  CycleStructurePerm
 114658      280        0  Size: for a list that is a collection
    287       20      290  Meth(CyclesOp)
    287        0      310  CyclesOp
     26        0      330  Size: for a conjugacy class
   2219       50      380  Size
      2        0      670  IsSubset: for two collections (loop over the ele*
     32        0      670  IsSubset
     48       10      670  IN: for a permutation, and a permutation group
      2       20      730  Meth(ClosureGroup)
      2        0      750  ClosureGroup
      1        0      780  DerivedSubgroup
      1        0      780  Meth(DerivedSubgroup)
      4        0      810  Meth(StabChainMutable)
     29        0      810  StabChainOp
      3      700      110  Meth(StabChainOp)
      1        0      820  Meth(IsSimpleGroup)
      1        0      820  Meth(IsSimple)
    552       10      830  Meth(StabChainImmutable)
     26      490      480  CentralizerOp: perm group,elm
     26        0      970  Meth(StabilizerOfExternalSet)
    107        0      970  CentralizerOp
    926       10      970  Meth(CentralizerOp)
    819     2100     2340  Meth(IN)
      1       10     4890  ConjugacyClasses: by random search
      1        0     5720  ConjugacyClasses: perm group
      2        0     5740  ConjugacyClasses
           6920           TOTAL
gap> DisplayProfile(StabChainOp,DerivedSubgroup); # only two functions
  count  self/ms  chld/ms  function
      1        0      780  DerivedSubgroup
     29        0      810  StabChainOp
           6920           OTHER
           6920           TOTAL
```

Note that profiling (even the command `ProfileOperationsAndMethods(true)`) can take substantial time and GAP will perform much more slowly when profiling than when not.

11 ▶ `DisplayCacheStats( )` F

displays statistics about the different caches used by the method selection.

12 ▶ `ClearCacheStats( )` F

clears all statistics about the different caches used by the method selection.

7.8 Information about the version used

1 ▶ `DisplayRevision( )` F

Displays the revision numbers of all loaded files from the library.

7.9 Test Files

Test files are used to check that GAP produces correct results in certain computations. A selection of test files for the library can be found in the `tst` directory of the GAP distribution.

1 ▶ `ReadTest(` *name-file* `)` O

reads a test file. A test file starts with a line

```
gap> START_TEST("arbitrary identifier string");
```

(Note that the `gap>` prompt is part of the line!) It continues by lines of GAP input and corresponding output. The input lines again start with the `gap>` prompt (or the `>` prompt if commands exceed one line). The output is exactly as would result from typing in the input interactively to a GAP session (on a screen with 80 characters per line).

The test file stops with a line

```
gap> STOP_TEST( "filename", 10000 );
```

Here the string `"filename"` should give the name of the test file. The number is a proportionality factor that is used to output a "GAPstone" speed ranking after the file has been completely processed. For the files provided with the distribution this scaling is roughly equalized to yield the same numbers as produced by `combinat.tst`.

7.10 Debugging Recursion

The GAP interpreter monitors the level of nesting of GAP functions during execution. By default, whenever this nesting reaches a multiple of 5000, GAP enters a break loop (6.3) allowing you to terminate the calculation, or enter `return;` to continue it.

```
gap> dive:= function(depth) if depth>1 then dive(depth-1); fi; return; end;
function( depth ) ... end
gap> dive(100);
gap> OnBreak:= function() Where(1); end; # shorter traceback
function(  ) ... end
gap> dive(6000);
recursion depth trap (5000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
gap> dive(11000);
recursion depth trap (5000)
 at
```

```
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
recursion depth trap (10000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
gap>
```

This behaviour can be controlled using the procedure

1 ▶ `SetRecursionTrapInterval(` *interval* `)` F

interval must be a non-negative small integer (between 0 and 2^{28}). An *interval* of 0 suppresses the monitoring of recursion altogether. In this case excessive recursion may cause GAP to crash.

```
gap> dive:= function(depth) if depth>1 then dive(depth-1); fi; return; end;
function( depth ) ... end
gap> SetRecursionTrapInterval(1000);
gap> dive(2500);
recursion depth trap (1000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
recursion depth trap (2000)
 at
dive( depth - 1 );
 called from
dive( depth - 1 ); called from
...
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you may 'return;' to continue
brk> return;
gap> SetRecursionTrapInterval(-1);
SetRecursionTrapInterval( <interval> ): <interval> must be a non-negative smal\
```

```
l integer
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <interval> via 'return <interval>;' to continue
brk> return ();
SetRecursionTrapInterval( <interval> ): <interval> must be a non-negative smal\
l integer
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace <interval> via 'return <interval>;' to continue
brk> return 0;
gap> dive(20000);
gap> dive(2000000);
Segmentation fault
```

7.11 Global Memory Information

The GAP environment provides automatic memory management, so that the programmer does not need to concern themselves with allocating space for objects, or recovering space when objects are no longer needed. The component of the kernel which provides this is called `GASMAN` (GAP Storage MANager). Messages reporting garbage collections performed by GASMAN can be switched on by the `-g` command line option (see section 3.1). There are also some facilities to access information from GASMAN from GAP programs.

1 ▶ `GasmanStatistics( )` F

`GasmanStatistics()` returns a record containing some information from the garbage collection mechanism. The record may contain up to two components: `full` and `partial`

The `full` component will be present if a full garbage collection has taken place since GAP started. It contains information about the most recent full garbage collection. It is a record, with six components: `livebags` contains the number of bags which survived the garbage collection; `livekb` contains the total number of kilobytes occupied by those bags; `deadbags` contains the total number of bags which were reclaimed by that garbage collection and all the partial garbage collections preceeding it, since the previous full garbage collection; `deadkb` contains the total number of kilobytes occupied by those bags; `freekb` reports the total number of kilobytes available in the GAP workspace for new objects and `totalkb` the actual size of the workspace.

These figures shouold be viewed with some caution. They are stored internally in fixed length integer formats, and `deadkb` and `deadbags` are liable to overflow if there are many partial collections before a full collection. Also, note that `livekb` and `freekb` will not usually add up to `totalkb`. The difference is essentially the space overhead of the memory management system.

The `partial` component will be present if there has been a partial garbage collection since the last full one. It is also a record with the same six components as `full`. In this case `deadbags` and `deadkb` refer only to the number and total size of the garbage bags reclaimed in this partial garbage collection and `livebags`and `livekb` only to the numbers and total size of the young bags that were considered for garbage collection, and survived.

2 ▶ `GasmanMessageStatus( )` F
▶ `SetGasmanMessageStatus(` *stat* `)` F

`GasmanMessageStatus()` returns one of the string "none", "full" or "all", depending on whether the garbage collector is currently set to print messages on no collections, full collections only or all collections.

`SetGasmanMessageStatus( stat )` sets the garbage collector messaging level. *stat* should be one of the strings "none", "full" or "all".

3 ▶ `GasmanLimits( )` F

`GasmanLimits()` returns a record with three components: `min` is the minimum workspace size as set by the `-m` command line option in kilobytes. The workspace size will never be reduced below this by the garbage collector. `max` is the maximum workspace size, as set by the '-o' command line option, also in kilobytes. If the workspace would need to grow past this point, GAP will enter a break loop to warn the user. A value of 0 indicates no limit.`kill` is the absolute maximum, set by the `-K` command line option. The workspace will never be allowed to grow past this limit.

8 Options Stack

GAP Supports a global Options system. This is intended as a way for the user to provide guidance to various algorithms that might be used in a computation. Such guidance should not change mathematically the specification of the computation to be performed, although it may change the algorithm used. A typical example is the selection of a strategy for the Todd-Coxeter coset enumeration procedure. An example of something not suited to the options mechanism is the imposition of exponent laws in the p-Quotient algorithm.

The basis of this system is a global stack of records. All the entries of each record are thought of as options settings, and the effective setting of an option is given by the topmost record in which the relevant field is bound.

The reason for the choice of a stack is the intended pattern of use:

```
PushOptions( rec( <stuff> ) );
DoSomething( <args> );
PopOptions();
```

This can be abbreviated, to `DoSomething(` *args* `:` *stuff* `);` with a small additional abbreviation of *stuff* permitted. See 4.10.2 for details. The full form can be used where the same options are to run across several calls, or where the `DoSomething` procedure is actually a binary operation, or other function with special syntax.

At some time, an options predicate or something of the kind may be added to method selection.

An alternative to this system is the use of additional optional arguments in procedure calls. This is not felt to be adequate because many procedure calls might cause, for example, a coset enumeration and each would need to make provision for the possibility of extra arguments. In this system the options are pushed when the user-level procedure is called, and remain in effect (unless altered) for all procedures called by it.

1 ▶ `PushOptions(` *options record* `)` F

This function pushes a record of options onto the global option stack. Note that `PushOption(rec(`*opt* `:= fail))` has the effect of resetting option *opt*, since an option that has never been set has the value `fail` returned by `ValueOptions`.

Note that there is no check for misspelt or undefined options.

2 ▶ `PopOptions( )` F

This function removes the top-most options record from the options stack.

3 ▶ `ResetOptionsStack( )` F

unbinds (i.e. removes) all the options records from the options stack.

Note: `ResetOptionsStack` should **not** be used within a function. Its intended use is to clean up the options stack in the event that the user has `quit` from a `break` loop, so leaving a stack of no-longer-needed options (see 6.3.1).

4 ▶ `ValueOption( opt )` F

This function is the main method of accessing the Options Stack; *opt* should be the name of an option, i.e. a string. A function which makes decisions which might be affected by options should examine the result of `ValueOption( opt )`. If *opt* has never been set then `fail` is returned.

5 ▶ `DisplayOptionsStack( )` F

This function prints a human-readable display of the complete options stack.

6 ▶ `InfoOptions` V

This info class can be used to enable messages about options being changed (level 1) or accessed (level 2).

The example below shows simple manipulation of the Options Stack, first using `PushOptions` and `PopOptions` and then using the special function calling syntax.

```
gap> foo := function()
> Print("myopt1 = ", ValueOption("myopt1"),
>       " myopt2 = ",ValueOption("myopt2"),"\n");
> end;
function(  ) ... end
gap> foo();
myopt1 = fail myopt2 = fail
gap> PushOptions(rec(myopt1 := 17));
gap> foo();
myopt1 = 17 myopt2 = fail
gap> DisplayOptionsStack();
[ rec(
      myopt1 := 17 ) ]
gap> PopOptions();
gap> foo();
myopt1 = fail myopt2 = fail
gap> foo( : myopt1, myopt2 := [Z(3),"aardvark"]);
myopt1 = true myopt2 = [ Z(3), "aardvark" ]
gap> DisplayOptionsStack();
[  ]
gap>
```

9 Files and Filenames

Files are identified by filenames, which are represented in GAP as strings. Filenames can be created directly by the user or a program, but of course this is operating system dependent.

Filenames for some files can be constructed in a system independent way using the following functions. This is done by first getting a directory object for the directory the file shall reside in, and then constructing the filename. However, it is sometimes necessary to construct filenames of files in subdirectories relative to a given directory object. In this case the directory separator is **always** `'/'` even under DOS or MacOS.

Section 9.3 describes how to construct directory objects for the common GAP and system directories. Using the command `Filename` described in section 9.4.1 it is possible to construct a filename pointing to a file in these directories. There are also functions to test for accessibility of files, see 9.6.

9.1 Portability

For portability filenames and directory names should be restricted to at most 8 alphanumerical characters optionally followed by a dot `'.'` and between 1 and 3 alphanumerical characters. Upper case letters should be avoided because some operating systems do not make any distinction between case, so that `NaMe`, `Name` and `name` all refer to the same file whereas some operating systems are case sensitive. To avoid problems only lower case characters should be used.

Another function which is system-dependent is:

1 ▶ `LastSystemError()` F

`LastSystemError` returns a record describing the last system error that has occurred. This record contains at least the component `message` which is a string. This message is, however, highly operating system dependent and should only be used as an informational message for the user.

9.2 GAP Root Directory

When starting GAP it is possible to specify various directories as root directories. In GAP's view of the world these directories are merged into one meta-directory. This directory is called GAP **root directory** in the following.

For example, if *root1*`;`*root2*`;...` is passed as argument to `-l` when GAP is started and GAP wants to locate a file `lib/group.gd` in the GAP root directory it will first check if the file exists in *root1*, if not, it checks *root2*, and so on.

This layout makes it possible to have one system-wide installation of GAP which is read-only but still allows users to modify individual files. Therefore instead of constructing an absolute path name to a file you should always use `DirectoriesLibrary` or `DirectoriesPackageLibrary` together with `Filename` to construct a filename for a file in the GAP root directory.

Example

Suppose that the system-wide installation lives in `/usr/local/lib/gap4` and you want to modify the file `lib/files.gd` without disturbing the system installation.

In this case create a new directory `/home/myhome/gap` containing a subdirectory `lib` which contains the modified `lib/files.gd`.

The directory/file structure now looks like

```
/usr/local/lib/gap4/
/usr/local/lib/gap4/lib/
/usr/local/lib/gap4/lib/files.gd
/home/myhome/gap/
/home/myhome/gap/lib
/home/myhome/gap/lib/files.gd
```

If you start GAP using (under UNIX)

```
you@unix> gap -l '/home/myhome/gap;/usr/local/lib/gap4'
```

then the file `/home/myhome/gap/lib/files.gd` will be used whenever GAP references the file with filename `lib/files.gd` in the GAP root directory.

This setup also allows one to easily install new GAP packages or bugfixes even if no access to the system GAP installation is possible. Simply unpack the files into "/home/myhome/gap".

9.3 Directories

1 ▶ `Directory(` *string* `)` O

returns a directory object for the string *string*. `Directory` understands `.` for "current directory", that is, the directory in which GAP was started. It also understands absolute paths.

If the variable `GAPInfo.UserHome` is defined (this may depend on the operating system) then `Directory` understands a string with a leading `~` character for a path relative to the user's home directory.

Paths are otherwise taken relative to the current directory.

2 ▶ `DirectoryTemporary(` *hint* `)` F
▶ `DirectoryTemporary( )` F

returns a directory object in the category `IsDirectory` for a **new** temporary directory. This is guaranteed to be newly created and empty immediately after the call to `DirectoryTemporary`. GAP will make a reasonable effort to **remove** this directory either when a garbage collection collects the directory object or upon termination of the GAP job that created the directory. *hint* can be used by `DirectoryTemporary` to construct the name of the directory but `DirectoryTemporary` is free to use only a part of *hint* or even ignore it completely.

If `DirectoryTemporary` is unable to create a new directory, `fail` is returned. In this case `LastSystemError` can be used to get information about the error.

3 ▶ `DirectoryCurrent( )` F

returns the directory object for the current directory.

4 ▶ `DirectoriesLibrary( )` F
▶ `DirectoriesLibrary(` *name* `)` F

returns the directory objects for the GAP library *name* as a list. *name* must be one of `"lib"` (the default), `"grp"`, `"prim"`, and so on. The string `""` is also legal and with this argument `DirectoriesLibrary` returns the list of GAP root directories; the return value of `DirectoriesLibrary("");` differs from `GAPInfo.RootPaths` in that the former is a list of directory objects and the latter a list of strings.

The directory *name* must exist in at least one of the root directories, otherwise `fail` is returned.

As the files in the GAP root directory (see 9.2) can be distributed into different directories in the filespace a list of directories is returned. In order to find an existing file in a GAP root directory you should pass that list to `Filename` (see 9.4.1) as the first argument. In order to create a filename for a new file inside a GAP root directory you should pass the first entry of that list. However, creating files inside the GAP root directory is not recommended, you should use `DirectoryTemporary` instead.

5 ▶ `DirectoriesSystemPrograms( )` F

`DirectoriesSystemPrograms` returns the directory objects for the list of directories where the system programs reside as a list. Under UNIX this would usually represent `$PATH`.

6 ▶ `DirectoryContents(` *name* `)` F

This function returns a list of filenames/directory names that reside in the directory with name *name* (given as a string). It is an error, if such a directory does not exist.

9.4 Filename

1 ▶ `Filename(` *dir*`,` *name* `)` O
▶ `Filename(` *list-of-dirs*`,` *name* `)` O

If the first argument is a directory object *dir*, `Filename` returns the (system dependent) filename as a string for the file with name *name* in the directory *dir*. `Filename` returns the filename regardless of whether the directory contains a file with name *name* or not.

If the first argument is a list *list-of-dirs* (possibly of length 1) of directory objects, then `Filename` searches the directories in order, and returns the filename for the file *name* in the first directory which contains a file *name* or `fail` if no directory contains a file *name*.

Examples

In order to locate the system program `date` use `DirectoriesSystemPrograms` together with the second form of `Filename`.

```
gap> path := DirectoriesSystemPrograms();;
gap> date := Filename( path, "date" );
"/bin/date"
```

In order to locate the library file `files.gd` use `DirectoriesLibrary` together with the second form of `Filename`.

```
gap> path := DirectoriesLibrary();;
gap> Filename( path, "files.gd" );
"./lib/files.gd"
```

In order to construct filenames for new files in a temporary directory use `DirectoryTemporary` together with the first form of `Filename`.

```
gap> tmpdir := DirectoryTemporary();;
gap> Filename( [ tmpdir ], "file.new" );
fail
gap> Filename( tmpdir, "file.new" );
"/var/tmp/tmp.0.021738.0001/file.new"
```

9.5 Special Filenames

The special filename `"*stdin*"` denotes the standard input, i.e., the stream through which the user enters commands to GAP. The exact behaviour of reading from `"*stdin*"` is operating system dependent, but usually the following happens. If GAP was started with no input redirection, statements are read from the terminal stream until the user enters the end of file character, which is usually *ctr*-'D'. Note that terminal streams are special, in that they may yield ordinary input **after** an end of file. Thus when control returns to the main read-eval-print loop the user can continue with GAP. If GAP was started with an input redirection, statements are read from the current position in the input file up to the end of the file. When control returns to the main read eval view loop the input stream will still return end of file, and GAP will terminate.

The special filename `"*errin*"` denotes the stream connected to the UNIX `stderr` output. This stream is usually connected to the terminal, even if the standard input was redirected, unless the standard error stream was also redirected, in which case opening of `"*errin*"` fails.

The special filename `"*stdout*"` can be used to print to the standard output.

The special filename `"*errout*"` can be used to print to the standard error output file, which is usually connected to the terminal, even if the standard output was redirected.

9.6 File Access

When the following functions return `false` one can use `LastSystemError` (see 9.1.1) to find out the reason (as provided by the operating system).

1 ▶ `IsExistingFile(` *name-file* `)` F

returns `true` if a file with the filename *name-file* exists and can be seen by the GAP process. Otherwise `false` is returned.

2 ▶ `IsReadableFile(` *name-file* `)` F

returns `true` if a file with the filename *name-file* exists **and** the GAP process has read permissions for the file, or `false` if this is not the case.

3 ▶ `IsWritableFile(` *name-file* `)` F

returns `true` if a file with the filename *name-file* exists **and** the GAP process has write permissions for the file, or `false` if this is not the case.

4 ▶ `IsExecutableFile(` *name-file* `)` F

returns `true` if a file with the filename *name-file* exists **and** the GAP process has execute permissions for the file, or `false` if this is not the case. Note that execute permissions do not imply that it is possible to execute the file, e.g., it may only be executable on a different machine.

5 ▶ `IsDirectoryPath(` *name-file* `)` F

returns `true` if the file with the filename *name-file* exists **and** is a directory and `false` otherwise. Note that this function does not check if the GAP process actually has write or execute permissions for the directory (you can use `IsWritableFile` (see 9.6.3), resp. `IsExecutableFile` (see 9.6.4) to check such permissions).

Examples

Note, in particular, how one may use `LastSystemError` (see 9.1.1) to discover the reason a file access function returns `false`.

```
gap> IsExistingFile( "/bin/date" );     # the file '/bin/date' exists
true
gap> IsExistingFile( "/bin/date.new" ); # the file '/bin/date.new' does not exist
false
gap> IsExistingFile( "/bin/date/new" ); # '/bin/date' is not a directory
false
gap> LastSystemError().message;
"Not a directory"
gap> IsReadableFile( "/bin/date" );     # the file '/bin/date' is readable
true
gap> IsReadableFile( "/bin/date.new" ); # the file '/bin/date.new' does not exist
false
gap> LastSystemError().message;
"No such file or directory"
gap> IsWritableFile( "/bin/date" );     # the file '/bin/date' is not writable ...
false
gap> IsExecutableFile( "/bin/date" );   # ... but executable
true
```

9.7 File Operations

1 ▶ `Read(` *name-file* `)` O

reads the input from the file with the filename *name-file*, which must be given as a string.

`Read` first opens the file *name-file*. If the file does not exist, or if GAP cannot open it, e.g., because of access restrictions, an error is signalled.

Then the contents of the file are read and evaluated, but the results are not printed. The reading and evaluations happens exactly as described for the main loop (see 6.1).

If a statement in the file causes an error a break loop is entered (see 6.3). The input for this break loop is not taken from the file, but from the input connected to the `stderr` output of GAP. If `stderr` is not connected to a terminal, no break loop is entered. If this break loop is left with `quit` (or *ctr*-`D`), GAP exits from the `Read` command, and from all enclosing `Read` commands, so that control is normally returned to an interactive prompt. The `QUIT` statement (see 6.7) can also be used in the break loop to exit GAP immediately.

Note that a statement must not begin in one file and end in another. I.e., *eof* (`end-of-file`) is not treated as whitespace, but as a special symbol that must not appear inside any statement.

Note that one file may very well contain a read statement causing another file to be read, before input is again taken from the first file. There is an operating system dependent maximum on the number of files that may be open simultaneously. Usually it is 15.

2 ▶ `ReadAsFunction(` *name-file* `)` O

reads the file with filename *name-file* as a function and returns this function.

Example

Suppose that the file `/tmp/example.g` contains the following

```
local a;

a := 10;
return a*10;
```

Reading the file as a function will not affect a global variable `a`.

```
gap> a := 1;
1
gap> ReadAsFunction("/tmp/example.g")();
100
gap> a;
1
```

3 ► `PrintTo(` *name-file*`[,` *obj1*`, ...] )` F

works like `Print` (see 6.2.2), except that the arguments *obj1*, ... (if present) are printed to the file with the name *name-file* instead of the standard output. This file must of course be writable by GAP. Otherwise an error is signalled. Note that `PrintTo` will **overwrite** the previous contents of this file if it already existed; in particular, `PrintTo` with just the *name-file* argument empties that file. `AppendTo` can be used to append to a file (see 9.7.4). There is an operating system dependent maximum on the number of output files that may be open simultaneously, usually this is 14.

4 ► `AppendTo(` *name-file*`[,` *obj1*`, ...] )` F

works like `PrintTo` (see 9.7.3), except that the output does not overwrite the previous contents of the file, but is appended to the file.

5 ► `LogTo(` *name-file* `)` O

causes the subsequent interaction to be logged to the file with the name *name-file*, i.e., everything you see on your terminal will also appear in this file. `LogTo` may also be used to log to a stream (see 10.4.5). This file must of course be writable by GAP, otherwise an error is signalled. Note that `LogTo` will overwrite the previous contents of this file if it already existed.

6 ► `LogTo()` M

In this form `LogTo` stops logging to a file or stream.

7 ► `InputLogTo(` *name-file* `)` O

causes the subsequent input to be logged to the file with the name *name-file*, i.e., everything you type on your terminal will also appear in this file. Note that `InputLogTo` and `LogTo` cannot be used at the same time while `InputLogTo` and `OutputLogTo` can. Note that `InputLogTo` will overwrite the previous contents of this file if it already existed.

8 ► `InputLogTo()` M

In this form `InputLogTo` stops logging to a file or stream.

9 ► `OutputLogTo(` *name-file* `)` O

causes the subsequent output to be logged to the file with the name *name-file*, i.e., everything GAP prints on your terminal will also appear in this file. Note that `OutputLogTo` and `LogTo` cannot be used at the same time while `InputLogTo` and `OutputLogTo` can. Note that `OutputLogTo` will overwrite the previous contents of this file if it already existed.

10 ► `OutputLogTo()` M

In this form `OutputLogTo` stops logging to a file or stream.

Note that one should be careful not to write to a logfile with `PrintTo` or `AppendTo`.

11 ► `CrcFile(` *name-file* `)` F

computes a checksum value for the file with filename *name-file* and returns this value as an integer. See Section 3.10 for an example. The function returns `fail` if a system error occurred, say, for example, if *name-file* does not exist. In this case the function `LastSystemError` (see 9.1.1) can be used to get information about the error.

12 ▶ `RemoveFile( name-file )` F

will remove the file with filename *name-file* and returns `true` in case of success. The function returns `fail` if a system error occurred, for example, if your permissions do not allow the removal of *name-file*. In this case the function `LastSystemError` (see 9.1.1) can be used to get information about the error.

13 ▶ `Reread( name-file )` F
▶ `REREADING`

In general, it is not possible to read the same GAP library file twice, or to read a compiled version after reading a GAP version, because crucial global variables are made read-only (see 4.9) and filters and methods are added to global tables.

A partial solution to this problem is provided by the function `Reread` (and related functions `RereadLib` etc.). `Reread( name-file )` sets the global variable `REREADING` to `true`, reads the file named by *name-file* and then resets `REREADING`. Various system functions behave differently when `REREADING` is set to true. In particular, assignment to read-only global variables is permitted, calls to `NewRepresentation` (see 3.2.1 in "Programming in GAP") and `NewInfoClass` (see 7.4.1) with parameters identical to those of an existing representation or info class will return the existing object, and methods installed with `InstallMethod` (see 2.2.1 in "Programming in GAP") may sometimes displace existing methods.

This function may not entirely produce the intended results, especially if what has changed is the super-representation of a representation or the requirements of a method. In these cases, it is necessary to restart GAP to read the modified file.

An additional use of `Reread` is to load the compiled version of a file for which the GAP language version had previously been read (or perhaps was included in a saved workspace). See 3.7 and 3.11 for more information.

10 Streams

Streams provide flexible access to GAP's input and output processing. An **input stream** takes characters from some source and delivers them to GAP which **reads** them from the stream. When an input stream has delivered all characters it is at `end-of-stream`. An **output stream** receives characters from GAP which **writes** them to the stream, and delivers them to some destination.

A major use of streams is to provide efficient and flexible access to files. Files can be read and written using `Read` and `AppendTo`, however the former only allows a complete file to be read as GAP input and the latter imposes a high time penalty if many small pieces of output are written to a large file. Streams allow input files in other formats to be read and processed, and files to be built up efficiently from small pieces of output. Streams may also be used for other purposes, for example to read from and print to GAP strings, or to read input directly from the user.

Any stream is either a **text stream**, which translates the `end-of-line` character (`'\n'`) to or from the system's representation of `end-of-line` (e.g., *new-line* under UNIX, *carriage-return* under MacOS, *carriage-return-new-line* under DOS), or a **binary stream**, which does not translate the `end-of-line` character. The processing of other unprintable characters by text streams is undefined. Binary streams pass them unchanged.

Note that binary streams are **@not yet implemented@**.

Whereas it is cheap to append to a stream, streams do consume system resources, and only a limited number can be open at any time, therefore it is necessary to close a stream as soon as possible using `CloseStream` described in Section 10.2.1. If creating a stream failed then `LastSystemError` (see 9.1.1) can be used to get information about the failure.

10.1 Categories for Streams and the StreamsFamily

1 ▶ `IsStream(` *obj* `)` C

Streams are GAP objects and all open streams, input, output, text and binary, lie in this category.

2 ▶ `IsClosedStream(` *obj* `)` C

When a stream is closed, its type changes to lie in 'IsClosedStream'. This category is used to install methods that trap accesses to closed streams.

3 ▶ `IsInputStream(` *obj* `)` C

All input streams lie in this category, and support input operations such as `ReadByte` (see 10.3)

4 ▶ `IsInputTextStream(` *obj* `)` C

All **text** input streams lie in this category. They translate new-line characters read.

5 ▶ `IsInputTextNone(` *obj* `)` C

It is convenient to use a category to distinguish dummy streams (see 10.9) from others. Other distinctions are usually made using representations

6 ► `IsOutputStream( obj )` C

All output streams lie in this category and support basic operations such as `WriteByte` (see 10.4)

7 ► `IsOutputTextStream( obj )` C

All **text** output streams lie in this category and translate new-line characters on output.

8 ► `IsOutputTextNone( obj )` C

It is convenient to use a category to distinguish dummy streams (see 10.9) from others. Other distinctions are usually made using representations

9 ► `StreamsFamily` V

All streams lie in the `StreamsFamily`

10.2 Operations applicable to All Streams

1 ► `CloseStream( stream )` O

In order to preserve system resources and to flush output streams every stream should be closed as soon as it is no longer used using `CloseStream`.

It is an error to try to read characters from or write characters to a closed stream. Closing a stream tells the GAP kernel and/or the operating system kernel that the file is no longer needed. This may be necessary because the GAP kernel and/or the operating system may impose a limit on how many streams may be open simultaneously.

2 ► `FileDescriptorOfStream( stream )` O

returns the UNIX file descriptor of the underlying file. This is mainly useful for the `UNIXSelect` function call (see 10.2.3). This is as of now only available on UNIX-like operating systems and only for streams to local processes and local files.

3 ► `UNIXSelect( inlist, outlist, exclist, timeoutsec, timeoutusec )` F

makes the UNIX C-library function `select` accessible from GAP for streams. The functionality is as described in the man page (see `man select`). The first three arguments must be lists containing UNIX file descriptors (integers) for streams. They can be obtained via `FileDescriptorOfStream` (see 10.2.2) for streams to local processes and to local files. The argument *timeoutsec* is a timeout in seconds as in the `struct timeval` on the C level. The argument *timeoutusec* is analogously in microseconds. The total timeout is the sum of both. If one of those timeout arguments is not a small integer then no timeout is applicable (`fail` is allowed for the timeout arguments).

The return value is the number of streams that are ready, this may be 0 if a timeout was specified. All file descriptors in the three lists that are not yet ready are replaced by `fail` in this function. So the lists are changed!

This function is not available on the Macintosh architecture and is only available if your operating system has `select`, which is detected during compilation of GAP.

10.3 Operations for Input Streams

Three operations normally used to read files: `Read`, `ReadAsFunction` and `ReadTest` can also be used to read GAP input from a stream. The input is immediately parsed and executed. When reading from a stream *str*, the GAP kernel generates calls to `ReadLine(`*str*`)` to supply text to the parser.

Three further operations: `ReadByte`, `ReadLine` and `ReadAll`, support reading characters from an input stream without parsing them. This can be used to read data in any format and process it in GAP.

Additional operations for input streams support detection of end of stream, and (for those streams for which it is appropriate) random access to the data.

1 ▶ `Read(` *input-text-stream* `)` O

reads the input-text-stream as input until `end-of-stream` occurs. See 9.7 for details.

2 ▶ `ReadAsFunction(` *input-text-stream* `)` O

reads the input-text-stream as function and returns this function. See 9.7 for details.

3 ▶ `ReadTest(` *input-text-stream* `)` O

reads the input-text-stream as test input until `end-of-stream` occurs. See 9.7 for details.

Example

```
gap> # a function with local ‘a’ does not change the global one
gap> a := 1;;
gap> i := InputTextString( "local a; a := 10; return a*10;" );;
gap> ReadAsFunction(i)();
100
gap> a;
1

gap> # reading it via ‘Read’ does
gap> i := InputTextString( "a := 10;" );;
gap> Read(i);
gap> a;
10
```

4 ▶ `ReadByte(` *input-stream* `)` O

`ReadByte` returns one character (returned as integer) from the input stream *stream-in*. `ReadByte` returns `fail` if there is no character available, in particular if it is at the end of a file.

If *stream-in* is the input stream of a input/output process, `ReadByte` may also return `fail` if no byte is currently available.

`ReadByte` is the basic operation for input streams. If a `ReadByte` method is installed for a user-defined type of stream which does not block, then all the other input stream operations will work (although possibly not at peak efficiency).

`ReadByte` will wait (block) until a byte is available. For instance if the stream is a connection to another process, it will wait for the process to output a byte.

5 ▶ `ReadLine(` *input-stream* `)` O

`ReadLine` returns one line (returned as string **with** the newline) from the input stream *input-stream*. `ReadLine` reads in the input until a newline is read or the end-of-stream is encountered.

If *input-stream* is the input stream of a input/output process, `ReadLine` may also return `fail` or return an incomplete line if the other process has not yet written any more. It will always wait (block) for at least one byte to be available, but will then return as much input as is available, up to a limit of one line

A default method is supplied for `ReadLine` which simply calls `ReadByte` repeatedly. This is only safe for streams that cannot block. The kernel uses calls to `ReadLine` to supply input to the parser when reading from a stream.

6 ► `ReadAll(` *input-stream* `)` O
► `ReadAll(` *input-stream* `,` *limit* `)` O

`ReadAll` returns all characters as string from the input stream *stream-in*. It waits (blocks) until at least one character is available from the stream, or until there is evidence that no characters will ever be available again. This last indicates that the stream is at end-of-stream. Otherwise, it reads as much input as it can from the stream without blocking further and returns it to the user. If the stream is already at end of file, so that no bytes are available, `fail` is returned. In the case of a file stream connected to a normal file (not a pseudo-tty or named pipe or similar), all the bytes should be immediately available and this function will read the remainder of the file.

With a second argument, at most *limit* bytes will be returned. Depending on the stream a bounded number of additional bytes may have been read into an internal buffer.

A default method is supplied for `ReadAll` which simply calls `ReadLine` repeatedly. This is only really safe for streams which cannot block. Other streams should install a method for ReadAll

Example

```
gap> i := InputTextString( "1Hallo\nYou\n1" );;
gap> ReadByte(i);
49
gap> CHAR_INT(last);
'1'
gap> ReadLine(i);
"Hallo\n"
gap> ReadLine(i);
"You\n"
gap> ReadLine(i);
"1"
gap> ReadLine(i);
fail
gap> ReadAll(i);
""
gap> RewindStream(i);;
gap> ReadAll(i);
"1Hallo\nYou\n1"
```

7 ► `IsEndOfStream(` *input-stream* `)` O

`IsEndOfStream` returns `true` if the input stream is at *end-of-stream*, and `false` otherwise. Note that `IsEndOfStream` might return `false` even if the next `ReadByte` fails.

8 ► `PositionStream(` *input-stream* `)` O

Some input streams, such as string streams and file streams attached to disk files, support a form of random access by way of the operations `PositionStream`, `SeekPositionStream` and `RewindStream`. `PositionStream` returns a non-negative integer denoting the current position in the stream (usually the number of characters **before** the next one to be read.

If this is not possible, for example for an input stream attached to standard input (normally the keyboard), then `fail` is returned

9 ▶ `RewindStream(` *input-stream* `)` O

`RewindStream` attempts to return an input stream to its starting condition, so that all the same characters can be read again. It returns `true` if the rewind succeeds and `fail` otherwise

A default method implements RewindStream using `SeekPositionStream`.

10 ▶ `SeekPositionStream(` *input-stream*`,` *pos* `)` O

`SeekPositionStream` attempts to rewind or wind forward an input stream to the specified position. This is not possible for all streams. It returns `true` if the seek is successful and `fail` otherwise.

10.4 Operations for Output Streams

1 ▶ `WriteByte(` *output-stream*`,` *byte* `)` O

writes the next character (given as **integer**) to the output stream *output-stream*. The function returns `true` if the write succeeds and `fail` otherwise.

`WriteByte` is the basic operation for input streams. If a `WriteByte` method is installed for a user-defined type of stream, then all the other output stream operations will work (although possibly not at peak efficiency).

2 ▶ `WriteLine(` *output-stream*`,` *string* `)` O

appends *string* to *output-stream*. A final newline is written. The function returns `true` if the write succeeds and `fail` otherwise.

A default method is installed which implements `WriteLine` by repeated calls to `WriteByte`.

3 ▶ `WriteAll(` *output-stream*`,` *string* `)` O

appends *string* to *output-stream*. No final newline is written. The function returns `true` if the write succeeds and `fail` otherwise. It will block as long as necessary for the write operation to complete (for example for a child process to clear its input buffer)

A default method is installed which implements `WriteAll` by repeated calls to `WriteByte`.

When printing or appending to a stream (using `PrintTo`, or `AppendTo` or when logging to a stream), the kernel generates a call to `WriteAll` for each line output.

Example

```
gap> str := "";; a := OutputTextString(str,true);;
gap> WriteByte(a,INT_CHAR('H'));
true
gap> WriteLine(a,"allo");
true
gap> WriteAll(a,"You\n");
true
gap> CloseStream(a);
gap> Print(str);
Hallo
You
```

4 ▶ `PrintTo(` *output-stream*`,` *arg1*`, ... )` F
▶ `AppendTo(` *output-stream*`,` *arg1*`, ... )` F

These functions work like `Print`, except that the output is appended to the output stream *output-stream*.

Example

```
gap> str := "";; a := OutputTextString(str,true);;
gap> AppendTo( a, (1,2,3), ":", Z(3) );
gap> CloseStream(a);
gap> Print( str, "\n" );
(1,2,3):Z(3)
```

5 ▶ `LogTo(` *stream* `)` O

causes the subsequent interaction to be logged to the output stream *stream*. It works in precisely the same way as it does for files (see 9.7.5).

6 ▶ `InputLogTo(` *stream* `)` O

causes the subsequent input to be logged to the output stream *stream*. It works just like it does for files (see 9.7.7).

7 ▶ `OutputLogTo(` *stream* `)` O

causes the subsequent output to be logged to the output stream *stream*. It works just like it does for files (see 9.7.9).

When text is being sent to an output text stream via `PrintTo`, `AppendTo`, `LogTo`, etc., it is, by default formatted just as it would be were it being printed to the screen. Thus, it is broken into lines of reasonable length at (where possible) sensible places, lines containing elements of lists or records are indented, and so forth. This is appropriate if the output is eventually to be viewed by a human, and harmless if it to passed as input to GAP, but may be unhelpful if the output is to be passed as input to another program. It is possible to turn off this behaviour for a stream using the `SetPrintFormattingStatus` operation, and to test whether it is on or off using `PrintFormattingStatus`.

8 ▶ `SetPrintFormattingStatus(` *stream*, *newstatus* `)` O

sets whether output sent to the output stream *stream* via `PrintTo`, `AppendTo`, etc. (but not `WriteByte`, `WriteLine` or `WriteAll`) will be formatted with line breaks and indentation. If the second argument *newstatus* is `true` then output will be so formatted, and if `false` then it will not.

9 ▶ `PrintFormattingStatus(` *stream* `)` O

returns `true` if output sent to the output stream *stream* via `PrintTo`, `AppendTo`, etc. (but not `WriteByte`, `WriteLine` or `WriteAll`) will be formatted with line breaks and indentation, and `false` otherwise (see 10.4.8).

Example

```
gap> s := "";; str := OutputTextString(s,false);;
gap> PrintTo(str,Primes{[1..30]});
gap> s;
"[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,\
 \n  73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]"
gap> Print(s,"\n");
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
  73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ]
gap> SetPrintFormattingStatus(str, false);
gap> PrintTo(str,Primes{[1..30]});
gap> s;
"[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,\
 \n  73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ][ 2, 3, 5, 7, 11, 13, 17, 19\
, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103\
```

```
, 107, 109, 113 ]"
gap> Print(s,"\n");
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
  73, 79, 83, 89, 97, 101, 103, 107, 109, 113 ][ 2, 3, 5, 7, 11, 13, 17, 19, 2\
3, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 1\
07, 109, 113 ]
```

10.5 File Streams

File streams are streams associated with files. An input file stream reads the characters it delivers from a file, an output file stream prints the characters it receives to a file. The following functions can be used to create such streams. They return `fail` if an error occurred, in this case `LastSystemError` (see 9.1.1) can be used to get information about the error.

1 ▶ `InputTextFile(` *name-file* `)` O

`InputTextFile(` *name-file* `)` returns an input stream in the category `IsInputTextStream` that delivers the characters from the file *name-file*.

2 ▶ `OutputTextFile(` *name-file*`,` *append* `)` O

`OutputTextFile(` *name-file*`,` *append* `)` returns an output stream in the category `IsOutputTextFile` that writes received characters to the file *name-file*. If *append* is `false`, then the file is emptied first, otherwise received characters are added at the end of the list.

Example

```
gap> # use a temporary directory
gap> name := Filename( DirectoryTemporary(), "test" );;
gap> # create an output stream, append output, and close again
gap> output := OutputTextFile( name, true );;
gap> AppendTo( output, "Hallo\n", "You\n" );
gap> CloseStream(output);
gap> # create an input, print complete contents of file, and close
gap> input := InputTextFile(name);;
gap> Print( ReadAll(input) );
Hallo
You
gap> CloseStream(input);
gap> # append a single line
gap> output := OutputTextFile( name, true );;
gap> AppendTo( output, "AppendLine\n" );
gap> # close output stream to flush the output
gap> CloseStream(output);
gap> # create an input, print complete contents of file, and close
gap> input := InputTextFile(name);;
gap> Print( ReadAll(input) );
Hallo
You
AppendLine
gap> CloseStream(input);
```

10.6 User Streams

The following two commands create streams which accept characters from, or deliver characters to, the user, via the keyboard or the GAP session display.

1 ▶ `InputTextUser( )` F

returns an input text stream which delivers characters typed by the user (or from the standard input device if it has been redirected). In normal circumstances, characters are delivered one by one as they are typed, without waiting until the end of a line. No prompts are printed.

2 ▶ `OutputTextUser( )` F

returns an output stream which delivers characters to the user's display (or the standard output device if it has been redirected). Each character is delivered immediately it is written, without waiting for a full line of output. Text written in this way is **not** written to the session log (see 9.7.5).

10.7 String Streams

String streams are streams associated with strings. An input string stream reads the characters it delivers from a string, an output string stream appends the characters it receives to a string. The following functions can be used to create such streams.

1 ▶ `InputTextString(` *string* `)` O

`InputTextString(` *string* `)`returns an input stream that delivers the characters from the string *string*. The *string* is not changed when reading characters from it and changing the *string* after the call to `InputTextString` has no influence on the input stream.

2 ▶ `OutputTextString(` *list*`,` *append* `)` O

returns an output stream that puts all received characters into the list *list*. If *append* is `false`, then the list is emptied first, otherwise received characters are added at the end of the list.

Example

```
gap> # read input from a string
gap> input := InputTextString( "Hallo\nYou\n" );;
gap> ReadLine(input);
"Hallo\n"
gap> ReadLine(input);
"You\n"
gap> # print to a string
gap> str := "";;
gap> out := OutputTextString( str, true );;
gap> PrintTo( out, 1, "\n", (1,2,3,4)(5,6), "\n" );
gap> CloseStream(out);
gap> Print( str );
1
(1,2,3,4)(5,6)
```

10.8 Input-Output Streams

Input-output streams capture bidirectional communications between GAP and another process, either locally or (@as yet unimplemented@) remotely.

Such streams support the basic operations of both input and output streams. They should provide some buffering, allowing output date to be written to the stream, even when input data is waiting to be read, but the amount of this buffering is operating system dependent, and the user should take care not to get too far ahead in writing, or behind in reading, or deadlock may occur.

1 ▶ `IsInputOutputStream(` *obj* `)` C

`IsInputOutputStream` is the Category of Input-Output Streams; it returns `true` if the *obj* is an input-output stream and `false` otherwise.

At present the only type of Input-Output streams that are implemented provide communication with a local child process, using a pseudo-tty. They are only available on UNIX systems.

Like other streams, write operations are blocking, read operations will block to get the first character, but not thereafter.

As far as possible, no translation is done on characters written to, or read from the stream, and no control characters have special effects, but the details of particular pseudo-tty implementations may effect this.

2 ▶ `InputOutputLocalProcess(` *dir*`,` *executable*`,` *args* `)` F

starts up a slave process, whose executable file is *executable*, with "command line" arguments *args* in the directory *dir*. (Suitable choices for *dir* are `DirectoryCurrent()` or `DirectoryTemporary()` (see Section 9.3); `DirectoryTemporary()` may be a good choice when *executable* generates output files that it doesn't itself remove afterwards.) `InputOutputLocalProcess` returns an InputOutputStream object. Bytes written to this stream are received by the slave process as if typed at a terminal on standard input. Bytes written to standard output by the slave process can be read from the stream.

When the stream if closed, the signal SIGTERM is delivered to the child process, which is expected to exit.

```
gap> d := DirectoryCurrent();
dir("./")
gap> f := Filename(DirectoriesSystemPrograms(), "rev");
"/usr/bin/rev"
gap> s := InputOutputLocalProcess(d,f,[]);
< input/output stream to rev >
gap> WriteLine(s,"The cat sat on the mat");
true
gap> Print(ReadLine(s));
tam eht no tas tac ehT
gap> x := ListWithIdenticalEntries(10000,'x');;
gap> ConvertToStringRep(x);
gap> WriteLine(s,x);
true
gap> WriteByte(s,INT_CHAR('\n'));
true
gap> y := ReadAll(s);;
gap> Length(y);
4095
gap> CloseStream(s);
gap> s;
< closed input/output stream to rev >
```

3 ▶ `ReadAllLine( ` *iostream*`[, ` *nofail*`][, ` *IsAllLine*`] )` O

For an input/output stream *iostream* `ReadAllLine` reads until a newline character if any input is found or returns `fail` if no input is found, i.e. if any input is found `ReadAllLine` is non-blocking.

If the argument *nofail* (which must be `false` or `true`) is provided and it is set to `true` then `ReadAllLine` will wait, if necessary, for input and never return `fail`.

If the argument *IsAllLine* (which must be a function that takes a string argument and returns either `true` or `false`) then it is used to determine what constitutes a whole line. The default behaviour is equivalent to passing the function

```
line -> 0 < Length(line) and line[Length(line)] = '\n'
```

for the *IsAllLine* argument. The purpose of the *IsAllLine* argument is to cater for the case where the input being read is from an external process that writes a "prompt" for data that does not terminate with a newline.

If the first argument is an input stream but not an input/output stream then `ReadAllLine` behaves as if `ReadLine` was called with just the first argument and any additional arguments are ignored.

10.9 Dummy Streams

The following two commands create dummy streams which will consume all characters and never deliver one.

1 ▶ `InputTextNone( )` F

returns a dummy input text stream, which delivers no characters, i.e., it is always at end of stream. Its main use is for calls to `Process` (see 11.1.1) when the started program does not read anything.

2 ▶ `OutputTextNone( )` F

returns a dummy output stream, which discards all received characters. Its main use is for calls to `Process` (see 11.1.1) when the started program does not write anything.

10.10 Handling of Streams in the Background

This section describes a feature of the GAP kernel that can be used to handle pending streams somehow "in the background". This is currently not available on the Macintosh architecture and only on operating systems that have `select`.

Right before GAP reads a keypress from the keyboard it calls a little subroutine that can handle streams that are ready to be read or ready to be written. This means that GAP can handle these streams during user input on the command line. Note that this does not work when GAP is in the middle of some calculation.

This feature is used in the following way. One can install handler functions for reading or writing streams. This is done via:

1 ▶ `InstallCharReadHookFunc( ` *stream*`, ` *mode*`, ` *func* `)` F

installs the function *func* as a handler function for the stream *stream*. The argument *mode* decides, for what operations on the stream this function is installed. *mode* must be a string, in which a letter `r` means "read", `w` means "write" and `x` means "exception", according to the `select` function call in the UNIX C-library (see `man select` and 10.2.3). More than one letter is allowed in *mode*. As described above the function is called in a situation when GAP is reading a character from the keyboard. Handler functions should not use much time to complete.

This functionality does not work on the Macintosh architecture and only works if the operating system has a `select` function.

Handlers can be removed via:

2► `UnInstallCharReadHookFunc(` *stream*`,` *func* `)` F

uninstalls the function *func* as a handler function for the stream *stream*. All instances are deinstalled, regardless of the mode of operation (read, write, exception).

This functionality does not work on the Macintosh architecture and only works if the operating system has a `select` function.

Note that handler functions must not return anything and get one integer argument, which refers to an index in one of the following arrays (according to whether the function was installed for input, output or exceptions on the stream). Handler functions usually should not output anything on the standard output because this ruins the command line during command line editing.

11 Processes

GAP can call other programs, such programs are called **processes**. There are two kinds of processes: First there are processes that are started, run and return a result, while GAP is suspended until the process terminates. Then there are processes that will run in parallel to GAP as subprocesses and GAP can communicate and control the processes using streams (see 10.8.2).

11.1 Process

1 ▶ `Process(` *dir*, *prg*, *stream-in*, *stream-out*, *options* `)` O

`Process` runs a new process and returns when the process terminates. It returns the return value of the process if the operating system supports such a concept.

The first argument *dir* is a directory object (see 9.3) which will be the current directory (in the usual UNIX or MSDOS sense) when the program is run. This will only matter if the program accesses files (including running other programs) via relative path names. In particular, it has nothing to do with finding the binary to run.

In general the directory will either be the current directory, which is returned by `DirectoryCurrent` (see 9.3.3) –this was the behaviour of GAP 3– or a temporary directory returned by `DirectoryTemporary` (see 9.3.2). If one expects that the process creates temporary or log files the latter should be used because GAP will attempt to remove these directories together with all the files in them when quitting.

If a program of a GAP package which does not only consist of GAP code needs to be launched in a directory relative to certain data libraries, then the first entry of `DirectoriesPackageLibrary` should be used. The argument of `DirectoriesPackageLibrary` should be the path to the data library relative to the package directory.

If a program calls other programs and needs to be launched in a directory containing the executables for such a GAP package then the first entry of `DirectoriesPackagePrograms` should be used.

The latter two alternatives should only be used if absolutely necessary because otherwise one risks accumulating log or core files in the package directory.

Examples

```
gap> path := DirectoriesSystemPrograms();;
gap> ls := Filename( path, "ls" );;
gap> stdin := InputTextUser();;
gap> stdout := OutputTextUser();;
gap> Process( path[1], ls, stdin, stdout, ["-c"] );;
awk    ls     mkdir

gap> # current directory, here the root directory
gap> Process( DirectoryCurrent(), ls, stdin, stdout, ["-c"] );;
bin    lib    trans  tst    CVS    grp    prim   thr    two
src    dev    etc    tbl    doc    pkg    small  tom
```

```
gap> # create a temporary directory
gap> tmpdir := DirectoryTemporary();;
gap> Process( tmpdir, ls, stdin, stdout, ["-c"] );;
gap> PrintTo( Filename( tmpdir, "emil" ) );
gap> Process( tmpdir, ls, stdin, stdout, ["-c"] );;
emil
```

prg is the filename of the program to launch, for portability it should be the result of `Filename` (see 9.4.1) and should pass `IsExecutableFile`. Note that `Process` does **no** searching through a list of directories, this is done by `Filename`.

stream-in is the input stream that delivers the characters to the process. For portability it should either be `InputTextNone` (if the process reads no characters), `InputTextUser`, the result of a call to `InputTextFile` from which no characters have been read, or the result of a call to `InputTextString`.

`Process` is free to consume **all** the input even if the program itself does not require any input at all.

stream-out is the output stream which receives the characters from the process. For portability it should either be `OutputTextNone` (if the process writes no characters), `OutputTextUser`, the result of a call to `OutputTextFile` to which no characters have been written, or the result of a call to `OutputTextString`.

options is a list of strings which are passed to the process as command line argument. Note that no substitutions are performed on the strings, i.e., they are passed immediately to the process and are not processed by a command interpreter (shell). Further note that each string is passed as one argument, even if it contains *space* characters. Note that input/output redirection commands are **not** allowed as *options*.

Examples

In order to find a system program use `DirectoriesSystemPrograms` together with `Filename`.

```
gap> path := DirectoriesSystemPrograms();;
gap> date := Filename( path, "date" );
"/bin/date"
```

Now execute `date` with no argument and no input, collect the output into a string stream.

```
gap> str := "";; a := OutputTextString(str,true);;
gap> Process( DirectoryCurrent(), date, InputTextNone(), a, [] );
0
gap> CloseStream(a);
gap> Print(str);
Fri Jul 11 09:04:23 MET DST 1997
```

11.2 Exec

1 ▶ `Exec(` *cmd*`,` *option1*`, ..., `*optionN*` )` F

`Exec` runs a shell in the current directory to execute the command given by the string *cmd* with options *option1*`, ..., `*optionN*.

```
gap> Exec( "date" );
Thu Jul 24 10:04:13 BST 1997
```

cmd is interpreted by the shell and therefore we can make use of the various features that a shell offers as in following example.

```
gap> Exec( "echo \"GAP is great!\" > foo" );
gap> Exec( "cat foo" );
GAP is great!
gap> Exec( "rm foo" );
```

`Exec` calls the more general operation `Process` (see 11.1.1). `Edit` (see 6.9.1) should be used to call an editor from within GAP.

12 Objects and Elements

An **object** is anything in GAP that can be assigned to a variable, so nearly everything in GAP is an object.

Different objects can be regarded as equal with respect to the equivalence relation `'='`, in this case we say that the objects describe the same **element**.

12.1 Objects

Nearly all things one deals with in GAP are **objects**. For example, an integer is an object, as is a list of integers, a matrix, a permutation, a function, a list of functions, a record, a group, a coset or a conjugacy class in a group.

Examples of things that are not objects are comments which are only lexical constructs, `while` loops which are only syntactical constructs, and expressions, such as `1 + 1`; but note that the value of an expression, in this case the integer `2`, is an object.

Objects can be assigned to variables, and everything that can be assigned to a variable is an object. Analogously, objects can be used as arguments of functions, and can be returned by functions.

1 ▶ `IsObject(` *obj* `)` C

`IsObject` returns `true` if the object *obj* is an object. Obviously it can never return `false`.

It can be used as a filter in `InstallMethod` (see 2.2 in "Programming in GAP") when one of the arguments can be anything.

12.2 Elements as equivalence classes

The equality operation "`=`" defines an equivalence relation on all GAP objects. The equivalence classes are called **elements**.

There are basically three reasons to regard different objects as equal. Firstly the same information may be stored in different places. Secondly the same information may be stored in different ways; for example, a polynomial can be stored sparsely or densely. Thirdly different information may be equal modulo a mathematical equivalence relation. For example, in a finitely presented group with the relation $a^2 = 1$ the different objects a and a^3 describe the same element.

As an example of all three reasons, consider the possibility of storing an integer in several places of the memory, of representing it as a fraction with denominator 1, or of representing it as a fraction with any denominator, and numerator a suitable multiple of the denominator.

12.3 Sets

In GAP there is no category whose definition corresponds to the mathematical property of being a set, however in the manual we will often refer to an object as a **set** in order to convey the fact that mathematically, we are thinking of it as a set. In particular, two sets A and B are equal if and only if, $x \in A \iff x \in B$.

There are two types of object in GAP which exhibit this kind of behaviour with respect to equality, namely domains (see Section 12.4) and lists whose elements are strictly sorted see `IsSSortedList` (see 21.17.4). In general, **set** in this manual will mean an object of one of these types.

More precisely: two domains can be compared with "`=`", the answer being `true` if and only if the sets of elements are equal (regardless of any additional structure) and; a domain and a list can be compared with "`=`", the answer being `true` if and only if the list is equal to the strictly sorted list of elements of the domain.

A discussion about sorted lists and sets can be found in the Reference Manual section "Sorted Lists and Sets" 21.19.

12.4 Domains

An especially important class of objects in GAP are those whose underlying mathematical abstraction is that of a structured set, for example a group, a conjugacy class, or a vector space. Such objects are called **domains**. The equality relation between domains is always equality **as sets**, so that two domains are equal if and only if they contain the same elements.

Domains play a central role in GAP. In a sense, the only reason that GAP supports objects such as integers and permutations is the wish to form domains of them and compute the properties of those domains.

Domains are described in Chapter 30.

12.5 Identical Objects

Two objects that are equal **as objects** (that is they actually refer to the same area of computer memory) and not only w.r.t. the equality relation '`=`' are called **identical**. Identical objects do of course describe the same element.

1 ▶ `IsIdenticalObj(` *obj1*`,` *obj2* `)` F

`IsIdenticalObj(` *obj1*`,` *obj2* `)` tests whether the objects *obj1* and *obj2* are identical (that is they are either equal immediate objects or are both stored at the same location in memory.

If two copies of a simple constant object (see section 12.6) are created, it is not defined whether GAP will actually store two equal but non-identical objects, or just a single object. For mutable objects, however, it is important to know whether two value refer to identical or non-identical objects, and the documentation of operations that return mutable values should make clear whether the values returned are new, or may be identical to values stored elsewhere.

```
gap> IsIdenticalObj( 10^6, 10^6);
true
gap> IsIdenticalObj( 10^12, 10^12);
false
gap> IsIdenticalObj( true, true);
true
```

Generally, one may compute with objects but think of the results in terms of the underlying elements because one is not interested in locations in memory, data formats or information beyond underlying equivalence relations. But there are cases where it is important to distinguish the relations identity and equality. This

is best illustrated with an example. (The reader who is not familiar with lists in GAP, in particular element access and assignment, is referred to Chapter 21.)

```
gap> l1:= [ 1, 2, 3 ];; l2:= [ 1, 2, 3 ];;
gap> l1 = l2;
true
gap> IsIdenticalObj( l1, l2 );
false
gap> l1[3]:= 4;; l1; l2;
[ 1, 2, 4 ]
[ 1, 2, 3 ]
gap> l1 = l2;
false
```

The two lists `l1` and `l2` are equal but not identical. Thus a change in `l1` does not affect `l2`.

```
gap> l1:= [ 1, 2, 3 ];; l2:= l1;;
gap> l1 = l2;
true
gap> IsIdenticalObj( l1, l2 );
true
gap> l1[3]:= 4;; l1; l2;
[ 1, 2, 4 ]
[ 1, 2, 4 ]
gap> l1 = l2;
true
```

Here, `l1` and `l2` are identical objects, so changing `l1` means a change to `l2` as well.

The library also provides:

2 ▶ `IsNotIdenticalObj( `*obj1*`, `*obj2*` )` F

tests whether the objects *obj1* and *objs2* are not identical.

12.6 Mutability and Copyability

An object in GAP is said to be **immutable** if its mathematical value (as defined by =) does not change under any operation. More explicitly, suppose a is immutable and O is some operation on a, then if $a = b$ evaluates to `true` before executing $O(a)$, $a = b$ also evaluates to `true` afterwards. (Examples for operations O that change mutable objects are `Add` and `Unbind` which are used to change list objects, see Chapter 21.) An immutable object **may** change, for example to store new information, or to adopt a more efficient representation, but this does not affect its behaviour under =.

There are two points here to note. Firstly, "operation" above refers to the functions and methods which can legitimately be applied to the object, and not the `!.` operation whereby virtually any aspect of any GAP level object may be changed. The second point which follows from this, is that when implementing new types of objects, it is the programmer's responsibility to ensure that the functions and methods they write never change immutable objects mathematically.

In fact, most objects with which one deals in GAP are immutable. For instance, the permutation `(1,2)` will never become a different permutation or a non-permutation (although a variable which previously had `(1,2)` stored in it may subsequently have some other value).

For many purposes, however, **mutable** objects are useful. These objects may be changed to represent different mathematical objects during their life. For example, mutable lists can be changed by assigning values to positions or by unbinding values at certain positions. Similarly, one can assign values to components of a mutable record, or unbind them.

1 ► `IsCopyable( obj )` C

If a mutable form of an object *obj* can be made in GAP, the object is called **copyable**. Examples of copyable objects are of course lists and records. A new mutable version of the object can always be obtained by the operation `ShallowCopy` (see 12.7).

Objects for which only an immutable form exists in GAP are called **constants**. Examples of constants are integers, permutations, and domains. Called with a constant as argument, `Immutable` and `ShallowCopy` return this argument.

2 ► `IsMutable( obj )` C

tests whether *obj* is mutable.

If an object is mutable then it is also copyable (see 12.6.1), and a `ShallowCopy` (see 12.7.1) method should be supplied for it. Note that `IsMutable` must not be implied by another filter, since otherwise `Immutable` would be able to create paradoxical objects in the sense that `IsMutable` for such an object is `false` but the filter that implies `IsMutable` is `true`.

In many situations, however, one wants to ensure that objects are **immutable**. For example, take the identity of a matrix group. Since this matrix may be referred to as the identity of the group in several places, it would be fatal to modify its entries, or add or unbind rows. We can obtain an immutable copy of an object with:

3 ► `Immutable( obj )` O

returns an immutable structural copy (see 12.7.2) of *obj* in which the subobjects are immutable **copies** of the subobjects of *obj*. If *obj* is immutable then `Immutable` returns *obj* itself.

GAP will complain with an error if one tries to change an immutable object.

4 ► `MakeImmutable( obj )` F

One can turn the (mutable or immutable) object *obj* into an immutable one with `MakeImmutable`; note that this also makes all subobjects of *obj* immutable, so one should call `MakeImmutable` only if *obj* and its mutable subobjects are newly created. If one is not sure about this, `Immutable` should be used.

Note that it is **not** possible to turn an immutable object into a mutable one; only mutable copies can be made (see 12.7).

Using `Immutable`, it is possible to store an immutable identity matrix or an immutable list of generators, and to pass around references to this immutable object safely. Only when a mutable copy is really needed does the actual object have to be duplicated. Compared to the situation without immutable objects, much unnecessary copying is avoided this way. Another advantage of immutability is that lists of immutable objects may remember whether they are sorted (see 21.19), which is not possible for lists of mutable objects.

Since the operation `Immutable` must work for any object in GAP, it follows that an immutable form of every object must be possible, even if it is not sensible, and user-defined objects must allow for the possibility of becoming immutable without notice.

Another interesting example of mutable (and thus copyable) objects is provided by **iterators**, see 28.7. (Of course an immutable form of an iterator is not very useful, but clearly `Immutable` will yield such an object.) Every call of `NextIterator` changes a mutable iterator until it is exhausted, and this is the only way to change an iterator. `ShallowCopy` for an iterator *iter* is defined so as to return a mutable iterator that has no mutable data in common with *iter*, and that behaves equally to *iter* w.r.t. `IsDoneIterator` and (if *iter* is mutable) `NextIterator`. Note that this meaning of the "shallow copy" of an iterator that is returned by `ShallowCopy` is not as obvious as for lists and records, and must be explicitly defined.

Many operations return immutable results, among those in particular attributes (see 13.5). Examples of attributes are `Size`, `Zero`, `AdditiveInverse`, `One`, and `Inverse`. Arithmetic operations, such as the binary infix operations `+`, `-`, `*`, `/`, `^`, `mod`, the unary `-`, and operations such as `Comm` and `LeftQuotient`, return

mutable results, **except** if all arguments are immutable. So the product of two matrices or of a vector and a matrix is immutable if and only if the two matrices or both the vector and the matrix are immutable (see also 21.11). There is one exception to this rule, which arises where the result is less deeply nested that at least one of the argument, where mutable arguments may sometimes lead to an immutable result. For instance, a mutable matrix with immutable rows, multiplied by an immutable vector gives an immutable vector result. The exact rules are given in 21.11.

It should be noted that `0 *` *obj* is equivalent to `ZeroSM(` *obj* `)`, `-`*obj* is equivalent to `AdditiveInverseSM(` *obj* `)`, *obj*`^0` is equivalent to `OneSM(` *obj*`)`, and *obj*`^-1` is equivalent to `InverseSM(` *obj* `)`. The "SM" stands for "same mutability", and indicates that the result is mutable if and only if the argument is mutable.

The operations `ZeroOp`, `AdditiveInverseOp`, `OneOp`, and `InverseOp` return **mutable** results whenever a mutable version of the result exists, contrary to the attributes `Zero`, `AdditiveInverse`, `One`, and `Inverse`.

If one introduces new arithmetic objects then one need not install methods for the attributes `One`, `Zero`, etc. The methods for the associated operations `OneOp` and `ZeroOp` will be called, and then the results made immutable.

All methods installed for the arithmetic operations must obey the rule about the mutability of the result. This means that one may try to avoid the perhaps expensive creation of a new object if both operands are immutable, and of course no problems of this kind arise at all in the (usual) case that the objects in question do not admit a mutable form, i.e., that these objects are not copyable.

In a few, relatively low-level algorithms, one wishes to treat a matrix partly as a data structure, and manipulate and change its entries. For this, the matrix needs to be mutable, and the rule that attribute values are immutable is an obstacle. For these situations, a number of additional operations are provided, for example `TransposedMatMutable` constructs a mutable matrix (contrary to the attribute `TransposedMat`), while `TriangulizeMat` modifies a mutable matrix (in place) into upper triangular form.

Note that being immutable does not forbid an object to store knowledge. For example, if it is found out that an immutable list is strictly sorted then the list may store this information. More precisely, an immutable object may change in any way, provided that it continues to represent the same mathematical object.

12.7 Duplication of Objects

1 ▶ `ShallowCopy(` *obj* `)` O

If GAP supports a mutable form of the object *obj* (see 12.6) then this is obtained by `ShallowCopy`. Otherwise `ShallowCopy` returns *obj* itself.

The subobjects of `ShallowCopy(` *obj* `)` are **identical** to the subobjects of *obj*. Note that if the object returned by `ShallowCopy` is mutable then it is always a **new** object. In particular, if the return value is mutable, then it is not **identical** with the argument *obj*, no matter whether *obj* is mutable or immutable. But of course the object returned by `ShallowCopy` is **equal** to *obj* w.r.t. the equality operator `=`.

Since `ShallowCopy` is an operation, the concrete meaning of "subobject" depends on the type of *obj*. But for any copyable object *obj*, the definition should reflect the idea of "first level copying".

The definition of `ShallowCopy` for lists (in particular for matrices) can be found in 21.7.

2 ▶ `StructuralCopy(` *obj* `)` F

In a few situations, one wants to make a **structural copy** *scp* of an object *obj*. This is defined as follows. *scp* and *obj* are identical if *obj* is immutable. Otherwise, *scp* is a mutable copy of *obj* such that each subobject of *scp* is a structural copy of the corresponding subobject of *obj*. Furthermore, if two subobjects of *obj* are identical then also the corresponding subobjects of *scp* are identical.

```
gap> obj:= [ [ 0, 1 ] ];;
gap> obj[2]:= obj[1];;
gap> obj[3]:= Immutable( obj[1] );;
gap> scp:= StructuralCopy( obj );;
gap> scp = obj; IsIdenticalObj( scp, obj );
true
false
gap> IsIdenticalObj( scp[1], obj[1] );
false
gap> IsIdenticalObj( scp[3], obj[3] );
true
gap> IsIdenticalObj( scp[1], scp[2] );
true
```

That both `ShallowCopy` and `StructuralCopy` return the argument *obj* itself if it is not copyable is consistent with this definition, since there is no way to change *obj* by modifying the result of any of the two functions, because in fact there is no way to change this result at all.

12.8 Other Operations Applicable to any Object

There are a number of general operations which can be applied, in principle, to any object in GAP. Some of these are documented elsewhere – see 26.5.1, 6.2.3 and 6.2.4. Others are mainly somewhat technical.

1 ► `SetName( obj, name )` F

for a suitable object *obj* sets that object to have name *name* (a string).

2 ► `Name( obj )` A

returns the name, a string, previously assigned to *obj* via a call to `SetName` (see 12.8.1). The name of an object is used **only** for viewing the object via this name.

There are no methods installed for computing names of objects, but the name may be set for suitable objects, using `SetName`.

```
gap> g := Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> SetName(g, "S3");
gap> g;
S3
gap> Name(g);
"S3"
```

3 ► `IsInternallyConsistent( obj )` O

For debugging purposes, it may be useful to check the consistency of an object *obj* that is composed from other (composed) objects.

There is a default method of `IsInternallyConsistent`, with rank zero, that returns `true`. So it is possible (and recommended) to check the consistency of subobjects of *obj* recursively by `IsInternallyConsistent`.

(Note that `IsInternallyConsistent` is not an attribute.)

13 Types of Objects

Every GAP object has a **type**. The type of an object is the information which is used to decide whether an operation is admissible or possible with that object as an argument, and if so, how it is to be performed (see Chapter 2 in "Programming in GAP").

For example, the types determine whether two objects can be multiplied and what function is called to compute the product. Analogously, the type of an object determines whether and how the size of the object can be computed. It is sometimes useful in discussing the type system, to identify types with the set of objects that have this type. Partial types can then also be regarded as sets, such that any type is the intersection of its parts.

The type of an object consists of two main parts, which describe different aspects of the object.

The **family** determines the relation of the object to other objects. For example, all permutations form a family. Another family consists of all collections of permutations, this family contains the set of permutation groups as a subset. A third family consists of all rational functions with coefficients in a certain family.

The other part of a type is a collection of **filters** (actually stored as a bit-list indicating, from the complete set of possible filters, which are included in this particular type). These filters are all treated equally by the method selection, but, from the viewpoint of their creation and use, they can be divided (with a small number of unimportant exceptions) into categories, representations, attribute testers and properties. Each of these is described in more detail below.

This chapter does not describe how types and their constituent parts can be created. Information about this topic can be found in "Programming in GAP" in Section 3.)

Note: Detailed understanding of the type system is not required to use GAP. It can be helpful, however, to understand how things work and why GAP behaves the way it does.

A discussion of the type system can be found in [BL98].

13.1 Families

The family of an object determines its relationship to other objects.

More precisely, the families form a partition of all GAP objects such that the following two conditions hold: objects that are equal w.r.t. `'='` lie in the same family; and the family of the result of an operation depends only on the families of its operands.

The first condition means that a family can be regarded as a set of elements instead of a set of objects. Note that this does not hold for categories and representations (see below), two objects that are equal w.r.t. `'='` need not lie in the same categories and representations. For example, a sparsely represented matrix can be equal to a densely represented matrix. Similarly, each domain is equal w.r.t. `'='` to the sorted list of its elements, but a domain is not a list, and a list is not a domain.

1 ▶ `FamilyObj( `*obj*` )` F

returns the family of the object *obj*.

The family of the object *obj* is itself an object, its family is the `FamilyOfFamilies`.

It should be emphasized that families may be created when they are needed. For example, the family of elements of a finitely presented group is created only after the presentation has been constructed. Thus families are the dynamic part of the type system, that is, the part that is not fixed after the initialisation of GAP.

Families can be parametrized. For example, the elements of each finitely presented group form a family of their own. Here the family of elements and the finitely presented group coincide when viewed as sets. Note that elements in different finitely presented groups lie in different families. This distinction allows GAP to forbid multiplications of elements in different finitely presented groups.

As a special case, families can be parametrized by other families. An important example is the family of **collections** that can be formed for each family. A collection consists of objects that lie in the same family, it is either a nonempty dense list of objects from the same family or a domain.

Note that every domain is a collection, that is, it is not possible to construct domains whose elements lie in different families. For example, a polynomial ring over the rationals cannot contain the integer `0` because the family that contains the integers does not contain polynomials. So one has to distinguish the integer zero from each zero polynomial.

Let us look at this example from a different viewpoint. A polynomial ring and its coefficients ring lie in different families, hence the coefficients ring cannot be embedded "naturally" into the polynomial ring in the sense that it is a subset. But it is possible to allow, e.g., the multiplication of an integer and a polynomial over the integers. The relation between the arguments, namely that one is a coefficient and the other a polynomial, can be detected from the relation of their families. Moreover, this analysis is easier than in a situation where the rationals would lie in one family together with all polynomials over the rationals, because then the relation of families would not distinguish the multiplication of two polynomials, the multiplication of two coefficients, and the multiplication of a coefficient with a polynomial. So the wish to describe relations between elements can be taken as a motivation for the introduction of families.

13.2 Filters

A **filter** is a special unary GAP function that returns either `true` or `false`, depending on whether or not the argument lies in the set defined by the filter. Filters are used to express different aspects of information about a GAP object, which are described below (see 13.3, 13.4, 13.5, 13.6, 13.7, 13.8).

Presently any filter in GAP is implemented as a function which corresponds to a set of positions in the bitlist which forms part of the type of each GAP object, and returns `true` if and only if the bitlist of the type of the argument has the value `true` at all of these positions.

The intersection (or meet) of two filters *filt1*, *filt2* is again a filter, it can be formed as

▶ *filt1* `and` *filt2*

See 20.3.3 for more details.

For example, `IsList and IsEmpty` is a filter that returns `true` if its argument is an empty list, and `false` otherwise. The filter `IsGroup` is defined as the intersection of the category `IsMagmaWithInverses` and the property `IsAssociative`.

A filter that is not the meet of other filters is called a **simple filter**. For example, each attribute tester (see 13.6) is a simple filter. Each simple filter corresponds to a position in the bitlist currently used as part of the data structure representing a type.

Every filter *filt* has a **rank**, which is used to define a ranking of the methods installed for an operation, see Section 2.2 in "Programming in GAP". The rank of a filter can be accessed as

1 ▶ `RankFilter(` *filt* `)` F

For simple filters, an **incremental rank** is defined when the filter is created, see the sections about the creation of filters 3.1, 3.2, 3.3, 3.4; all in "Programming in GAP". For an arbitrary filter, its rank is given by

the sum of the incremental ranks of the **involved** simple filters; in addition to the implied filters, these are also the required filters of attributes (again see the sections about the creation of filters). In other words, for the purpose of computing the rank and **only** for this purpose, attribute testers are treated as if they would imply the requirements of their attributes.

2 ▶ `NamesFilter( filt )` F

`NamesFilter` returns a list of names of the **implied** simple filters of the filter *filt*, these are all those simple filters *imp* such that every object in *filt* also lies in *imp*. For implications between filters, see 13.2.3 as well as sections 2.7, 3.1, 3.2, 3.3 in "Programming in GAP"

3 ▶ `ShowImpliedFilters( filter )` F

Displays information about the filters that may be implied by *filter*. They are given by their names. `ShowImpliedFilters` first displays the names of all filters that are unconditionally implied by *filter*. It then displays implications that require further filters to be present (indicating by `+` the required further filters). The function displays only first-level implications, implications that follow in turn are not displayed (though GAP will do these).

```
gap> ShowImpliedFilters(IsMatrix);
Implies:
   IsGeneralizedRowVector
   IsNearAdditiveElementWithInverse
   IsAdditiveElement
   IsMultiplicativeElement


May imply with:
+IsGF2MatrixRep
   IsOrdinaryMatrix

+CategoryCollections(CategoryCollections(IsAdditivelyCommutativeElement))
   IsAdditivelyCommutativeElement

+IsInternalRep
   IsOrdinaryMatrix
```

13.3 Categories

The **categories** of an object are filters (see 13.2) determine what operations an object admits. For example, all integers form a category, all rationals form a category, and all rational functions form a category. An object which claims to lie in a certain category is accepting the requirement that it should have methods for certain operations (and perhaps that their behaviour should satisfy certain axioms). For example, an object lying in the category `IsList` must have methods for `Length`, `IsBound\[\]` and the list element access operation `\[\]`.

An object can lie in several categories. For example, a row vector lies in the categories `IsList` and `IsVector`; each list lies in the category `IsCopyable`, and depending on whether or not it is mutable, it may lie in the category `IsMutable`. Every domain lies in the category `IsDomain`.

Of course some categories of a mutable object may change when the object is changed. For example, after assigning values to positions of a mutable non-dense list, this list may become part of the category `IsDenseList`.

However, if an object is immutable then the set of categories it lies in is fixed.

All categories in the library are created during initialization, in particular they are not created dynamically at runtime.

The following list gives an overview of important categories of arithmetic objects. Indented categories are to be understood as subcategories of the non indented category listed above it.

```
IsObject
    IsExtLElement
    IsExtRElement
        IsMultiplicativeElement
            IsMultiplicativeElementWithOne
                IsMultiplicativeElementWithInverse
    IsExtAElement
        IsAdditiveElement
            IsAdditiveElementWithZero
                IsAdditiveElementWithInverse
```

Every object lies in the category `IsObject`.

The categories `IsExtLElement` and `IsExtRElement` contain objects that can be multiplied with other objects via `*` from the left and from the right, respectively. These categories are required for the operands of the operation `*`.

The category `IsMultiplicativeElement` contains objects that can be multiplied from the left and from the right with objects from the same family. `IsMultiplicativeElementWithOne` contains objects *obj* for which a multiplicatively neutral element can be obtained by taking the zeroth power *obj*`^0`. `IsMultiplicativeElementWithInverse` contains objects *obj* for which a multiplicative inverse can be obtained by forming *obj*`^-1`.

Likewise, the categories `IsExtAElement`, `IsAdditiveElement`, `IsAdditiveElementWithZero`, and `IsAdditiveElementWithInverse` contain objects that can be added via `+` to other objects, objects that can be added to objects of the same family, objects for which an additively neutral element can be obtained by multiplication with zero, and objects for which an additive inverse can be obtained by multiplication with `-1`.

So a vector lies in `IsExtLElement`, `IsExtRElement`, and `IsAdditiveElementWithInverse`. A ring element must additionally lie in `IsMultiplicativeElement`.

As stated above it is not guaranteed by the categories of objects whether the result of an operation with these objects as arguments is defined. For example, the category `IsMatrix` is a subcategory of `IsMultiplicativeElementWithInverse`. Clearly not every matrix has a multiplicative inverse. But the category `IsMatrix` makes each matrix an admissible argument of the operation `Inverse`, which may sometimes return 'fail'. Likewise, two matrices can be multiplied only if they are of appropriate shapes.

Analogous to the categories of arithmetic elements, there are categories of domains of these elements.

```
IsObject
    IsDomain
        IsMagma
            IsMagmaWithOne
                IsMagmaWithInversesIfNonzero
                    IsMagmaWithInverses
        IsAdditiveMagma
            IsAdditiveMagmaWithZero
                IsAdditiveMagmaWithInverses
        IsExtLSet
        IsExtRSet
```

Of course `IsDomain` is a subcategory of `IsObject`. A domain that is closed under multiplication `*` is called a magma and it lies in the category `IsMagma`. If a magma is closed under taking the identity, it lies in

IsMagmaWithOne, and if it is also closed under taking inverses, it lies in IsMagmaWithInverses. The category IsMagmaWithInversesIfNonzero denotes closure under taking inverses only for nonzero elements, every division ring lies in this category.

Note that every set of categories constitutes its own notion of generation, for example a group may be generated as a magma with inverses by some elements, but to generate it as a magma with one it may be necessary to take the union of these generators and their inverses.

1 ► CategoriesOfObject(*object*) O

returns a list of the names of the categories in which *object* lies.

```
gap> g:=Group((1,2),(1,2,3));;
gap> CategoriesOfObject(g);
[ "IsListOrCollection", "IsCollection", "IsExtLElement",
  "CategoryCollections(IsExtLElement)", "IsExtRElement",
  "CategoryCollections(IsExtRElement)",
  "CategoryCollections(IsMultiplicativeElement)",
  "CategoryCollections(IsMultiplicativeElementWithOne)",
  "CategoryCollections(IsMultiplicativeElementWithInverse)",
  "CategoryCollections(IsAssociativeElement)",
  "CategoryCollections(IsFiniteOrderElement)", "IsGeneralizedDomain",
  "CategoryCollections(IS_PERM)", "IsMagma", "IsMagmaWithOne",
  "IsMagmaWithInversesIfNonzero", "IsMagmaWithInverses" ]
```

13.4 Representation

The **representation** of an object is a set of filters (see 13.2) that determines how an object is actually represented. For example, a matrix or a polynomial can be stored sparsely or densely; all dense polynomials form a representation. An object which claims to lie in a certain representation is accepting the requirement that certain fields in the data structure be present and have specified meanings.

GAP distinguishes four essentially different ways to represent objects. First there are the representations IsInternalRep for internal objects such as integers and permutations, and IsDataObjectRep for other objects that are created and whose data are accessible only by kernel functions. The data structures underlying such objects cannot be manipulated at the GAP level.

All other objects are either in the representation IsComponentObjectRep or in the representation IsPositionalObjectRep, see 3.9 and 3.10 in "Programming in GAP".

An object can belong to several representations in the sense that it lies in several subrepresentations of IsComponentObjectRep or of IsPositionalObjectRep. The representations to which an object belongs should form a chain and either two representations are disjoint or one is contained in the other. So the subrepresentations of IsComponentObjectRep and IsPositionalObjectRep each form trees. In the language of Object Oriented Programming, we support only single inheritance for representations.

These trees are typically rather shallow, since for one representation to be contained in another implies that all the components of the data structure implied by the containing representation, are present in, and have the same meaning in, the smaller representation (whose data structure presumably contains some additional components).

Objects may change their representation, for example a mutable list of characters can be converted into a string.

All representations in the library are created during initialization, in particular they are not created dynamically at runtime.

Examples of subrepresentations of `IsPositionalObjectRep` are `IsModulusRep`, which is used for residue classes in the ring of integers, and `IsDenseCoeffVectorRep`, which is used for elements of algebras that are defined by structure constants.

An important subrepresentation of `IsComponentObjectRep` is `IsAttributeStoringRep`, which is used for many domains and some other objects. It provides automatic storing of all attribute values (see below).

1 ▶ `RepresentationsOfObject(` *object* `)` O

returns a list of the names of the representations *object* has.

```
gap> g:=Group((1,2),(1,2,3));;
gap> RepresentationsOfObject(g);
[ "IsComponentObjectRep", "IsAttributeStoringRep" ]
```

13.5 Attributes

The attributes of an object are filters (see 13.2) that describe knowledge about it.

An attribute is a unary operation without side-effects.

An object may store values of its attributes once they have been computed, and claim that it knows these values. Note that "store" and "know" have to be understood in the sense that it is very cheap to get such a value when the attribute is called again.

The stored value of an attribute is in general immutable (see 12.6), except if the attribute had been specally constructed as "mutable attribute".

It depends on the representation of an object (see 13.4) which attribute values it stores. An object in the representation `IsAttributeStoringRep` stores **all** attribute values once they are computed. Moreover, for an object in this representation, subsequent calls to an attribute will return the **same** object; this is achieved via a special method for each attribute setter that stores the attribute value in an object in `IsAttributeStoringRep`, and a special method for the attribute itself that fetches the stored attribute value. (These methods are called the "system setter" and the "system getter" of the attribute, respectively.)

Note also that it is impossible to get rid of a stored attribute value because the system may have drawn conclusions from the old attribute value, and just removing the value might leave the data structures in an inconsistent state. If necessary, a new object can be constructed.

Properties are a special form of attributes that have the value `true` or `false`, see section 13.7.

All attributes in the library are created during initialization, in particular they are not created dynamically at runtime.

Examples of attributes for multiplicative elements are `Inverse`, `One`, and `Order`. `Size` is an attribute for domains, `Centre` is an attribute for magmas, and `DerivedSubgroup` is an attribute for groups.

1 ▶ `KnownAttributesOfObject(` *object* `)` O

returns a list of the names of the attributes whose values are known for *object*.

```
gap> g:=Group((1,2),(1,2,3));;Size(g);;
gap> KnownAttributesOfObject(g);
[ "Size", "OneImmutable", "NrMovedPoints", "MovedPoints",
  "GeneratorsOfMagmaWithInverses", "MultiplicativeNeutralElement", "Pcgs",
  "GeneralizedPcgs", "StabChainMutable", "StabChainOptions" ]
```

Several attributes have methods for more than one argument. For example `IsTransitive` (see 39.9.1) is an attribute for a G-set that can also be called for the two arguments, being a group G and its operation domain. If attributes are called with more than one argument then the return value is not stored in any of the arguments.

13.6 Setter and Tester for Attributes

For every attribute two further operations, the **attribute setter** and the **attribute tester** are defined.

To check whether an object belongs to an attribute *attr*, the tester

1 ▶ `Tester( attr )` O

of the attribute is used; this is a filter (see 13.2) that returns `true` or `false`, depending on whether or not the value of *attr* for the object is known. For example, `Tester( Size )( obj )` is `true` if the size of the object *obj* is known.

To store a value for the attribute *attr* in an object, the setter

2 ▶ `Setter( attr )` O

of the attribute is used. The setter is called automatically when the attribute value has been computed for the first time. One can also call the setter explicitly, for example, `Setter( Size )( obj, val )` sets *val* as size of the object *obj* if the size was not yet known.

For each attribute *attr* that is declared with `DeclareAttribute` resp. `DeclareProperty` (see 3.17 in "Programming in GAP"), tester and setter are automatically made accessible by the names `Has`*attr* and `Set`*attr*, respectively. For example, the tester for `Size` is called `HasSize`, and the setter is called `SetSize`.

```
gap> g:=Group((1,2,3,4),(1,2));;Size(g);
24
gap> HasSize(g);
true
gap> SetSize(g,99);
gap> Size(g);
24
```

For two properties *prop1* and *prop2*, the intersection *prop1* `and` *prop2* (see 13.2) is again a property for which a setter and a tester exist. Setting the value of this intersection to `true` for a GAP object means to set the values of *prop1* and *prop2* to `true` for this object.

```
gap> prop:= IsFinite and IsCommutative;
<Operation "<<and-filter>>">
gap> g:= Group( (1,2,3), (4,5) );;
gap> Tester( prop )( g );
false
gap> Setter( prop )( g, true );
gap> Tester( prop )( g );  prop( g );
true
true
```

It is **not allowed** to set the value of such an intersection to `false` for an object.

```
gap> Setter( prop )( Rationals, false );
You cannot set an "and-filter" except to true
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can type 'return true;' to set all components true
(but you might really want to reset just one component) to continue
brk>
```

3 ▶ `AttributeValueNotSet( attr, obj )` F

If the value of the attribute *attr* is already stored for *obj*, `AttributeValueNotSet` simply returns this value. Otherwise the value of *attr*`( obj )` is computed and returned **without storing it** in *obj*. This can be useful

when "large" attribute values (such as element lists) are needed only once and shall not be stored in the object.

```
gap> HasAsSSortedList(g);
false
gap> AttributeValueNotSet(AsSSortedList,g);
[ (), (4,5), (1,2,3), (1,2,3)(4,5), (1,3,2), (1,3,2)(4,5) ]
gap> HasAsSSortedList(g);
false
```

The normal behaviour of attributes (when called with just one argument) is that once a method has been selected and executed, and has returned a value the setter of the attribute is called, to (possibly) store the computed value. In special circumstances, this behaviour can be altered dynamically on an attribute-by-attribute basis, using the functions `DisableAttributeValueStoring` and `EnableAttributeValueStoring`.

In general, the code in the library assumes, for efficiency, but not for correctness, that attribute values **will** be stored (in suitable objects), so disabling storing may cause substantial computations to be repeated.

4 ▶ `InfoAttributes` V

This info class (together with `InfoWarning`; see 7.4.6) is used for messages about attribute storing being disabled (at level 2) or enabled (level 3). It may be used in the future for other messages concerning changes to attribute behaviour.

5 ▶ `DisableAttributeValueStoring(` *attr* `)` F

disables the usual call of `Setter(` *attr* `)` when a method for *attr* returns a value. In consequence the values will never be stored. Note that *attr* must be an attribute and **not** a property.

6 ▶ `EnableAttributeValueStoring(` *attr* `)` F

enables the usual call of `Setter(` *attr* `)` when a method for *attr* returns a value. In consequence the values may be stored. This will usually have no effect unless `DisableAttributeValueStoring` has previously been used for *attr*. Note that *attr* must be an attribute and **not** a property.

```
gap> g := Group((1,2,3,4,5),(1,2,3));
Group([ (1,2,3,4,5), (1,2,3) ])
gap> KnownAttributesOfObject(g);
[ "LargestMovedPoint", "GeneratorsOfMagmaWithInverses",
  "MultiplicativeNeutralElement" ]
gap> SetInfoLevel(InfoAttributes,3);
gap> DisableAttributeValueStoring(Size);
#I  Disabling value storing for Size
gap> Size(g);
60
gap> KnownAttributesOfObject(g);
[ "OneImmutable", "LargestMovedPoint", "NrMovedPoints", "MovedPoints",
  "GeneratorsOfMagmaWithInverses", "MultiplicativeNeutralElement",
  "StabChainMutable", "StabChainOptions" ]
gap> Size(g);
60
gap> EnableAttributeValueStoring(Size);
#I  Enabling value storing for Size
gap> Size(g);
60
gap> KnownAttributesOfObject(g);
```

```
[ "Size", "OneImmutable", "LargestMovedPoint", "NrMovedPoints",
  "MovedPoints", "GeneratorsOfMagmaWithInverses",
  "MultiplicativeNeutralElement", "StabChainMutable", "StabChainOptions" ]
```

13.7 Properties

The properties of an object are those of its attributes (see 13.5) whose values can only be `true` or `false`.

The main difference between attributes and properties is that a property defines two sets of objects, namely the usual set of all objects for which the value is known, and the set of all objects for which the value is known to be `true`.

(Note that it makes no sense to consider a third set, namely the set of objects for which the value of a property is `true` whether or not it is known, since there may be objects for which the containment in this set cannot be decided.)

For a property *prop*, the containment of an object *obj* in the first set is checked again by applying `Tester(` *prop* `)` to *obj*, and *obj* lies in the second set if and only if `Tester(` *prop* `)(` *obj* `)` `and` *prop*`(` *obj* `)` is `true`.

If a property value is known for an immutable object then this value is also stored, as part of the type of the object. To some extent, property values of mutable objects also can be stored, for example a mutable list all of whose entries are immutable can store whether it is strictly sorted. When the object is mutated (for example by list assignment) the type may need to be adjusted.

Important properties for domains are `IsAssociative`, `IsCommutative`, `IsAnticommutative`, `IsLDistributive`, and `IsRDistributive`, which mean that the multiplication of elements in the domain satisfies $(a*b)*c = a*(b*c)$, $a*b = b*a$, $a*b = -(b*a)$, $a*(b+c) = a*b + a*c$, and $(a+b)*c = a*c + b*c$, respectively, for all a, b, c in the domain.

1 ▶ `KnownPropertiesOfObject(` *object* `)` O

returns a list of the names of the properties whose values are known for *object*.

2 ▶ `KnownTruePropertiesOfObject(` *object* `)` O

returns a list of the names of the properties known to be `true` for *object*.

```
gap> g:=Group((1,2),(1,2,3));;
gap> KnownPropertiesOfObject(g);
[ "IsFinite", "CanEasilyCompareElements", "CanEasilySortElements",
  "IsDuplicateFree", "IsGeneratorsOfMagmaWithInverses", "IsAssociative",
  "IsSimpleSemigroup", "IsFinitelyGeneratedGroup",
  "IsSubsetLocallyFiniteGroup", "KnowsHowToDecompose", "IsChainTypeGroup",
  "IsStabChainViaChainSubgroup" ]
gap> Size(g);
6
gap> KnownPropertiesOfObject(g);
[ "IsEmpty", "IsTrivial", "IsNonTrivial", "IsFinite",
  "CanEasilyCompareElements", "CanEasilySortElements", "IsDuplicateFree",
  "IsGeneratorsOfMagmaWithInverses", "IsAssociative", "IsSimpleSemigroup",
  "IsFinitelyGeneratedGroup", "IsSubsetLocallyFiniteGroup",
  "KnowsHowToDecompose", "IsPerfectGroup", "IsSolvableGroup",
  "IsPolycyclicGroup", "IsChainTypeGroup", "IsStabChainViaChainSubgroup" ]
gap> KnownTruePropertiesOfObject(g);
[ "IsNonTrivial", "IsFinite", "CanEasilyCompareElements",
  "CanEasilySortElements", "IsDuplicateFree",
```

```
"IsGeneratorsOfMagmaWithInverses", "IsAssociative", "IsSimpleSemigroup",
"IsFinitelyGeneratedGroup", "IsSubsetLocallyFiniteGroup",
"KnowsHowToDecompose", "IsSolvableGroup", "IsPolycyclicGroup" ]
```

13.8 Other Filters

There are situations where one wants to express a kind of knowledge that is based on some heuristic.

For example, the filters (see 13.2) `CanEasilyTestMembership` and `CanEasilyComputePcgs` are defined in the GAP library. Note that such filters do not correspond to a mathematical concept, contrary to properties (see 13.7). Also it need not be defined what "easily" means for an arbitrary GAP object, and in this case one cannot compute the value for an arbitrary GAP object. In order to access this kind of knowledge as a part of the type of an object, GAP provides filters for which the value is `false` by default, and it is changed to `true` in certain situations, either explicitly (for the given object) or via a logical implication (see 2.7 in "Programming in GAP") from other filters.

For example, a `true` value of `CanEasilyComputePcgs` for a group means that certain methods are applicable that use a pcgs (see 43.1) for the group. There are logical implications to set the filter value to `true` for permutation groups that are known to be solvable, and for groups that have already a (sufficiently nice) pcgs stored. In the case one has a solvable matrix group and wants to enable methods that use a pcgs, one can set the `CanEasilyComputePcgs` value to `true` for this particular group.

A filter *filt* of the kind described here is different from the other filters introduced in the previous sections. In particular, *filt* is not a category (see 13.3) or a property (see 13.7) because its value may change for a given object, and *filt* is not a representation (see 13.4) because it has nothing to do with the way an object is made up from some data. *filt* is similar to an attribute tester (see 13.6), the only difference is that *filt* does not refer to an attribute value; note that *filt* is also used in the same way as an attribute tester; namely, the `true` value may be required for certain methods to be applicable.

13.9 Types

We stated above (see 13) that, for an object *obj*, its **type** is formed from its family and its filters. There is a also a third component, used in a few situations, namely defining data of the type.

1 ▶ `TypeObj(` *obj* `)` F

returns the type of the object *obj*.

The type of an object is itself an object.

Two types are equal if and only if the two families are identical, the filters are equal, and, if present, also the defining data of the types are equal.

The last part of the type, defining data, has not been mentioned before and seems to be of minor importance. It can be used, e.g., for cosets Ug of a group U, where the type of each coset may contain the group U as defining data. As a consequence, two such cosets mod U and V can have the same type only if $U = V$. The defining data of the type *type* can be accessed as

2 ▶ `DataType(` *type* `)` F

14 Integers

One of the most fundamental datatypes in every programming language is the integer type. GAP is no exception.

GAP integers are entered as a sequence of decimal digits optionally preceded by a + sign for positive integers or a - sign for negative integers. The size of integers in GAP is only limited by the amount of available memory, so you can compute with integers having thousands of digits.

```
gap> -1234;
-1234
gap> 123456789012345678901234567890123456789012345678901234567890;
123456789012345678901234567890123456789012345678901234567890
```

Many more functions that are mainly related to the prime residue group of integers modulo an integer are described in chapter 15, and functions dealing with combinatorics can be found in chapter 17.

1 ▶ `Integers` V
▶ `PositiveIntegers` V
▶ `NonnegativeIntegers` V

These global variables represent the ring of integers and the semirings of positive and nonnegative integers, respectively.

```
gap> Size( Integers ); 2 in Integers;
infinity
true
```

2 ▶ `IsIntegers(` *obj* `)` C
▶ `IsPositiveIntegers(` *obj* `)` C
▶ `IsNonnegativeIntegers(` *obj* `)` C

are the defining categories for the domains `Integers`, `PositiveIntegers`, and `NonnegativeIntegers`.

```
gap> IsIntegers( Integers );  IsIntegers( Rationals );  IsIntegers( 7 );
true
false
false
```

`Integers` is a subset of `Rationals`, which is a subset of `Cyclotomics`. See Chapter 18 for arithmetic operations and comparison of integers.

14.1 Elementary Operations for Integers

1 ► `IsInt( obj )` C

Every rational integer lies in the category `IsInt`, which is a subcategory of `IsRat` (see 16).

2 ► `IsPosInt( obj )` C

Every positive integer lies in the category `IsPosInt`.

3 ► `Int( elm )` A

`Int` returns an integer *int* whose meaning depends on the type of *elm*.

If *elm* is a rational number (see 16) then *int* is the integer part of the quotient of numerator and denominator of *elm* (see 14.2.1).

If *elm* is an element of a finite prime field (see Chapter 57) then *int* is the smallest nonnegative integer such that *elm* `=` *int* `*` `One(` *elm* `)`.

If *elm* is a string (see Chapter 26) consisting of digits `'0'`, `'1'`, ..., `'9'` and `'-'` (at the first position) then *int* is the integer described by this string. The operation `String` (see 26.5.1) can be used to compute a string for rational integers, in fact for all cyclotomics.

```
gap> Int( 4/3 );  Int( -2/3 );
1
0
gap> int:= Int( Z(5) );  int * One( Z(5) );
2
Z(5)
gap> Int( "12345" );  Int( "-27" );  Int( "-27/3" );
12345
-27
fail
```

4 ► `IsEvenInt( n )` F

tests if the integer n is divisible by 2.

5 ► `IsOddInt( n )` F

tests if the integer n is not divisible by 2.

6 ► `AbsInt( n )` F

`AbsInt` returns the absolute value of the integer n, i.e., n if n is positive, -n if n is negative and 0 if n is 0.

`AbsInt` is a special case of the general operation `EuclideanDegree` see 54.6.2).

See also 18.1.6.

```
gap> AbsInt( 33 );
33
gap> AbsInt( -214378 );
214378
gap> AbsInt( 0 );
0
```

7 ► `SignInt( n )` F

`SignInt` returns the sign of the integer n, i.e., 1 if n is positive, -1 if n is negative and 0 if n is 0.

```
gap> SignInt( 33 );
```

```
1
gap> SignInt( -214378 );
-1
gap> SignInt( 0 );
0
```

8 ▶ `LogInt( n, base )` F

`LogInt` returns the integer part of the logarithm of the positive integer n with respect to the positive integer *base*, i.e., the largest positive integer *exp* such that $base^{exp} \leq n$. `LogInt` will signal an error if either n or *base* is not positive.

For *base* 2 this is very efficient because the internal binary representation of the integer is used.

```
gap> LogInt( 1030, 2 );
10
gap> 2^10;
1024
gap> LogInt( 1, 10 );
0
```

9 ▶ `RootInt( n )` F
▶ `RootInt( n, k )` F

`RootInt` returns the integer part of the kth root of the integer n. If the optional integer argument k is not given it defaults to 2, i.e., `RootInt` returns the integer part of the square root in this case.

If n is positive, `RootInt` returns the largest positive integer r such that $r^k \leq n$. If n is negative and k is odd `RootInt` returns `-RootInt( -`n`, `k `)`. If n is negative and k is even `RootInt` will cause an error. `RootInt` will also cause an error if k is 0 or negative.

```
gap> RootInt( 361 );
19
gap> RootInt( 2 * 10^12 );
1414213
gap> RootInt( 17000, 5 );
7
gap> 7^5;
16807
```

10 ▶ `SmallestRootInt( n )` F

`SmallestRootInt` returns the smallest root of the integer n.

The smallest root of an integer n is the integer r of smallest absolute value for which a positive integer k exists such that $n = r^k$.

```
gap> SmallestRootInt( 2^30 );
2
gap> SmallestRootInt( -(2^30) );
-4
```

Note that $(-2)^{30} = +(2^{30})$.

```
gap> SmallestRootInt( 279936 );
6
gap> LogInt( 279936, 6 );
7
gap> SmallestRootInt( 1001 );
1001
```

11 ▶ `Random( Integers )`

`Random` for integers returns pseudo random integers between -10 and 10 distributed according to a binomial distribution. To generate uniformly distributed integers from a range, use the construct 'Random([*low* .. *high*])'. (Also see 28.6.1.)

14.2 Quotients and Remainders

1 ▶ `QuoInt( n, m )` F

`QuoInt` returns the integer part of the quotient of its integer operands.

If n and m are positive `QuoInt( n, m )` is the largest positive integer q such that $q * m \leq n$. If n or m or both are negative the absolute value of the integer part of the quotient is the quotient of the absolute values of n and m, and the sign of it is the product of the signs of n and m.

`QuoInt` is used in a method for the general operation `EuclideanQuotient` (see 54.6.3).

```
gap> QuoInt(5,3);  QuoInt(-5,3);  QuoInt(5,-3);  QuoInt(-5,-3);
1
-1
-1
1
```

2 ▶ `BestQuoInt( n, m )` F

`BestQuoInt` returns the best quotient q of the integers n and m. This is the quotient such that $n-q*m$ has minimal absolute value. If there are two quotients whose remainders have the same absolute value, then the quotient with the smaller absolute value is chosen.

```
gap> BestQuoInt( 5, 3 );  BestQuoInt( -5, 3 );
2
-2
```

3 ▶ `RemInt( n, m )` F

`RemInt` returns the remainder of its two integer operands.

If m is not equal to zero `RemInt( n, m ) = n - m * QuoInt( n, m )`. Note that the rules given for `QuoInt` imply that `RemInt( n, m )` has the same sign as n and its absolute value is strictly less than the absolute value of m. Note also that `RemInt( n, m ) = n mod m` when both n and m are nonnegative. Dividing by 0 signals an error.

`RemInt` is used in a method for the general operation `EuclideanRemainder` (see 54.6.4).

```
gap> RemInt(5,3);  RemInt(-5,3);  RemInt(5,-3);  RemInt(-5,-3);
2
-2
2
-2
```

4 ▶ `GcdInt( m, n )` F

`GcdInt` returns the greatest common divisor of its two integer operands m and n, i.e., the greatest integer that divides both m and n. The greatest common divisor is never negative, even if the arguments are. We define `GcdInt( m, 0 ) = GcdInt( 0, m ) = AbsInt( m )` and `GcdInt( 0, 0 ) = 0`.

`GcdInt` is a method used by the general function `Gcd` (see 54.7.1).

```
gap> GcdInt( 123, 66 );
3
```

5 ► `Gcdex(` *m*, *n* `)` F

returns a record *g* describing the extended greatest common divisor of *m* and *n*. The component `gcd` is this gcd, the components `coeff1` and `coeff2` are integer cofactors such that *g*`.gcd = `*g*`.coeff1 * `*m*` + `*g*`.coeff2 * `*n*, and the components `coeff3` and `coeff4` are integer cofactors such that `0 = `*g*`.coeff3 * `*m*` + `*g*`.coeff4 * `*n*.

If *m* and *n* both are nonzero, `AbsInt(` *g*`.coeff1 )` is less than or equal to `AbsInt(`*n*`) / (2 * `*g*`.gcd)` and `AbsInt(` *g*`.coeff2 )` is less than or equal to `AbsInt(`*m*`) / (2 * `*g*`.gcd)`.

If *m* or *n* or both are zero `coeff3` is `-`*n*` / `*g*`.gcd` and `coeff4` is *m*` / `*g*`.gcd`.

The coefficients always form a unimodular matrix, i.e., the determinant *g*`.coeff1 * `*g*`.coeff4 - `*g*`.coeff3 * `*g*`.coeff2` is 1 or −1.

```
gap> Gcdex( 123, 66 );
rec( gcd := 3, coeff1 := 7, coeff2 := -13, coeff3 := -22, coeff4 := 41 )
```

This means $3 = 7 * 123 - 13 * 66$, $0 = -22 * 123 + 41 * 66$.

```
gap> Gcdex( 0, -3 );
rec( gcd := 3, coeff1 := 0, coeff2 := -1, coeff3 := 1, coeff4 := 0 )
gap> Gcdex( 0, 0 );
rec( gcd := 0, coeff1 := 1, coeff2 := 0, coeff3 := 0, coeff4 := 1 )
```

6 ► `LcmInt(` *m*, *n* `)` F

returns the least common multiple of the integers *m* and *n*.

`LcmInt` is a method used by the general function `Lcm`.

```
gap> LcmInt( 123, 66 );
2706
```

7 ► `CoefficientsQadic(` *i*, *q* `)` F

returns the *q*-adic representation of the integer *i* as a list *l* of coefficients where $i = \sum_{j=0} q^j \cdot l[j+1]$.

8 ► `CoefficientsMultiadic(` *ints*, *int* `)` F

returns the multiadic expansion of the integer *int* modulo the integers given in *ints* (in ascending order). It returns a list of coefficients in the **reverse** order to that in *ints*.

9 ► `ChineseRem(` *moduli*, *residues* `)` F

`ChineseRem` returns the combination of the *residues* modulo the *moduli*, i.e., the unique integer *c* from `[0..Lcm(`*moduli*`)-1]` such that *c* `= `*residues*`[i]` modulo *moduli*`[i]` for all *i*, if it exists. If no such combination exists `ChineseRem` signals an error.

Such a combination does exist if and only if *residues*[*i*]=*residues*[*k*] mod Gcd(*moduli*[*i*],*moduli*[*k*]) for every pair i, k. Note that this implies that such a combination exists if the moduli are pairwise relatively prime. This is called the Chinese remainder theorem.

```
gap> ChineseRem( [ 2, 3, 5, 7 ], [ 1, 2, 3, 4 ] );
53
gap> ChineseRem( [ 6, 10, 14 ], [ 1, 3, 5 ] );
103

gap> ChineseRem( [ 6, 10, 14 ], [ 1, 2, 3 ] );
Error, the residues must be equal modulo 2 called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> gap>
```

10 ▶ PowerModInt(r, e, m) F

returns $r^e \pmod m$ for integers r,e and m ($e \geq 0$). Note that using r ^ e mod m will generally be slower, because it can not reduce intermediate results the way PowerModInt does but would compute r^e first and then reduce the result afterwards.

PowerModInt is a method for the general operation PowerMod.

14.3 Prime Integers and Factorization

1 ▶ Primes V

Primes is a strictly sorted list of the 168 primes less than 1000.

This is used in IsPrimeInt and FactorsInt to cast out small primes quickly.

```
gap> Primes[1];
2
gap> Primes[100];
541
```

2 ▶ IsPrimeInt(n) F
▶ IsProbablyPrimeInt(n) F

IsPrimeInt returns false if it can prove that n is composite and true otherwise. By convention IsPrimeInt(0) = IsPrimeInt(1) = false and we define IsPrimeInt(-n) = IsPrimeInt(n).

IsPrimeInt will return true for every prime n. IsPrimeInt will return false for all composite $n < 10^{13}$ and for all composite n that have a factor $p < 1000$. So for integers $n < 10^{13}$, IsPrimeInt is a proper primality test. It is conceivable that IsPrimeInt may return true for some composite $n > 10^{13}$, but no such n is currently known. So for integers $n > 10^{13}$, IsPrimeInt is a probable-primality test. IsPrimeInt will issue a warning when its argument is probably prime but not a proven prime. (The function IsProbablyPrimeInt will do the same calculations but not issue a warning.) The warning can be switched off by SetInfoLevel(InfoPrimeInt, 0);, the default level is 1.

If composites that fool IsPrimeInt do exist, they would be extremely rare, and finding one by pure chance might be less likely than finding a bug in GAP. We would appreciate being informed about any example of a composite number n for which IsPrimeInt returns true.

IsPrimeInt is a deterministic algorithm, i.e., the computations involve no random numbers, and repeated calls will always return the same result. IsPrimeInt first does trial divisions by the primes less than 1000.

Then it tests that n is a strong pseudoprime w.r.t. the base 2. Finally it tests whether n is a Lucas pseudoprime w.r.t. the smallest quadratic nonresidue of n. A better description can be found in the comment in the library file `integer.gi`.

The time taken by `IsPrimeInt` is approximately proportional to the third power of the number of digits of n. Testing numbers with several hundreds digits is quite feasible.

`IsPrimeInt` is a method for the general operation `IsPrime`.

Remark: In future versions of GAP we hope to change the definition of `IsPrimeInt` to return `true` only for proven primes (currently, we lack a sufficiently good primality proving function). In applications, use explicitly `IsPrimeInt` or `IsProbablePrimeInt` with this change in mind.

```
gap> IsPrimeInt( 2^31 - 1 );
true
gap> IsPrimeInt( 10^42 + 1 );
false
```

3 ▶ `IsPrimePowerInt( n )` F

`IsPrimePowerInt` returns `true` if the integer n is a prime power and `false` otherwise.

n is a **prime power** if there exists a prime p and a positive integer i such that $p^i = n$. If n is negative the condition is that there must exist a negative prime p and an odd positive integer i such that $p^i = n$. 1 and -1 are not prime powers.

Note that `IsPrimePowerInt` uses `SmallestRootInt` (see 14.1.10) and a probable-primality test (see 14.3.2).

```
gap> IsPrimePowerInt( 31^5 );
true
gap> IsPrimePowerInt( 2^31-1 );  # 2^31-1 is actually a prime
true
gap> IsPrimePowerInt( 2^63-1 );
false
gap> Filtered( [-10..10], IsPrimePowerInt );
[ -8, -7, -5, -3, -2, 2, 3, 4, 5, 7, 8, 9 ]
```

4 ▶ `NextPrimeInt( n )` F

`NextPrimeInt` returns the smallest prime which is strictly larger than the integer n.

Note that `NextPrimeInt` uses a probable-primality test (see 14.3.2).

```
gap> NextPrimeInt( 541 ); NextPrimeInt( -1 );
547
2
```

5 ▶ `PrevPrimeInt( n )` F

`PrevPrimeInt` returns the largest prime which is strictly smaller than the integer n.

Note that `PrevPrimeInt` uses a probable-primality test (see 14.3.2).

```
gap> PrevPrimeInt( 541 ); PrevPrimeInt( 1 );
523
-2
```

6 ▶ `FactorsInt( n )` F
▶ `FactorsInt( n : RhoTrials := trials )` F

`FactorsInt` returns a list of prime factors of the integer n.

If the ith power of a prime divides n this prime appears i times. The list is sorted, that is the smallest prime factors come first. The first element has the same sign as n, the others are positive. For any integer n it holds that `Product( FactorsInt(` n `) ) =` n.

Note that `FactorsInt` uses a probable-primality test (see 14.3.2). Thus `FactorsInt` might return a list which contains composite integers. In such a case you will get a warning about the use of a probable prime. You can switch off these warnings by `SetInfoLevel(InfoPrimeInt, 0);`.

The time taken by `FactorsInt` is approximately proportional to the square root of the second largest prime factor of n, which is the last one that `FactorsInt` has to find, since the largest factor is simply what remains when all others have been removed. Thus the time is roughly bounded by the fourth root of n. `FactorsInt` is guaranteed to find all factors less than 10^6 and will find most factors less than 10^{10}. If n contains multiple factors larger than that `FactorsInt` may not be able to factor n and will then signal an error.

`FactorsInt` is used in a method for the general operation `Factors`.

In the second form, FactorsInt calls FactorsRho with a limit of *trials* on the number of trials is performs. The default is 8192.

```
gap> FactorsInt( -Factorial(6) );
[ -2, 2, 2, 2, 3, 3, 5 ]
gap> Set( FactorsInt( Factorial(13)/11 ) );
[ 2, 3, 5, 7, 13 ]
gap> FactorsInt( 2^63 - 1 );
[ 7, 7, 73, 127, 337, 92737, 649657 ]
gap> FactorsInt( 10^42 + 1 );
#I  IsPrimeInt: probably prime, but not proven: 4458192223320340849
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
```

7 ▶ `PrintFactorsInt(` n `)` F

prints the prime factorization of the integer n in human-readable form.

```
gap> PrintFactorsInt( Factorial( 7 ) ); Print( "\n" );
2^4*3^2*5*7
```

8 ▶ `PrimePowersInt(` n `)` F

returns the prime factorization of the integer n as a list $[p_1, e_1, \ldots, p_n, e_n]$ with $n = \prod_{i=1}^{n} p_i^{e_i}$.

```
gap> PrimePowersInt( Factorial( 7 ) );
[ 2, 4, 3, 2, 5, 1, 7, 1 ]
```

9 ▶ `DivisorsInt(` n `)` F

`DivisorsInt` returns a list of all divisors of the integer n. The list is sorted, so that it starts with 1 and ends with n. We define that `Divisors( -`n `) = Divisors(` n `)`.

Since the set of divisors of 0 is infinite calling `DivisorsInt( 0 )` causes an error.

`DivisorsInt` may call `FactorsInt` to obtain the prime factors. `Sigma` and `Tau` (see 15.4.1 and 15.4.2) compute the sum and the number of positive divisors, respectively.

```
gap> DivisorsInt( 1 ); DivisorsInt( 20 ); DivisorsInt( 541 );
[ 1 ]
[ 1, 2, 4, 5, 10, 20 ]
[ 1, 541 ]
```

14.4 Residue Class Rings

1 ▶ *r* `/` *s* `mod` *n*

If *r*, *s* and *n* are integers, *r* `/` *s* as a reduced fraction is *p* `/` *q*, and *q* and *n* are coprime, then *r* `/` *s* `mod` *n* is defined to be the product of *p* and the inverse of *q* modulo *n*. See Section 4.12 for more details and definitions.

With the above definition, `4 / 6 mod 32` equals `2 / 3 mod 32` and hence exists (and is equal to 22), despite the fact that 6 has no inverse modulo 32.

2 ▶ `ZmodnZ(` *n* `)` F
▶ `ZmodpZ(` *p* `)` F
▶ `ZmodpZNC(` *p* `)` F

`ZmodnZ` returns a ring *R* isomorphic to the residue class ring of the integers modulo the positive integer *n*. The element corresponding to the residue class of the integer *i* in this ring can be obtained by $i * \texttt{One}(R)$, and a representative of the residue class corresponding to the element $x \in R$ can be computed by $\texttt{Int}(x)$.

`ZmodnZ(` *n* `)` is equivalent to `Integers mod` *n*.

`ZmodpZ` does the same if the argument *p* is a prime integer, additionally the result is a field. `ZmodpZNC` omits the check whether *p* is a prime.

Each ring returned by these functions contains the whole family of its elements if *n* is not a prime, and is embedded into the family of finite field elements of characteristic *n* if *n* is a prime.

3 ▶ `ZmodnZObj(` *Fam*`,` *r* `)` O
▶ `ZmodnZObj(` *r*`,` *n* `)` O

If the first argument is a residue class family *Fam* then `ZmodnZObj` returns the element in *Fam* whose coset is represented by the integer *r*. If the two arguments are an integer *r* and a positive integer *n* then `ZmodnZObj` returns the element in `ZmodnZ(` *n* `)` (see 14.4.2) whose coset is represented by the integer *r*.

```
gap> r:= ZmodnZ(15);
(Integers mod 15)
gap> fam:=ElementsFamily(FamilyObj(r));;
gap> a:= ZmodnZObj(fam,9);
ZmodnZObj( 9, 15 )
gap> a+a;
ZmodnZObj( 3, 15 )
gap> Int(a+a);
3
```

4 ▶ `IsZmodnZObj(` *obj* `)` C
▶ `IsZmodnZObjNonprime(` *obj* `)` C
▶ `IsZmodpZObj(` *obj* `)` C
▶ `IsZmodpZObjSmall(` *obj* `)` C
▶ `IsZmodpZObjLarge(` *obj* `)` C

The elements in the rings Z/nZ are in the category `IsZmodnZObj`. If *n* is a prime then the elements are of course also in the category `IsFFE` (see 57.1.1), otherwise they are in `IsZmodnZObjNonprime`. `IsZmodpZObj` is an abbreviation of `IsZmodnZObj and IsFFE`. This category is the disjoint union of `IsZmodpZObjSmall` and `IsZmodpZObjLarge`, the former containing all elements with *n* at most `MAXSIZE_GF_INTERNAL`.

The reasons to distinguish the prime case from the nonprime case are

– that objects in `IsZmodnZObjNonprime` have an external representation (namely the residue in the range $[0, 1, \ldots, n-1]$),

– that the comparison of elements can be defined as comparison of the residues, and

– that the elements lie in a family of type `IsZmodnZObjNonprimeFamily` (note that for prime n, the family must be an `IsFFEFamily`).

The reasons to distinguish the small and the large case are that for small n the elements must be compatible with the internal representation of finite field elements, whereas we are free to define comparison as comparison of residues for large n.

Note that we **cannot** claim that every finite field element of degree 1 is in `IsZmodnZObj`, since finite field elements in internal representation may not know that they lie in the prime field.

The residue class rings are rings, thus all operations for rings (see Chapter 54) apply. See also Chapters 57 and 15.

15 Number Theory

GAP provides a couple of elementary number theoretic functions. Most of these deal with the group of integers coprime to m, called the **prime residue group**. $\phi(m)$ (see 15.1.2) is the order of this group, $\lambda(m)$ (see 15.1.3) the exponent. If and only if m is 2, 4, an odd prime power p^e, or twice an odd prime power $2p^e$, this group is cyclic. In this case the generators of the group, i.e., elements of order $\phi(m)$, are called **primitive roots** (see 15.2.3, 15.2.4).

Note that neither the arguments nor the return values of the functions listed below are groups or group elements in the sense of GAP. The arguments are simply integers.

1 ► `InfoNumtheor` V

`InfoNumtheor` is the info class (see 7.4) for the functions in the number theory chapter.

15.1 Prime Residues

1 ► `PrimeResidues(` m `)` F

`PrimeResidues` returns the set of integers from the range `0..Abs(`m`)-1` that are coprime to the integer m.

`Abs(`m`)` must be less than 2^{28}, otherwise the set would probably be too large anyhow.

```
gap> PrimeResidues( 0 );  PrimeResidues( 1 );  PrimeResidues( 20 );
[  ]
[ 0 ]
[ 1, 3, 7, 9, 11, 13, 17, 19 ]
```

2 ► `Phi(` m `)` F

`Phi` returns the number $\phi(m)$ of positive integers less than the positive integer m that are coprime to m.

Suppose that $m = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k}$. Then $\phi(m)$ is $p_1^{e_1-1}(p_1-1)p_2^{e_2-1}(p_2-1)\cdots p_k^{e_k-1}(p_k-1)$.

```
gap> Phi( 12 );
4
gap> Phi( 2^13-1 );  # this proves that 2^(13)-1 is a prime
8190
gap> Phi( 2^15-1 );
27000
```

3 ► `Lambda(` m `)` F

`Lambda` returns the exponent $\lambda(m)$ of the group of prime residues modulo the integer m.

$\lambda(m)$ is the smallest positive integer l such that for every a relatively prime to m we have $a^l \equiv 1 \pmod{m}$. Fermat's theorem asserts $a^{\phi(m)} \equiv 1 \pmod{m}$; thus $\lambda(m)$ divides $\phi(m)$ (see 15.1.2).

Carmichael's theorem states that λ can be computed as follows: $\lambda(2) = 1$, $\lambda(4) = 2$ and $\lambda(2^e) = 2^{e-2}$ if $3 \le e$, $\lambda(p^e) = (p-1)p^{e-1}$ (i.e. $\phi(m)$) if p is an odd prime and $\lambda(n * m) = \mathtt{Lcm}(\lambda(n), \lambda(m))$ if n, m are coprime.

Composites for which $\lambda(m)$ divides $m-1$ are called Carmichaels. If $6k+1$, $12k+1$ and $18k+1$ are primes their product is such a number. There are only 1547 Carmichaels below 10^{10} but 455052511 primes.

```
gap> Lambda( 10 );
4
gap> Lambda( 30 );
4
gap> Lambda( 561 );  # 561 is the smallest Carmichael number
80
```

4 ▶ `GeneratorsPrimeResidues( n )` F

Let n be a positive integer. `GeneratorsPrimeResidues` returns a description of generators of the group of prime residues modulo n. The return value is a record with components

`primes`:
: a list of the prime factors of n,

`exponents`:
: a list of the exponents of these primes in the factorization of n, and

`generators`:
: a list describing generators of the group of prime residues; for the prime factor 2, either a primitive root or a list of two generators is stored, for each other prime factor of n, a primitive root is stored.

```
gap> GeneratorsPrimeResidues( 1 );
rec( primes := [  ], exponents := [  ], generators := [  ] )
gap> GeneratorsPrimeResidues( 4*3 );
rec( primes := [ 2, 3 ], exponents := [ 2, 1 ], generators := [ 7, 5 ] )
gap> GeneratorsPrimeResidues( 8*9*5 );
rec( primes := [ 2, 3, 5 ], exponents := [ 3, 2, 1 ],
  generators := [ [ 271, 181 ], 281, 217 ] )
```

15.2 Primitive Roots and Discrete Logarithms

1 ▶ `OrderMod( n, m )` F

`OrderMod` returns the multiplicative order of the integer n modulo the positive integer m. If n and m are not coprime the order of n is not defined and `OrderMod` will return 0.

If n and m are relatively prime the multiplicative order of n modulo m is the smallest positive integer i such that $n^i \equiv 1 \pmod{m}$. If the group of prime residues modulo m is cyclic then each element of maximal order is called a primitive root modulo m (see 15.2.4).

`OrderMod` usually spends most of its time factoring m and $\phi(m)$ (see 14.3.6).

```
gap> OrderMod( 2, 7 );
3
gap> OrderMod( 3, 7 );  # 3 is a primitive root modulo 7
6
```

2 ▶ `LogMod( n, r, m )` F
▶ `LogModShanks( n, r, m )` F

computes the discrete r-logarithm of the integer n modulo the integer m. It returns a number l such that $r^l \equiv n \pmod{m}$ if such a number exists. Otherwise `fail` is returned.

`LogModShanks` uses the Baby Step - Giant Step Method of Shanks (see for example section 5.4.1 in [Coh93] and in general requires more memory than a call to `LogMod`.

```
gap> l:= LogMod( 2, 5, 7 );  5^l mod 7 = 2;
4
true
gap> LogMod( 1, 3, 3 );  LogMod( 2, 3, 3 );
0
fail
```

3 ▶ `PrimitiveRootMod( `*m*`[, `*start*`] )` F

`PrimitiveRootMod` returns the smallest primitive root modulo the positive integer *m* and `fail` if no such primitive root exists. If the optional second integer argument *start* is given `PrimitiveRootMod` returns the smallest primitive root that is strictly larger than *start*.

```
gap> PrimitiveRootMod( 409 );      # largest primitive root for a prime less than 2000
21
gap> PrimitiveRootMod( 541, 2 );
10
gap> PrimitiveRootMod( 337, 327 ); # 327 is the largest primitive root mod 337
fail
gap> PrimitiveRootMod( 30 );       # there exists no primitive root modulo 30
fail
```

4 ▶ `IsPrimitiveRootMod( `*r*`, `*m*` )` F

`IsPrimitiveRootMod` returns `true` if the integer *r* is a primitive root modulo the positive integer *m* and `false` otherwise. If *r* is less than 0 or larger than *m* it is replaced by its remainder.

```
gap> IsPrimitiveRootMod( 2, 541 );
true
gap> IsPrimitiveRootMod( -539, 541 );  # same computation as above;
true
gap> IsPrimitiveRootMod( 4, 541 );
false
gap> ForAny( [1..29], r -> IsPrimitiveRootMod( r, 30 ) );
false
gap> # there is no a primitive root modulo 30
```

15.3 Roots Modulo Integers

1 ▶ `Jacobi( `*n*`, `*m*` )` F

`Jacobi` returns the value of the **Jacobi symbol** of the integer *n* modulo the integer *m*.

Suppose that $m = p_1 p_2 \cdots p_k$ is a product of primes, not necessarily distinct. Then for *n* coprime to *m* the Jacobi symbol is defined by $J(n/m) = L(n/p_1)L(n/p_2)\cdots L(n/p_k)$, where $L(n/p)$ is the Legendre symbol (see 15.3.2). By convention $J(n/1) = 1$. If the gcd of *n* and *m* is larger than 1 we define $J(n/m) = 0$.

If *n* is a **quadratic residue** modulo *m*, i.e., if there exists an *r* such that $r^2 \equiv n \pmod{m}$ then $J(n/m) = 1$. However, $J(n/m) = 1$ implies the existence of such an *r* only if *m* is a prime.

`Jacobi` is very efficient, even for large values of n and m, it is about as fast as the Euclidean algorithm (see 54.7.1).

```
gap> Jacobi( 11, 35 );  # 9^2 = 11 mod 35
1
gap> Jacobi( 6, 35 );   # it is -1, thus there is no r such that r^2 = 6 mod 35
-1
gap> Jacobi( 3, 35 );   # it is 1 even though there is no r with r^2 = 3 mod 35
1
```

2 ▶ `Legendre( n, m )` F

`Legendre` returns the value of the **Legendre symbol** of the integer n modulo the positive integer m.

The value of the Legendre symbol $L(n/m)$ is 1 if n is a **quadratic residue** modulo m, i.e., if there exists an integer r such that $r^2 \equiv n \pmod{m}$ and -1 otherwise.

If a root of n exists it can be found by `RootMod` (see 15.3.3).

While the value of the Legendre symbol usually is only defined for m a prime, we have extended the definition to include composite moduli too. The Jacobi symbol (see 15.3.1) is another generalization of the Legendre symbol for composite moduli that is much cheaper to compute, because it does not need the factorization of m (see 14.3.6).

A description of the Jacobi symbol, the Legendre symbol, and related topics can be found in [Bak84].

```
gap> Legendre( 5, 11 );  # 4^2 = 5 mod 11
1
gap> Legendre( 6, 11 );  # it is -1, thus there is no r such that r^2 = 6 mod 11
-1
gap> Legendre( 3, 35 );  # it is -1, thus there is no r such that r^2 = 3 mod 35
-1
```

3 ▶ `RootMod( n[, k], m )` F

`RootMod` computes a kth root of the integer n modulo the positive integer m, i.e., a r such that $r^k \equiv n \pmod{m}$. If no such root exists `RootMod` returns `fail`. If only the arguments n and m are given, the default value for k is 2.

In the current implementation k must be a prime.

A square root of n exists only if `Legendre(n,m) = 1` (see 15.3.2). If m has r different prime factors then there are 2^r different roots of n mod m. It is unspecified which one `RootMod` returns. You can, however, use `RootsMod` (see 15.3.4) to compute the full set of roots.

`RootMod` is efficient even for large values of m, in fact the most time is usually spent factoring m (see 14.3.6).

```
gap> RootMod( 64, 1009 );  # note 'RootMod' does not return 8 in this case but -8;
1001
gap> RootMod( 64, 3, 1009 );
518
gap> RootMod( 64, 5, 1009 );
656
gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ),
>          x -> x mod 1009 );  # set of all square roots of 64 mod 1009
[ 1001, 8 ]
```

4 ▶ `RootsMod( n[, k], m )` F

`RootsMod` computes the set of kth roots of the integer n modulo the positive integer m, i.e., a r such that $r^k \equiv n \pmod{m}$. If only the arguments n and m are given, the default value for k is 2.

In the current implementation k must be a prime.

```
gap> RootsMod( 1, 7*31 );  # the same as 'RootsUnityMod( 7*31 )'
[ 1, 92, 125, 216 ]
gap> RootsMod( 7, 7*31 );
[ 21, 196 ]
gap> RootsMod( 5, 7*31 );
[  ]
gap> RootsMod( 1, 5, 7*31 );
[ 1, 8, 64, 78, 190 ]
```

5 ▶ `RootsUnityMod( [`k`, ] `m` )` F

`RootsUnityMod` returns the set of k-th roots of unity modulo the positive integer m, i.e., the list of all solutions r of $r^k \equiv n \pmod{m}$. If only the argument m is given, the default value for k is 2.

In general there are k^n such roots if the modulus m has n different prime factors p such that $p \equiv 1 \pmod{k}$. If k^2 divides m then there are k^{n+1} such roots; and especially if $k = 2$ and 8 divides m there are 2^{n+2} such roots.

In the current implementation k must be a prime.

```
gap> RootsUnityMod( 7*31 );  RootsUnityMod( 3, 7*31 );
[ 1, 92, 125, 216 ]
[ 1, 25, 32, 36, 67, 149, 156, 191, 211 ]
gap> RootsUnityMod( 5, 7*31 );
[ 1, 8, 64, 78, 190 ]
gap> List( RootMod( 64, 1009 ) * RootsUnityMod( 1009 ),
>          x -> x mod 1009 );  # set of all square roots of 64 mod 1009
[ 1001, 8 ]
```

15.4 Multiplicative Arithmetic Functions

1 ▶ `Sigma( `n` )` F

`Sigma` returns the sum of the positive divisors of the nonzero integer n.

`Sigma` is a multiplicative arithmetic function, i.e., if n and m are relatively prime we have $\sigma(nm) = \sigma(n)\sigma(m)$.

Together with the formula $\sigma(p^e) = (p^{e+1} - 1)/(p - 1)$ this allows us to compute $\sigma(n)$.

Integers n for which $\sigma(n) = 2n$ are called perfect. Even perfect integers are exactly of the form $2^{n-1}(2^n - 1)$ where $2^n - 1$ is prime. Primes of the form $2^n - 1$ are called **Mersenne primes**, the known ones are obtained for $n = 2$, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, and 859433. It is not known whether odd perfect integers exist, however [BC89] show that any such integer must have at least 300 decimal digits.

`Sigma` usually spends most of its time factoring n (see 14.3.6).

```
gap> Sigma( 1 );
1
gap> Sigma( 1009 );  # 1009 is a prime
1010
gap> Sigma( 8128 ) = 2*8128;  # 8128 is a perfect number
true
```

2 ▶ `Tau( `n` )` F

`Tau` returns the number of the positive divisors of the nonzero integer n.

`Tau` is a multiplicative arithmetic function, i.e., if n and m are relative prime we have $\tau(nm) = \tau(n)\tau(m)$. Together with the formula $\tau(p^e) = e + 1$ this allows us to compute $\tau(n)$.

`Tau` usually spends most of its time factoring n (see 14.3.6).

```
gap> Tau( 1 );
1
gap> Tau( 1013 );  # thus 1013 is a prime
2
gap> Tau( 8128 );
14
gap> Tau( 36 );  # the result is odd if and only if the argument is a perfect square
9
```

3 ▶ `MoebiusMu( n )` F

`MoebiusMu` computes the value of Moebius inversion function for the nonzero integer n. This is 0 for integers which are not squarefree, i.e., which are divided by a square r^2. Otherwise it is 1 if n has a even number and -1 if n has an odd number of prime factors.

The importance of μ stems from the so called inversion formula. Suppose $f(n)$ is a multiplicative arithmetic function defined on the positive integers and let $g(n) = \sum_{d|n} f(d)$. Then $f(n) = \sum_{d|n} \mu(d)g(n/d)$. As a special case we have $\phi(n) = \sum_{d|n} \mu(d)n/d$ since $n = \sum_{d|n} \phi(d)$ (see 15.1.2).

`MoebiusMu` usually spends all of its time factoring n (see 14.3.6).

```
gap> MoebiusMu( 60 );  MoebiusMu( 61 );  MoebiusMu( 62 );
0
-1
1
```

15.5 Continued Fractions

1 ▶ `ContinuedFractionExpansionOfRoot( P, n )` F

The first n terms of the continued fraction expansion of the only positive real root of the polynomial P with integer coefficients. The leading coefficient of P must be positive and the value of P at 0 must be negative. If the degree of P is 2 and $n = 0$, the function computes one period of the continued fraction expansion of the root in question. Anything may happen if P has three or more positive real roots.

```
gap> x := Indeterminate(Integers);;
gap> ContinuedFractionExpansionOfRoot(x^2-7,20);
[ 2, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1, 4, 1, 1, 1 ]
gap> ContinuedFractionExpansionOfRoot(x^2-7,0);
[ 2, 1, 1, 1, 4 ]
gap> ContinuedFractionExpansionOfRoot(x^3-2,20);
[ 1, 3, 1, 5, 1, 1, 4, 1, 1, 8, 1, 14, 1, 10, 2, 1, 4, 12, 2, 3 ]
gap> ContinuedFractionExpansionOfRoot(x^5-x-1,50);
[ 1, 5, 1, 42, 1, 3, 24, 2, 2, 1, 16, 1, 11, 1, 1, 2, 31, 1, 12, 5, 1, 7, 11,
  1, 4, 1, 4, 2, 2, 3, 4, 2, 1, 1, 11, 1, 41, 12, 1, 8, 1, 1, 1, 1, 1, 9, 2,
  1, 5, 4 ]
```

2 ▶ `ContinuedFractionApproximationOfRoot( P, n )` F

The nth continued fraction approximation of the only positive real root of the polynomial P with integer coefficients. The leading coefficient of P must be positive and the value of P at 0 must be negative. Anything may happen if P has three or more positive real roots.

```
gap> ContinuedFractionApproximationOfRoot(x^2-2,10);
3363/2378
gap> 3363^2-2*2378^2;
1
gap> z := ContinuedFractionApproximationOfRoot(x^5-x-1,20);
499898783527/428250732317
gap> z^5-z-1;
486192462527432755459620441970617283/
1440424738231984242169735755880570903111698782624263126135 7
```

15.6 Miscellaneous

1 ▶ `TwoSquares( n )` F

`TwoSquares` returns a list of two integers $x \leq y$ such that the sum of the squares of x and y is equal to the nonnegative integer n, i.e., $n = x^2 + y^2$. If no such representation exists `TwoSquares` will return `fail`. `TwoSquares` will return a representation for which the gcd of x and y is as small as possible. It is not specified which representation `TwoSquares` returns, if there is more than one.

Let a be the product of all maximal powers of primes of the form $4k + 3$ dividing n. A representation of n as a sum of two squares exists if and only if a is a perfect square. Let b be the maximal power of 2 dividing n or its half, whichever is a perfect square. Then the minimal possible gcd of x and y is the square root c of ab. The number of different minimal representation with $x \leq y$ is 2^{l-1}, where l is the number of different prime factors of the form $4k + 1$ of n.

The algorithm first finds a square root r of -1 modulo $n/(ab)$, which must exist, and applies the Euclidean algorithm to r and n. The first residues in the sequence that are smaller than $\sqrt{n/(ab)}$ times c are a possible pair x and y.

Better descriptions of the algorithm and related topics can be found in [Wag90] and [Zag90].

```
gap> TwoSquares( 5 );
[ 1, 2 ]
gap> TwoSquares( 11 );  # there is no representation
fail
gap> TwoSquares( 16 );
[ 0, 4 ]
gap> TwoSquares( 45 );  # 3 is the minimal possible gcd because 9 divides 45
[ 3, 6 ]
gap> TwoSquares( 125 );  # it is not [5,10] because their gcd is not minimal
[ 2, 11 ]
gap> TwoSquares( 13*17 );  # [10,11] would be the other possible representation
[ 5, 14 ]
gap> TwoSquares( 848654483879497562821 );  # 848654483879497562821 is prime
#I  IsPrimeInt: probably prime, but not proven: 848654483879497562821
#I  FactorsInt: used the following factor(s) which are probably primes:
#I        848654483879497562821
[ 6305894639, 28440994650 ]
```

16 Rational Numbers

The **rationals** form a very important field. On the one hand it is the quotient field of the integers (see chapter 14). On the other hand it is the prime field of the fields of characteristic zero (see chapter 58).

The former comment suggests the representation actually used. A rational is represented as a pair of integers, called **numerator** and **denominator**. Numerator and denominator are **reduced**, i.e., their greatest common divisor is 1. If the denominator is 1, the rational is in fact an integer and is represented as such. The numerator holds the sign of the rational, thus the denominator is always positive.

Because the underlying integer arithmetic can compute with arbitrary size integers, the rational arithmetic is always exact, even for rationals whose numerators and denominators have thousands of digits.

```
gap> 2/3;
2/3
gap> 66/123;  # numerator and denominator are made relatively prime
22/41
gap> 17/-13;  # the numerator carries the sign;
-17/13
gap> 121/11;  # rationals with denominator 1 (after cancelling) are integers
11
```

1 ▶ `Rationals` V
▶ `IsRationals( obj )` P

`Rationals` is the field $\mathbb{Q}$ of rational integers, as a set of cyclotomic numbers, see Chapter 18 for basic operations, Functions for the field `Rationals` can be found in the chapters 56 and 58.

`IsRationals` returns `true` for a prime field that consists of cyclotomic numbers –for example the GAP object `Rationals`– and `false` for all other GAP objects.

```
gap> Size( Rationals ); 2/3 in Rationals;
infinity
true
```

16.1 Elementary Operations for Rationals

1 ▶ `IsRat( obj )` C

Every rational number lies in the category `IsRat`, which is a subcategory of `IsCyc` (see 18).

```
gap> IsRat( 2/3 );
true
gap> IsRat( 17/-13 );
true
gap> IsRat( 11 );
true
gap> IsRat( IsRat );  # 'IsRat' is a function, not a rational
false
```

2 ▶ `IsPosRat( obj )` C

Every positive rational number lies in the category `IsPosRat`.

3 ▶ `IsNegRat( obj )` C

Every negative rational number lies in the category `IsNegRat`.

4 ▶ `NumeratorRat( rat )` F

`NumeratorRat` returns the numerator of the rational *rat*. Because the numerator holds the sign of the rational it may be any integer. Integers are rationals with denominator 1, thus `NumeratorRat` is the identity function for integers.

```
gap> NumeratorRat( 2/3 );
2
gap> NumeratorRat( 66/123 );  # numerator and denominator are made relatively prime
22
gap> NumeratorRat( 17/-13 );  # the numerator holds the sign of the rational
-17
gap> NumeratorRat( 11 );      # integers are rationals with denominator 1
11
```

5 ▶ `DenominatorRat( rat )` F

`DenominatorRat` returns the denominator of the rational *rat*. Because the numerator holds the sign of the rational the denominator is always a positive integer. Integers are rationals with the denominator 1, thus `DenominatorRat` returns 1 for integers.

```
gap> DenominatorRat( 2/3 );
3
gap> DenominatorRat( 66/123 );  # numerator and denominator are made relatively prime
41
gap> DenominatorRat( 17/-13 );  # the denominator holds the sign of the rational
13
gap> DenominatorRat( 11 );      # integers are rationals with denominator 1
1
```

6 ▶ `Rat( elm )` A

`Rat` returns a rational number *rat* whose meaning depends on the type of *elm*.

If *elm* is a string consisting of digits `'0'`, `'1'`, ..., `'9'` and `'-'` (at the first position), `'/'` and the decimal dot `'.'` then *rat* is the rational described by this string. The operation `String` (see 26.5.1) can be used to compute a string for rational numbers, in fact for all cyclotomics.

```
gap> Rat( "1/2" );  Rat( "35/14" );  Rat( "35/-27" );  Rat( "3.14159" );
1/2
5/2
-35/27
314159/100000
```

7 ▶ `Random( Rationals )`

`Random` for rationals returns pseudo random rationals which are the quotient of two random integers. See the description of `Random` for integers (14.1.11) for details. (Also see 28.6.1.)

17 Combinatorics

This chapter describes the functions that deal with combinatorics. We mainly concentrate on two areas. One is about **selections**, that is the ways one can select elements from a set. The other is about **partitions**, that is the ways one can partition a set into the union of pairwise disjoint subsets.

17.1 Combinatorial Numbers

1 ▶ `Factorial( n )` F

returns the **factorial** $n!$ of the positive integer n, which is defined as the product $1 \cdot 2 \cdot 3 \cdots n$.

$n!$ is the number of permutations of a set of n elements. $1/n!$ is the coefficient of x^n in the formal series e^x, which is the generating function for factorial.

```
gap> List( [0..10], Factorial );
[ 1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800 ]
gap> Factorial( 30 );
265252859812191058636308480000000
```

`PermutationsList` (see 17.2.9) computes the set of all permutations of a list.

2 ▶ `Binomial( n, k )` F

returns the **binomial coefficient** $\binom{n}{k}$ of integers n and k, which is defined as $n!/(k!(n-k)!)$ (see 17.1.1). We define $\binom{0}{0} = 1$, $\binom{n}{k} = 0$ if $k < 0$ or $n < k$, and $\binom{n}{k} = (-1)^k \binom{-n+k-1}{k}$ if $n < 0$, which is consistent with the equivalent definition $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.

$\binom{n}{k}$ is the number of combinations with k elements, i.e., the number of subsets with k elements, of a set with n elements. $\binom{n}{k}$ is the coefficient of the term x^k of the polynomial $(x+1)^n$, which is the generating function for $\binom{n}{\cdot}$, hence the name.

```
gap> List( [0..4], k->Binomial( 4, k ) );  # Knuth calls this the trademark of Binomial
[ 1, 4, 6, 4, 1 ]
gap> List( [0..6], n->List( [0..6], k->Binomial( n, k ) ) );;
gap> PrintArray( last );  # the lower triangle is called Pascal's triangle
[ [  1,  0,  0,  0,  0,  0,  0 ],
  [  1,  1,  0,  0,  0,  0,  0 ],
  [  1,  2,  1,  0,  0,  0,  0 ],
  [  1,  3,  3,  1,  0,  0,  0 ],
  [  1,  4,  6,  4,  1,  0,  0 ],
  [  1,  5, 10, 10,  5,  1,  0 ],
  [  1,  6, 15, 20, 15,  6,  1 ] ]
gap> Binomial( 50, 10 );
10272278170
```

`NrCombinations` (see 17.2.1) is the generalization of `Binomial` for multisets. `Combinations` (see 17.2.1) computes the set of all combinations of a multiset.

3 ▶ `Bell( n )` F

returns the **Bell number** $B(n)$. The Bell numbers are defined by $B(0) = 1$ and the recurrence $B(n+1) = \sum_{k=0}^{n} \binom{n}{k} B(k)$.

$B(n)$ is the number of ways to partition a set of n elements into pairwise disjoint nonempty subsets (see 17.2.13). This implies of course that $B(n) = \sum_{k=0}^{n} S_2(n,k)$ (see 17.1.6). $B(n)/n!$ is the coefficient of x^n in the formal series e^{e^x-1}, which is the generating function for $B(n)$.

```
gap> List( [0..6], n -> Bell( n ) );
[ 1, 1, 2, 5, 15, 52, 203 ]
gap> Bell( 14 );
190899322
```

4 ▶ `Bernoulli( n )` F

returns the n-th **Bernoulli number** B_n, which is defined by $B_0 = 1$ and $B_n = -\sum_{k=0}^{n-1} \binom{n+1}{k} B_k/(n+1)$.

$B_n/n!$ is the coefficient of x^n in the power series of $x/e^x - 1$. Except for $B_1 = -1/2$ the Bernoulli numbers for odd indices are zero.

```
gap> Bernoulli( 4 );
-1/30
gap> Bernoulli( 10 );
5/66
gap> Bernoulli( 12 );  # there is no simple pattern in Bernoulli numbers
-691/2730
gap> Bernoulli( 50 );  # and they grow fairly fast
4950572052410796482124775 25/66
```

5 ▶ `Stirling1( n, k )` F

returns the **Stirling number of the first kind** $S_1(n,k)$ of the integers n and k. Stirling numbers of the first kind are defined by $S_1(0,0) = 1$, $S_1(n,0) = S_1(0,k) = 0$ if $n, k \neq 0$ and the recurrence $S_1(n,k) = (n-1)S_1(n-1,k) + S_1(n-1,k-1)$.

$S_1(n,k)$ is the number of permutations of n points with k cycles. Stirling numbers of the first kind appear as coefficients in the series $n!\binom{x}{n} = \sum_{k=0}^{n} S_1(n,k)x^k$ which is the generating function for Stirling numbers of the first kind. Note the similarity to $x^n = \sum_{k=0}^{n} S_2(n,k)k!\binom{x}{k}$ (see 17.1.6). Also the definition of S_1 implies $S_1(n,k) = S_2(-k,-n)$ if $n, k < 0$. There are many formulae relating Stirling numbers of the first kind to Stirling numbers of the second kind, Bell numbers, and Binomial coefficients.

```
gap> List( [0..4], k -> Stirling1( 4, k ) );  # Knuth calls this the trademark of S_1
[ 0, 6, 11, 6, 1 ]
gap> List( [0..6], n->List( [0..6], k->Stirling1( n, k ) ) );;
gap> # note the similarity with Pascal's triangle for the Binomial numbers
gap> PrintArray( last );
[ [    1,    0,    0,    0,    0,    0,    0 ],
  [    0,    1,    0,    0,    0,    0,    0 ],
  [    0,    1,    1,    0,    0,    0,    0 ],
  [    0,    2,    3,    1,    0,    0,    0 ],
  [    0,    6,   11,    6,    1,    0,    0 ],
  [    0,   24,   50,   35,   10,    1,    0 ],
  [    0,  120,  274,  225,   85,   15,    1 ] ]
gap> Stirling1(50,10);
10162302092636749005904379711930994404340550538050366562736 5376
```

6 ▶ `Stirling2( n, k )` F

returns the **Stirling number of the second kind** $S_2(n,k)$ of the integers n and k. Stirling numbers of the second kind are defined by $S_2(0,0) = 1$, $S_2(n,0) = S_2(0,k) = 0$ if $n,k \neq 0$ and the recurrence $S_2(n,k) = kS_2(n-1,k) + S_2(n-1,k-1)$.

$S_2(n,k)$ is the number of ways to partition a set of n elements into k pairwise disjoint nonempty subsets (see 17.2.13). Stirling numbers of the second kind appear as coefficients in the expansion of $x^n = \sum_{k=0}^{n} S_2(n,k)k!\binom{x}{k}$. Note the similarity to $n!\binom{x}{n} = \sum_{k=0}^{n} S_1(n,k)x^k$ (see 17.1.5). Also the definition of S_2 implies $S_2(n,k) = S_1(-k,-n)$ if $n,k < 0$. There are many formulae relating Stirling numbers of the second kind to Stirling numbers of the first kind, Bell numbers, and Binomial coefficients.

```
gap> List( [0..4], k->Stirling2( 4, k ) );  # Knuth calls this the trademark of S_2
[ 0, 1, 7, 6, 1 ]
gap> List( [0..6], n->List( [0..6], k->Stirling2( n, k ) ) );;
gap> # note the similarity with Pascal's triangle for the Binomial numbers
gap> PrintArray( last );
[ [   1,   0,   0,   0,   0,   0,   0 ],
  [   0,   1,   0,   0,   0,   0,   0 ],
  [   0,   1,   1,   0,   0,   0,   0 ],
  [   0,   1,   3,   1,   0,   0,   0 ],
  [   0,   1,   7,   6,   1,   0,   0 ],
  [   0,   1,  15,  25,  10,   1,   0 ],
  [   0,   1,  31,  90,  65,  15,   1 ] ]
gap> Stirling2( 50, 10 );
2615471651586288129201277739657799378172701l
```

17.2 Combinations, Arrangements and Tuples

1 ▶ `Combinations( mset [, k] )` F

returns the set of all combinations of the multiset *mset* (a list of objects which may contain the same object several times) with k elements; if k is not given it returns all combinations of *mset*.

A **combination** of *mset* is an unordered selection without repetitions and is represented by a sorted sublist of *mset*. If *mset* is a proper set, there are $\binom{|mset|}{k}$ (see 17.1.2) combinations with k elements, and the set of all combinations is just the **powerset** of *mset*, which contains all **subsets** of *mset* and has cardinality $2^{|mset|}$.

2 ▶ `NrCombinations( mset [, k] )` F

returns the number of `Combinations(`*mset*`,`k`)`.

```
gap> Combinations( [1,2,2,3] );
[ [  ], [ 1 ], [ 1, 2 ], [ 1, 2, 2 ], [ 1, 2, 2, 3 ], [ 1, 2, 3 ], [ 1, 3 ],
  [ 2 ], [ 2, 2 ], [ 2, 2, 3 ], [ 2, 3 ], [ 3 ] ]
gap> NrCombinations( [1..52], 5 );  # number of different hands in a game of poker
2598960
```

The function `Arrangements` (see 17.2.3) computes ordered selections without repetitions, `UnorderedTuples` (see 17.2.5) computes unordered selections with repetitions and `Tuples` (see 17.2.7) computes ordered selections with repetitions.

3 ▶ `Arrangements( mset [, k] )` F

returns the set of arrangements of the multiset *mset* that contain k elements. If k is not given it returns all arrangements of *mset*.

An **arrangement** of *mset* is an ordered selection without repetitions and is represented by a list that contains only elements from *mset*, but maybe in a different order. If *mset* is a proper set there are $|mset|!/(|mset|-k)!$ (see 17.1.1) arrangements with k elements.

4 ▶ `NrArrangements( mset [, k] )` F

returns the number of `Arrangements(`*mset*`,`*k*`)`.

As an example of arrangements of a multiset, think of the game Scrabble. Suppose you have the six characters of the word `settle` and you have to make a four letter word. Then the possibilities are given by

```
gap> Arrangements( ["s","e","t","t","l","e"], 4 );
[ [ "e", "e", "l", "s" ], [ "e", "e", "l", "t" ], [ "e", "e", "s", "l" ],
  [ "e", "e", "s", "t" ], [ "e", "e", "t", "l" ], [ "e", "e", "t", "s" ],
  ... 93 more possibilities ...
  [ "t", "t", "l", "s" ], [ "t", "t", "s", "e" ], [ "t", "t", "s", "l" ] ]
```

Can you find the five proper English words, where `lets` does not count? Note that the fact that the list returned by `Arrangements` is a proper set means in this example that the possibilities are listed in the same order as they appear in the dictionary.

```
gap> NrArrangements( ["s","e","t","t","l","e"] );
523
```

The function `Combinations` (see 17.2.1) computes unordered selections without repetitions, `UnorderedTuples` (see 17.2.5) computes unordered selections with repetitions and `Tuples` (see 17.2.7) computes ordered selections with repetitions.

5 ▶ `UnorderedTuples( set, k )` F

returns the set of all unordered tuples of length k of the set *set*.

An **unordered tuple** of length k of *set* is a unordered selection with repetitions of *set* and is represented by a sorted list of length k containing elements from *set*. There are $\binom{|set|+k-1}{k}$ (see 17.1.2) such unordered tuples.

Note that the fact that `UnorderedTuples` returns a set implies that the last index runs fastest. That means the first tuple contains the smallest element from *set* k times, the second tuple contains the smallest element of *set* at all positions except at the last positions, where it contains the second smallest element from *set* and so on.

6 ▶ `NrUnorderedTuples( set, k )` F

returns the number of `UnorderedTuples(`*set*`,`*k*`)`.

As an example for unordered tuples think of a poker-like game played with 5 dice. Then each possible hand corresponds to an unordered five-tuple from the set [1..6]

```
gap> NrUnorderedTuples( [1..6], 5 );
252
gap> UnorderedTuples( [1..6], 5 );
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 1, 2 ], [ 1, 1, 1, 1, 3 ], [ 1, 1, 1, 1, 4 ],
  [ 1, 1, 1, 1, 5 ], [ 1, 1, 1, 1, 6 ], [ 1, 1, 1, 2, 2 ], [ 1, 1, 1, 2, 3 ],
  ... 100 more tuples ...
  [ 1, 3, 5, 5, 6 ], [ 1, 3, 5, 6, 6 ], [ 1, 3, 6, 6, 6 ], [ 1, 4, 4, 4, 4 ],
  ... 100 more tuples ...
  [ 3, 3, 5, 5, 5 ], [ 3, 3, 5, 5, 6 ], [ 3, 3, 5, 6, 6 ], [ 3, 3, 6, 6, 6 ],
  ... 32 more tuples ...
  [ 5, 5, 5, 6, 6 ], [ 5, 5, 6, 6, 6 ], [ 5, 6, 6, 6, 6 ], [ 6, 6, 6, 6, 6 ] ]
```

The function `Combinations` (see 17.2.1) computes unordered selections without repetitions, `Arrangements` (see 17.2.3) computes ordered selections without repetitions and `Tuples` (see 17.2.7) computes ordered selections with repetitions.

7 ▶ `Tuples( `*set*`, `*k*` )` F

returns the set of all ordered tuples of length k of the set *set*.

An **ordered tuple** of length k of *set* is an ordered selection with repetition and is represented by a list of length k containing elements of *set*. There are $|set|^k$ such ordered tuples.

Note that the fact that `Tuples` returns a set implies that the last index runs fastest. That means the first tuple contains the smallest element from *set* k times, the second tuple contains the smallest element of *set* at all positions except at the last positions, where it contains the second smallest element from *set* and so on.

8 ▶ `NrTuples( `*set*`, `*k*` )` F

returns the number of `Tuples(`*set*`,`*k*`)`.

```
gap> Tuples( [1,2,3], 2 );
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 3, 1 ],
  [ 3, 2 ], [ 3, 3 ] ]
gap> NrTuples( [1..10], 5 );
100000
```

`Tuples(`*set*`,`*k*`)` can also be viewed as the k-fold cartesian product of *set* (see 21.20.15).

The function `Combinations` (see 17.2.1) computes unordered selections without repetitions, `Arrangements` (see 17.2.3) computes ordered selections without repetitions, and finally the function `UnorderedTuples` (see 17.2.5) computes unordered selections with repetitions.

9 ▶ `PermutationsList( `*mset*` )` F

`PermutationsList` returns the set of permutations of the multiset *mset*.

A **permutation** is represented by a list that contains exactly the same elements as *mset*, but possibly in different order. If *mset* is a proper set there are $|mset|!$ (see 17.1.1) such permutations. Otherwise if the first elements appears k_1 times, the second element appears k_2 times and so on, the number of permutations is $|mset|!/(k_1!k_2!\ldots)$, which is sometimes called multinomial coefficient.

10 ▶ `NrPermutationsList( `*mset*` )` F

returns the number of `PermutationsList(`*mset*`)`.

```
gap> PermutationsList( [1,2,3] );
[ [ 1, 2, 3 ], [ 1, 3, 2 ], [ 2, 1, 3 ], [ 2, 3, 1 ], [ 3, 1, 2 ],
  [ 3, 2, 1 ] ]
gap> PermutationsList( [1,1,2,2] );
[ [ 1, 1, 2, 2 ], [ 1, 2, 1, 2 ], [ 1, 2, 2, 1 ], [ 2, 1, 1, 2 ],
  [ 2, 1, 2, 1 ], [ 2, 2, 1, 1 ] ]
gap> NrPermutationsList( [1,2,2,3,3,3,4,4,4,4] );
12600
```

The function `Arrangements` (see 17.2.3) is the generalization of `PermutationsList` that allows you to specify the size of the permutations. `Derangements` (see 17.2.11) computes permutations that have no fixpoints.

11 ▶ `Derangements( `*list*` )` F

returns the set of all derangements of the list *list*.

A **derangement** is a fixpointfree permutation of *list* and is represented by a list that contains exactly the same elements as *list*, but in such an order that the derangement has at no position the same element as *list*. If the list *list* contains no element twice there are exactly $|list|!(1/2! - 1/3! + 1/4! - \cdots + (-1)^n/n!)$ derangements.

Note that the ratio `NrPermutationsList([1..n])/NrDerangements([1..n])`, which is $n!/(n!(1/2!-1/3!+1/4!-\cdots+(-1)^n/n!))$ is an approximation for the base of the natural logarithm $e = 2.7182818285\ldots$, which is correct to about n digits.

12 ▶ `NrDerangements(` *list* `)` F

returns the number of `Derangements(`*list*`)`.

As an example of derangements suppose that you have to send four different letters to four different people. Then a derangement corresponds to a way to send those letters such that no letter reaches the intended person.

```
gap> Derangements( [1,2,3,4] );
[ [ 2, 1, 4, 3 ], [ 2, 3, 4, 1 ], [ 2, 4, 1, 3 ], [ 3, 1, 4, 2 ],
  [ 3, 4, 1, 2 ], [ 3, 4, 2, 1 ], [ 4, 1, 2, 3 ], [ 4, 3, 1, 2 ],
  [ 4, 3, 2, 1 ] ]
gap> NrDerangements( [1..10] );
1334961
gap> Int( 10^7*NrPermutationsList([1..10])/last );
27182816
gap> Derangements( [1,1,2,2,3,3] );
[ [ 2, 2, 3, 3, 1, 1 ], [ 2, 3, 1, 3, 1, 2 ], [ 2, 3, 1, 3, 2, 1 ],
  [ 2, 3, 3, 1, 1, 2 ], [ 2, 3, 3, 1, 2, 1 ], [ 3, 2, 1, 3, 1, 2 ],
  [ 3, 2, 1, 3, 2, 1 ], [ 3, 2, 3, 1, 1, 2 ], [ 3, 2, 3, 1, 2, 1 ],
  [ 3, 3, 1, 1, 2, 2 ] ]
gap> NrDerangements( [1,2,2,3,3,3,4,4,4,4] );
338
```

The function `PermutationsList` (see 17.2.9) computes all permutations of a list.

13 ▶ `PartitionsSet(` *set* `[,` *k*`] )` F

returns the set of all unordered partitions of the set *set* into k pairwise disjoint nonempty sets. If k is not given it returns all unordered partitions of *set* for all k.

An **unordered partition** of *set* is a set of pairwise disjoint nonempty sets with union *set* and is represented by a sorted list of such sets. There are $B(|set|)$ (see 17.1.3) partitions of the set *set* and $S_2(|set|, k)$ (see 17.1.6) partitions with k elements.

14 ▶ `NrPartitionsSet(` *set* `[,` *k*`] )` F

returns the number of `PartitionsSet(`*set*`,`*k*`)`.

```
gap> PartitionsSet( [1,2,3] );
[ [ [ 1 ], [ 2 ], [ 3 ] ], [ [ 1 ], [ 2, 3 ] ], [ [ 1, 2 ], [ 3 ] ],
  [ [ 1, 2, 3 ] ], [ [ 1, 3 ], [ 2 ] ] ]
gap> PartitionsSet( [1,2,3,4], 2 );
[ [ [ 1 ], [ 2, 3, 4 ] ], [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 2, 3 ], [ 4 ] ],
  [ [ 1, 2, 4 ], [ 3 ] ], [ [ 1, 3 ], [ 2, 4 ] ], [ [ 1, 3, 4 ], [ 2 ] ],
  [ [ 1, 4 ], [ 2, 3 ] ] ]
gap> NrPartitionsSet( [1..6] );
203
gap> NrPartitionsSet( [1..10], 3 );
```

```
9330
```

Note that `PartitionsSet` does currently not support multisets and that there is currently no ordered counterpart.

15 ▶ `Partitions(` *n* `[,` *k*`] )` F

returns the set of all (unordered) partitions of the positive integer n into sums with k summands. If k is not given it returns all unordered partitions of *set* for all k.

An **unordered partition** is an unordered sum $n = p_1 + p_2 + \cdots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order, i.e., $p_1 >= p_2 >= \ldots >= p_k$. We write $p \vdash n$. There are approximately $e^{\pi\sqrt{2/3n}}/4\sqrt{3}n$ such partitions.

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore $p(n) :=$ `NrPartitions`(n) is the number of conjugacy classes of the symmetric group on n points.

Ramanujan found the identities $p(5i + 4) = 0 \bmod 5$, $p(7i + 5) = 0 \bmod 7$ and $p(11i + 6) = 0 \bmod 11$ and many other fascinating things about the number of partitions.

Do not call `Partitions` with an n much larger than 40, in which case there are 37338 partitions, since the list will simply become too large.

16 ▶ `NrPartitions(` *n* `[,` *k*`] )` F

returns the number of `Partitions(`*set*`,`*k*`)`.

```
gap> Partitions( 7 );
[ [ 1, 1, 1, 1, 1, 1, 1 ], [ 2, 1, 1, 1, 1, 1 ], [ 2, 2, 1, 1, 1 ],
  [ 2, 2, 2, 1 ], [ 3, 1, 1, 1, 1 ], [ 3, 2, 1, 1 ], [ 3, 2, 2 ],
  [ 3, 3, 1 ], [ 4, 1, 1, 1 ], [ 4, 2, 1 ], [ 4, 3 ], [ 5, 1, 1 ], [ 5, 2 ],
  [ 6, 1 ], [ 7 ] ]
gap> Partitions( 8, 3 );
[ [ 3, 3, 2 ], [ 4, 2, 2 ], [ 4, 3, 1 ], [ 5, 2, 1 ], [ 6, 1, 1 ] ]
gap> NrPartitions( 7 );
15
gap> NrPartitions( 100 );
190569292
```

The function `OrderedPartitions` (see 17.2.17) is the ordered counterpart of `Partitions`.

17 ▶ `OrderedPartitions(` *n* `[,` *k*`] )` F

returns the set of all ordered partitions of the positive integer n into sums with k summands. If k is not given it returns all ordered partitions of *set* for all k.

An **ordered partition** is an ordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $[p_1, p_2, \ldots, p_k]$. There are totally 2^{n-1} ordered partitions and $\binom{n-1}{k-1}$ (see 17.1.2) ordered partitions with k summands.

Do not call `OrderedPartitions` with an n much larger than 15, the list will simply become too large.

18 ▶ `NrOrderedPartitions( n [, k] )` F

returns the number of `OrderedPartitions(`*set*`,`*k*`)`.

```
gap> OrderedPartitions( 5 );
[ [ 1, 1, 1, 1, 1 ], [ 1, 1, 1, 2 ], [ 1, 1, 2, 1 ], [ 1, 1, 3 ],
  [ 1, 2, 1, 1 ], [ 1, 2, 2 ], [ 1, 3, 1 ], [ 1, 4 ], [ 2, 1, 1, 1 ],
  [ 2, 1, 2 ], [ 2, 2, 1 ], [ 2, 3 ], [ 3, 1, 1 ], [ 3, 2 ], [ 4, 1 ], [ 5 ] ]
gap> OrderedPartitions( 6, 3 );
[ [ 1, 1, 4 ], [ 1, 2, 3 ], [ 1, 3, 2 ], [ 1, 4, 1 ], [ 2, 1, 3 ],
  [ 2, 2, 2 ], [ 2, 3, 1 ], [ 3, 1, 2 ], [ 3, 2, 1 ], [ 4, 1, 1 ] ]
gap> NrOrderedPartitions(20);
524288
```

The function `Partitions` (see 17.2.15) is the unordered counterpart of `OrderedPartitions`.

19 ▶ `PartitionsGreatestLE( n, m )` F

returns the set of all (unordered) partitions of the integer n having parts less or equal to the integer m.

20 ▶ `PartitionsGreatestEQ( n, m )` F

returns the set of all (unordered) partitions of the integer n having greatest part equal to the integer m.

21 ▶ `RestrictedPartitions( n, set [, k] )` F

In the first form `RestrictedPartitions` returns the set of all restricted partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set *set*. If k is not given all restricted partitions for all k are returned.

A **restricted partition** is like an ordinary partition (see 17.2.15) an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order. The difference is that here the p_i must be elements from the set *set*, while for ordinary partitions they may be elements from `[1..n]`.

22 ▶ `NrRestrictedPartitions( n, set [, k] )` F

returns the number of `RestrictedPartitions(`n`,`*set*`,`k`)`.

```
gap> RestrictedPartitions( 8, [1,3,5,7] );
[ [ 1, 1, 1, 1, 1, 1, 1, 1 ], [ 3, 1, 1, 1, 1, 1 ], [ 3, 3, 1, 1 ],
  [ 5, 1, 1, 1 ], [ 5, 3 ], [ 7, 1 ] ]
gap> NrRestrictedPartitions(50,[1,2,5,10,20,50]);
451
```

The last example tells us that there are 451 ways to return 50 pence change using 1,2,5,10,20 and 50 pence coins.

23 ▶ `SignPartition( pi )` F

returns the sign of a permutation with cycle structure pi.

This function actually describes a homomorphism from the symmetric group S_n into the cyclic group of order 2, whose kernel is exactly the alternating group A_n (see 40.3.1). Partitions of sign 1 are called **even** partitions while partitions of sign -1 are called **odd**.

```
gap> SignPartition([6,5,4,3,2,1]);
-1
```

24 ▶ `AssociatedPartition( pi )` F

`AssociatedPartition` returns the associated partition of the partition pi which is obtained by transposing the corresponding Young diagram.

```
gap> AssociatedPartition([4,2,1]);
[ 3, 2, 1, 1 ]
gap> AssociatedPartition([6]);
[ 1, 1, 1, 1, 1, 1 ]
```

25 ▶ `PowerPartition( pi, k )` F

`PowerPartition` returns the partition corresponding to the k-th power of a permutation with cycle structure pi.

Each part l of pi is replaced by $d = \gcd(l, k)$ parts l/d. So if pi is a partition of n then pi^k also is a partition of n. `PowerPartition` describes the powermap of symmetric groups.

```
gap> PowerPartition([6,5,4,3,2,1], 3);
[ 5, 4, 2, 2, 2, 2, 1, 1, 1, 1 ]
```

26 ▶ `PartitionTuples( n, r )` F

`PartitionTuples` returns the list of all r-tuples of partitions which together form a partition of n.

r–tuples of partitions describe the classes and the characters of wreath products of groups with r conjugacy classes with the symmetric group S_n.

27 ▶ `NrPartitionTuples( n, r )` F

returns the number of `PartitionTuples( n, r )`.

```
gap> PartitionTuples(3, 2);
[ [ [ 1, 1, 1 ], [  ] ], [ [ 1, 1 ], [ 1 ] ], [ [ 1 ], [ 1, 1 ] ],
  [ [  ], [ 1, 1, 1 ] ], [ [ 2, 1 ], [  ] ], [ [ 1 ], [ 2 ] ],
  [ [ 2 ], [ 1 ] ], [ [  ], [ 2, 1 ] ], [ [ 3 ], [  ] ], [ [  ], [ 3 ] ] ]
```

17.3 Fibonacci and Lucas Sequences

1 ▶ `Fibonacci( n )` F

returns the nth number of the **Fibonacci sequence**. The Fibonacci sequence F_n is defined by the initial conditions $F_1 = F_2 = 1$ and the recurrence relation $F_{n+2} = F_{n+1} + F_n$. For negative n we define $F_n = (-1)^{n+1}F_{-n}$, which is consistent with the recurrence relation.

Using generating functions one can prove that $F_n = \phi^n - 1/\phi^n$, where ϕ is $(\sqrt{5}+1)/2$, i.e., one root of $x^2 - x - 1 = 0$. Fibonacci numbers have the property $Gcd(F_m, F_n) = F_{Gcd(m,n)}$. But a pair of Fibonacci numbers requires more division steps in Euclid's algorithm (see 54.7.1) than any other pair of integers of the same size. `Fibonacci(`k`)` is the special case `Lucas(1,-1,`k`)[1]` (see 17.3.2).

```
gap> Fibonacci( 10 );
55
gap> Fibonacci( 35 );
9227465
gap> Fibonacci( -10 );
-55
```

2 ▶ `Lucas( P, Q, k )` F

returns the k-th values of the **Lucas sequence** with parameters P and Q, which must be integers, as a list of three integers.

Let α, β be the two roots of $x^2 - Px + Q$ then we define $Lucas(P, Q, k)[1] = U_k = (\alpha^k - \beta^k)/(\alpha - \beta)$ and $Lucas(P, Q, k)[2] = V_k = (\alpha^k + \beta^k)$ and as a convenience $Lucas(P, Q, k)[3] = Q^k$.

The following recurrence relations are easily derived from the definition $U_0 = 0$, $U_1 = 1$, $U_k = PU_{k-1} - QU_{k-2}$ and $V_0 = 2$, $V_1 = P$, $V_k = PV_{k-1} - QV_{k-2}$. Those relations are actually used to define `Lucas` if $\alpha = \beta$.

Also the more complex relations used in `Lucas` can be easily derived $U_{2k} = U_k V_k$, $U_{2k+1} = (PU_{2k} + V_{2k})/2$ and $V_{2k} = V_k^2 - 2Q^k$, $V_{2k+1} = ((P^2 - 4Q)U_{2k} + PV_{2k})/2$.

`Fibonacci(`k`)` (see 17.3.1) is simply `Lucas(1,-1,`k`)[1]`. In an abuse of notation, the sequence `Lucas(1,-1,`k`)[2]` is sometimes called the Lucas sequence.

```
gap> List( [0..10], i -> Lucas(1,-2,i)[1] );     # 2^k - (-1)^k)/3
[ 0, 1, 1, 3, 5, 11, 21, 43, 85, 171, 341 ]
gap> List( [0..10], i -> Lucas(1,-2,i)[2] );     # 2^k + (-1)^k
[ 2, 1, 5, 7, 17, 31, 65, 127, 257, 511, 1025 ]
gap> List( [0..10], i -> Lucas(1,-1,i)[1] );     # Fibonacci sequence
[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ]
gap> List( [0..10], i -> Lucas(2,1,i)[1] );      # the roots are equal
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
```

17.4 Permanent of a Matrix

1 ▶ `Permanent(` *mat* `)` F

returns the **permanent** of the matrix *mat*. The permanent is defined by $\sum_{p \in Symm(n)} \prod_{i=1}^{n} mat[i][i^p]$.

Note the similarity of the definition of the permanent to the definition of the determinant (see 24.3.4). In fact the only difference is the missing sign of the permutation. However the permanent is quite unlike the determinant, for example it is not multilinear or alternating. It has however important combinatorial properties.

```
gap> Permanent( [[0,1,1,1],
>      [1,0,1,1],
>      [1,1,0,1],
>      [1,1,1,0]] );  # inefficient way to compute 'NrDerangements([1..4])'
9
gap> Permanent( [[1,1,0,1,0,0,0],
>      [0,1,1,0,1,0,0],
>      [0,0,1,1,0,1,0],
>      [0,0,0,1,1,0,1],
>      [1,0,0,0,1,1,0],
>      [0,1,0,0,0,1,1],
>      [1,0,1,0,0,0,1]] );  # 24 permutations fit the projective plane of order 2
24
```

18 Cyclotomic Numbers

GAP admits computations in abelian extension fields of the rational number field $\mathbb{Q}$, that is fields with abelian Galois group over $\mathbb{Q}$. These fields are subfields of **cyclotomic fields** $\mathbb{Q}(e_n)$ where $e_n = e^{2\pi i/n}$ is a primitive complex n-th root of unity. The elements of these fields are called **cyclotomics**.

Information concerning operations for domains of cyclotomics, for example certain integral bases of fields of cyclotomics, can be found in Chapter 58. For more general operations that take a field extension as a —possibly optional— argument, e.g., `Trace` or `Coefficients`, see Chapter 56.

18.1 Operations for Cyclotomics

1 ▶ `E(` *n* `)` F

`E` returns the primitive n-th root of unity $e_n = e^{2\pi i/n}$. Cyclotomics are usually entered as sums of roots of unity, with rational coefficients, and irrational cyclotomics are displayed in the same way. (For special cyclotomics, see 18.4.)

```
gap> E(9); E(9)^3; E(6); E(12) / 3;
-E(9)^4-E(9)^7
E(3)
-E(3)^2
-1/3*E(12)^7
```

A particular basis is used to express cyclotomics, see 58.3; note that `E(9)` is **not** a basis element, as the above example shows.

2 ▶ `Cyclotomics` V

is the domain of all cyclotomics.

```
gap> E(9) in Cyclotomics; 37 in Cyclotomics; true in Cyclotomics;
true
true
false
```

As the cyclotomics are field elements the usual arithmetic operators `+`,`-`,`*` and `/` (and `^` to take powers by integers) are applicable. Note that `^` does **not** denote the conjugation of group elements, so it is **not** possible to explicitly construct groups of cyclotomics. (However, it is possible to compute the inverse and the multiplicative order of a nonzero cyclotomic.) Also, taking the k-th power of a root of unity z defines a Galois automorphism if and only if k is coprime to the conductor of z.

```
gap> E(5) + E(3); (E(5) + E(5)^4) ^ 2; E(5) / E(3); E(5) * E(3);
-E(15)^2-2*E(15)^8-E(15)^11-E(15)^13-E(15)^14
-2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
E(15)^13
E(15)^8
gap> Order( E(5) ); Order( 1+E(5) );
5
infinity
```

3 ▶ `IsCyclotomic( obj )` C
▶ `IsCyc( obj )` C

Every object in the family `CyclotomicsFamily` lies in the category `IsCyclotomic`. This covers integers, rationals, proper cyclotomics, the object `infinity` (see 18.2.1), and unknowns (see Chapter 19). All these objects except `infinity` and unknowns lie also in the category `IsCyc`, `infinity` lies in (and can be detected from) the category `IsInfinity`, and unknowns lie in `IsUnknown`.

```
gap> IsCyclotomic(0); IsCyclotomic(1/2*E(3)); IsCyclotomic( infinity );
true
true
true
gap> IsCyc(0); IsCyc(1/2*E(3)); IsCyc( infinity );
true
true
false
```

4 ▶ `IsIntegralCyclotomic( obj )` P

A cyclotomic is called **integral** or a **cyclotomic integer** if all coefficients of its minimal polynomial over the rationals are integers. Since the underlying basis of the external representation of cyclotomics is an integral basis (see 58.3), the subring of cyclotomic integers in a cyclotomic field is formed by those cyclotomics for which the external representation is a list of integers. For example, square roots of integers are cyclotomic integers (see 18.4), any root of unity is a cyclotomic integer, character values are always cyclotomic integers, but all rationals which are not integers are not cyclotomic integers.

```
gap> r:= ER( 5 );               # The square root of 5 is a cyclotomic integer.
E(5)-E(5)^2-E(5)^3+E(5)^4
gap> IsIntegralCyclotomic( r );  # It has integral coefficients.
true
gap> r2:= 1/2 * r;              # This is not a cyclotomic integer, ...
1/2*E(5)-1/2*E(5)^2-1/2*E(5)^3+1/2*E(5)^4
gap> IsIntegralCyclotomic( r2 );
false
gap> r3:= 1/2 * r - 1/2;        # ... but this is one.
E(5)+E(5)^4
gap> IsIntegralCyclotomic( r3 );
true
```

The operation `Int` can be used to find a cyclotomic integer near to an arbitrary cyclotomic. For rationals, `Int` returns the largest integer smaller or equal to the argument.

```
gap> Int( E(5)+1/2*E(5)^2 ); Int( 2/3*E(7)+3/2*E(4) );
E(5)
E(4)
```

The operation `String` returns for a cyclotomic a string corresponding to the way the cyclotomic is printed by `ViewObj` and `PrintObj`.

```
gap> String( E(5)+1/2*E(5)^2 ); String( 17/3 );
"E(5)+1/2*E(5)^2"
"17/3"
```

5 ▶ `Conductor(` *cyc* `)` A
▶ `Conductor(` *C* `)` A

For an element *cyc* of a cyclotomic field, `Conductor` returns the smallest integer n such that *cyc* is contained in the n-th cyclotomic field. For a collection C of cyclotomics (for example a dense list of cyclotomics or a field of cyclotomics), `Conductor` returns the smallest integer n such that all elements of C are contained in the n-th cyclotomic field.

```
gap> Conductor( 0 ); Conductor( E(10) ); Conductor( E(12) );
1
5
12
```

6 ▶ `AbsoluteValue(` *cyc* `)` A

returns the absolute value of a cyclotomic number *cyc*. At the moment only methods for rational numbers exist.

```
gap> AbsoluteValue(-3);
3
```

7 ▶ `RoundCyc(` *cyc* `)` O

is a cyclotomic integer z (see 18.1.4) near to the cyclotomic *cyc* in the sense that the i-th coefficient in the external representation (see 18.1.8) of z is `Int( c+1/2 )` where `c` is the i-th coefficient in the external representation of *cyc*. Expressed in terms of the Zumbroich basis (see 58.3), the coefficients of *cyc* w.r.t. this basis are rounded.

```
gap> RoundCyc( E(5)+1/2*E(5)^2 ); RoundCyc( 2/3*E(7)+3/2*E(4) );
E(5)+E(5)^2
-2*E(28)^3+E(28)^4-2*E(28)^11-2*E(28)^15-2*E(28)^19-2*E(28)^23-2*E(28)^27
```

8 ▶ `CoeffsCyc(` *cyc*`,` *N* `)` F

Let *cyc* be a cyclotomic with conductor n. If N is not a multiple of n then `CoeffsCyc` returns `fail` because *cyc* cannot be expressed in terms of N-th roots of unity. Otherwise `CoeffsCyc` returns a list of length N with entry at position j equal to the coefficient of $e^{2\pi i(j-1)/N}$ if this root belongs to the N-th Zumbroich basis (see 58.3), and equal to zero otherwise. So we have *cyc* `= CoeffsCyc(`*cyc*`,`*N*`) * List( [1..`*N*`], j -> E(`*N*`)^(j-1) )`.

```
gap> cyc:= E(5)+E(5)^2;
E(5)+E(5)^2
gap> CoeffsCyc( cyc, 5 );  CoeffsCyc( cyc, 15 );  CoeffsCyc( cyc, 7 );
[ 0, 1, 1, 0, 0 ]
[ 0, -1, 0, 0, 0, 0, 0, 0, -1, 0, 0, -1, 0, -1, 0 ]
fail
```

9 ▶ `DenominatorCyc( cyc )` F

For a cyclotomic number *cyc* (see 18.1.3), this function returns the smallest positive integer n such that n `*` *cyc* is a cyclotomic integer (see 18.1.4). For rational numbers *cyc*, the result is the same as that of `DenominatorRat` (see 16.1.5).

10 ▶ `ExtRepOfObj( cyc )`

```
gap> ExtRepOfObj( E(5) ); CoeffsCyc( E(5), 15 );
[ 0, 1, 0, 0, 0 ]
[ 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, -1, 0 ]
gap> CoeffsCyc( 1+E(3), 9 ); CoeffsCyc( E(5), 7 );
[ 0, 0, 0, 0, 0, 0, -1, 0, 0 ]
fail
```

11 ▶ `DescriptionOfRootOfUnity( root )` F

Given a cyclotomic *root* that is known to be a root of unity (this is **not** checked), `DescriptionOfRootOfUnity` returns a list $[n, e]$ of coprime positive integers such that *root* $= \mathtt{E}(n)^e$ holds.

```
gap> E(9);  DescriptionOfRootOfUnity( E(9) );
-E(9)^4-E(9)^7
[ 9, 1 ]
gap> DescriptionOfRootOfUnity( -E(3) );
[ 6, 5 ]
```

12 ▶ `IsGaussInt( x )` F

`IsGaussInt` returns `true` if the object x is a Gaussian integer (see 58.5.1) and `false` otherwise. Gaussian integers are of the form a `+` b`*E(4)`, where a and b are integers.

13 ▶ `IsGaussRat( x )` F

`IsGaussRat` returns `true` if the object x is a Gaussian rational (see 58.1.3) and `false` otherwise. Gaussian rationals are of the form a `+` b`*E(4)`, where a and b are rationals.

`DefaultField` (see 56.1.4) for cyclotomics is defined to return the smallest **cyclotomic** field containing the given elements.

```
gap> Field( E(5)+E(5)^4 );  DefaultField( E(5)+E(5)^4 );
NF(5,[ 1, 4 ])
CF(5)
```

18.2 Infinity

1 ▶ `IsInfinity( obj )` C
▶ `infinity` V

`infinity` is a special GAP object that lies in `CyclotomicsFamily`. It is larger than all other objects in this family. `infinity` is mainly used as return value of operations such as `Size` and `Dimension` for infinite and infinite dimensional domains, respectively.

Note that **no** arithmetic operations are provided for `infinity`, in particular there is no problem to define what `0 * infinity` or `infinity - infinity` means.

Often it is useful to distinguish `infinity` from "proper" cyclotomics. For that, `infinity` lies in the category `IsInfinity` but not in `IsCyc`, and the other cyclotomics lie in the category `IsCyc` but not in `IsInfinity`.

```
gap> s:= Size( Rationals );
infinity
gap> s = infinity; IsCyclotomic( s ); IsCyc( s ); IsInfinity( s );
true
true
false
true
gap> s in Rationals; s > 17;
false
true
gap> Set( [ s, 2, s, E(17), s, 19 ] );
[ 2, 19, E(17), infinity ]
```

18.3 Comparisons of Cyclotomics

To compare cyclotomics, the operators `<`, `<=`, `=`, `>=`, `>` and `<>` can be used, the result will be `true` if the first operand is smaller, smaller or equal, equal, larger or equal, larger, or unequal, respectively, and `false` otherwise.

Cyclotomics are ordered as follows: The relation between rationals is the natural one, rationals are smaller than irrational cyclotomics, and `infinity` is the largest cyclotomic. For two irrational cyclotomics with different conductors, the one with smaller conductor is regarded as smaller. Two irrational cyclotomics with same conductor are compared via their external representation.

For comparisons of cyclotomics and other GAP objects, see Section 4.11.

```
gap> E(5) < E(6);      # the latter value has conductor 3
false
gap> E(3) < E(3)^2;    # both have conductor 3, compare the ext. repr.
false
gap> 3 < E(3); E(5) < E(7);
true
true
```

18.4 ATLAS Irrationalities

1 ▶ `EB( n )` F
▶ `EC( n )` F
▶ `ED( n )` F
▶ `EE( n )` F
▶ `EF( n )` F
▶ `EG( n )` F
▶ `EH( n )` F

For N a positive integer, let $z = \mathtt{E}(N) = \exp(2\pi i/N)$. The following so-called **atomic irrationalities** (see Chapter 7, Section 10 of [CCN+85]) can be entered using functions. (Note that the values are not necessary irrational.)

$$\begin{array}{lllll}
\texttt{EB}(N) & = & b_N & = & \frac{1}{2}\sum_{j=1}^{N-1} z^{j^2}, \quad N \equiv 1 \pmod 2 \\
\texttt{EC}(N) & = & c_N & = & \frac{1}{3}\sum_{j=1}^{N-1} z^{j^3}, \quad N \equiv 1 \pmod 3 \\
\texttt{ED}(N) & = & d_N & = & \frac{1}{4}\sum_{j=1}^{N-1} z^{j^4}, \quad N \equiv 1 \pmod 4 \\
\texttt{EE}(N) & = & e_N & = & \frac{1}{5}\sum_{j=1}^{N-1} z^{j^5}, \quad N \equiv 1 \pmod 5 \\
\texttt{EF}(N) & = & f_N & = & \frac{1}{6}\sum_{j=1}^{N-1} z^{j^6}, \quad N \equiv 1 \pmod 6 \\
\texttt{EG}(N) & = & g_N & = & \frac{1}{7}\sum_{j=1}^{N-1} z^{j^7}, \quad N \equiv 1 \pmod 7 \\
\texttt{EH}(N) & = & h_N & = & \frac{1}{8}\sum_{j=1}^{N-1} z^{j^8}, \quad N \equiv 1 \pmod 8
\end{array}$$

(Note that in $c_N, \ldots, h_N$, N must be a prime.)

2 ▶ `EI( n )` F
▶ `ER( n )` F

For a rational number N, `ER` returns the square root $\sqrt{N}$ of N, and `EI` returns $\sqrt{-N}$. By the chosen embedding of cyclotomic fields into the complex numbers, `ER` returns the positive square root if N is positive, and if N is negative then `ER(`N`) = EI(-`N`)`. In any case, `EI(`N`) = E(4) * ER(`N`)`.

`ER` is installed as method for the operation `Sqrt` (see 30.12.5) for rational argument.

From a theorem of Gauss we know that

$$b_N = \begin{cases} \frac{1}{2}(-1+\sqrt{N}) & \text{if} \quad N \equiv 1 \pmod 4 \\ \frac{1}{2}(-1+i\sqrt{N}) & \text{if} \quad N \equiv -1 \pmod 4 \end{cases}$$

So $\sqrt{N}$ can be computed from b_N (see 18.4.1).

3 ▶ `EY( n[, d] )` F
▶ `EX( n[, d] )` F
▶ `EW( n[, d] )` F
▶ `EV( n[, d] )` F
▶ `EU( n[, d] )` F
▶ `ET( n[, d] )` F
▶ `ES( n[, d] )` F

For given N, let $n_k = n_k(N)$ be the first integer with multiplicative order exactly k modulo N, chosen in the order of preference

$$1, -1, 2, -2, 3, -3, 4, -4, \ldots.$$

We define

$$\begin{array}{lllll}
\texttt{EY}(N) & = & y_n & = & z + z^n & (n = n_2) \\
\texttt{EX}(N) & = & x_n & = & z + z^n + z^{n^2} & (n = n_3) \\
\texttt{EW}(N) & = & w_n & = & z + z^n + z^{n^2} + z^{n^3} & (n = n_4) \\
\texttt{EV}(N) & = & v_n & = & z + z^n + z^{n^2} + z^{n^3} + z^{n^4} & (n = n_5) \\
\texttt{EU}(N) & = & u_n & = & z + z^n + z^{n^2} + \ldots + z^{n^5} & (n = n_6) \\
\texttt{ET}(N) & = & t_n & = & z + z^n + z^{n^2} + \ldots + z^{n^6} & (n = n_7) \\
\texttt{ES}(N) & = & s_n & = & z + z^n + z^{n^2} + \ldots + z^{n^7} & (n = n_8)
\end{array}$$

4 ▶ `EM( n[, d] )` F
▶ `EL( n[, d] )` F
▶ `EK( n[, d] )` F
▶ `EJ( n[, d] )` F

$$\begin{array}{lllll}
\texttt{EM}(N) & = & m_n & = & z - z^n & (n = n_2) \\
\texttt{EL}(N) & = & l_n & = & z - z^n + z^{n^2} - z^{n^3} & (n = n_4) \\
\texttt{EK}(N) & = & k_n & = & z - z^n + \ldots - z^{n^5} & (n = n_6) \\
\texttt{EJ}(N) & = & j_n & = & z - z^n + \ldots - z^{n^7} & (n = n_8)
\end{array}$$

5 ▶ `NK( n, k, d )` F

Let $n_k^{(d)} = n_k^{(d)}(N)$ be the $d+1$-th integer with multiplicative order exactly k modulo N, chosen in the order of preference defined above; we write $n_k = n_k^{(0)}, n'_k = n_k^{(1)}, n''_k = n_k^{(2)}$ and so on. These values can be computed as `NK(`N`,`k`,`d`)`$= n_k^{(d)}(N)$; if there is no integer with the required multiplicative order, `NK` returns `fail`.

The algebraic numbers

$$y'_N = y_N^{(1)}, y''_N = y_N^{(2)}, \ldots, x'_N, x''_N, \ldots, j'_N, j''_N, \ldots$$

are obtained on replacing n_k in the above definitions by $n'_k, n''_k, \ldots$; they can be entered as

$$\begin{aligned} \texttt{EY}(N,d) &= y_N^{(d)} \\ \texttt{EX}(N,d) &= x_N^{(d)} \\ &\vdots \\ \texttt{EJ}(N,d) &= j_n^{(d)} \end{aligned}$$

6 ▶ `AtlasIrrationality( `*irratname*` )` F

Let *irratname* be a string that describes an irrational value as described in Chapter 6, Section 10 of [CCN+85], that is, a linear combination of the atomic irrationalities introduced above. (The following definition is mainly copied from [CCN+85].) If q_N is such a value (e.g. y''_{24}) then linear combinations of algebraic conjugates of q_N are abbreviated as in the following examples:

`2qN + 3&5 − 4&7 + &9`	means	$2q_N + 3q_N^{*5} - 4q_N^{*7} + q_N^{*9}$
`4qN&3&5&7 − 3&4`	means	$4(q_N + q_N^{*3} + q_N^{*5} + q_N^{*7}) - 3q_N^{*11}$
`4qN * 3&5 + &7`	means	$4(q_N^{*3} + q_N^{*5}) + q_N^{*7}$

To explain the "ampersand" syntax in general we remark that "&k" is interpreted as q_N^{*k}, where q_N is the most recently named atomic irrationality, and that the scope of any premultiplying coefficient is broken by a + or − sign, but not by & or $*k$. The algebraic conjugations indicated by the ampersands apply directly to the **atomic** irrationality q_N, even when, as in the last example, q_N first appears with another conjugacy $*k$.

```
gap> EW(16,3); EW(17,2); ER(3); EI(3); EY(5); EB(9);
0
E(17)+E(17)^4+E(17)^13+E(17)^16
-E(12)^7+E(12)^11
E(3)-E(3)^2
E(5)+E(5)^4
1
gap> AtlasIrrationality( "b7*3" );
E(7)^3+E(7)^5+E(7)^6
gap> AtlasIrrationality( "y'''24" );
E(24)-E(24)^19
gap> AtlasIrrationality( "-3y'''24*13&5" );
3*E(8)-3*E(8)^3
gap> AtlasIrrationality( "3y'''24*13-2&5" );
-3*E(24)-2*E(24)^11+2*E(24)^17+3*E(24)^19
gap> AtlasIrrationality( "3y'''24*13-&5" );
-3*E(24)-E(24)^11+E(24)^17+3*E(24)^19
gap> AtlasIrrationality( "3y'''24*13-4&5&7" );
-7*E(24)-4*E(24)^11+4*E(24)^17+7*E(24)^19
gap> AtlasIrrationality( "3y'''24&7" );
6*E(24)-6*E(24)^19
```

18.5 Galois Conjugacy of Cyclotomics

1 ▶ `GaloisCyc( cyc, k )` O
▶ `GaloisCyc( list, k )` O

For a cyclotomic *cyc* and an integer k, `GaloisCyc` returns the cyclotomic obtained by raising the roots of unity in the Zumbroich basis representation of *cyc* to the k-th power. If k is coprime to the integer n, `GaloisCyc( ., k )` acts as a Galois automorphism of the n-th cyclotomic field (see 58.4); to get the Galois automorphisms themselves, use `GaloisGroup` (see 56.3.1).

The **complex conjugate** of *cyc* is `GaloisCyc( cyc, -1 )`, which can also be computed using `ComplexConjugate` (see 18.5.2).

For a list or matrix *list* of cyclotomics, `GaloisCyc` returns the list obtained by applying `GaloisCyc` to the entries of *list*.

2 ▶ `ComplexConjugate( z )` A

For a cyclotomic number z, `ComplexConjugate` returns `GaloisCyc( z, -1 )`. For a quaternion $z = c_1 e + c_2 i + c_3 j + c_4 k$, `ComplexConjugate` returns $c_1 e - c_2 i - c_3 j - c_4 k$.

```
gap> GaloisCyc( E(5) + E(5)^4, 2 );
E(5)^2+E(5)^3
gap> GaloisCyc( E(5), -1 );           # the complex conjugate
E(5)^4
gap> GaloisCyc( E(5) + E(5)^4, -1 );  # this value is real
E(5)+E(5)^4
gap> GaloisCyc( E(15) + E(15)^4, 3 );
E(5)+E(5)^4
gap> ComplexConjugate( E(7) );
E(7)^6
```

3 ▶ `StarCyc( cyc )` F

If the cyclotomic *cyc* is an irrational element of a quadratic extension of the rationals then `StarCyc` returns the unique Galois conjugate of *cyc* that is different from *cyc*, otherwise `fail` is returned. In the first case, the return value is often called *cyc*$*$ (see 69.11).

```
gap> StarCyc( EB(5) ); StarCyc( E(5) );
E(5)^2+E(5)^3
fail
```

4 ▶ `Quadratic( cyc )` F

Let *cyc* be a cyclotomic integer that lies in a quadratic extension field of the rationals. Then we have $cyc = (a + b\sqrt{n})/d$ for integers a, b, n, d, such that d is either 1 or 2. In this case, `Quadratic` returns a record with the components `a`, `b`, `root`, `d`, `ATLAS`, and `display`; the values of the first four are a, b, n, and d, the `ATLAS` value is a (not necessarily shortest) representation of *cyc* in terms of the ATLAS irrationalities $b_{|n|}$, $i_{|n|}$, $r_{|n|}$, and the `display` value is a string that expresses *cyc* in GAP notation, corresponding to the value of the `ATLAS` component.

If *cyc* is not a cyclotomic integer or does not lie in a quadratic extension field of the rationals then `fail` is returned.

If the denominator d is 2 then necessarily n is congruent to 1 modulo 4, and r_n, i_n are not possible; we have *cyc* `= x + y * EB( root )` with `y = b`, `x = ( a + b ) / 2`.

If $d = 1$, we have the possibilities $i_{|n|}$ for $n < -1$, $a + b * i$ for $n = -1$, $a + b * r_n$ for $n > 0$. Furthermore if n is congruent to 1 modulo 4, also $cyc = (a + b) + 2 * b * b_{|n|}$ is possible; the shortest string of these is taken as the value for the component `ATLAS`.

```
gap> Quadratic( EB(5) ); Quadratic( EB(27) );
rec( a := -1, b := 1, root := 5, d := 2, ATLAS := "b5",
  display := "(-1+ER(5))/2" )
rec( a := -1, b := 3, root := -3, d := 2, ATLAS := "1+3b3",
  display := "(-1+3*ER(-3))/2" )
gap> Quadratic(0); Quadratic( E(5) );
rec( a := 0, b := 0, root := 1, d := 1, ATLAS := "0", display := "0" )
fail
```

5 ▶ `GaloisMat( mat )` A

Let *mat* be a matrix of cyclotomics. `GaloisMat` calculates the complete orbits under the operation of the Galois group of the (irrational) entries of *mat*, and the permutations of rows corresponding to the generators of the Galois group.

If some rows of *mat* are identical, only the first one is considered for the permutations, and a warning will be printed.

`GaloisMat` returns a record with the components `mat`, `galoisfams`, and `generators`.

`mat`:
: a list with initial segment being the rows of *mat* (**not** shallow copies of these rows); the list consists of full orbits under the action of the Galois group of the entries of *mat* defined above. The last rows in the list are those not contained in *mat* but must be added in order to complete the orbits; so if the orbits were already complete, *mat* and `mat` have identical rows.

`galoisfams`:
: a list that has the same length as the `mat` component, its entries are either 1, 0, -1, or lists. `galoisfams[i] = 1` means that `mat[i]` consists of rationals, i.e. `[ mat[i] ]` forms an orbit; `galoisfams[i] = -1` means that `mat[i]` contains unknowns (see Chapter 19); in this case `[ mat[i] ]` is regarded as an orbit, too, even if `mat[i]` contains irrational entries; if `galoisfams`$[i] = [l_1, l_2]$ is a list then `mat[i]` is the first element of its orbit in `mat`, l_1 is the list of positions of rows that form the orbit, and l_2 is the list of corresponding Galois automorphisms (as exponents, not as functions), so we have `mat`$[l_1[j]][k] =$ `GaloisCyc`$($`mat`$[i][k], l_2[j])$; `galoisfams[i] = 0` means that `mat[i]` is an element of a nontrivial orbit but not the first element of it.

`generators`:
: a list of permutations generating the permutation group corresponding to the action of the Galois group on the rows of `mat`.

In the following example we temporarily increase the line length limit from its default value 80 to 84 in order to get a nicer output format.

```
gap> SizeScreen([ 84, ]);;
gap> GaloisMat( [ [ E(3), E(4) ] ] );
rec(
  mat := [ [ E(3), E(4) ], [ E(3), -E(4) ], [ E(3)^2, E(4) ], [ E(3)^2, -E(4) ] ],
  galoisfams := [ [ [ 1, 2, 3, 4 ], [ 1, 7, 5, 11 ] ], 0, 0, 0 ],
  generators := [ (1,2)(3,4), (1,3)(2,4) ] )
gap> SizeScreen([ 80, ]);;
gap> GaloisMat( [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ] ] );
rec( mat := [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ],
  galoisfams := [ 1, [ [ 2, 3 ], [ 1, 2 ] ], 0 ], generators := [ (2,3) ] )
```

6 ▶ `RationalizedMat( mat )` A

returns the list of rationalized rows of *mat*, which must be a matrix of cyclotomics. This is the set of sums over orbits under the action of the Galois group of the entries of *mat* (see 18.5.5), so the operation may be viewed as a kind of trace on the rows.

Note that no two rows of *mat* should be equal.

```
gap> mat:= [ [ 1, 1, 1 ], [ 1, E(3), E(3)^2 ], [ 1, E(3)^2, E(3) ] ];;
gap> RationalizedMat( mat );
[ [ 1, 1, 1 ], [ 2, -1, -1 ] ]
```

18.6 Internally Represented Cyclotomics

The implementation of an **internally represented cyclotomic** is based on a list of length equal to its conductor. This means that the internal representation of a cyclotomic does **not** refer to the smallest number field but the smallest **cyclotomic** field containing it. The reason for this is the wish to reflect the natural embedding of two cyclotomic fields into a larger one that contains both. With such embeddings, it is easy to construct the sum or the product of two arbitrary cyclotomics (in possibly different fields) as an element of a cyclotomic field.

The disadvantage of this approach is that the arithmetical operations are quite expensive, so the use of internally represented cyclotomics is not recommended for doing arithmetics over number fields, such as calculations with matrices of cyclotomics. But internally represented cyclotomics are good enough for dealing with irrationalities in character tables (see chapter 69).

For the representation of cyclotomics one has to recall that the n-th cyclotomic field $\mathbb{Q}(e_n)$ is a vector space of dimension $\varphi(n)$ over the rationals where φ denotes Euler's phi-function (see 15.1.2).

A special integral basis of cyclotomic fields is chosen that allows one to easily convert arbitrary sums of roots of unity into the basis, as well as to convert a cyclotomic represented w.r.t. the basis into the smallest possible cyclotomic field. This basis is accessible in GAP, see 58.3 for more information and references.

Note that the set of all n-th roots of unity is linearly dependent for $n > 1$, so multiplication is **not** the multiplication of the group ring $\mathbb{Q}\langle e_n\rangle$; given a $\mathbb{Q}$-basis of $\mathbb{Q}(e_n)$ the result of the multiplication (computed as multiplication of polynomials in e_n, using $(e_n)^n = 1$) will be converted to the basis.

```
gap> E(5) * E(5)^2; ( E(5) + E(5)^4 ) * E(5)^2;
E(5)^3
E(5)+E(5)^3
gap> ( E(5) + E(5)^4 ) * E(5);
-E(5)-E(5)^3-E(5)^4
```

An internally represented cyclotomic is always represented in the smallest cyclotomic field it is contained in. The internal coefficients list coincides with the external representation returned by `ExtRepOfObj`.

Since the conductor of internally represented cyclotomics must be in the filter `IsSmallIntRep`, the biggest possible (though not very useful) conductor is $2^{28} - 1$. So the maximal cyclotomic field implemented in GAP is not really the field $\mathbb{Q}^{ab}$.

```
gap> IsSmallIntRep( 2^28-1 );
true
gap> IsSmallIntRep( 2^28 );
false
```

It should be emphasized that one disadvantage of representing a cyclotomic in the smallest **cyclotomic** field (and not in the smallest field) is that arithmetic operations in a fixed small extension field of the rational number field are comparatively expensive. For example, take a prime integer p and suppose that we want to work with a matrix group over the field $\mathbb{Q}(\sqrt{p})$. Then each matrix entry could be described by two rational coefficients, whereas the representation in the smallest cyclotomic field requires $p-1$ rational coefficients for each entry. So it is worth thinking about using elements in a field constructed with `AlgebraicExtension` (see 65.1.1) when natural embeddings of cyclotomic fields are not needed.

19 Unknowns

Sometimes the result of an operation does not allow further computations with it. In many cases, then an error is signalled, and the computation is stopped.

This is not appropriate for some applications in character theory. For example, if one wants to induce a character of a group to a supergroup (see 70.9.3) but the class fusion is only a parametrized map (see Chapter 71), there may be values of the induced character which are determined by the fusion map, whereas other values are not known.

For this and other situations, GAP provides the data type **unknown**. An object of this type, further on called an **unknown**, may stand for any cyclotomic (see Chapter 18), in particular its family (see 13.1) is `CyclotomicsFamily`.

Unknowns are parametrized by positive integers. When a GAP session is started, no unknowns exist.

The only ways to create unknowns are to call the function `Unknown` or a function that calls it, or to do arithmetical operations with unknowns.

GAP objects containing unknowns will contain **fixed** unknowns when they are printed to files, i.e., function calls `Unknown( n )` instead of `Unknown()`. So be careful to read files printed in different GAP sessions, since there may be the same unknown at different places.

The rest of this chapter contains information about the unknown constructor, the category, and comparison of and arithmetical operations for unknowns; more is not known about unknowns in GAP.

1 ▶ `Unknown( )` O
▶ `Unknown(` *n* `)` O

In the first form `Unknown` returns a new unknown value, i.e., the first one that is larger than all unknowns which exist in the current GAP session.

In the second form `Unknown` returns the n-th unknown; if it did not exist yet, it is created.

2 ▶ `LargestUnknown` V

`LargestUnknown` is the largest n that is used in any `Unknown(` *n* `)` in the current GAP session. This is used in `Unknown` which increments this value when asked to make a new unknown.

3 ▶ `IsUnknown(` *obj* `)` C

is the category of unknowns in GAP.

```
gap> Unknown();  List( [ 1 .. 20 ], i -> Unknown() );;
Unknown(1)
gap> Unknown();   # note that we have already created 21 unknowns.
Unknown(22)
gap> Unknown(2000);  Unknown();
Unknown(2000)
Unknown(2001)
gap> LargestUnknown;
2001
```

```
gap> IsUnknown( Unknown );  IsUnknown( Unknown() );
false
true
```

Unknowns can be **compared** via = and < with all cyclotomics and with certain other GAP objects (see 4.11). We have `Unknown(` *n* `) >= Unknown(` *m* `)` if and only if *n* >= *m* holds; unknowns are larger than all cyclotomics that are not unknowns.

```
gap> Unknown() >= Unknown();  Unknown(2) < Unknown(3);
false
true
gap> Unknown() > 3;  Unknown() > E(3);
true
true
gap> Unknown() > Z(8);  Unknown() > [];
false
false
```

The usual arithmetic operations +, -, * and / are defined for addition, subtraction, multiplication and division of unknowns and cyclotomics. The result will be a new unknown except in one of the following cases.

Multiplication with zero yields zero, and multiplication with one or addition of zero yields the old unknown. **Note** that division by an unknown causes an error, since an unknown might stand for zero.

As unknowns are cyclotomics, dense lists of unknowns and other cyclotomics are row vectors and they can be added and multiplied in the usual way. Consequently, lists of such row vectors of equal length are (ordinary) matrices (see 24.1.2).

20 Booleans

The two main **boolean** values are `true` and `false`. They stand for the **logical** values of the same name. They appear as values of the conditions in `if`-statements and `while`-loops. Booleans are also important as return values of **filters** (see 13.2) such as `IsFinite` and `IsBool`. Note that it is a convention that the name of a function that returns `true` or `false` according to the outcome, starts with `Is`.

For technical reasons, also the value `fail` (see 20.1.1) is regarded as a boolean.

1 ▶ `IsBool(` *obj* `)` C

tests whether *obj* is `true`, `false` or `fail`.

```
gap> IsBool( true );  IsBool( false );  IsBool( 17 );
true
true
false
```

20.1 Fail

1 ▶ `fail` V

The value `fail` is used to indicate situations when an operation could not be performed for the given arguments, either because of shortcomings of the arguments or because of restrictions in the implementation or computability. So for example `Position` (see 21.16.1) will return `fail` if the point searched for is not in the list.

`fail` is simply an object that is different from every other object than itself.

For technical reasons, `fail` is a boolean value. But note that `fail` cannot be used to form boolean expressions with `and`, `or`, and `not` (see 20.3 below), and `fail` cannot appear in boolean lists (see Chapter 22).

20.2 Comparisons of Booleans

1 ▶ *bool1* `=` *bool2*
▶ *bool1* `<>` *bool2*

The equality operator `=` evaluates to `true` if the two boolean values *bool1* and *bool2* are equal, i.e., both are `true` or both are `false` or both `fail`, and `false` otherwise. The inequality operator `<>` evaluates to `true` if the two boolean values *bool1* and *bool2* are different and `false` otherwise. This operation is also called the **exclusive or**, because its value is `true` if exactly one of *bool1* or *bool2* is `true`.

You can compare boolean values with objects of other types. Of course they are never equal.

```
gap> true = false;
false
gap> false = (true = fail);
true
gap> true <> 17;
true
```

2 ► *bool1* `<` *bool2*

The ordering of boolean values is defined by `true < false < fail`. For the comparison of booleans with other GAP objects, see Section 4.11.

```
gap> true < false;  fail >= false;
true
true
```

20.3 Operations for Booleans

The following boolean operations are only applicable to `true` and `false`.

1 ► *bool1* `or` *bool2*

The logical operator `or` evaluates to `true` if at least one of the two boolean operands *bool1* and *bool2* is `true` and to `false` otherwise.

`or` first evaluates *bool1*. If the value is neither `true` nor `false` an error is signalled. If the value is `true`, then `or` returns `true` **without** evaluating *bool2*. If the value is `false`, then `or` evaluates *bool2*. Again, if the value is neither `true` nor `false` an error is signalled. Otherwise `or` returns the value of *bool2*. This **short-circuited** evaluation is important if the value of *bool1* is `true` and evaluation of *bool2* would take much time or cause an error.

`or` is associative, i.e., it is allowed to write *b1* `or` *b2* `or` *b3*, which is interpreted as (*b1* `or` *b2*) `or` *b3*. `or` has the lowest precedence of the logical operators. All logical operators have lower precedence than the comparison operators `=`, `<`, `in`, etc.

```
gap> true or false;
true
gap> false or false;
false
gap> i := -1;; l := [1,2,3];;
gap> if i <= 0 or l[i] = false then     # this does not cause an error,
>    Print("aha\n"); fi;               # because 'l[i]' is not evaluated
aha
```

2 ► *bool1* `and` *bool2*

The logical operator `and` evaluates to `true` if both boolean operands *bool1* and *bool2* are `true` and to `false` otherwise.

`and` first evaluates *bool1*. If the value is neither `true` nor `false` an error is signalled. If the value is `false`, then `and` returns `false` **without** evaluating *bool2*. If the value is `true`, then `and` evaluates *bool2*. Again, if the value is neither `true` nor `false` an error is signalled. Otherwise `and` returns the value of *bool2*. This **short-circuited** evaluation is important if the value of *bool1* is `false` and evaluation of *bool2* would take much time or cause an error.

`and` is associative, i.e., it is allowed to write *b1* `and` *b2* `and` *b3*, which is interpreted as (*b1* `and` *b2*) `and` *b3*. `and` has higher precedence than the logical `or` operator, but lower than the unary logical `not` operator. All logical operators have lower precedence than the comparison operators `=`, `<`, `in`, etc.

```
gap> true and false;
false
gap> true and true;
true
gap> false and 17;  # this does not cause an error, because '17' is never looked at
false
```

3 ▶ *fil1* `and` *fil2*

`and` can also be applied to filters. It returns a filter that when applied to some argument x, tests *fil1*`(`x`)` `and` *fil2*`(`x`)`.

```
gap> andfilt:= IsPosRat and IsInt;;
gap> andfilt( 17 );  andfilt( 1/2 );
true
false
```

4 ▶ `not` *bool*

The logical operator `not` returns `true` if the boolean value *bool* is `false` and `true` otherwise. An error is signalled if *bool* does not evaluate to `true` or `false`.

`not` has higher precedence than the other logical operators, `or` and `and`. All logical operators have lower precedence than the comparison operators `=`, `<`, `in`, etc.

```
gap> true and false;
false
gap> not true;
false
gap> not false;
true
```

21 Lists

Lists are the most important way to treat objects together. A **list** arranges objects in a definite order. So each list implies a partial mapping from the integers to the elements of the list. I.e., there is a first element of a list, a second, a third, and so on. Lists can occur in mutable or immutable form, see 12.6 for the concept of mutability, and 21.7 for the case of lists.

This chapter deals mainly with the aspect of lists in GAP as **data structures**. Chapter 28 tells more about the **collection** aspect of certain lists, and more about lists as **arithmetic objects** can be found in the chapters 23 and 24.

Lists are used to implement ranges (see 21.22), sets (see 21.19), strings (see 26), row vectors (see 23), and matrices (see 24); Boolean lists (see 22) are a further special kind of lists.

Several operations for lists, such as `Intersection` and `Random`, will be described in Chapter 28, in particular see 28.2.

21.1 List Categories

A list can be written by writing down the elements in order between square brackets `[`, `]`, and separating them with commas `,`. An **empty list**, i.e., a list with no elements, is written as `[]`.

```
gap> [ 1, 2, 3 ];              # a list with three elements
[ 1, 2, 3 ]
gap> [ [], [ 1 ], [ 1, 2 ] ];  # a list may contain other lists
[ [  ], [ 1 ], [ 1, 2 ] ]
```

Each list constructed this way is mutable (see 12.6).

1 ▶ `IsList( obj )` C

tests whether *obj* is a list.

```
gap> IsList( [ 1, 3, 5, 7 ] );  IsList( 1 );
true
false
```

2 ▶ `IsDenseList( obj )` C

A list is **dense** if it has no holes, i.e., contains an element at every position up to the length. It is absolutely legal to have lists with holes. They are created by leaving the entry between the commas empty. Holes at the end of a list are ignored. Lists with holes are sometimes convenient when the list represents a mapping from a finite, but not consecutive, subset of the positive integers.

```
gap> IsDenseList( [ 1, 2, 3 ] );
true
gap> l := [ , 4, 9,, 25,, 49,,,, 121 ];;  IsDenseList( l );
false
gap> l[3];
9
gap> l[4];
List Element: <list>[4] must have an assigned value
not in any function
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' after assigning a value to continue
brk> l[4] := 16;;  # assigning a value
brk> return;       # to escape the break-loop
16
gap>
```

Observe that requesting the value of `l[4]`, which was not assigned, caused the entry of a `break`-loop (see Section 6.3). After assigning a value and typing `return;`, GAP is finally able to comply with our request (by responding with `16`).

3 ▶ `IsHomogeneousList(` *obj* `)` C

returns `true` if *obj* is a list and it is homogeneous, or `false` otherwise.

A **homogeneous** list is a dense list whose elements lie in the same family (see 13.1). The empty list is homogeneous but not a collection (see 28), a nonempty homogeneous list is also a collection.

```
gap> IsHomogeneousList( [ 1, 2, 3 ] );  IsHomogeneousList( [] );
true
true
gap> IsHomogeneousList( [ 1, false, () ] );
false
```

4 ▶ `IsTable(` *obj* `)` C

A **table** is a nonempty list of homogeneous lists which lie in the same family. Typical examples of tables are matrices (see 24).

```
gap> IsTable( [ [ 1, 2 ], [ 3, 4 ] ] );        # in fact a matrix
true
gap> IsTable( [ [ 1 ], [ 2, 3 ] ] );           # not rectangular but a table
true
gap> IsTable( [ [ 1, 2 ], [ () , (1,2) ] ] );  # not homogeneous
false
```

5 ▶ `IsConstantTimeAccessList(` *list* `)` C

This category indicates whether the access to each element of the list *list* will take roughly the same time. This is implied for example by `IsList and IsInternalRep`, so all strings, Boolean lists, ranges, and internally represented plain lists are in this category.

But also other enumerators (see 21.23) can lie in this category if they guarantee constant time access to their elements.

21.2 Basic Operations for Lists

The basic operations for lists are element access (see 21.3), assignment of elements to a list (see 21.4), fetching the length of a list (see 21.17.5), the test for a hole at a given position, and unbinding an element at a given position (see 21.5).

The term basic operation means that each other list operation can be formulated in terms of the basic operations. (But note that usually a more efficient method than this one is implemented.)

Any GAP object *list* in the category `IsList` (see 21.1.1) is regarded as a list, and if methods for the basic list operations are installed for *list* then *list* can be used also for the other list operations.

For internally represented lists, kernel methods are provided for the basic list operations. For other lists, it is possible to install appropriate methods for these operations. This permits the implementation of lists that do not need to store all list elements (see also 21.23); for example, the elements might be described by an algorithm, such as the elements list of a group. For this reduction of space requirements, however, a price in access time may have to be paid (see 21.17.6).

1 ▶ `\[\](` *list*`,` *pos* `)` O
▶ `IsBound\[\](` *list*`,` *pos* `)` O
▶ `\[\]\:\=(` *list*`,` *pos*`,` *val* `)` O
▶ `Unbind\[\](` *list*`,` *pos* `)` O

These operations implement element access, test for element boundedness, list element assignment, and removal of the element at position *pos*. In all cases, the index *pos* must be a positive integer.

Note that the special characters `[`, `]`, `:`, and `=` must be escaped with a backslash `\` (see 4.3); so `\[\]` denotes the operation for element access in a list, whereas `[]` denotes an empty list. (Maybe the variable names involving special characters look strange, but nevertheless they are quite suggestive.)

`\[\](` *list*`,` *pos* `)` is equivalent to *list*`[` *pos* `]`, which clearly will usually be preferred; the former is useful mainly if one wants to access the operation itself, for example if one wants to install a method for element access in a special kind of lists.

Similarly, `IsBound\[\]` is used explicitly mainly in method installations. In other situations, one can simply call `IsBound`, which then delegates to `IsBound\[\]` if the first argument is a list, and to `IsBound\.` if the first argument is a record.

Analogous statements hold for `\[\]\:\=` and `Unbind\[\]`.

21.3 List Elements

1 ▶ *list*`[` *pos* `]`

The above construct evaluates to the *pos*-th element of the list *list*. *pos* must be a positive integer. List indexing is done with origin 1, i.e., the first element of the list is the element at position 1.

```
gap> l := [ 2, 3, 5, 7, 11, 13 ];;  l[1];  l[2];  l[6];
2
3
13
```

If *list* is not a list, or *pos* does not evaluate to a positive integer, or *list*`[`*pos*`]` is unbound an error is signalled.

2 ▶ *list*`{` *poss* `}`

The above construct evaluates to a new list *new* whose first element is *list*`[`*poss*`[1]]`, whose second element is *list*`[`*poss*`[2]]`, and so on. *poss* must be a dense list of positive integers. However, it does not need to be sorted and may contain duplicate elements. If for any i, *list*`[` *poss*`[`i`]` `]` is unbound, an error is signalled.

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[4..6]};  l{[1,7,1,8]};
[ 7, 11, 13 ]
[ 2, 17, 2, 19 ]
```

The result is a **new** list, that is not identical to any other list. The elements of that list, however, are identical to the corresponding elements of the left operand (see 21.6).

It is possible to nest such **sublist extractions**, as can be seen in the following example.

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;  m{[1,2,3]}{[3,2]};
[ [ 3, 2 ], [ 6, 5 ], [ 9, 8 ] ]
gap> l := m{[1,2,3]};; l{[3,2]};
[ [ 7, 8, 9 ], [ 4, 5, 6 ] ]
```

Note the difference between the two examples. The latter extracts elements 1, 2, and 3 from m and then extracts the elements 3 and 2 from **this list**. The former extracts elements 1, 2, and 3 from m and then extracts the elements 3 and 2 from **each of those element lists**.

To be precise: With each selector `[`*pos*`]` or `{`*poss*`}` we associate a **level** that is defined as the number of selectors of the form `{`*poss*`}` to its left in the same expression. For example

```
     l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1     1      1     2
```

Then a selector *list*`[`*pos*`]` of level *level* is computed as `ListElement(`*list*`,`*pos*`,`*level*`)`, where `ListElement` is defined as follows. (Note that `ListElement` is **not** a GAP function.)

```
ListElement := function ( list, pos, level )
 if level = 0 then
  return list[pos];
 else
  return List( list, elm -> ListElement(elm,pos,level-1) );
 fi;
end;
```

and a selector *list*`{`*poss*`}` of level *level* is computed as `ListElements(`*list*`,`*poss*`,`*level*`)`, where `ListElements` is defined as follows. (Note that `ListElements` is **not** a GAP function.)

```
ListElements := function ( list, poss, level )
 if level = 0 then
  return list{poss};
  else
   return List( list, elm -> ListElements(elm,poss,level-1) );
  fi;
end;
```

3 ▶ `\{\}(` *list*, *poss* `)` O

This operation implements **sublist access**. For any list, the default method is to loop over the entries in the list *poss*, and to delegate to the element access operation. (For the somewhat strange variable name, cf. 21.2.)

21.4 List Assignment

1 ► *list*`[` *pos* `] :=` *object*`;`

The list element assignment assigns the object *object*, which can be of any type, to the list entry at the position *pos*, which must be a positive integer, in the mutable (see 12.6) list *list*. That means that accessing the *pos*-th element of the list *list* will return *object* after this assignment.

```
gap> l := [ 1, 2, 3 ];;
gap> l[1] := 3;; l;             # assign a new object
[ 3, 2, 3 ]
gap> l[2] := [ 4, 5, 6 ];; l;   # <object> may be of any type
[ 3, [ 4, 5, 6 ], 3 ]
gap> l[ l[1] ] := 10;; l;       # <index> may be an expression
[ 3, [ 4, 5, 6 ], 10 ]
```

If the index *pos* is larger than the length of the list *list* (see 21.17.5), the list is automatically enlarged to make room for the new element. Note that it is possible to generate lists with holes that way.

```
gap> l[4] := "another entry";; l;  # <list> is enlarged
[ 3, [ 4, 5, 6 ], 10, "another entry" ]
gap> l[ 10 ] := 1;; l;             # now <list> has a hole
[ 3, [ 4, 5, 6 ], 10, "another entry",,,,,, 1 ]
```

The function `Add` (see 21.4.4) should be used if you want to add an element to the end of the list.

Note that assigning to a list changes the list, thus this list must be mutable (see 12.6). See 21.6 for subtleties of changing lists.

If *list* does not evaluate to a list, *pos* does not evaluate to a positive integer or *object* is a call to a function which does not return a value (for example `Print`) an error is signalled.

2 ► *list*`{` *poss* `} :=` *objects*`;`

The sublist assignment assigns the object *objects*`[1]`, which can be of any type, to the list *list* at the position *poss*`[1]`, the object *objects*`[2]` to *list*`[`*poss*`[2]]`, and so on. *poss* must be a dense list of positive integers, it need, however, not be sorted and may contain duplicate elements. *objects* must be a dense list and must have the same length as *poss*.

```
gap> l := [ 2, 3, 5, 7, 11, 13, 17, 19 ];;
gap> l{[1..4]} := [10..13];; l;
[ 10, 11, 12, 13, 11, 13, 17, 19 ]
gap> l{[1,7,1,10]} := [ 1, 2, 3, 4 ];; l;
[ 3, 11, 12, 13, 11, 13, 2, 19,, 4 ]
```

It is possible to nest such sublist assignments, as can be seen in the following example.

```
gap> m := [ [1,2,3], [4,5,6], [7,8,9], [10,11,12] ];;
gap> m{[1,2,3]}{[3,2]} := [ [11,12], [13,14], [15,16] ];; m;
[ [ 1, 12, 11 ], [ 4, 14, 13 ], [ 7, 16, 15 ], [ 10, 11, 12 ] ]
```

The exact behaviour is defined in the same way as for list extractions (see 21.3). Namely with each selector `[`*pos*`]` or `{`*poss*`}` we associate a **level** that is defined as the number of selectors of the form `{`*poss*`}` to its left in the same expression. For example

```
      l[pos1]{poss2}{poss3}[pos4]{poss5}[pos6]
level   0      0      1     1      1     2
```

Then a list assignment *list*`[`*pos*`] := `*vals*`;` of level *level* is computed as `ListAssignment(` *list*`,` *pos*`,` *vals*`,` *level* `)`, where `ListAssignment` is defined as follows. (Note that `ListAssignment` is **not** a GAP function.)

```
ListAssignment := function ( list, pos, vals, level )
 local i;
 if level = 0 then
  list[pos] := vals;
 else
  for i in [1..Length(list)] do
   ListAssignment( list[i], pos, vals[i], level-1 );
  od;
 fi;
end;
```

and a list assignment *list*`{`*poss*`} := `*vals* of level *level* is computed as `ListAssignments(` *list*`,` *poss*`,` *vals*`,` *level* `)`, where `ListAssignments` is defined as follows. (Note that `ListAssignments` is **not** a GAP function.)

```
ListAssignments := function ( list, poss, vals, level )
 local i;
 if level = 0 then
  list{poss} := vals;
 else
  for i in [1..Length(list)] do
   ListAssignments( list[i], poss, vals[i], level-1 );
  od;
 fi;
end;
```

3 ► `\{\}\:\=(` *list*`,` *poss*`,` *val* `)` O

This operation implements sublist assignment. For any list, the default method is to loop over the entries in the list *poss*, and to delegate to the element assignment operation. (For the somewhat strange variable name, cf. 21.2.)

4 ► `Add(` *list*`,` *obj* `)` O

adds the element *obj* to the end of the mutable list *list*, i.e., it is equivalent to the assignment *list*`[ Length(`*list*`) + 1 ] := `*obj*, see 21.4.1. Nothing is returned by `Add`, the function is only called for its side effect.

```
gap> l := [ 2, 3, 5 ];; Add( l, 7 ); l;
[ 2, 3, 5, 7 ]
```

5 ► `Append(` *list1*`,` *list2* `)` O

adds the elements of the list *list2* to the end of the mutable list *list1*, see 21.4.2. *list2* may contain holes, in which case the corresponding entries in *list1* will be left unbound. `Append` returns nothing, it is only called for its side effect.

Note that `Append` changes its first argument, while `Concatenation` (see 21.20.1) creates a new list and leaves its arguments unchanged.

```
gap> l := [ 2, 3, 5 ];; Append( l, [ 7, 11, 13 ] ); l;
[ 2, 3, 5, 7, 11, 13 ]
gap> Append( l, [ 17,, 23 ] ); l;
[ 2, 3, 5, 7, 11, 13, 17,, 23 ]
```

21.5 IsBound and Unbind for Lists

1 ▶ `IsBound( `*list*`[`*n*`] )` M

`IsBound` returns `true` if the list *list* has a element at the position *n*, and `false` otherwise. *list* must evaluate to a list, otherwise an error is signalled.

```
gap> l := [ , 2, 3, , 5, , 7, , , , 11 ];;
gap> IsBound( l[7] );
true
gap> IsBound( l[4] );
false
gap> IsBound( l[101] );
false
```

2 ▶ `Unbind( `*list*`[`*n*`] )` M

`Unbind` deletes the element at the position *n* in the mutable list *list*. That is, after execution of `Unbind`, *list* no longer has an assigned value at the position *n*. Thus `Unbind` can be used to produce holes in a list. Note that it is not an error to unbind a nonexisting list element. *list* must evaluate to a list, otherwise an error is signalled.

```
gap> l := [ , 2, 3, 5, , 7, , , , 11 ];;
gap> Unbind( l[3] ); l;
[ , 2,, 5,, 7,,,, 11 ]
gap> Unbind( l[4] ); l;
[ , 2,,,, 7,,,, 11 ]
```

Note that `IsBound` and `Unbind` are special in that they do not evaluate their argument, otherwise `IsBound` would always signal an error when it is supposed to return `false` and there would be no way to tell `Unbind` which component to remove.

21.6 Identical Lists

With the list assignment (see 21.4) it is possible to change a mutable list. This section describes the semantic consequences of this fact. (See also 12.5.)

First we define what it means when we say that "an object is changed". You may think that in the following example the second assignment changes the integer.

```
i := 3;
i := i + 1;
```

But in this example it is not the **integer** `3` which is changed, by adding one to it. Instead the **variable** `i` is changed by assigning the value of `i+1`, which happens to be `4`, to `i`. The same thing happens in the following example

```
l := [ 1, 2 ];
l := [ 1, 2, 3 ];
```

The second assignment does not change the first list, instead it assigns a new list to the variable `l`. On the other hand, in the following example the list **is** changed by the second assignment.

```
l := [ 1, 2 ];
l[3] := 3;
```

To understand the difference, think of a variable as a name for an object. The important point is that a list can have several names at the same time. An assignment *var*`:=`*list*`;` means in this interpretation that *var*

is a name for the object *list*. At the end of the following example `l2` still has the value `[ 1, 2 ]` as this list has not been changed and nothing else has been assigned to it.

```
l1 := [ 1, 2 ];
l2 := l1;
l1 := [ 1, 2, 3 ];
```

But after the following example the list for which `l2` is a name has been changed and thus the value of `l2` is now `[ 1, 2, 3 ]`.

```
l1 := [ 1, 2 ];
l2 := l1;
l1[3] := 3;
```

We say that two lists are **identical** if changing one of them by a list assignment also changes the other one. This is slightly incorrect, because if **two** lists are identical, there are actually only two names for **one** list. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical lists must be equal, because there is only one list with two different names. Thus identity is an equivalence relation that is a refinement of equality. Identity of objects can be detected using `IsIdenticalObj`, see 12.5.

Let us now consider under which circumstances two lists are identical.

If you enter a list literal then the list denoted by this literal is a new list that is not identical to any other list. Thus in the following example `l1` and `l2` are not identical, though they are equal of course.

```
l1 := [ 1, 2 ];
l2 := [ 1, 2 ];
```

Also in the following example, no lists in the list `l` are identical.

```
l := [];
for i in [1..10] do l[i] := [ 1, 2 ]; od;
```

If you assign a list to a variable no new list is created. Thus the list value of the variable on the left hand side and the list on the right hand side of the assignment are identical. So in the following example `l1` and `l2` are identical lists.

```
l1 := [ 1, 2 ];
l2 := l1;
```

If you pass a list as an argument, the old list and the argument of the function are identical. Also if you return a list from a function, the old list and the value of the function call are identical. So in the following example `l1` and `l2` are identical lists:

```
l1 := [ 1, 2 ];
f := function ( l ) return l; end;
l2 := f( l1 );
```

If you change a list it keeps its identity. Thus if two lists are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two lists that are not identical will never become identical if you change one of them. So in the following example both `l1` and `l2` are changed, and are still identical.

```
l1 := [ 1, 2 ];
l2 := l1;
l1[1] := 2;
```

21.7 Duplication of Lists

Here we describe the meaning of `ShallowCopy` and `StructuralCopy` for lists. For the general definition of these functions, see 12.7.

The subobjects (see 12.7.1) of a list are exactly its elements.

This means that for any list *list*, `ShallowCopy` returns a mutable **new** list *new* that is **not identical** to any other list (see 21.6), and whose elements are identical to the elements of *list*.

Analogously, for a **mutable** list *list*, `StructuralCopy` returns a mutable **new** list *scp* that is **not identical** to any other list, and whose elements are structural copies (defined recursively) of the elements of *list*; an element of *scp* is mutable (and then a **new** list) if and only if the corresponding element of *list* is mutable.

In both cases, modifying the copy *new* resp. *scp* by assignments (see 21.4) does not modify the original object *list*.

`ShallowCopy` basically executes the following code for lists.

```
new := [];
for i in [ 1 .. Length( list ) ] do
  if IsBound( list[i] ) then
    new[i] := list[i];
  fi;
od;

gap> list1 := [ [ 1, 2 ], [ 3, 4 ] ];;  list2 := ShallowCopy( list1 );;
gap> IsIdenticalObj( list1, list2 );
false
gap> IsIdenticalObj( list1[1], list2[1] );
true
gap> list2[1] := 0;;  list1;  list2;
[ [ 1, 2 ], [ 3, 4 ] ]
[ 0, [ 3, 4 ] ]
```

`StructuralCopy` basically executes the following code for lists.

```
new := [];
for i in [ 1 .. Length( list ) ] do
  if IsBound( list[i] ) then
    new[i] := StructuralCopy( list[i] );
  fi;
od;

gap> list1 := [ [ 1, 2 ], [ 3, 4 ] ];;  list2 := StructuralCopy( list1 );;
gap> IsIdenticalObj( list1, list2 );
false
gap> IsIdenticalObj( list1[1], list2[1] );
false
gap> list2[1][1] := 0;;  list1;  list2;
[ [ 1, 2 ], [ 3, 4 ] ]
[ [ 0, 2 ], [ 3, 4 ] ]
```

The above code is not entirely correct. If the object *list* contains a mutable object twice this object is not copied twice, as would happen with the above definition, but only once. This means that the copy *new* and the object *list* have exactly the same structure when viewed as a general graph.

```
gap> sub := [ 1, 2 ];; list1 := [ sub, sub ];;
gap> list2 := StructuralCopy( list1 );
[ [ 1, 2 ], [ 1, 2 ] ]
gap> list2[1][1] := 0;; list2;
[ [ 0, 2 ], [ 0, 2 ] ]
gap> list1;
[ [ 1, 2 ], [ 1, 2 ] ]
```

21.8 Membership Test for Lists

1► *obj* `in` *list*

tests whether there is a positive integer *index* such that *list*`[` *index* `] =` *obj*.

If the list *list* knows that it is strictly sorted (see 21.17.4), the membership test is much quicker, because a binary search can be used instead of the linear search used for arbitrary lists.

```
gap> 1 in [ 2, 2, 1, 3 ];  1 in [ 4, -1, 0, 3 ];
true
false
gap> s := SSortedList( [2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32] );;
gap> 17 in s;  # uses binary search and only 4 comparisons
false
```

For finding the position of an element in a list, see 21.16.

21.9 Enlarging Internally Represented Lists

Section 21.4 told you (among other things) that it is possible to assign beyond the logical end of a mutable list, automatically enlarging the list. This section tells you how this is done for internally represented lists.

It would be extremely wasteful to make all lists large enough so that there is room for all assignments, because some lists may have more than 100000 elements, while most lists have less than 10 elements.

On the other hand suppose every assignment beyond the end of a list would be done by allocating new space for the list and copying all entries to the new space. Then creating a list of 1000 elements by assigning them in order, would take half a million copy operations and also create a lot of garbage that the garbage collector would have to reclaim.

So the following strategy is used. If a list is created it is created with exactly the correct size. If a list is enlarged, because of an assignment beyond the end of the list, it is enlarged by at least *length*`/8 + 4` entries. Therefore the next assignments beyond the end of the list do not need to enlarge the list. For example creating a list of 1000 elements by assigning them in order, would now take only 32 enlargements.

The result of this is of course that the **physical length** of a list may be larger than the **logical length**, which is usually called simply the length of the list. Aside from the implications for the performance you need not be aware of the physical length. In fact all you can ever observe, for example by calling `Length` (see 21.17.5), is the logical length.

Suppose that `Length` would have to take the physical length and then test how many entries at the end of a list are unassigned, to compute the logical length of the list. That would take too much time. In order to make `Length`, and other functions that need to know the logical length, more efficient, the length of a list is stored along with the list.

21.10 Comparisons of Lists

1 ▶ *list1* `=` *list2*
▶ *list1* `<>` *list2*

Two lists *list1* and *list2* are equal if and only if for every index i, either both entries *list1*`[`i`]` and *list2*`[`i`]` are unbound, or both are bound and are equal, i.e., *list1*`[`i`] =` *list2*`[`i`]` is `true`.

```
gap> [ 1, 2, 3 ] = [ 1, 2, 3 ];
true
gap> [ , 2, 3 ] = [ 1, 2, ];
false
gap> [ 1, 2, 3 ] = [ 3, 2, 1 ];
false
```

This definition will cause problems with lists which are their own entries. Comparing two such lists for equality may lead to an infinite recursion in the kernel if the list comparison has to compare the list entries which are in fact the lists themselves, and then GAP crashes.

2 ▶ *list1* `<` *list2*
▶ *list1* `<=` *list2*

Lists are ordered **lexicographically**. Unbound entries are smaller than any bound entry. That implies the following behaviour. Let i be the smallest positive integer i such that *list1* and *list2* at position i differ, i.e., either exactly one of *list1*`[i]`, *list2*`[i]` is bound or both entries are bound and differ. Then *list1* is less than *list2* if either *list1*`[`i`]` is unbound (and *list2*`[`i`]` is not) or both are bound and *list1*`[`i`] <` *list2*`[`i`]` is `true`.

```
gap> [ 1, 2, 3, 4 ] < [ 1, 2, 4, 8 ]; # <list1>[3] < <list2>[3]
true
gap> [ 1, 2, 3 ] < [ 1, 2, 3, 4 ];    # <list1>[4] is unbound and therefore very small
true
gap> [ 1, , 3, 4 ] < [ 1, 2, 3 ];     # <list1>[2] is unbound and therefore very small
true
```

Note that for comparing two lists with `<` or `<=`, the (relevant) list elements must be comparable with `<`, which is usually **not** the case for objects in different families, see 13.1. Also for the possibility to compare lists with other objects, see 13.1.

21.11 Arithmetic for Lists

It is convenient to have arithmetic operations for lists, in particular because in GAP row vectors and matrices are special kinds of lists. However, it is the wide variety of list objects because of which we prescribe arithmetic operations **not for all** of them. (Keep in mind that "list" means just an object in the category `IsList`, see 21.1.1.)

(Due to the intended generality and flexibility, the definitions given in the following sections are quite technical. But for not too complicated cases such as matrices (see 24.2) and row vectors (see 23.1) whose entries aren't lists, the resulting behaviour should be intuitive.)

For example, we want to deal with matrices which can be added and multiplied in the usual way, via the infix operators `+` and `*`; and we want also Lie matrices, with the same additive behaviour but with the multiplication defined by the Lie bracket. Both kinds of matrices shall be lists, with the usual access to their rows, with `Length` (see 21.17.5) returning the number of rows etc.

For the categories and attributes that control the arithmetic behaviour of lists, see 21.12.

For the definition of return values of additive and multiplicative operations whose arguments are lists in these filters, see 21.13 and 21.14, respectively. It should be emphasized that these sections describe only what the return values are, and not how they are computed.

For the mutability status of the return values, see 21.15. (Note that this is not dealt with in the sections about the result values.)

Further details about the special cases of row vectors and matrices can be found in 23.1 and in 24.2, the compression status is dealt with in 23.2 and 24.13.

21.12 Filters Controlling the Arithmetic Behaviour of Lists

The arithmetic behaviour of lists is controlled by their types. The following categories and attributes are used for that.

Note that we distinguish additive and multiplicative behaviour. For example, Lie matrices have the usual additive behaviour but not the usual multiplicative behaviour.

1 ► `IsGeneralizedRowVector(` *list* `)` C

For a list *list*, the value `true` for `IsGeneralizedRowVector` indicates that the additive arithmetic behaviour of *list* is as defined in 21.13, and that the attribute `NestingDepthA` (see 21.12.4) will return a nonzero value when called with *list*.

2 ► `IsMultiplicativeGeneralizedRowVector(` *list* `)` C

For a list *list*, the value `true` for `IsMultiplicativeGeneralizedRowVector` indicates that the multiplicative arithmetic behaviour of *list* is as defined in 21.14, and that the attribute `NestingDepthM` (see 21.12.5) will return a nonzero value when called with *list*.

Note that these filters do **not** enable default methods for addition or multiplication (cf. 21.12.3).

```
gap> IsList( "abc" ); IsGeneralizedRowVector( "abc" );
true
false
gap> liemat:= LieObject( [ [ 1, 2 ], [ 3, 4 ] ] );
LieObject( [ [ 1, 2 ], [ 3, 4 ] ] )
gap> IsGeneralizedRowVector( liemat );
true
gap> IsMultiplicativeGeneralizedRowVector( liemat );
false
gap> bas:= CanonicalBasis( FullRowSpace( Rationals, 3 ) );
CanonicalBasis( ( Rationals^3 ) )
gap> IsMultiplicativeGeneralizedRowVector( bas );
true
```

3 ► `IsListDefault(` *list* `)` C

For a list *list*, `IsListDefault` indicates that the default methods for arithmetic operations of lists, such as pointwise addition and multiplication as inner product or matrix product, shall be applicable to *list*.

`IsListDefault` implies `IsGeneralizedRowVector` and `IsMultiplicativeGeneralizedRowVector`.

All internally represented lists are in this category, and also all lists in the representations `IsGF2VectorRep`, `Is8BitVectorRep`, `IsGF2MatrixRep`, and `Is8BitMatrixRep` (see 23.2 and 24.13). Note that the result of an arithmetic operation with lists in `IsListDefault` will in general be an internally represented list, so most "wrapped list objects" will not lie in `IsListDefault`.

```
gap> v:= [ 1, 2 ];;  m:= [ v, 2*v ];;
gap> IsListDefault( v );  IsListDefault( m );
true
true
gap> IsListDefault( bas );  IsListDefault( liemat );
true
false
```

4 ► `NestingDepthA( obj )` A

For a GAP object *obj*, `NestingDepthA` returns the **additive nesting depth** of *obj*. This is defined recursively as the integer 0 if *obj* is not in `IsGeneralizedRowVector`, as the integer 1 if *obj* is an empty list in `IsGeneralizedRowVector`, and as 1 plus the additive nesting depth of the first bound entry in *obj* otherwise.

5 ► `NestingDepthM( obj )` A

For a GAP object *obj*, `NestingDepthM` returns the **multiplicative nesting depth** of *obj*. This is defined recursively as the integer 0 if *obj* is not in `IsMultiplicativeGeneralizedRowVector`, as the integer 1 if *obj* is an empty list in `IsMultiplicativeGeneralizedRowVector`, and as 1 plus the multiplicative nesting depth of the first bound entry in *obj* otherwise.

```
gap> NestingDepthA( v );  NestingDepthM( v );
1
1
gap> NestingDepthA( m );  NestingDepthM( m );
2
2
gap> NestingDepthA( liemat );  NestingDepthM( liemat );
2
0
gap> l1:= [ [ 1, 2 ], 3 ];;  l2:= [ 1, [ 2, 3 ] ];;
gap> NestingDepthA( l1 );  NestingDepthM( l1 );
2
2
gap> NestingDepthA( l2 );  NestingDepthM( l2 );
1
1
```

21.13 Additive Arithmetic for Lists

In this general context, we define the results of additive operations only in the following situations. For unary operations (zero and additive inverse), the unique argument must be in `IsGeneralizedRowVector`; for binary operations (addition and subtraction), at least one argument must be in `IsGeneralizedRowVector`, and the other either is not a list or also in `IsGeneralizedRowVector`.

(For non-list GAP objects, defining the results of unary operations is not an issue here, and if at least one argument is a list not in `IsGeneralizedRowVector`, it shall be left to this argument whether the result in question is defined and what it is.)

Zero

The zero (see 30.10.3) of a list x in `IsGeneralizedRowVector` is defined as the list whose entry at position i is the zero of $x[i]$ if this entry is bound, and is unbound otherwise.

```
gap> Zero( [ 1, 2, 3 ] );  Zero( [ [ 1, 2 ], 3 ] );  Zero( liemat );
[ 0, 0, 0 ]
[ [ 0, 0 ], 0 ]
LieObject( [ [ 0, 0 ], [ 0, 0 ] ] )
```

AdditiveInverse

The additive inverse (see 30.10.9) of a list x in `IsGeneralizedRowVector` is defined as the list whose entry at position i is the additive inverse of $x[i]$ if this entry is bound, and is unbound otherwise.

```
gap> AdditiveInverse( [ 1, 2, 3 ] );  AdditiveInverse( [ [ 1, 2 ], 3 ] );
[ -1, -2, -3 ]
[ [ -1, -2 ], -3 ]
```

Addition

If x and y are in `IsGeneralizedRowVector` and have the same additive nesting depth (see 21.12.4), the sum $x + y$ is defined **pointwise**, in the sense that the result is a list whose entry at position i is $x[i] + y[i]$ if these entries are bound, is a shallow copy (see 12.7.1) of $x[i]$ or $y[i]$ if the other argument is not bound at position i, and is unbound if both x and y are unbound at position i.

If x is in `IsGeneralizedRowVector` and y is in `IsGeneralizedRowVector` and has lower additive nesting depth, or is neither a list nor a domain, the sum $x+y$ is defined as a list whose entry at position i is $x[i]+y$ if x is bound at position i, and is unbound if not. The equivalent holds in the reversed case, where the order of the summands is kept, as addition is not always commutative.

```
gap> 1 + [ 1, 2, 3 ];  [ 1, 2, 3 ] + [ 0, 2, 4 ];  [ 1, 2 ] + [ Z(2) ];
[ 2, 3, 4 ]
[ 1, 4, 7 ]
[ 0*Z(2), 2 ]
gap> l1:= [ 1, , 3, 4 ];;             l2:= [ , 2, 3, 4, 5 ];;
gap> l3:= [ [ 1, 2 ], , [ 5, 6 ] ];;  l4:= [ , [ 3, 4 ], [ 5, 6 ] ];;
gap> NestingDepthA( l1 );  NestingDepthA( l2 );
1
1
gap> NestingDepthA( l3 );  NestingDepthA( l4 );
2
2
gap> l1 + l2;
[ 1, 2, 6, 8, 5 ]
gap> l1 + l3;
[ [ 2, 2, 3, 4 ],, [ 6, 6, 3, 4 ] ]
gap> l2 + l4;
[ , [ 3, 6, 3, 4, 5 ], [ 5, 8, 3, 4, 5 ] ]
gap> l3 + l4;
[ [ 1, 2 ], [ 3, 4 ], [ 10, 12 ] ]
gap> l1 + [];
[ 1,, 3, 4 ]
```

Subtraction

For two GAP objects x and y of which one is in `IsGeneralizedRowVector` and the other is also in `IsGeneralizedRowVector` or is neither a list nor a domain, $x - y$ is defined as $x + (-y)$.

```
gap> l1 - l2;
[ 1, -2, 0, 0, -5 ]
gap> l1 - l3;
[ [ 0, -2, 3, 4 ],, [ -4, -6, 3, 4 ] ]
gap> l2 - l4;
[ , [ -3, -2, 3, 4, 5 ], [ -5, -4, 3, 4, 5 ] ]
gap> l3 - l4;
[ [ 1, 2 ], [ -3, -4 ], [ 0, 0 ] ]
gap> l1 - [];
[ 1,, 3, 4 ]
```

21.14 Multiplicative Arithmetic for Lists

In this general context, we define the results of multiplicative operations only in the following situations. For unary operations (one and inverse), the unique argument must be in `IsMultiplicativeGeneralizedRowVector`; for binary operations (multiplication and division), at least one argument must be in `IsMultiplicativeGeneralizedRowVector`, and the other either not a list or also in `IsMultiplicativeGeneralizedRowVector`.

(For non-list GAP objects, defining the results of unary operations is not an issue here, and if at least one argument is a list not in `IsMultiplicativeGeneralizedRowVector`, it shall be left to this argument whether the result in question is defined and what it is.)

One

The one (see 30.10.2) of a dense list x in `IsMultiplicativeGeneralizedRowVector` such that x has even multiplicative nesting depth and has the same length as each of its rows is defined as the usual identity matrix on the outer two levels, that is, an identity matrix of the same dimensions, with diagonal entries `One`$(x[1][1])$ and off-diagonal entries `Zero`$(x[1][1])$.

```
gap> One( [ [ 1, 2 ], [ 3, 4 ] ] );
[ [ 1, 0 ], [ 0, 1 ] ]
gap> One( [ [ [ [ 1 ] ], [ [ 2 ] ] ], [ [ [ 3 ] ], [ [ 4 ] ] ] ] );
[ [ [ [ 1 ] ], [ [ 0 ] ] ], [ [ [ 0 ] ], [ [ 1 ] ] ] ]
```

Inverse

The inverse (see 30.10.8) of an invertible square table x in `IsMultiplicativeGeneralizedRowVector` whose entries lie in a common field is defined as the usual inverse y, i.e., a square matrix over the same field such that xy and yx is equal to `One`(x).

```
gap> Inverse( [ [ 1, 2 ], [ 3, 4 ] ] );
[ [ -2, 1 ], [ 3/2, -1/2 ] ]
```

Multiplication

There are three possible computations that might be triggered by a multiplication involving a list in `IsMultiplicativeGeneralizedRowVector`. Namely, $x * y$ might be

(I) the inner product $x[1] * y[1] + x[2] * y[2] + \cdots + x[n] * y[n]$, where summands are omitted for which the entry in x or y is unbound (if this leaves no summand then the multiplication is an error), or

(L) the left scalar multiple, i.e., a list whose entry at position i is $x * y[i]$ if y is bound at position i, and is unbound if not, or

(R) the right scalar multiple, i.e., a list whose entry at position i is $x[i] * y$ if x is bound at position i, and is unbound if not.

Our aim is to generalize the basic arithmetic of simple row vectors and matrices, so we first summarize the situations that shall be covered.

```
      scl   vec   mat
--------------------
scl         (L)   (L)
vec   (R)   (I)   (I)
mat   (R)   (R)   (R)
```

This means for example that the product of a scalar (scl) with a vector (vec) or a matrix (mat) is computed according to (L). Note that this is asymmetric.

Now we can state the general multiplication rules.

If exactly one argument is in `IsMultiplicativeGeneralizedRowVector` then we regard the other argument (which is then neither a list nor a domain) as a scalar, and specify result (L) or (R), depending on ordering.

In the remaining cases, both x and y are in `IsMultiplicativeGeneralizedRowVector`, and we distinguish the possibilities by their multiplicative nesting depths. An argument with **odd** multiplicative nesting depth is regarded as a vector, and an argument with **even** multiplicative nesting depth is regarded as a scalar or a matrix.

So if both arguments have odd multiplicative nesting depth, we specify result (I).

If exactly one argument has odd nesting depth, the other is treated as a scalar if it has lower multiplicative nesting depth, and as a matrix otherwise. In the former case, we specify result (L) or (R), depending on ordering; in the latter case, we specify result (L) or (I), depending on ordering.

We are left with the case that each argument has even multiplicative nesting depth. If the two depths are equal, we treat the computation as a matrix product, and specify result (R). Otherwise, we treat the less deeply nested argument as a scalar and the other as a matrix, and specify result (L) or (R), depending on ordering.

```
gap> [ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ] * (1,4);
[ (1,4), (1,4)(2,3), (1,2,4), (1,2,3,4), (1,3,2,4), (1,3,4) ]
gap> [ 1, 2, , 4 ] * 2;
[ 2, 4,, 8 ]
gap> [ 1, 2, 3 ] * [ 1, 3, 5, 7 ];
22
gap> m:= [ [ 1, 2 ], 3 ];;  m * m;
[ [ 7, 8 ], [ [ 3, 6 ], 9 ] ]
gap> m * m = [ m[1] * m, m[2] * m ];
true
gap> n:= [ 1, [ 2, 3 ] ];;  n * n;
14
gap> n * n = n[1] * n[1] + n[2] * n[2];
true
```

Division

For two GAP objects x and y of which one is in `IsMultiplicativeGeneralizedRowVector` and the other is also in `IsMultiplicativeGeneralizedRowVector` or is neither a list nor a domain, x/y is defined as $x * y^{-1}$.

```
gap> [ 1, 2, 3 ] / 2;  [ 1, 2 ] / [ [ 1, 2 ], [ 3, 4 ] ];
[ 1/2, 1, 3/2 ]
[ 1, 0 ]
```

mod

If x and y are in `IsMultiplicativeGeneralizedRowVector` and have the same multiplicative nesting depth (see 21.12.5), x`mod`y is defined **pointwise**, in the sense that the result is a list whose entry at position i is $x[i]$`mod`$y[i]$ if these entries are bound, is a shallow copy (see 12.7.1) of $x[i]$ or $y[i]$ if the other argument is not bound at position i, and is unbound if both x and y are unbound at position i.

If x is in `IsMultiplicativeGeneralizedRowVector` and y is in `IsMultiplicativeGeneralizedRowVector` and has lower multiplicative nesting depth or is neither a list nor a domain, x`mod`y is defined as a list whose entry at position i is $x[i]$`mod`y if x is bound at position i, and is unbound if not. The equivalent holds in the reversed case, where the order of the arguments is kept.

```
gap> 4711 mod [ 2, 3,, 5, 7 ];
[ 1, 1,, 1, 0 ]
gap> [ 2, 3, 4, 5, 6 ] mod 3;
[ 2, 0, 1, 2, 0 ]
gap> [ 10, 12, 14, 16 ] mod [ 3, 5, 7 ];
[ 1, 2, 0, 16 ]
```

Left Quotient

For two GAP objects x and y of which one is in `IsMultiplicativeGeneralizedRowVector` and the other is also in `IsMultiplicativeGeneralizedRowVector` or is neither a list nor a domain, `LeftQuotient`(x, y) is defined as $x^{-1} * y$.

```
gap> LeftQuotient( [ [ 1, 2 ], [ 3, 4 ] ], [ 1, 2 ] );
[ 0, 1/2 ]
```

21.15 Mutability Status and List Arithmetic

Many results of arithmetic operations, when applied to lists, are again lists, and it is of interest whether their entries are mutable or not (if applicable). Note that the mutability status of the result itself is already defined by the general rule for any result of an arithmetic operation, not only for lists (see 12.6).

However, we do **not** define exactly the mutability status for each element on each level of a nested list returned by an arithmetic operation. (Of course it would be possible to define this recursively, but since the methods used are in general not recursive, in particular for efficient multiplication of compressed matrices, such a general definition would be a burden in these cases.) Instead we consider, for a list x in `IsGeneralizedRowVector`, the sequence $x = x_1, x_2, \ldots x_n$ where x_{i+1} is the first bound entry in x_i if exists (that is, if x_i is a nonempty list), and n is the largest i such that x_i lies in `IsGeneralizedRowVector`. The **immutability level** of x is defined as infinity if x is immutable, and otherwise the number of x_i which are immutable. (So the immutability level of a mutable empty list is 0.)

Thus a fully mutable matrix has immutability level 0, and a mutable matrix with immutable first row has immutability level 1 (independent of the mutability of other rows).

The immutability level of the result of any of the binary operations discussed here is the minimum of the immutability levels of the arguments, provided that objects of the required mutability status exist in GAP.

Moreover, the results have a "homogeneous" mutability status, that is, if the first bound entry at nesting depth i is immutable (mutable) then all entries at nesting depth i are immutable (mutable, provided that a mutable version of this entry exists in GAP).

Thus the sum of two mutable matrices whose first rows are mutable is a matrix all of whose rows are mutable, and the product of two matrices whose first rows are immutable is a matrix all of whose rows are immutable, independent of the mutability status of the other rows of the arguments.

For example, the sum of a matrix (mutable or immutable, i.e., of immutability level one of 0, 1, or 2) and a mutable row vector (i.e., immutability level 0) is a fully mutable matrix. The product of two mutable row vectors of integers is an integer, and since GAP does not support mutable integers, the result is immutable.

For unary arithmetic operations, there are three operations available, an attribute that returns an immutable result (`Zero`, `AdditiveInverse`, `One`, `Inverse`), an operation that returns a result that is mutable (`ZeroOp`, `AdditiveInverseOp`, `OneOp`, `InverseOp`), and an operation whose result has the same immutability level as the argument (`ZeroSM`, `AdditiveInverseSM`, `OneSM`, `InverseSM`). The last kind of operations is equivalent to the corresponding infix operations `0 *` *list*, `-` *list*, *list*`^0`, and *list*`^-1`. (This holds not only for lists, see 12.6.)

```
gap> IsMutable( l1 );  IsMutable( 2 * Immutable( [ 1, 2, 3 ] ) );
true
false
gap> IsMutable( l2 );  IsMutable( l3 );
true
true
```

An example motivating the mutability rule is the use of syntactic constructs such as *obj* `*` *list* and `-` *list* as an elegant and efficient way to create mutable lists needed for further manipulations from mutable lists. In particular one can construct a mutable zero vector of length n by `0 * [ 1 .. ` n `]`. The latter can be done also using `ListWithIdenticalEntries`.

1 ▶ `ListWithIdenticalEntries(` *n*`,` *obj* `)` F

is a list *list* of length n that has the object *obj* stored at each of the positions from 1 to n. Note that all elements of *lists* are identical, see 21.6.

```
gap> ListWithIdenticalEntries( 10, 0 );
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

21.16 Finding Positions in Lists

1 ▶ `Position(` *list*`,` *obj*`[,` *from*`] )` O

returns the position of the first occurrence *obj* in *list*, or *fail* if *obj* is not contained in *list*. If a starting index *from* is given, it returns the position of the first occurrence starting the search **after** position *from*.

Each call to the two argument version is translated into a call of the three argument version, with third argument the integer zero `0`. (Methods for the two argument version must be installed as methods for the version with three arguments, the third being described by `IsZeroCyc`.)

```
gap> Position( [ 2, 2, 1, 3 ], 1 );
3
gap> Position( [ 2, 1, 1, 3 ], 1 );
2
gap> Position( [ 2, 1, 1, 3 ], 1, 2 );
3
gap> Position( [ 2, 1, 1, 3 ], 1, 3 );
fail
```

2 ▶ `PositionCanonical(` *list*`,` *obj* `)` O

returns the position of the canonical associate of *obj* in *list*. The definition of this associate depends on *list*. For internally represented lists it is defined as the element itself (and `PositionCanonical` thus defaults to

`Position`, see 21.16.1), but for example for certain enumerators (see 21.23) other canonical associates can be defined.

For example `RightTransversal` defines the canonical associate to be the element in the transversal defining the same coset of a subgroup in a group.

```
gap> g:=Group((1,2,3,4),(1,2));;u:=Subgroup(g,[(1,2)(3,4),(1,3)(2,4)]);;
gap> rt:=RightTransversal(g,u);;AsList(rt);
[ (), (3,4), (2,3), (2,3,4), (2,4,3), (2,4) ]
gap> Position(rt,(1,2));
fail
gap> PositionCanonical(rt,(1,2));
2
```

3 ▶ `PositionNthOccurrence(` *list*, *obj*, *n* `)` O

returns the position of the n-th occurrence of *obj* in *list* and returns `fail` if *obj* does not occur n times.

```
gap> PositionNthOccurrence([1,2,3,2,4,2,1],1,1);
1
gap> PositionNthOccurrence([1,2,3,2,4,2,1],1,2);
7
gap> PositionNthOccurrence([1,2,3,2,4,2,1],2,3);
6
gap> PositionNthOccurrence([1,2,3,2,4,2,1],2,4);
fail
```

4 ▶ `PositionSorted(` *list*, *elm* `)` O
▶ `PositionSorted(` *list*, *elm*, *func* `)` O

In the first form `PositionSorted` returns the position of the element *elm* in the sorted list *list*.

In the second form `PositionSorted` returns the position of the element *elm* in the list *list*, which must be sorted with respect to *func*. *func* must be a function of two arguments that returns `true` if the first argument is less than the second argument and `false` otherwise.

`PositionSorted` returns *pos* such that $list[pos-1] < elm$ and $elm \leq list[pos]$. That means, if *elm* appears once in *list*, its position is returned. If *elm* appears several times in *list*, the position of the first occurrence is returned. If *elm* is not an element of *list*, the index where *elm* must be inserted to keep the list sorted is returned.

`PositionSorted` uses binary search, whereas `Position` can in general use only linear search, see the remark at the beginning of 21.19. For sorting lists, see 21.18, for testing whether a list is sorted, see 21.17.3 and 21.17.4.

```
gap> PositionSorted( [1,4,5,5,6,7], 0 );
1
gap> PositionSorted( [1,4,5,5,6,7], 2 );
2
gap> PositionSorted( [1,4,5,5,6,7], 4 );
2
gap> PositionSorted( [1,4,5,5,6,7], 5 );
3
gap> PositionSorted( [1,4,5,5,6,7], 8 );
7
```

5 ▶ `PositionSet( `*list*`, `*obj*` )` F
▶ `PositionSet( `*list*`, `*obj*`, `*func*` )` F

`PositionSet` is a slight variation of `PositionSorted`. The only difference to `PositionSorted` is that `PositionSet` returns `fail` if *obj* is not in *list*.

```
gap> PositionSet( [1,4,5,5,6,7], 0 );
fail
gap> PositionSet( [1,4,5,5,6,7], 2 );
fail
gap> PositionSet( [1,4,5,5,6,7], 4 );
2
gap> PositionSet( [1,4,5,5,6,7], 5 );
3
gap> PositionSet( [1,4,5,5,6,7], 8 );
fail
```

6 ▶ `PositionProperty( `*list*`, `*func*` )` O

returns the first position of an element in the list *list* for which the property tester function *func* returns `true`.

```
gap> PositionProperty( [10^7..10^8], IsPrime );
20
gap> PositionProperty( [10^5..10^6],
>        n -> not IsPrime(n) and IsPrimePowerInt(n) );
490
```

`First` (see 21.20.20) allows you to extract the first element of a list that satisfies a certain property.

7 ▶ `PositionBound( `*list*` )` O

returns the first index for which an element is bound in the list *list*. For the empty list it returns `fail`.

```
gap> PositionBound([1,2,3]);
1
gap> PositionBound([,1,2,3]);
2
```

8 ▶ `PositionNot( `*list*`, `*val*`[, `*from-minus-one*`] )` O

For a list *list* and an object *val*, `PositionNot` returns the smallest nonnegative integer n such that *list*$[n]$ is either unbound or not equal to *val*. If a nonnegative integer is given as optional argument *from-minus-one* then the first position larger than *from-minus-one* with this property is returned.

9 ▶ `PositionNonZero( `*vec*` )` O

For a row vector *vec*, `PositionNonZero` returns the position of the first non-zero element of *vec*, or `Length(`*vec*`)+1` if all entries of *vec* are zero.

`PositionNonZero` implements a special case of `PositionNot` (see 21.16.8). Namely, the element to be avoided is the zero element, and the list must be (at least) homogeneous because otherwise the zero element cannot be specified implicitly.

```
gap> l:= [ 1, 1, 2, 3, 2 ];;  PositionNot( l, 1 );
3
gap> PositionNot( l, 1, 4 );  PositionNot( l, 2, 5 );
5
6
gap> PositionNonZero( l );  PositionNonZero( [ 2, 3, 4, 5 ] * Z(2) );
1
2
```

10 ▶ `PositionSublist( `*list*`, `*sub*` )` O
▶ `PositionSublist( `*list*`, `*sub*`, `*from*` )` O

returns the smallest index in the list *list* at which a sublist equal to *sub* starts. If *sub* does not occur the operation returns `fail`. The second version starts searching **after** position *from*.

To determine whether *sub* matches *list* at a particular position, use `IsMatchingSublist` instead (see 21.17.1).

21.17 Properties and Attributes for Lists

1 ▶ `IsMatchingSublist( `*list*`, `*sub*` )` O
▶ `IsMatchingSublist( `*list*`, `*sub*`, `*at*` )` O

returns `true` if *sub* matches a sublist of *list* from position 1 (or position *at*, in the case of the second version), or `false`, otherwise. If *sub* is empty `true` is returned. If *list* is empty but *sub* is non-empty `false` is returned.

If you actually want to know whether there is an *at* for which `IsMatchingSublist( `*list*`, `*sub*`, `*at*` )` is true, use a construction like `PositionSublist( `*list*`, `*sub*` ) <> fail` instead (see 21.16.10); it's more efficient.

Note: A list that contains mutable objects (like lists or records) **cannot** store attribute values that depend on the values of its entries, such as whether it is homogeneous, sorted, or strictly sorted, as changes in any of its entries could change such property values, like the following example shows.

```
gap> l:=[[1],[2]];
[ [ 1 ], [ 2 ] ]
gap> IsSSortedList(l);
true
gap> l[1][1]:=3;
3
gap> IsSSortedList(l);
false
```

For such lists these property values must be computed anew each time the property is asked for. For example, if *list* is a list of mutable row vectors then the call of `Position` (see 21.16.1) with *list* as first argument cannot take advantage of the fact that *list* is in fact sorted. One solution is to call explicitly `PositionSorted` (see 21.16.4) in such a situation, another solution is to replace *list* by an immutable copy using `Immutable` (see 12.6).

2 ▶ `IsDuplicateFree( `*obj*` )` P
▶ `IsDuplicateFreeList( `*obj*` )` P

`IsDuplicateFree(`*obj*`);` returns `true` if *obj* is both a list or collection, and it is duplicate free; otherwise it returns `false`. `IsDuplicateFreeList` is a synonym for `IsDuplicateFree and IsList`.

A list is **duplicate free** if it is dense and does not contain equal entries in different positions. Every domain (see 12.4) is duplicate free.

3 ▶ `IsSortedList( obj )` P

returns `true` if *obj* is a list and it is sorted, or `false` otherwise.

A list *list* is **sorted** if it is dense (see 21.1.2) and satisfies the relation $list[i] \leq list[j]$ whenever $i < j$. Note that a sorted list is not necessarily duplicate free (see 21.17.2 and 21.17.4).

Many sorted lists are in fact homogeneous (see 21.1.3), but also non-homogeneous lists may be sorted (see 30.11).

4 ▶ `IsSSortedList( obj )` P
▶ `IsSet( obj )` P

returns `true` if *obj* is a list and it is strictly sorted, or `false` otherwise. `IsSSortedList` is short for "is strictly sorted list"; `IsSet` is just a synonym for `IsSSortedList`.

A list *list* is **strictly sorted** if it is sorted (see 21.17.3) and satisfies the relation $list[i] \lneq list[j]$ whenever $i < j$. In particular, such lists are duplicate free (see 21.17.2).

In sorted lists, membership test and computing of positions can be done by binary search, see 21.19.

(Currently there is little special treatment of lists that are sorted but not strictly sorted. In particular, internally represented lists will **not** store that they are sorted but not strictly sorted.)

5 ▶ `Length( list )` A

returns the **length** of the list *list*, which is defined to be the index of the last bound entry in *list*.

6 ▶ `ConstantTimeAccessList( list )` A

`ConstantTimeAccessList` returns an immutable list containing the same elements as the list *list* (which may have holes) in the same order. If *list* is already a constant time access list, `ConstantTimeAccessList` returns an immutable copy of *list* directly. Otherwise it puts all elements and holes of *list* into a new list and makes that list immutable.

21.18 Sorting Lists

1 ▶ `Sort( list )` O
▶ `Sort( list, func )` O

sorts the list *list* in increasing order. In the first form `Sort` uses the operator `<` to compare the elements. (If the list is not homogeneous it is the users responsibility to ensure that `<` is defined for all element pairs, see 30.11) In the second form `Sort` uses the function *func* to compare elements. *func* must be a function taking two arguments that returns `true` if the first is regarded as strictly smaller than the second, and `false` otherwise.

`Sort` does not return anything, it just changes the argument *list*. Use `ShallowCopy` (see 12.7.1) if you want to keep *list*. Use `Reversed` (see 21.20.7) if you want to get a new list sorted in decreasing order.

It is possible to sort lists that contain multiple elements which compare equal. It is not guaranteed that those elements keep their relative order, i.e., `Sort` is not stable.

```
gap> list := [ 5, 4, 6, 1, 7, 5 ];; Sort( list ); list;
[ 1, 4, 5, 5, 6, 7 ]
gap> list := [ [0,6], [1,2], [1,3], [1,5], [0,4], [3,4] ];;
gap> Sort( list, function(v,w) return v*v < w*w; end );
gap> list;  # sorted according to the Euclidian distance from [0,0]
[ [ 1, 2 ], [ 1, 3 ], [ 0, 4 ], [ 3, 4 ], [ 1, 5 ], [ 0, 6 ] ]
gap> list := [ [0,6], [1,3], [3,4], [1,5], [1,2], [0,4], ];;
gap> Sort( list, function(v,w) return v[1] < w[1]; end );
gap> list;  # note the random order of the elements with equal first component
```

```
[ [ 0, 6 ], [ 0, 4 ], [ 1, 3 ], [ 1, 5 ], [ 1, 2 ], [ 3, 4 ] ]
```

2 ▶ `SortParallel( list, list2 )` O
▶ `SortParallel( list, list2, func )` O

sorts the list *list1* in increasing order just as `Sort` (see 21.18.1) does. In parallel it applies the same exchanges that are necessary to sort *list1* to the list *list2*, which must of course have at least as many elements as *list1* does.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := [ 2, 3, 5, 7, 8, 9 ];;
gap> SortParallel( list1, list2 );
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
gap> list2;  # note: [ 7, 3, 2, 9, 5, 8 ] or [ 7, 3, 9, 2, 5, 8 ] are possible results
[ 7, 3, 2, 9, 5, 8 ]
```

3 ▶ `Sortex( list )` O

sorts the list *list* via the operator< and returns a permutation that can be applied to *list* to obtain the sorted list. (If the list is not homogeneous it is the user's responsibility to ensure that < is defined for all element pairs, see 30.11)

`Permuted` (see 21.20.16) allows you to rearrange a list according to a given permutation.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := ShallowCopy( list1 );;
gap> perm := Sortex( list1 );
(1,3,5,6,4)
gap> list1;
[ 1, 4, 5, 5, 6, 7 ]
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]
```

4 ▶ `SortingPerm( list )` A

`SortingPerm` returns the same as `Sortex( list )` (see 21.18.3) but does **not** change the argument.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := ShallowCopy( list1 );;
gap> perm := SortingPerm( list1 );
(1,3,5,6,4)
gap> list1;
[ 5, 4, 6, 1, 7, 5 ]
gap> Permuted( list2, perm );
[ 1, 4, 5, 5, 6, 7 ]
```

Currently GAP uses shellsort.

21.19 Sorted Lists and Sets

Searching objects in a list works much quicker if the list is known to be sorted. Currently GAP exploits the sortedness of a list automatically only if the list is **strictly sorted**, which is indicated by the property `IsSSortedList`, see 21.17.4.

Remember that a list of **mutable** objects cannot store that it is strictly sorted but has to test it anew whenever it is asked whether it is sorted, see the remark in 21.17. Therefore GAP cannot take advantage of the sortedness of a list if this list has mutable entries. Moreover, if a sorted list *list* with mutable elements is used as an argument of a function that **expects** this argument to be sorted, for example `UniteSet` or `RemoveSet` (see 21.19.6, 21.19.5), then it is checked whether *list* is in fact sorted; this check can have the effect actually to slow down the computations, compared to computations with sorted lists of immutable elements or computations that do not involve functions that do automatically check sortedness.

Strictly sorted lists are used to represent **sets** in GAP. More precisely, a strictly sorted list is called a **proper set** in the following, in order to avoid confusion with domains (see 12.4) which also represent sets.

In short proper sets are represented by sorted lists without holes and duplicates in GAP. Note that we guarantee this representation, so you may make use of the fact that a set is represented by a sorted list in your functions.

In some contexts (for example see 17), we also want to talk about multisets. A **multiset** is like a set, except that an element may appear several times in a multiset. Such multisets are represented by sorted lists without holes that may have duplicates.

This section lists only those functions that are defined exclusively for proper sets. Set theoretic functions for general collections, such as `Intersection` and `Union`, are described in Chapter 28. In particular, for the construction of proper sets, see 28.2.6 and 28.2.9. For finding positions in sorted lists, see 21.16.4.

1► *obj* `in` *list*

The element test for strictly sorted lists uses binary search.

The following functions, if not explicitly stated differently, take two arguments, *set* and *obj*, where *set* must be a proper set, otherwise an error is signalled; If the second argument *obj* is a list that is not a proper set then `Set` (see 28.2.6) is silently applied to it first (see 28.2.6).

2► `IsEqualSet(` *list1*`,` *list2* `)` O

tests whether *list1* and *list2* are equal **when viewed as sets**, that is if every element of *list1* is an element of *list2* and vice versa. Either argument of `IsEqualSet` may also be a list that is not a proper set, in which case `Set` (see 28.2.6) is applied to it first.

If both lists are proper sets then they are of course equal if and only if they are also equal as lists. Thus `IsEqualSet(` *list1*`,` *list2* `)` is equivalent to `Set(` *list1* `) = Set(` *list2* `)` (see 28.2.6), but the former is more efficient.

```
gap> IsEqualSet( [2,3,5,7,11], [11,7,5,3,2] );
true
gap> IsEqualSet( [2,3,5,7,11], [2,3,5,7,11,13] );
false
```

3► `IsSubsetSet(` *list1*`,` *list2* `)` O

tests whether every element of *list2* is contained in *list1*. Either argument of `IsSubsetSet` may also be a list that is not a proper set, in which case `Set` (see 28.2.6) is applied to it first.

4► `AddSet(` *set*`,` *obj* `)` O

adds the element *obj* to the proper set *set*. If *obj* is already contained in *set* then *set* is not changed. Otherwise *obj* is inserted at the correct position such that *set* is again a proper set afterwards.

Note that *obj* must be in the same family as each element of *set*.

```
gap> s := [2,3,7,11];;
gap> AddSet( s, 5 );  s;
[ 2, 3, 5, 7, 11 ]
gap> AddSet( s, 13 );  s;
[ 2, 3, 5, 7, 11, 13 ]
gap> AddSet( s, 3 );  s;
[ 2, 3, 5, 7, 11, 13 ]
```

5 ▶ `RemoveSet( set, obj )` O

removes the element *obj* from the proper set *set*. If *obj* is not contained in *set* then *set* is not changed. If *obj* is an element of *set* it is removed and all the following elements in the list are moved one position forward.

```
gap> s := [ 2, 3, 4, 5, 6, 7 ];;
gap> RemoveSet( s, 6 ); s;
[ 2, 3, 4, 5, 7 ]
gap> RemoveSet( s, 10 ); s;
[ 2, 3, 4, 5, 7 ]
```

6 ▶ `UniteSet( set, list )` O

unites the proper set *set* with *list*. This is equivalent to adding all elements of *list* to *set* (see 21.19.4).

```
gap> set := [ 2, 3, 5, 7, 11 ];;
gap> UniteSet( set, [ 4, 8, 9 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> UniteSet( set, [ 16, 9, 25, 13, 16 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 25 ]
```

7 ▶ `IntersectSet( set, list )` O

intersects the proper set *set* with *list*. This is equivalent to removing from *set* all elements of *set* that are not contained in *list*.

```
gap> set := [ 2, 3, 4, 5, 7, 8, 9, 11, 13, 16 ];;
gap> IntersectSet( set, [ 3, 5, 7, 9, 11, 13, 15, 17 ] );  set;
[ 3, 5, 7, 9, 11, 13 ]
gap> IntersectSet( set, [ 9, 4, 6, 8 ] );  set;
[ 9 ]
```

8 ▶ `SubtractSet( set, list )` O

subtracts *list* from the proper set *set*. This is equivalent to removing from *set* all elements of *list*.

```
gap> set := [ 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ];;
gap> SubtractSet( set, [ 6, 10 ] );  set;
[ 2, 3, 4, 5, 7, 8, 9, 11 ]
gap> SubtractSet( set, [ 9, 4, 6, 8 ] );  set;
[ 2, 3, 5, 7, 11 ]
```

There are nondestructive counterparts of the functions `UniteSet`, `IntersectSet`, and `SubtractSet` available for proper sets. These are `UnionSet`, `IntersectionSet`, and `Difference`. The former two are methods for the more general operations `Union` and `Intersection` (see 28.4.3, 28.4.2), the latter is itself an operation (see 28.4.4).

The result of `IntersectionSet` and `UnionSet` is always a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the first argument *set*. If *set* is not a proper set it is not specified to which of a number of equal elements in *set* the element in the result is identical (see 21.6).

21.20 Operations for Lists

Several of the following functions expect the first argument to be either a list or a collection (see 28), with possibly slightly different meaning for lists and non-list collections. For these functions, the list case is indicated by an argument named *list*, and the collection case by one named *C*.

1 ▶ `Concatenation( `*list1*`, `*list2*`, ... )` F
▶ `Concatenation( `*list*` )` F

In the first form `Concatenation` returns the concatenation of the lists *list1*, *list2*, etc. The **concatenation** is the list that begins with the elements of *list1*, followed by the elements of *list2*, and so on. Each list may also contain holes, in which case the concatenation also contains holes at the corresponding positions.

In the second form *list* must be a dense list of lists *list1*, *list2*, etc., and `Concatenation` returns the concatenation of those lists.

The result is a new mutable list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of *list1*, *list2*, etc. (see 21.6).

Note that `Concatenation` creates a new list and leaves its arguments unchanged, while `Append` (see 21.4.5) changes its first argument. For computing the union of proper sets, `Union` can be used, see 28.4.3 and 21.19.

```
gap> Concatenation( [ 1, 2, 3 ], [ 4, 5 ] );
[ 1, 2, 3, 4, 5 ]
gap> Concatenation( [2,3,,5,,7], [11,,13,,,,17,,19] );
[ 2, 3,, 5,, 7, 11,, 13,,,, 17,, 19 ]
gap> Concatenation( [ [1,2,3], [2,3,4], [3,4,5] ] );
[ 1, 2, 3, 2, 3, 4, 3, 4, 5 ]
```

2 ▶ `Compacted( `*list*` )` O

returns a new mutable list that contains the elements of *list* in the same order but omitting the holes.

```
gap> l:=[,1,,,3,,,4,[5,,,6],7];;  Compacted( l );
[ 1, 3, 4, [ 5,,, 6 ], 7 ]
```

3 ▶ `Collected( `*list*` )` O

returns a new list *new* that contains for each element *elm* of the list *list* a list of length two, the first element of this is *elm* itself and the second element is the number of times *elm* appears in *list*. The order of those pairs in *new* corresponds to the ordering of the elements elm, so that the result is sorted.

For all pairs of elements in *list* the comparison via `<` must be defined.

```
gap> Factors( Factorial( 10 ) );
[ 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7 ]
gap> Collected( last );
[ [ 2, 8 ], [ 3, 4 ], [ 5, 2 ], [ 7, 1 ] ]
gap> Collected( last );
[ [ [ 2, 8 ], 1 ], [ [ 3, 4 ], 1 ], [ [ 5, 2 ], 1 ], [ [ 7, 1 ], 1 ] ]
```

4 ▶ `DuplicateFreeList( `*list*` )` O
▶ `Unique( `*list*` )` O

returns a new mutable list whose entries are the elements of the list *list* with duplicates removed. `DuplicateFreeList` only uses the `=` comparison and will not sort the result. Therefore `DuplicateFreeList` can be used even if the elements of *list* do not lie in the same family. `Unique` is an alias for `DuplicateFreeList`.

```
gap> l:=[1,Z(3),1,"abc",Group((1,2,3),(1,2)),Z(3),Group((1,2),(2,3))];;
gap> DuplicateFreeList( l );
[ 1, Z(3), "abc", Group([ (1,2,3), (1,2) ]) ]
```

5 ▶ `AsDuplicateFreeList( `*list*` )` A

returns the same result as `DuplicateFreeList` (see 21.20.4), except that the result is immutable.

6 ▶ `Flat( `*list*` )` O

returns the list of all elements that are contained in the list *list* or its sublists. That is, `Flat` first makes a new empty list *new*. Then it loops over the elements *elm* of *list*. If *elm* is not a list it is added to *new*, otherwise `Flat` appends `Flat( `*elm*` )` to *new*.

```
gap> Flat( [ 1, [ 2, 3 ], [ [ 1, 2 ], 3 ] ] );
[ 1, 2, 3, 1, 2, 3 ]
gap> Flat( [ ] );
[ ]
```

(To reconstruct a matrix from a `Flat`tened list, the sublist operator can be used:

```
gap> l:=[9..14];;w:=2;; # w is the length of each row
gap> sub:=[1..w];;List([1..Length(l)/w],i->l{(i-1)*w+sub});
[ [ 9, 10 ], [ 11, 12 ], [ 13, 14 ] ]
```

)

7 ▶ `Reversed( `*list*` )` F

returns a new mutable list, containing the elements of the dense list *list* in reversed order.

The argument list is unchanged. The result list is a new list, that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 21.6).

`Reversed` implements a special case of list assignment, which can also be formulated in terms of the operator (see 21.4).

```
gap> Reversed( [ 1, 4, 9, 5, 6, 7 ] );
[ 7, 6, 5, 9, 4, 1 ]
```

8 ▶ `IsLexicographicallyLess( `*list1*`, `*list2*` )` F

Let *list1* and *list2* be two dense lists, but not necessarily homogeneous (see 21.1.2, 21.1.3), such that for each i, the entries in both lists at position i can be compared via `<`. `IsLexicographicallyLess` returns `true` if *list1* is smaller than *list2* w.r.t. lexicographical ordering, and `false` otherwise.

9 ▶ `Apply( `*list*`, `*func*` )` F

`Apply` applies the function *func* to every element of the dense and mutable list *list*, and replaces each element entry by the corresponding return value.

`Apply` changes its argument. The nondestructive counterpart of `Apply` is `List` (see 21.20.17).

```
gap> l:= [ 1, 2, 3 ];;  Apply( l, i -> i^2 );  l;
[ 1, 4, 9 ]
```

10 ▶ `Perform( `*list*`, `*func*` )` O

`Perform( `*list*`, `*func*` )` applies func to every element of *list*, discarding any return values. It does not return a value.

```
gap> l := [1, 2, 3];; Perform(l,
> function(x) if IsPrimeInt(x) then Print(x,"\n"); fi; end);
2
3
```

11 ▶ `PermListList(` *list1*`,` *list2* `)` F

returns a permutation p of `[ 1 .. Length(` *list1* `) ]` such that *list1*`[i^`p`] = `*list2*`[i]`. It returns `fail` if there is no such permutation.

```
gap> list1 := [ 5, 4, 6, 1, 7, 5 ];;
gap> list2 := [ 4, 1, 7, 5, 5, 6 ];;
gap> perm := PermListList(list1, list2);
(1,2,4)(3,5,6)
gap> Permuted( list2, perm );
[ 5, 4, 6, 1, 7, 5 ]
```

12 ▶ `Maximum(` *obj1*`,` *obj2* `...)` F
▶ `Maximum(` *list* `)` F

In the first form `Maximum` returns the **maximum** of its arguments, i.e., one argument *obj* for which $obj \geq obj1$, $obj \geq obj2$ etc. In the second form `Maximum` takes a homogeneous list *list* and returns the maximum of the elements in this list.

```
gap> Maximum( -123, 700, 123, 0, -1000 );
700
gap> Maximum( [ -123, 700, 123, 0, -1000 ] );
700
gap> Maximum( [1,2], [0,15], [1,5], [2,-11] );  # lists are compared elementwise
[ 2, -11 ]
```

13 ▶ `Minimum(` *obj1*`,` *obj2* `...)` F
▶ `Minimum(` *list* `)` F

In the first form `Minimum` returns the **minimum** of its arguments, i.e., one argument *obj* for which $obj \leq obj1$, $obj \leq obj2$ etc. In the second form `Minimum` takes a homogeneous list *list* and returns the minimum of the elements in this list.

Note that for both `Maximum` and `Minimum` the comparison of the objects *obj1*, *obj2* etc. must be defined; for that, usually they must lie in the same family (see 13.1).

```
gap> Minimum( -123, 700, 123, 0, -1000 );
-1000
gap> Minimum( [ -123, 700, 123, 0, -1000 ] );
-1000
gap> Minimum( [ 1, 2 ], [ 0, 15 ], [ 1, 5 ], [ 2, -11 ] );
[ 0, 15 ]
```

14 ▶ `MaximumList(` *list* `)` O
▶ `MinimumList(` *list* `)` O

return the maximum resp. the minimum of the elements in the list *list*. They are the operations called by `Maximum` resp. `Minimum`. Methods can be installed for special kinds of lists. For example, there are special methods to compute the maximum resp. the minimum of a range (see 21.22).

15 ▶ `Cartesian(` *list1*`,` *list2* `...)` F
▶ `Cartesian(` *list* `)` F

In the first form `Cartesian` returns the cartesian product of the lists *list1*, *list2*, etc.

In the second form *list* must be a list of lists *list1*, *list2*, etc., and `Cartesian` returns the cartesian product of those lists.

The **cartesian product** is a list *cart* of lists *tup*, such that the first element of *tup* is an element of *list1*, the second element of *tup* is an element of *list2*, and so on. The total number of elements in *cart* is the product of the lengths of the argument lists. In particular *cart* is empty if and only if at least one of the argument lists is empty. Also *cart* contains duplicates if and only if no argument list is empty and at least one contains duplicates.

The last index runs fastest. That means that the first element *tup1* of *cart* contains the first element from *list1*, from *list2* and so on. The second element *tup2* of *cart* contains the first element from *list1*, the first from *list2*, an so on, but the last element of *tup2* is the second element of the last argument list. This implies that *cart* is a proper set if and only if all argument lists are proper sets (see 21.19).

The function `Tuples` (see 17.2.7) computes the k-fold cartesian product of a list.

```
gap> Cartesian( [1,2], [3,4], [5,6] );
[ [ 1, 3, 5 ], [ 1, 3, 6 ], [ 1, 4, 5 ], [ 1, 4, 6 ], [ 2, 3, 5 ],
  [ 2, 3, 6 ], [ 2, 4, 5 ], [ 2, 4, 6 ] ]
gap> Cartesian( [1,2,2], [1,1,2] );
[ [ 1, 1 ], [ 1, 1 ], [ 1, 2 ], [ 2, 1 ], [ 2, 1 ], [ 2, 2 ], [ 2, 1 ],
  [ 2, 1 ], [ 2, 2 ] ]
```

16 ▶ `Permuted(` *list*`,` *perm* `)` O

returns a new list *new* that contains the elements of the list *list* permuted according to the permutation *perm*. That is *new*`[`*i* `^` *perm*`] = `*list*`[`*i*`]`.

`Sortex` (see 21.18.3) allows you to compute a permutation that must be applied to a list in order to get the sorted list.

```
gap> Permuted( [ 5, 4, 6, 1, 7, 5 ], (1,3,5,6,4) );
[ 1, 4, 5, 5, 6, 7 ]
```

17 ▶ `List(` *list* `)` F
▶ `List(` *C* `)` F
▶ `List(` *list*`,` *func* `)` F

In the first form, where *list* is a list (not necessarily dense or homogeneous), `List` returns a new mutable list *new* that contains the elements (and the holes) of *list* in the same order; thus `List` does the same as `ShallowCopy` (see 12.7.1) in this case.

In the second form, where C is a collection (see 28) that is not a list, `List` returns a new mutable list *new* such that `Length(` *new* `)` is the number of different elements of C, and *new* contains the different elements of C in an unspecified order which may change for repeated calls. *new*`[`*pos*`]` executes in constant time (see 21.1.5), and the size of *new* is proportional to its length. The generic method for this case is `ShallowCopy( Enumerator(` *C* `) )`.

In the third form, for a dense list *list* and a function *func*, which must take exactly one argument, `List` returns a new mutable list *new* given by $new[i] = func(list[i])$.

```
gap> List( [1,2,3], i -> i^2 );
[ 1, 4, 9 ]
gap> List( [1..10], IsPrime );
[ false, true, true, false, true, false, true, false, false, false ]
```

18 ▶ `Filtered(` *list*`,` *func* `)` F
▶ `Filtered(` *C*`,` *func* `)` F

returns a new list that contains those elements of the list *list* or collection C (see 28), respectively, for which the unary function *func* returns `true`.

If the first argument is a list, the order of the elements in the result is the same as the order of the corresponding elements of *list*. If an element for which *func* returns `true` appears several times in *list* it will also appear the same number of times in the result. *list* may contain holes, they are ignored by `Filtered`.

For each element of *list* resp. *C*, *func* must return either `true` or `false`, otherwise an error is signalled.

The result is a new list that is not identical to any other list. The elements of that list however are identical to the corresponding elements of the argument list (see 21.6).

List assignment using the operator (see 21.4) can be used to extract elements of a list according to indices given in another list.

```
gap> Filtered( [1..20], IsPrime );
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
[ 3, 4, 4, 7 ]
gap> Filtered( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>              n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
[ 3, 7 ]
gap> Filtered( Group( (1,2), (1,2,3) ), x -> Order( x ) = 2 );
[ (2,3), (1,2), (1,3) ]
```

19 ▶ `Number(` *list* `)` F
▶ `Number(` *list*`,` *func* `)` F
▶ `Number(` *C*`,` *func* `)` F

In the first form, `Number` returns the number of bound entries in the list *list*. For dense lists `Number`, `Length` (see 21.17.5), and `Size` (see 28.3.6) return the same value; for lists with holes `Number` returns the number of bound entries, `Length` returns the largest index of a bound entry, and `Size` signals an error.

In the last two forms, `Number` returns the number of elements of the list *list* resp. the collection *C* for which the unary function *func* returns `true`. If an element for which *func* returns `true` appears several times in *list* it will also be counted the same number of times.

For each element of *list* resp. *C*, *func* must return either `true` or `false`, otherwise an error is signalled.

`Filtered` (see 21.20.18) allows you to extract the elements of a list that have a certain property.

```
gap> Number( [ 2, 3, 5, 7 ] );
4
gap> Number( [, 2, 3,, 5,, 7,,,, 11 ] );
5
gap> Number( [1..20], IsPrime );
8
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ], IsPrimePowerInt );
4
gap> Number( [ 1, 3, 4, -4, 4, 7, 10, 6 ],
>            n -> IsPrimePowerInt(n) and n mod 2 <> 0 );
2
gap> Number( Group( (1,2), (1,2,3) ), x -> Order( x ) = 2 );
3
```

20 ▶ `First(` *list*`,` *func* `)` F

`First` returns the first element of the list *list* for which the unary function *func* returns `true`. *list* may contain holes. *func* must return either `true` or `false` for each element of *list*, otherwise an error is signalled. If *func* returns `false` for all elements of *list* then `First` returns `fail`.

`PositionProperty` (see 21.16.6) allows you to find the position of the first element in a list that satisfies a certain property.

```
gap> First( [10^7..10^8], IsPrime );
10000019
gap> First( [10^5..10^6],
>      n -> not IsPrime(n) and IsPrimePowerInt(n) );
100489
gap> First( [ 1 .. 20 ], x -> x < 0 );
fail
gap> First( [ fail ], x -> x = fail );
fail
```

21 ▶ `ForAll(` *list*`,` *func* `)` F
▶ `ForAll(` *C*`,` *func* `)` F

tests whether the unary function *func* returns `true` for all elements in the list *list* resp. the collection *C*.

```
gap> ForAll( [1..20], IsPrime );
false
gap> ForAll( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAll( [2..14], n -> IsPrimePowerInt(n) or n mod 2 = 0 );
true
gap> ForAll( Group( (1,2), (1,2,3) ), i -> SignPerm(i) = 1 );
false
```

22 ▶ `ForAny(` *list*`,` *func* `)` F
▶ `ForAny(` *C*`,` *func* `)` F

tests whether the unary function *func* returns `true` for at least one element in the list *list* resp. the collection *C*.

```
gap> ForAny( [1..20], IsPrime );
true
gap> ForAny( [2,3,4,5,8,9], IsPrimePowerInt );
true
gap> ForAny( [2..14],
>    n -> IsPrimePowerInt(n) and n mod 5 = 0 and not IsPrime(n) );
false
gap> ForAny( Integers, i ->     i > 0
>                          and ForAll( [0,2..4], j -> IsPrime(i+j) ) );
true
```

23 ▶ `Product(` *list*`[,` *init*`] )` F
▶ `Product(` *C*`[,` *init*`] )` F
▶ `Product(` *list*`,` *func*`[,` *init*`] )` F
▶ `Product(` *C*`,` *func*`[,` *init*`] )` F

In the first two forms `Product` returns the product of the elements of the dense list *list* resp. the collection *C* (see 28). In the last two forms `Product` applies the function *func*, which must be a function taking one argument, to the elements of the dense list *list* resp. the collection *C*, and returns the product of the results. In either case `Product` returns `1` if the first argument is empty.

The general rules for arithmetic operations apply (see 21.15), so the result is immutable if and only if all summands are immutable.

If *list* or C contains exactly one element then this element (or its image under *func* if applicable) itself is returned, not a shallow copy of this element.

If an additional initial value *init* is given, `Product` returns the product of *init* and the elements of the first argument resp. of their images under the function *func*. This is useful for example if the first argument is empty and a different identity than `1` is desired, in which case *init* is returned.

```
gap> Product( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
9699690
gap> Product( [1..10], x->x^2 );
13168189440000
gap> Product( [ (1,2), (1,3), (1,4), (2,3), (2,4), (3,4) ] );
(1,4)(2,3)
gap> Product( GF(8) );
0*Z(2)
```

24 ▶ `Sum(` *list*`[,` *init*`]` `)` F
▶ `Sum(` *C*`[,` *init*`]` `)` F
▶ `Sum(` *list*`,` *func*`[,` *init*`]` `)` F
▶ `Sum(` *C*`,` *func*`[,` *init*`]` `)` F

In the first two forms `Sum` returns the sum of the elements of the dense list *list* resp. the collection C (see 28). In the last two forms `Sum` applies the function *func*, which must be a function taking one argument, to the elements of the dense list *list* resp. the collection C, and returns the sum of the results. In either case `Sum` returns `0` if the first argument is empty.

The general rules for arithmetic operations apply (see 21.15), so the result is immutable if and only if all summands are immutable.

If *list* or C contains exactly one element then this element (or its image under *func* if applicable) itself is returned, not a shallow copy of this element.

If an additional initial value *init* is given, `Sum` returns the sum of *init* and the elements of the first argument resp. of their images under the function *func*. This is useful for example if the first argument is empty and a different zero than `0` is desired, in which case *init* is returned.

```
gap> Sum( [ 2, 3, 5, 7, 11, 13, 17, 19 ] );
77
gap> Sum( [1..10], x->x^2 );
385
gap> Sum( [ [1,2], [3,4], [5,6] ] );
[ 9, 12 ]
gap> Sum( GF(8) );
0*Z(2)
```

25 ▶ `Iterated(` *list*`,` *func* `)` O

returns the result of the iterated application of the function *func*, which must take two arguments, to the elements of the list *list*. More precisely `Iterated` returns the result of the following application, $f(\cdots f(f(list[1], list[2]), list[3]), \ldots, list[n])$.

```
gap> Iterated( [ 126, 66, 105 ], Gcd );
3
```

26 ▶ `ListN(` *list1*`,` *list2*`, ...,` *listn*`,` *f* `)` F

Applies the n-argument function *func* to the lists. That is, `ListN` returns the list whose ith entry is $f(list1[i], list2[i], \ldots, listn[i])$.

```
gap> ListN( [1,2], [3,4], \+ );
[ 4, 6 ]
```

21.21 Advanced List Manipulations

The following functions are generalizations of `List` (see 21.20.17), `Set` (see 28.2.6), `Sum` (see 21.20.24), and `Product` (see 21.20.23).

1 ▶ `ListX(` *arg1*`,` *arg2*`, ...` *argn*`,` *func* `)` O

`ListX` returns a new list constructed from the arguments.

Each of the arguments *arg1*, *arg2*, ... *argn* must be one of the following:

a list or collection
: this introduces a new for-loop in the sequence of nested for-loops and if-statements;

a function returning a list or collection
: this introduces a new for-loop in the sequence of nested for-loops and if-statements, where the loop-range depends on the values of the outer loop-variables; or

a function returning `true` or `false`
: this introduces a new if-statement in the sequence of nested for-loops and if-statements.

The last argument *func* must be a function, it is applied to the values of the loop-variables and the results are collected.

Thus `ListX(` *list*`,` *func* `)` is the same as `List(` *list*`,` *func* `)`, and `ListX(` *list*`,` *func*`, x -> x )` is the same as `Filtered(` *list*`,` *func* `)`.

As a more elaborate example, assume *arg1* is a list or collection, *arg2* is a function returning `true` or `false`, *arg3* is a function returning a list or collection, and *arg4* is another function returning `true` or `false`, then

```
result := ListX( arg1, arg2, arg3, arg4, func );
```

is equivalent to

```
result := [];
for v1 in arg1 do
  if arg2( v1 ) then
    for v2 in arg3( v1 ) do
      if arg4( v1, v2 ) then
        Add( result, func( v1, v2 ) );
      fi;
    od;
  fi;
od;
```

The following example shows how `ListX` can be used to compute all pairs and all strictly sorted pairs of elements in a list.

```
gap> l:= [ 1, 2, 3, 4 ];;
gap> pair:= function( x, y ) return [ x, y ]; end;;
gap> ListX( l, l, pair );
[ [ 1, 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 1 ], [ 2, 2 ], [ 2, 3 ],
  [ 2, 4 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ], [ 4, 1 ], [ 4, 2 ],
  [ 4, 3 ], [ 4, 4 ] ]
```

In the following example, `<` is the comparison operation:

```
gap> ListX( l, l, \<, pair );
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 2, 4 ], [ 3, 4 ] ]
```

2 ► `SetX(` *arg1*`,` *arg2*`, ...` *func* `)` O

The only difference between `SetX` and `ListX` is that the result list of `SetX` is strictly sorted.

3 ► `SumX(` *arg1*`,` *arg2*`, ...` *func* `)` O

`SumX` returns the sum of the elements in the list obtained by `ListX` when this is called with the same arguments.

4 ► `ProductX(` *arg1*`,` *arg2*`, ...` *func* `)` O

`ProductX` returns the product of the elements in the list obtained by `ListX` when this is called with the same arguments.

21.22 Ranges

A **range** is a dense list of integers in arithmetic progression (or degression). This is a list of integers such that the difference between consecutive elements is a nonzero constant. Ranges can be abbreviated with the syntactic construct `[` *first*`,` *second* `..` *last* `]` or, if the difference between consecutive elements is 1, as `[` *first* `..` *last* `]`.

If *first* `>` *last*, `[`*first*`..`*last*`]` is the empty list, which by definition is also a range; also, if *second* `>` *first* `>` *last* or *second* `<` *first* `<` *last*, then `[`*first*`,`*second*`..`*last*`]` is the empty list. If *first* `=` *last*, `[`*first*`,`*second*`..`*last*`]` is a singleton list, which is a range, too. Note that *last* `-` *first* must be divisible by the increment *second* `-` *first*, otherwise an error is signalled.

Note also that a range is just a special case of a list. Thus you can access elements in a range (see 21.3), test for membership etc. You can even assign to such a range if it is mutable (see 21.4). Of course, unless you assign *last* `+` *second*`-`*first* to the entry *range*`[Length(`*range*`)+1]`, the resulting list will no longer be a range.

```
gap> r := [10..20];
[ 10 .. 20 ]
gap> Length( r );
11
gap> r[3];
12
gap> 17 in r;
true
gap> r[12] := 25;; r;  # r is no longer a range
[ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25 ]
gap> r := [1,3..17];
[ 1, 3 .. 17 ]
```

```
gap> Length( r );
9
gap> r[4];
7
gap> r := [0,-1..-9];
[ 0, -1 .. -9 ]
gap> r[5];
-4
gap> r := [ 1, 4 .. 32 ];
Range: <last>-<first> (31) must be divisible by <inc> (3)
```

Most often ranges are used in connection with the `for`-loop (see 4.19). Here the construct

`for` *var* `in [`*first*`..`*last*`] do` *statements* `od`

replaces the

`for` *var* `from` *first* `to` *last* `do` *statements* `od`

which is more usual in other programming languages.

```
gap> s := [];; for i in [10..20] do Add( s, i^2 ); od; s;
[ 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400 ]
```

Note that a range with *last* `>=` *first* is at the same time also a proper set (see 21.19), because it contains no holes or duplicates and is sorted, and also a row vector (see 23), because it contains no holes and all elements are integers.

1 ▶ `IsRange(` *obj* `)` C

tests if the object *obj* is a range, i.e. is a dense list of integers that is also a range (see 21.22 for a definition of "range").

```
gap> IsRange( [1,2,3] );  IsRange( [7,5,3,1] );
true
true
gap> IsRange( [1,2,4,5] );  IsRange( [1,,3,,5,,7] );
false
false
gap> IsRange( [] );  IsRange( [1] );
true
true
```

2 ▶ `ConvertToRangeRep(` *list* `)` F

For some lists the GAP kernel knows that they are in fact ranges. Those lists are represented internally in a compact way instead of the ordinary way.

If *list* is a range then `ConvertToRangeRep` changes the representation of *list* to this compact representation.

This is important since this representation needs only 12 bytes for the entire range while the ordinary representation needs 4*length* bytes.

Note that a list that is represented in the ordinary way might still be a range. It is just that GAP does not know this. The following rules tell you under which circumstances a range is represented in the compact way, so you can write your program in such a way that you make best use of this compact representation for ranges.

Lists created by the syntactic construct `[` *first*`,` *second* `..` *last* `]` are of course known to be ranges and are represented in the compact way.

If you call `ConvertToRangeRep` for a list represented the ordinary way that is indeed a range, the representation is changed from the ordinary to the compact representation. A call of `ConvertToRangeRep` for a list that is not a range is ignored.

If you change a mutable range that is represented in the compact way, by assignment, `Add` or `Append`, the range will be converted to the ordinary representation, even if the change is such that the resulting list is still a proper range.

Suppose you have built a proper range in such a way that it is represented in the ordinary way and that you now want to convert it to the compact representation to save space. Then you should call `ConvertToRangeRep` with that list as an argument. You can think of the call to `ConvertToRangeRep` as a hint to GAP that this list is a proper range.

```
gap> r:= [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ];
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
gap> ConvertToRangeRep( r );  r;
[ 1 .. 10 ]
gap> l:= [ 1, 2, 4, 5 ];;  ConvertToRangeRep( l );  l;
[ 1, 2, 4, 5 ]
```

21.23 Enumerators

An **enumerator** is an immutable list that need not store its elements explicitly but knows, from a set of basic data, how to determine the i-th element and the position of a given object. A typical example of this is a vector space over a finite field with q elements, say, for which it is very easy to enumerate all elements using q-adic expansions of integers.

Using this enumeration can be even quicker than a binary search in a sorted list of vectors:

1 ▶ `IsQuickPositionList(` *list* `)` F

This filter indicates that a position test in *list* is quicker than about 5 or 6 element comparisons for "smaller". If this is the case it can be beneficial to use `Position` in *list* and a bit list than ordered lists to represent subsets of *list*.

On the one hand, element access to an enumerator may take more time than element access to an internally represented list containing the same elements. On the other hand, an enumerator may save a vast amount of memory. Take for example a permutation group of size a few millions. Even for moderate degree it is unlikely that a list of all its elements will fit into memory whereas it is no problem to construct an enumerator from a stabilizer chain (see 41.5).

There are situations where one only wants to loop over the elements of a domain, without using the special facilities of an enumerator, namely the particular order of elements and the possibility to find the position of elements. For such cases, GAP provides iterators (see 28.7).

The functions `Enumerator` and `EnumeratorSorted` (see 28.2.2 and 28.2.3) return enumerators of domains. Most of the special implementations of enumerators in the GAP library are based on the general interface that is provided by `EnumeratorByFunctions` (see 28.2.4); one generic example is `EnumeratorByBasis` (see 59.5.5), which can be used to get an enumerator of a finite dimensional free module.

Also enumerators for non-domains can be implemented via `EnumeratorByFunctions`; for a discussion, see 3.12 in "Programming in GAP".

22 Boolean Lists

This chapter describes boolean lists. A **boolean list** is a list that has no holes and contains only the boolean values `true` and `false` (see Chapter 20). In function names we call boolean lists **blist** for brevity.

1 ▶ `IsBlist( obj )` C

A boolean list ("blist") is a list that has no holes and contains only `true` and `false`. If a list is known to be a boolean list by a test with `IsBlist` it is stored in a compact form. See 22.4.

```
gap> IsBlist( [ true, true, false, false ] );
true
gap> IsBlist( [] );
true
gap> IsBlist( [false,,true] );  # has holes
false
gap> IsBlist( [1,1,0,0] );      # contains not only boolean values
false
gap> IsBlist( 17 );             # is not even a list
false
```

Boolean lists are lists and all operations for lists are therefore applicable to boolean lists.

Boolean lists can be used in various ways, but maybe the most important application is their use for the description of **subsets** of finite sets. Suppose *set* is a finite set, represented as a list. Then a subset *sub* of *set* is represented by a boolean list *blist* of the same length as *set* such that *blist*`[`*i*`]` is `true` if *set*`[`*i*`]` is in *sub* and `false` otherwise.

22.1 Boolean Lists Representing Subsets

1 ▶ `BlistList( list, sub )` F

returns a new boolean list that describes the list *sub* as a sublist of the dense list *list*. That is `BlistList` returns a boolean list *blist* of the same length as *list* such that *blist*`[`*i*`]` is `true` if *list*`[`*i*`]` is in *sub* and `false` otherwise.

list need not be a proper set (see 21.19), even though in this case `BlistList` is most efficient. In particular *list* may contain duplicates. *sub* need not be a proper sublist of *list*, i.e., *sub* may contain elements that are not in *list*. Those elements of course have no influence on the result of `BlistList`.

```
gap> BlistList( [1..10], [2,3,5,7] );
[ false, true, true, false, true, false, true, false, false, false ]
gap> BlistList( [1,2,3,4,5,2,8,6,4,10], [4,8,9,16] );
[ false, false, false, true, false, false, true, false, true, false ]
```

See also 22.3.2.

2 ▶ `ListBlist(` *list*`,` *blist* `)` O

returns the sublist *sub* of the list *list*, which must have no holes, represented by the boolean list *blist*, which must have the same length as *list*. *sub* contains the element *list*`[`*i*`]` if *blist*`[`*i*`]` is `true` and does not contain the element if *blist*`[`*i*`]` is `false`. The order of the elements in *sub* is the same as the order of the corresponding elements in *list*.

```
gap> ListBlist([1..8],[false,true,true,true,true,false,true,true]);
[ 2, 3, 4, 5, 7, 8 ]
gap> ListBlist( [1,2,3,4,5,2,8,6,4,10],
> [false,false,false,true,false,false,true,false,true,false] );
[ 4, 8, 4 ]
```

3 ▶ `SizeBlist(` *blist* `)` F

returns the number of entries of the boolean list *blist* that are `true`. This is the size of the subset represented by the boolean list *blist*.

```
gap> SizeBlist( [ false, true, false, true, false ] );
2
```

4 ▶ `IsSubsetBlist(` *blist1*`,` *blist2* `)` F

returns `true` if the boolean list *blist2* is a subset of the boolean list *list1*, which must have equal length, and `false` otherwise. *blist2* is a subset of *blist1* if *blist1*`[`*i*`] = `*blist1*`[`*i*`] or` *blist2*`[`*i*`]` for all *i*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> IsSubsetBlist( blist1, blist2 );
false
gap> blist2 := [ true, false, false, false ];;
gap> IsSubsetBlist( blist1, blist2 );
true
```

22.2 Set Operations via Boolean Lists

1 ▶ `UnionBlist(` *blist1*`,` *blist2*`[, ...] )` F
▶ `UnionBlist(` *list* `)` F

In the first form `UnionBlist` returns the union of the boolean lists *blist1*, *blist2*, etc., which must have equal length. The **union** is a new boolean list such that *union*`[`*i*`] = `*blist1*`[`*i*`] or` *blist2*`[`*i*`] or ...`.

The second form takes the union of all blists (which as for the first form must have equal length) in the list *list*.

2 ▶ `IntersectionBlist(` *blist1*`,` *blist2*`[, ...] )` F
▶ `IntersectionBlist(` *list* `)` F

In the first form `IntersectionBlist` returns the intersection of the boolean lists *blist1*, *blist2*, etc., which must have equal length. The **intersection** is a new blist such that *inter*`[`*i*`] = `*blist1*`[`*i*`] and` *blist2*`[`*i*`] and ...`.

In the second form *list* must be a list of boolean lists *blist1*, *blist2*, etc., which must have equal length, and `IntersectionBlist` returns the intersection of those boolean lists.

3 ▶ `DifferenceBlist(` *blist1*`,` *blist2* `)` F

returns the asymmetric set difference (exclusive or) of the two boolean lists *blist1* and *blist2*, which must have equal length. The **asymmetric set difference** is a new boolean list such that *union*`[`*i*`] = `*blist1*`[`*i*`] and not` *blist2*`[`*i*`]`.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> UnionBlist( blist1, blist2 );
[ true, true, true, false ]
gap> IntersectionBlist( blist1, blist2 );
[ true, false, false, false ]
gap> DifferenceBlist( blist1, blist2 );
[ false, true, false, false ]
```

22.3 Function that Modify Boolean Lists

1 ▶ `UniteBlist(` *blist1*`,` *blist2* `)` F

`UniteBlist` unites the boolean list *blist1* with the boolean list *blist2*, which must have the same length. This is equivalent to assigning *blist1*`[`*i*`] := `*blist1*`[`*i*`] or `*blist2*`[`*i*`]` for all *i*. `UniteBlist` returns nothing, it is only called to change *blist1*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> UniteBlist( blist1, blist2 );
gap> blist1;
[ true, true, true, false ]
```

2 ▶ `UniteBlistList(` *list*`,` *blist*`,` *sub* `)` F

works like `UniteBlist(`*blist*`,BlistList(`*list*`,`*sub*`))`. As no intermediate blist is created, the performance is better than the separate function calls.

The function `UnionBlist` (see 22.2.1) is the nondestructive counterpart to the procedure `UniteBlist`.

3 ▶ `IntersectBlist(` *blist1*`,` *blist2* `)` F

intersects the boolean list *blist1* with the boolean list *blist2*, which must have the same length. This is equivalent to assigning *blist1*`[`*i*`]:= `*blist1*`[`*i*`] and `*blist2*`[`*i*`]` for all *i*. `IntersectBlist` returns nothing, it is only called to change *blist1*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> IntersectBlist( blist1, blist2 );
gap> blist1;
[ true, false, false, false ]
```

The function `IntersectionBlist` (see 22.2.2) is the nondestructive counterpart to the procedure `IntersectBlist`.

4 ▶ `SubtractBlist(` *blist1*`,` *blist2* `)` F

subtracts the boolean list *blist2* from the boolean list *blist1*, which must have equal length. This is equivalent to assigning *blist1*`[`*i*`] := `*blist1*`[`*i*`] and not `*blist2*`[`*i*`]` for all *i*. `SubtractBlist` returns nothing, it is only called to change *blist1*.

```
gap> blist1 := [ true, true, false, false ];;
gap> blist2 := [ true, false, true, false ];;
gap> SubtractBlist( blist1, blist2 );
gap> blist1;
[ false, true, false, false ]
```

The function `DifferenceBlist` (see 22.2.3) is the nondestructive counterpart to the procedure `SubtractBlist`.

22.4 More about Boolean Lists

We defined a boolean list as a list that has no holes and contains only `true` and `false`. There is a special internal representation for boolean lists that needs only 1 bit for each entry. This bit is set if the entry is `true` and reset if the entry is `false`. This representation is of course much more compact than the ordinary representation of lists, which needs (at least) 32 bits per entry.

Not every boolean list is represented in this compact representation. It would be too much work to test every time a list is changed, whether this list has become a boolean list. This section tells you under which circumstances a boolean list is represented in the compact representation, so you can write your functions in such a way that you make best use of the compact representation.

The results of `BlistList`, `UnionBlist`, `IntersectionBlist` and `DifferenceBlist` are known to be boolean lists by construction, and thus are represented in the compact representation upon creation.

If an argument of `IsBlist`, `IsSubsetBlist`, `ListBlist`, `UnionBlist`, `IntersectionBlist`, `DifferenceBlist`, `UniteBlist`, `IntersectBlist` and `SubtractBlist` is a list represented in the ordinary representation, it is tested to see if it is in fact a boolean list. If it is not, `IsBlist` returns `false` and the other functions signal an error. If it is, the representation of the list is changed to the compact representation.

If you change a boolean list that is represented in the compact representation by assignment (see 21.4) or `Add` (see 21.4.4) in such a way that the list remains a boolean list it will remain represented in the compact representation. Note that changing a list that is not represented in the compact representation, whether it is a boolean list or not, in such a way that the resulting list becomes a boolean list, will never change the representation of the list.

23 Row Vectors

Just as in mathematics, a vector in GAP is any object which supports appropriate addition and scalar multiplication operations (see Chapter 59). As in mathematics, an especially important class of vectors are those represented by a list of coefficients with respect to some basis. These correspond roughly to the GAP concept of **row vectors**.

1 ▶ `IsRowVector(` *obj* `)` C

A **row vector** is a vector (see 30.14.14) that is also a homogeneous list of odd additive nesting depth (see 21.12). Typical examples are lists of integers and rationals, lists of finite field elements of the same characteristic, and lists of polynomials from a common polynomial ring. Note that matrices are **not** regarded as row vectors, because they have even additive nesting depth.

The additive operations of the vector must thus be compatible with that for lists, implying that the list entries are the coefficients of the vector with respect to some basis.

Note that not all row vectors admit a multiplication via `*` (which is to be understood as a scalar product); for example, class functions are row vectors but the product of two class functions is defined in a different way. For the installation of a scalar product of row vectors, the entries of the vector must be ring elements; note that the default method expects the row vectors to lie in `IsRingElementList`, and this category may not be implied by `IsRingElement` for all entries of the row vector (see the comment for `IsVector` in 30.14.14).

Note that methods for special types of row vectors really must be installed with the requirement `IsRowVector`, since `IsVector` may lead to a rank of the method below that of the default method for row vectors (see file `lib/vecmat.gi`).

```
gap> IsRowVector([1,2,3]);
true
```

Because row vectors are just a special case of lists, all operations and functions for lists are applicable to row vectors as well (see Chapter 21). This especially includes accessing elements of a row vector (see 21.3), changing elements of a mutable row vector (see 21.4), and comparing row vectors (see 21.10).

Note that, unless your algorithms specifically require you to be able to change entries of your vectors, it is generally better and faster to work with immutable row vectors. See Section 12.6 for more details.

23.1 Operators for Row Vectors

The rules for arithmetic operations involving row vectors are in fact special cases of those for the arithmetic of lists, as given in Section 21.11 and the following sections, here we reiterate that definition, in the language of vectors.

Note that the additive behaviour sketched below is defined only for lists in the category `IsGeneralizedRowVector`, and the multiplicative behaviour is defined only for lists in the category `IsMultiplicativeGeneralizedRowVector` (see 21.12).

1 ▶ *vec1* `+` *vec2* O

returns the sum of the two row vectors *vec1* and *vec2*. Probably the most usual situation is that *vec1* and *vec2* have the same length and are defined over a common field; in this case the sum is a new row vector over the same field where each entry is the sum of the corresponding entries of the vectors.

In more general situations, the sum of two row vectors need not be a row vector, for example adding an integer vector *vec1* and a vector *vec2* over a finite field yields the list of pointwise sums, which will be a mixture of finite field elements and integers if *vec1* is longer than *vec2*.

2 ▶ *scalar* `+` *vec* O
▶ *vec* `+` *scalar* O

returns the sum of the scalar *scalar* and the row vector *vec*. Probably the most usual situation is that the elements of *vec* lie in a common field with *scalar*; in this case the sum is a new row vector over the same field where each entry is the sum of the scalar and the corresponding entry of the vector.

More general situations are for example the sum of an integer scalar and a vector over a finite field, or the sum of a finite field element and an integer vector.

```
gap> [ 1, 2, 3 ] + [ 1/2, 1/3, 1/4 ];
[ 3/2, 7/3, 13/4 ]
gap>  [ 1/2, 3/2, 1/2 ] + 1/2;
[ 1, 2, 1 ]
```

3 ▶ *vec1* `-` *vec2* O
▶ *scalar* `-` *vec* O
▶ *vec* `-` *scalar* O

Subtracting a vector or scalar is defined as adding its additive inverse, so the statements for the addition hold likewise.

```
gap> [ 1, 2, 3 ] - [ 1/2, 1/3, 1/4 ];
[ 1/2, 5/3, 11/4 ]
gap> [ 1/2, 3/2, 1/2 ] - 1/2;
[ 0, 1, 0 ]
```

4 ▶ *scalar* `*` *vec* O
▶ *vec* `*` *scalar* O

returns the product of the scalar *scalar* and the row vector *vec*. Probably the most usual situation is that the elements of *vec* lie in a common field with *scalar*; in this case the product is a new row vector over the same field where each entry is the product of the scalar and the corresponding entry of the vector.

More general situations are for example the product of an integer scalar and a vector over a finite field, or the product of a finite field element and an integer vector.

```
gap> [ 1/2, 3/2, 1/2 ] * 2;
[ 1, 3, 1 ]
```

5 ▶ *vec1* `*` *vec2* O

returns the standard scalar product of *vec1* and *vec2*, i.e., the sum of the products of the corresponding entries of the vectors. Probably the most usual situation is that *vec1* and *vec2* have the same length and are defined over a common field; in this case the sum is an element of this field.

More general situations are for example the inner product of an integer vector and a vector over a finite field, or the inner product of two row vectors of different lengths.

```
gap> [ 1, 2, 3 ] * [ 1/2, 1/3, 1/4 ];
23/12
```

For the mutability of results of arithmetic operations, see 12.6.

Further operations with vectors as operands are defined by the matrix operations (see 24.2).

6 ► `NormedRowVector( v )` A

returns a scalar multiple w = c * v of the row vector v with the property that the first nonzero entry of w is an identity element in the sense of `IsOne`.

```
gap> NormedRowVector([5,2,3]);
[ 1, 2/5, 3/5 ]
```

23.2 Row Vectors over Finite Fields

GAP can use compact formats to store row vectors over fields of order at most 256, based on those used by the Meat-Axe [Rin93]. This format also permits extremely efficient vector arithmetic. On the other hand element access and assignment is significantly slower than for plain lists.

The function `ConvertToVectorRep` is used to convert a list into a compressed vector, or to rewrite a compressed vector over another field. Note that this function is **much** faster when it is given a field (or field size) as an argument, rather than having to scan the vector and try to decide the field. Supplying the field can also avoid errors and/or loss of performance, when one vector from some collection happens to have all of its entries over a smaller field than the "natural" field of the problem.

1 ► `ConvertToVectorRep( list )` F
► `ConvertToVectorRep( list , field )` F
► `ConvertToVectorRep( list , fieldsize )` F
► `ConvertToVectorRepNC( list )` F
► `ConvertToVectorRepNC( list , field )` F
► `ConvertToVectorRepNC( list , fieldsize )` F

`ConvertToVectorRep( list )` converts *list* to an internal vector representation if possible.

`ConvertToVectorRep( list , field )` converts *list* to an internal vector representation appropriate for a vector over *field*.

It is forbidden to call this function unless *list* is a plain list or a vector, *field* a field, and all elements of *list* lie in *field*, violation of this condition can lead to unpredictable behaviour or a system crash. (Setting the assertion level to at least 2 might catch some violations before a crash, see 7.5.1.)

Instead of a *field* also its size *fieldsize* may be given.

list may already be a compressed vector. In this case, if no *field* or *fieldsize* is given, then nothing happens. If one is given then the vector is rewritten as a compressed vector over the given *field* unless it has the filter `IsLockedRepresentationVector`, in which case it is not changed.

The return value is the size of the field over which the vector ends up written, if it is written in a compressed representation.

In this example, we first create a row vector and then ask GAP to rewrite it, first over GF(2) and then over GF(4).

```
gap> v := [Z(2)^0,Z(2),Z(2),0*Z(2)];
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> RepresentationsOfObject(v);
[ "IS_PLIST_REP", "IsInternalRep" ]
gap> ConvertToVectorRep(v);
2
gap> v;
<a GF2 vector of length 4>
gap> ConvertToVectorRep(v,4);
4
gap> v;
[ Z(2)^0, Z(2)^0, Z(2)^0, 0*Z(2) ]
gap> RepresentationsOfObject(v);
[ "IsDataObjectRep", "Is8BitVectorRep" ]
```

A vector in the special representation over $GF(2)$ is always viewed as `<a GF2 vector of length ...>`. Over fields of orders 3 to 256, a vector of length 10 or less is viewed as the list of its coefficients, but a longer one is abbreviated.

Arithmetic operations (see 21.11 and the following sections) preserve the compression status of row vectors in the sense that if all arguments are compressed row vectors written over the same field and the result is a row vector then also the result is a compressed row vector written over this field.

2 ► `NumberFFVector(` *vec*, *sz* `)` O

returns an integer that gives the position of the finite field row vector (*vec*) in the sorted list of all row vectors over the field with *sz* elements in the same dimension as *vec*. `NumberFFVector` returns `fail` if the vector cannot be represented over the field with *sz* elements.

23.3 Coefficient List Arithmetic

The following operations all perform arithmetic on row vectors. given as homogeneous lists of the same length, containing elements of a commutative ring.

There are two reasons for using `AddRowVector` in preference to arithmetic operators. Firstly, the three argument form has no single-step equivalent. Secondly `AddRowVector` changes its first argument in-place, rather than allocating a new vector to hold the result, and may thus produce less garbage.

1 ► `AddRowVector(` *dst*, *src*, `[` *mul* `[,` *from*, *to*`]] )` O

Adds the product of *src* and *mul* to *dst*, changing *dst*. If *from* and *to* are given then only the index range `[`*from*`..`*to*`]` is guaranteed to be affected. Other indices MAY be affected, if it is more convenient to do so. Even when *from* and *to* are given, *dst* and *src* must be row vectors of the **same** length.

If *mul* is not given either then this Operation simply adds *src* to *dst*.

2 ► `AddCoeffs(` *list1*, *poss1*, *list2*, *poss2*, *mul* `)` O
► `AddCoeffs(` *list1*, *list2*, *mul* `)` O
► `AddCoeffs(` *list1*, *list2* `)` O

`AddCoeffs` adds the entries of *list2*`{`*poss2*`}`, multiplied by the scalar *mul*, to *list1*`{`*poss1*`}`. Non-existing entries in *list1* are assumed to be zero. The position of the right-most non-zero element is returned.

If the ranges *poss1* and *poss2* are not given, they are assumed to span the whole vectors. If the scalar *mul* is omitted, one is used as a default.

Note that it is the responsibility of the caller to ensure that the *list2* has elements at position *poss2* and that the result (in *list1*) will be a dense list.

The function is free to remove trailing (right-most) zeros.

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;AddCoeffs(l,m);
4
gap> l;
[ 6, 8, 10, 4 ]
```

3 ▶ `MultRowVector(` *list1*`,` *poss1*`,` *list2*`,` *poss2*`,` *mul* `)` O
▶ `MultRowVector(` *list*`,` *mul* `)` O

The five-argument version of this Operation replaces *list1*`[`*poss1*`[`*i*`]]` by *mul*`*`*list2*`[`*poss2*`[`*i*`]]` for *i* between 1 and `Length(`*poss1*`)`.

The two-argument version simply multiplies each element of *list*, in-place, by *mul*.

4 ▶ `CoeffsMod(` *list1*`, [`*len1*`, ]` *mod* `)` O

returns the coefficient list obtained by reducing the entries in *list1* modulo *mod*. After reducing it shrinks the list to remove trailing zeroes.

```
gap> l:=[1,2,3,4];;CoeffsMod(l,2);
[ 1, 0, 1 ]
```

23.4 Shifting and Trimming Coefficient Lists

The following functions change coefficient lists by shifting or trimming.

1 ▶ `LeftShiftRowVector(` *list*`,` *shift* `)` O

changes *list* by assigning *list*`[i]:=`*list*`[i+`*shift*`]` and removing the last *shift* entries of the result.

2 ▶ `RightShiftRowVector(` *list*`,` *shift*`,` *fill* `)` O

changes *list* by assigning *list*`[i+`*shift*`]:=`*list*`[i]` and filling each of the *shift* first entries with *fill*.

3 ▶ `ShrinkRowVector(` *list* `)` O

removes trailing zeroes from the list *list*.

4 ▶ `RemoveOuterCoeffs(` *list*`,` *coef* `)` O

removes *coef* at the beginning and at the end of *list* and returns the number of elements removed at the beginning.

```
gap> l:=[1,1,2,1,2,1,1,2,1];;RemoveOuterCoeffs(l,1);
2
gap> l;
[ 2, 1, 2, 1, 1, 2 ]
```

23.5 Functions for Coding Theory

The following functions perform operations on Finite fields vectors considered as code words in a linear code.

1 ▶ `WeightVecFFE(` *vec* `)` O

returns the weight of the finite field vector *vec*, i.e. the number of nonzero entries.

2 ▶ `DistanceVecFFE(` *vec1*`,` *vec2* `)` O

returns the distance between the two vectors *vec1* and *vec2*, which must have the same length and whose elements must lie in a common field. The distance is the number of places where *vec1* and *vec2* differ.

3 ▶ `DistancesDistributionVecFFEsVecFFE(` *vecs*`,` *vec* `)` O

returns the distances distribution of the vector *vec* to the vectors in the list *vecs*. All vectors must have the same length, and all elements must lie in a common field. The distances distribution is a list d of length `Length(`*vec*`)+1`, such that the value d`[`i`]` is the number of vectors in *vecs* that have distance i`+1` to *vec*.

4 ▶ `DistancesDistributionMatFFEVecFFE(` *mat*`,` f`,` *vec* `)` O

returns the distances distribution of the vector *vec* to the vectors in the vector space generated by the rows of the matrix *mat* over the finite field f. The length of the rows of *mat* and the length of *vec* must be equal, and all elements must lie in f. The rows of *mat* must be linearly independent. The distances distribution is a list d of length `Length(`*vec*`)+1`, such that the value d`[`i`]` is the number of vectors in the vector space generated by the rows of *mat* that have distance i`+1` to *vec*.

5 ▶ `AClosestVectorCombinationsMatFFEVecFFE(` *mat*`,` f`,` *vec*`,` l`,` *stop* `)` O
▶ `AClosestVectorCombinationsMatFFEVecFFECoords(` *mat*`,` f`,` *vec*`,` l`,` *stop* `)` O

These functions run through the f-linear combinations of the vectors in the rows of the matrix *mat* that can be written as linear combinations of exactly l rows (that is without using zero as a coefficient). The length of the rows of *mat* and the length of *vec* must be equal, and all elements must lie in f. The rows of *mat* must be linearly independent. `AClosestVectorCombinationsMatFFEVecFFE` returns a vector from these that is closest to the vector *vec*. If it finds a vector of distance at most *stop*, which must be a nonnegative integer, then it stops immediately and returns this vector.

`AClosestVectorCombinationsMatFFEVecFFECoords` returns a length 2 list containing the same closest vector and also a vector v with exactly l non-zero entries, such that v times *mat* is the closest vector.

6 ▶ `CosetLeadersMatFFE(` *mat*`,` f `)` O

returns a list of representatives of minimal weight for the cosets of a code. *mat* must be a **check matrix** for the code, the code is defined over the finite field f. All rows of *mat* must have the same length, and all elements must lie in f. The rows of *mat* must be linearly independent.

23.6 Vectors as coefficients of polynomials

A list of ring elements can be interpreted as a row vector or the list of coefficients of a polynomial. There are a couple of functions that implement arithmetic operations based on these interpretations. GAP contains proper support for polynomials (see 64), the operations described in this section are on a lower level.

The following operations all perform arithmetic on univariate polynomials given by their coefficient lists. These lists can have different lengths but must be dense homogeneous lists containing elements of a commutative ring. Not all input lists may be empty.

In the following descriptions we will always assume that *list1* is the coefficient list of the polynomial *pol1* and so forth. If length parameter *leni* is not given, it is set to the length of *listi* by default.

1 ▶ `ValuePol( coeff, x )` F

Let *coeff* be the coefficients list of a univariate polynomial f, and x a ring element. Then `ValuePol` returns the value $f(x)$.

The coefficient of x^i is assumed to be stored at position $i+1$ in the coefficients list.

```
gap> ValuePol([1,2,3],4);
57
```

2 ▶ `ProductCoeffs( list1, [len1, ] list2 [, len2] )` O

Let *pol1* (and *pol2*) be polynomials given by the first *len1* (*len2*) entries of the coefficient list *list2* (*list2*). If *len1* and *len2* are omitted, they default to the lengths of *list1* and *list2*. This operation returns the coefficient list of the product of *pol1* and *pol2*.

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;ProductCoeffs(l,m);
[ 5, 16, 34, 52, 45, 28 ]
```

3 ▶ `ReduceCoeffs( list1 [, len1], list2 [, len2] )` O

changes *list1* to the coefficient list of the remainder when dividing *pol1* by *pol2*. This operation changes *list1* which therefore must be a mutable list. The operations returns the position of the last non-zero entry of the result but is not guaranteed to remove trailing zeroes.

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;ReduceCoeffs(l,m);
2
gap> l;
[ 64/49, -24/49, 0, 0 ]
```

4 ▶ `ReduceCoeffsMod( list1, [len1, ] list2, [len2, ] mod )` O

changes *list1* to the coefficient list of the remainder when dividing *pol1* by *pol2* modulo *mod*. *mod* must be a positive integer. This operation changes *list1* which therefore must be a mutable list. The operations returns the position of the last non-zero entry of the result but is not guaranteed to remove trailing zeroes.

```
gap> l:=[1,2,3,4];;m:=[5,6,7];;ReduceCoeffsMod(l,m,3);
1
gap> l;
[ 1, 0, 0, 0 ]
```

5 ▶ `PowerModCoeffs( list1[, len1], exp, list2[, len2] )` O

Let p_1 and p_2 be polynomials whose coefficients are given by the first *len1* resp. *len2* entries of the lists *list1* and *list2*, respectively. If *len1* and *len2* are omitted, they default to the lengths of *list1* and *list2*. Let *exp* be a positive integer. `PowerModCoeffs` returns the coefficient list of the remainder when dividing the *exp*-th power of p_1 by p_2. The coefficients are reduced already while powers are computed, therefore avoiding an explosion in list length.

```
gap> l:= [1,2,3,4];; m:= [5,6,7];; PowerModCoeffs(l,5,m);
[ -839462813696/678223072849, -7807439437824/678223072849 ]
gap> EuclideanRemainder( UnivariatePolynomial( Rationals, l )^5,
>        UnivariatePolynomial( Rationals, m ) );
-7807439437824/678223072849*x_1-839462813696/678223072849
```

6 ▶ `ShiftedCoeffs( list, shift )` O

produces a new coefficient list *new* obtained by the rule *new*`[i+`*shift*`]:=`*list*`[i]` and filling initial holes by the appropriate zero.

```
gap> l:=[1,2,3];;ShiftedCoeffs(l,2);ShiftedCoeffs(l,-2);
[ 0, 0, 1, 2, 3 ]
[ 3 ]
```

7 ► `ShrinkCoeffs(` *list* `)` O

removes trailing zeroes from *list*. It returns the position of the last non-zero entry, that is the length of *list* after the operation.

```
gap> l:=[1,0,0];;ShrinkCoeffs(l);l;
1
[ 1 ]
```

24 Matrices

Matrices are represented in GAP by lists of row vectors (see 23). The vectors must all have the same length, and their elements must lie in a common ring. However, since checking rectangularness can be expensive functions and methods of operations for matrices often will not give an error message for non-rectangular lists of lists – in such cases the result is undefined.

Because matrices are just a special case of lists, all operations and functions for lists are applicable to matrices also (see chapter 21). This especially includes accessing elements of a matrix (see 21.3), changing elements of a matrix (see 21.4), and comparing matrices (see 21.10).

Note that, since a matrix is a list of lists, the behaviour of `ShallowCopy` for matrices is just a special case of `ShallowCopy` for lists (see 21.7); called with an immutable matrix *mat*, `ShallowCopy` returns a mutable matrix whose rows are identical to the rows of *mat*. In particular the rows are still immutable. To get a matrix whose rows are mutable, one can use `List(` *mat*`, ShallowCopy )`.

1 ▶ `InfoMatrix` V

The info class for matrix operations is `InfoMatrix`.

24.1 Categories of Matrices

1 ▶ `IsMatrix(` *obj* `)` C

A **matrix** is a list of lists of equal length whose entries lie in a common ring.

Note that matrices may have different multiplications, besides the usual matrix product there is for example the Lie product. So there are categories such as `IsOrdinaryMatrix` and `IsLieMatrix` (see 24.1.2, 24.1.3) that describe the matrix multiplication. One can form the product of two matrices only if they support the same multiplication.

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
gap> IsMatrix(mat);
true
```

Note also the filter `IsTable` (see section 21.1.4) which may be more appropriate than `IsMatrix` for some purposes.

Note that the empty list '[ ]' and more complex "empty" structures such as `[[ ]]` are **not** matrices, although special methods allow them be used in place of matrices in some situations. See 24.4.3 below.

```
gap> [[0]]*[[]];
[ [  ] ]
gap> IsMatrix([[]]);
false
```

2 ▶ `IsOrdinaryMatrix(` *obj* `)` C

An **ordinary matrix** is a matrix whose multiplication is the ordinary matrix multiplication.

Each matrix in internal representation is in the category `IsOrdinaryMatrix`, and arithmetic operations with objects in `IsOrdinaryMatrix` produce again matrices in `IsOrdinaryMatrix`.

Note that we want that Lie matrices shall be matrices that behave in the same way as ordinary matrices, except that they have a different multiplication. So we must distinguish the different matrix multiplications, in order to be able to describe the applicability of multiplication, and also in order to form a matrix of the appropriate type as the sum, difference etc. of two matrices which have the same multiplication.

3 ► `IsLieMatrix(` *mat* `)` C

A **Lie matrix** is a matrix whose multiplication is given by the Lie bracket. (Note that a matrix with ordinary matrix multiplication is in the category `IsOrdinaryMatrix`, see 24.1.2.)

Each matrix created by `LieObject` is in the category `IsLieMatrix`, and arithmetic operations with objects in `IsLieMatrix` produce again matrices in `IsLieMatrix`.

24.2 Operators for Matrices

The rules for arithmetic operations involving matrices are in fact special cases of those for the arithmetic of lists, given in Section 21.11 and the following sections, here we reiterate that definition, in the language of vectors and matrices.

Note that the additive behaviour sketched below is defined only for lists in the category `IsGeneralizedRowVector`, and the multiplicative behaviour is defined only for lists in the category `IsMultiplicativeGeneralizedRowVector` (see 21.12).

1 ► *mat1* `+` *mat2* O

returns the sum of the two matrices *mat1* and *mat2*, Probably the most usual situation is that *mat1* and *mat2* have the same dimensions and are defined over a common field; in this case the sum is a new matrix over the same field where each entry is the sum of the corresponding entries of the matrices.

In more general situations, the sum of two matrices need not be a matrix, for example adding an integer matrix *mat1* and a matrix *mat2* over a finite field yields the table of pointwise sums, which will be a mixture of finite field elements and integers if *mat1* has bigger dimensions than *mat2*.

2 ► *scalar* `+` *mat* O
► *mat* `+` *scalar* O

returns the sum of the scalar *scalar* and the matrix *mat*. Probably the most usual situation is that the entries of *mat* lie in a common field with *scalar*; in this case the sum is a new matrix over the same field where each entry is the sum of the scalar and the corresponding entry of the matrix.

More general situations are for example the sum of an integer scalar and a matrix over a finite field, or the sum of a finite field element and an integer matrix.

3 ► *mat1* `-` *mat2*
► *scalar* `-` *mat* O
► *mat* `-` *scalar* O

Subtracting a matrix or scalar is defined as adding its additive inverse, so the statements for the addition hold likewise.

4 ► *scalar* `*` *mat* O
► *mat* `*` *scalar* O

returns the product of the scalar *scalar* and the matrix *mat*. Probably the most usual situation is that the elements of *mat* lie in a common field with *scalar*; in this case the product is a new matrix over the same field where each entry is the product of the scalar and the corresponding entry of the matrix.

More general situations are for example the product of an integer scalar and a matrix over a finite field, or the product of a finite field element and an integer matrix.

5 ► *vec* `*` *mat* O

returns the product of the row vector *vec* and the matrix *mat*. Probably the most usual situation is that *vec* and *mat* have the same lengths and are defined over a common field, and that all rows of *mat* have the same length m, say; in this case the product is a new row vector of length m over the same field which is the sum of the scalar multiples of the rows of *mat* with the corresponding entries of *vec*.

More general situations are for example the product of an integer vector and a matrix over a finite field, or the product of a vector over a finite field and an integer matrix.

6 ► *mat* `*` *vec* O

returns the product of the matrix *mat* and the row vector *vec*. (This is the standard product of a matrix with a **column** vector.) Probably the most usual situation is that the length of *vec* and of all rows of *mat* are equal, and that the elements of *mat* and *vec* lie in a common field; in this case the product is a new row vector of the same length as *mat* and over the same field which is the sum of the scalar multiples of the columns of *mat* with the corresponding entries of *vec*.

More general situations are for example the product of an integer matrix and a vector over a finite field, or the product of a matrix over a finite field and an integer vector.

7 ► *mat1* `*` *mat2* O

This form evaluates to the (Cauchy) product of the two matrices *mat1* and *mat2*. Probably the most usual situation is that the number of columns of *mat1* equals the number of rows of *mat2*, and that the elements of *mat* and *vec* lie in a common field; if *mat1* is a matrix with m rows and n columns, say, and *mat2* is a matrix with n rows and o columns, the result is a new matrix with m rows and o columns. The element in row i at position j of the product is the sum of $mat1[i][l] * mat2[l][j]$, with l running from 1 to n.

8 ► `Inverse(` *mat* `)` O

returns the inverse of the matrix *mat*, which must be an invertible square matrix. If *mat* is not invertible then `fail` is returned.

9 ► *mat1* `/` *mat2* O
► *scalar* `/` *mat* O
► *mat* `/` *scalar* O
► *vec* `/` *mat* O

In general, *left* `/` *right* is defined as *left* `*` *right*`^-1`. Thus in the above forms the right operand must always be invertible.

10 ► *mat* `^` *int* O
► *mat1* `^` *mat2* O
► *vec* `^` *mat* O

Powering a square matrix *mat* by an integer *int* yields the *int*-th power of *mat*; if *int* is negative then *mat* must be invertible, if *int* is `0` then the result is the identity matrix `One(` *mat* `)`, even if *mat* is not invertible.

Powering a square matrix *mat1* by an invertible square matrix *mat2* of the same dimensions yields the conjugate of *mat1* by *mat2*, i.e., the matrix *mat2*`^-1 *` *mat1* `*` *mat2*.

Powering a row vector *vec* by a matrix *mat* is in every respect equivalent to *vec* `*` *mat*. This operations reflects the fact that matrices act naturally on row vectors by multiplication from the right, and that the powering operator is GAP's standard for group actions.

11 ▶ `Comm(` *mat1*`,` *mat2* `)` O

returns the commutator of the square invertible matrices *mat1* and *mat2* of the same dimensions and over a common field, which is the matrix *mat1*`^-1 *` *mat2*`^-1 *` *mat1* `*` *mat2*.

The following cases are still special cases of the general list arithmetic defined in 21.11.

12 ▶ *scalar* `+` *matlist* O
▶ *matlist* `+` *scalar* O
▶ *scalar* `-` *matlist* O
▶ *matlist* `-` *scalar* O
▶ *scalar* `*` *matlist* O
▶ *matlist* `*` *scalar* O
▶ *matlist* `/` *scalar* O

A scalar *scalar* may also be added, subtracted, multiplied with, or divided into a list *matlist* of matrices. The result is a new list of matrices where each matrix is the result of performing the operation with the corresponding matrix in *matlist*.

13 ▶ *mat* `*` *matlist* O
▶ *matlist* `*` *mat* O

A matrix *mat* may also be multiplied with a list *matlist* of matrices. The result is a new list of matrices, where each entry is the product of *mat* and the corresponding entry in *matlist*.

14 ▶ *matlist* `/` *mat* O

Dividing a list *matlist* of matrices by an invertible matrix *mat* evaluates to *matlist* `*` *mat*`^-1`.

15 ▶ *vec* `*` *matlist* O

returns the product of the vector *vec* and the list of matrices *mat*. The lengths l of *vec* and *matlist* must be equal. All matrices in *matlist* must have the same dimensions. The elements of *vec* and the elements of the matrices in *matlist* must lie in a common ring. The product is the sum over *vec*`[`i`]` `*` *matlist*`[`i`]` with i running from 1 to l.

For the mutability of results of arithmetic operations, see 12.6.

24.3 Properties and Attributes of Matrices

1 ▶ `DimensionsMat(` *mat* `)` A

is a list of length 2, the first being the number of rows, the second being the number of columns of the matrix *mat*.

```
gap> DimensionsMat([[1,2,3],[4,5,6]]);
[ 2, 3 ]
```

2 ▶ `DefaultFieldOfMatrix(` *mat* `)` A

For a matrix *mat*, `DefaultFieldOfMatrix` returns either a field (not necessarily the smallest one) containing all entries of *mat*, or `fail`.

If *mat* is a matrix of finite field elements or a matrix of cyclotomics, `DefaultFieldOfMatrix` returns the default field generated by the matrix entries (see 57.3 and 18.1).

```
gap> DefaultFieldOfMatrix([[Z(4),Z(8)]]);
GF(2^6)
```

3 ▶ `TraceMat( mat )` F
▶ `Trace( mat )` F

The trace of a square matrix is the sum of its diagonal entries.

```
gap> TraceMat([[1,2,3],[4,5,6],[7,8,9]]);
15
```

4 ▶ `DeterminantMat( mat )` A
▶ `Determinant( mat )` F

returns the determinant of the square matrix *mat*.

These methods assume implicitly that *mat* is defined over an integral domain whose quotient field is implemented in GAP. For matrices defined over an arbitrary commutative ring with one see 24.3.6.

5 ▶ `DeterminantMatDestructive( mat )` O

Does the same as `DeterminantMat`, with the difference that it may destroy its argument. The matrix *mat* must be mutable.

```
gap> DeterminantMat([[1,2],[2,1]]);
-3
gap> mm:= [[1,2],[2,1]];;
gap> DeterminantMatDestructive( mm );
-3
gap> mm;
[ [ 1, 2 ], [ 0, -3 ] ]
```

6 ▶ `DeterminantMatDivFree( mat )` O

returns the determinant of a square matrix *mat* over an arbitrary commutative ring with one using the division free method of Mahajan and Vinay [MV97].

7 ▶ `IsMonomialMatrix( mat )` P

A matrix is monomial if and only if it has exactly one nonzero entry in every row and every column.

```
gap> IsMonomialMatrix([[0,1],[1,0]]);
true
```

8 ▶ `IsDiagonalMat( mat )` O

returns true if mat has only zero entries off the main diagonal, false otherwise.

9 ▶ `IsUpperTriangularMat( mat )` O

returns true if mat has only zero entries below the main diagonal, false otherwise.

10 ▶ `IsLowerTriangularMat( mat )` O

returns true if mat has only zero entries below the main diagonal, false otherwise.

24.4 Matrix Constructions

1 ▶ `IdentityMat(` *m* `[,` *F*`] )` F

returns a (mutable) $m \times m$ identity matrix over the field given by F (i.e. the smallest field containing the element F or F itself if it is a field).

2 ▶ `NullMat(` *m*`,` *n* `[,` *F*`] )` F

returns a (mutable) $m \times n$ null matrix over the field given by F.

```
gap> IdentityMat(3,1);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> NullMat(3,2,Z(3));
[ [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ], [ 0*Z(3), 0*Z(3) ] ]
```

3 ▶ `EmptyMatrix(` *char* `)` F

is an empty (ordinary) matrix in characteristic *char* that can be added to or multiplied with empty lists (representing zero-dimensional row vectors). It also acts (via `^`) on empty lists.

```
gap> EmptyMatrix(5);
EmptyMatrix( 5 )
gap> AsList(last);
[  ]
```

4 ▶ `DiagonalMat(` *vector* `)` F

returns a diagonal matrix *mat* with the diagonal entries given by *vector*.

```
gap> DiagonalMat([1,2,3]);
[ [ 1, 0, 0 ], [ 0, 2, 0 ], [ 0, 0, 3 ] ]
```

5 ▶ `PermutationMat(` *perm*`,` *dim* `[,` *F* `] )` F

returns a matrix in dimension *dim* over the field given by F (i.e. the smallest field containing the element F or F itself if it is a field) that represents the permutation *perm* acting by permuting the basis vectors as it permutes points.

```
gap> PermutationMat((1,2,3),4,1);
[ [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ] ]
```

6 ▶ `TransposedMatImmutable(` *mat* `)` A
▶ `TransposedMatAttr(` *mat* `)` AM
▶ `TransposedMat(` *mat* `)` AM
▶ `TransposedMatMutable(` *mat* `)` O
▶ `TransposedMatOp(` *mat* `)` O

These functions all return the transposed of the matrix *mat*, i.e., a matrix *trans* such that *trans*`[`*i*`][`*k*`] =` *mat*`[`*k*`][`*i*`]` holds.

They differ only w.r.t. the mutability of the result.

`TransposedMat` is an attribute and hence returns an immutable result. `TransposedMatMutable` is guaranteed to return a new **mutable** matrix.

`TransposedMatImmutable` and `TransposedMatAttr` are synonyms of `TransposedMat`, and `TransposedMatOp` is a synonym of `TransposedMatMutable`, in analogy to operations such as `Zero` (see 30.10.3).

7 ▶ `TransposedMatDestructive( mat )` O

If *mat* is a mutable matrix, then the transposed is computed by swapping the entries in *mat*. In this way *mat* gets changed. In all other cases the transposed is computed by `TransposedMat`.

```
gap> TransposedMat([[1,2,3],[4,5,6],[7,8,9]]);
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
gap> mm:= [[1,2,3],[4,5,6],[7,8,9]];;
gap> TransposedMatDestructive( mm );
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
gap> mm;
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
```

8 ▶ `KroneckerProduct( mat1, mat2 )` O

The Kronecker product of two matrices is the matrix obtained when replacing each entry *a* of *mat1* by the product *a*mat2* in one matrix.

```
gap> KroneckerProduct([[1,2]],[[5,7],[9,2]]);
[ [ 5, 7, 10, 14 ], [ 9, 2, 18, 4 ] ]
```

9 ▶ `ReflectionMat( coeffs )` F
▶ `ReflectionMat( coeffs, root )` F
▶ `ReflectionMat( coeffs, conj )` F
▶ `ReflectionMat( coeffs, conj, root )` F

Let *coeffs* be a row vector. `ReflectionMat` returns the matrix of the reflection in this vector.

More precisely, if *coeffs* is the coefficients of a vector v w.r.t. a basis B (see 59.4.2), say, then the returned matrix describes the reflection in v w.r.t. B as a map on a row space, with action from the right.

The optional argument *root* is a root of unity that determines the order of the reflection. The default is a reflection of order 2. For triflections one should choose a third root of unity etc. (see 18.1.1).

conj is a function of one argument that conjugates a ring element. The default is `ComplexConjugate`.

The matrix of the reflection in v is defined as

$$M = I_n + \overline{v^{tr}} \cdot \frac{w-1}{v\overline{v^{tr}}} \cdot v$$

where w = `root`, n is the length of the coefficient list, and $\overline{}$ denotes the conjugation.

10 ▶ `PrintArray( array )` F

pretty-prints the array *array*.

11 ▶ `MutableIdentityMat( m [, F] )` F

returns a (mutable) $m \times m$ identity matrix over the field given by F. This is identical to `IdentityMat` and is present in GAP 4.1 only for the sake of compatibility with beta-releases. It should **not** be used in new code.

12 ▶ `MutableNullMat( m, n [, F] )` F

returns a (mutable) $m \times n$ null matrix over the field given by F. This is identical to `NullMat` and is present in GAP 4.1 only for the sake of compatibility with beta-releases. It should **not** be used in new code.

24.5 Random Matrices

1 ▶ `RandomMat(` *m*, *n* `[,` *R*`] )` F

`RandomMat` returns a new mutable random matrix with m rows and n columns with elements taken from the ring R, which defaults to `Integers`.

2 ▶ `RandomInvertibleMat(` *m* `[,` *R*`] )` F

`RandomInvertibleMat` returns a new mutable invertible random matrix with m rows and columns with elements taken from the ring R, which defaults to `Integers`.

3 ▶ `RandomUnimodularMat(` *m* `)` F

returns a new random mutable $m \times m$ matrix with integer entries that is invertible over the integers.

```
gap> RandomMat(2,3,GF(3));
[ [ 0*Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3)^0 ] ]
gap> RandomInvertibleMat(4);
[ [ 4, 3, 5, 3 ], [ 1, -2, 1, -1 ], [ -2, 4, -1, -3 ], [ -1, 1, 2, 0 ] ]
```

24.6 Matrices Representing Linear Equations and the Gaussian Algorithm

1 ▶ `RankMat(` *mat* `)` A

If *mat* is a matrix whose rows span a free module over the ring generated by the matrix entries and their inverses then `RankMat` returns the dimension of this free module. Otherwise `fail` is returned.

Note that `RankMat` may perform a Gaussian elimination. For large rational matrices this may take very long, because the entries may become very large.

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> RankMat(mat);
2
```

2 ▶ `TriangulizeMat(` *mat* `)` O

applies the Gaussian Algorithm to the mutable matrix *mat* and changes *mat* such that it is in upper triangular normal form (sometimes called "Hermite normal form").

```
gap> m:=TransposedMatMutable(mat);
[ [ 1, 4, 7 ], [ 2, 5, 8 ], [ 3, 6, 9 ] ]
gap> TriangulizeMat(m);m;
[ [ 1, 0, -1 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]
```

3 ▶ `NullspaceMat(` *mat* `)` A
▶ `TriangulizedNullspaceMat(` *mat* `)` A

returns a list of row vectors that form a basis of the vector space of solutions to the equation *vec*`*`*mat*`=0`. The result is an immutable matrix. This basis is not guaranteed to be in any specific form.

The variant `TriangulizedNullspaceMat` returns a basis of the nullspace in triangulized form as is often needed for algorithms.

4 ▶ `NullspaceMatDestructive(` *mat* `)` O
▶ `TriangulizedNullspaceMatDestructive(` *mat* `)` O

This function does the same as `NullspaceMat`. However, the latter function makes a copy of *mat* to avoid having to change it. This function does not do that; it returns the null space and may destroy *mat*; this saves a lot of memory in case *mat* is big. The matrix *mat* must be mutable.

The variant `TriangulizedNullspaceMatDestructive` returns a basis of the nullspace in triangulized form. It may destroy the matrix *mat*.

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> NullspaceMat(mat);
[ [ 1, -2, 1 ] ]
gap> mm:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> NullspaceMatDestructive( mm );
[ [ 1, -2, 1 ] ]
gap> mm;
[ [ 1, 2, 3 ], [ 0, -3, -6 ], [ 0, 0, 0 ] ]
```

5 ▶ `SolutionMat( `*mat*`, `*vec*` )` O

returns a row vector x that is a solution of the equation x `*` *mat* `=` *vec*. It returns `fail` if no such vector exists.

6 ▶ `SolutionMatDestructive( `*mat*`, `*vec*` )` O

Does the same as `SolutionMat( `*mat*`, `*vec*` )` except that it may destroy the matrix *mat*. The matrix *mat* must be mutable.

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> SolutionMat(mat,[3,5,7]);
[ 5/3, 1/3, 0 ]
gap> mm:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> SolutionMatDestructive( mm, [3,5,7] );
[ 5/3, 1/3, 0 ]
gap> mm;
[ [ 1, 2, 3 ], [ 0, -3, -6 ], [ 0, 0, 0 ] ]
```

7 ▶ `BaseFixedSpace( `*mats*` )` F

`BaseFixedSpace` returns a list of row vectors that form a base of the vector space V such that $vM = v$ for all v in V and all matrices M in the list *mats*. (This is the common eigenspace of all matrices in *mats* for the eigenvalue 1.)

```
gap> BaseFixedSpace([[[1,2],[0,1]]]);
[ [ 0, 1 ] ]
```

24.7 Eigenvectors and eigenvalues

1 ▶ `GeneralisedEigenvalues( `F`, `A` )` O
▶ `GeneralizedEigenvalues( `F`, `A` )` O

The generalised eigenvalues of the matrix A over the field F.

2 ▶ `GeneralisedEigenspaces( `F`, `A` )` O
▶ `GeneralizedEigenspaces( `F`, `A` )` O

The generalised eigenspaces of the matrix A over the field F.

3 ▶ `Eigenvalues( `F`, `A` )` O

The eigenvalues of the matrix A over the field F.

4 ▶ `Eigenspaces( `F`, `A` )` O

The eigenspaces of the matrix A over the field F.

5 ▶ `Eigenvectors( `F`, `A` )` O

The eigenspaces of the matrix A over the field F.

24.8 Elementary Divisors

See also chapter 25.

1 ► `ElementaryDivisorsMat( [`*ring*`, ] `*mat*` )` O
► `ElementaryDivisorsMatDestructive( `*ring*`, `*mat*` )` F

`ElementaryDivisors` returns a list of the elementary divisors, i.e., the unique d with d`[`i`]` divides d`[`i`+1]` and *mat* is equivalent to a diagonal matrix with the elements d`[`i`]` on the diagonal. The operations are performed over the ring *ring*, which must contain all matrix entries. For compatibility reasons it can be omitted and defaults to `Integers`.

The function `ElementaryDivisorsMatDestructive` produces the same result but in the process destroys the contents of *mat*.

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> ElementaryDivisorsMat(mat);
[ 1, 3, 0 ]
gap> x:=X(Rationals,"x");;
gap> mat:=mat*One(x)-x*mat^0;
[ [ -x+1, 2, 3 ], [ 4, -x+5, 6 ], [ 7, 8, -x+9 ] ]
gap> ElementaryDivisorsMat(PolynomialRing(Rationals,1),mat);
[ 1, 1, x^3-15*x^2-18*x ]
gap> mat:=KroneckerProduct(CompanionMat((x-1)^2),CompanionMat((x^3-1)*(x-1)));;
gap> mat:=mat*One(x)-x*mat^0;
[ [ -x, 0, 0, 0, 0, 0, 0, 1 ], [ 0, -x, 0, 0, -1, 0, 0, -1 ],
  [ 0, 0, -x, 0, 0, -1, 0, 0 ], [ 0, 0, 0, -x, 0, 0, -1, -1 ],
  [ 0, 0, 0, -1, -x, 0, 0, -2 ], [ 1, 0, 0, 1, 2, -x, 0, 2 ],
  [ 0, 1, 0, 0, 0, 2, -x, 0 ], [ 0, 0, 1, 1, 0, 0, 2, -x+2 ] ]
gap> ElementaryDivisorsMat(PolynomialRing(Rationals,1),mat);
[ 1, 1, 1, 1, 1, 1, x-1, x^7-x^6-2*x^4+2*x^3+x-1 ]
```

2 ► `DiagonalizeMat( `*ring*`, `*mat*` )` O

brings the mutable matrix *mat*, considered as a matrix over *ring*, into diagonal form by elementary row and column operations.

```
gap> m:=[[1,2],[2,1]];;
gap> DiagonalizeMat(Integers,m);m;
[ [ 1, 0 ], [ 0, 3 ] ]
```

24.9 Echelonized Matrices

1 ► `SemiEchelonMat( `*mat*` )` A

A matrix over a field F is in semi-echelon form if the first nonzero element in each row is the identity of F, and all values exactly below these pivots are the zero of F.

`SemiEchelonMat` returns a record that contains information about a semi-echelonized form of the matrix *mat*.

The components of this record are

`vectors`
: list of row vectors, each with pivot element the identity of F,

`heads`
: list that contains at position i, if nonzero, the number of the row for that the pivot element is in column i.

2 ▶ `SemiEchelonMatDestructive(` *mat* `)` O

This does the same as `SemiEchelonMat(` *mat* `)`, except that it may (and probably will) destroy the matrix *mat*.

```
gap> mm:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> SemiEchelonMatDestructive( mm );
rec( heads := [ 1, 2, 0 ], vectors := [ [ 1, 2, 3 ], [ 0, 1, 2 ] ] )
gap> mm;
[ [ 1, 2, 3 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]
```

3 ▶ `SemiEchelonMatTransformation(` *mat* `)` A

does the same as `SemiEchelonMat` but additionally stores the linear transformation T performed on the matrix. The additional components of the result are

`coeffs`
: a list of coefficients vectors of the `vectors` component, with respect to the rows of *mat*, that is, `coeffs * mat` is the `vectors` component.

`relations`
: a list of basis vectors for the (left) null space of *mat*.

```
gap> SemiEchelonMatTransformation([[1,2,3],[0,0,1]]);
rec( heads := [ 1, 0, 2 ], vectors := [ [ 1, 2, 3 ], [ 0, 0, 1 ] ],
  coeffs := [ [ 1, 0 ], [ 0, 1 ] ], relations := [  ] )
```

4 ▶ `SemiEchelonMats(` *mats* `)` O

A list of matrices over a field F is in semi-echelon form if the list of row vectors obtained on concatenating the rows of each matrix is a semi-echelonized matrix (see 24.9.1).

`SemiEchelonMats` returns a record that contains information about a semi-echelonized form of the list *mats* of matrices.

The components of this record are

`vectors`
: list of matrices, each with pivot element the identity of F,

`heads`
: matrix that contains at position $[i,j]$, if nonzero, the number of the matrix that has the pivot element in this position

5 ▶ `SemiEchelonMatsDestructive(` *mats* `)` O

Does the same as `SemiEchelonmats`, except that it may destroy its argument. Therefore the argument must be a list of matrices that re mutable.

24.10 Matrices as Basis of a Row Space

1 ▶ `BaseMat( mat )` A

returns a basis for the row space generated by the rows of *mat* in the form of an immutable matrix.

2 ▶ `BaseMatDestructive( mat )` O

Does the same as `BaseMat`, with the difference that it may destroy the matrix *mat*. The matrix *mat* must be mutable.

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> BaseMat(mat);
[ [ 1, 2, 3 ], [ 0, 1, 2 ] ]
gap> mm:= [[1,2,3],[4,5,6],[5,7,9]];;
gap> BaseMatDestructive( mm );
[ [ 1, 2, 3 ], [ 0, 1, 2 ] ]
gap> mm;
[ [ 1, 2, 3 ], [ 0, 1, 2 ], [ 0, 0, 0 ] ]
```

3 ▶ `BaseOrthogonalSpaceMat( mat )` A

Let V be the row space generated by the rows of *mat* (over any field that contains all entries of *mat*). `BaseOrthogonalSpaceMat( mat )` computes a base of the orthogonal space of V.

The rows of *mat* need not be linearly independent.

4 ▶ `SumIntersectionMat( M1, M2 )` O

performs Zassenhaus' algorithm to compute bases for the sum and the intersection of spaces generated by the rows of the matrices *M1*, *M2*.

returns a list of length 2, at first position a base of the sum, at second position a base of the intersection. Both bases are in semi-echelon form (see 24.9).

```
gap> SumIntersectionMat(mat,[[2,7,6],[5,9,4]]);
[ [ [ 1, 2, 3 ], [ 0, 1, 2 ], [ 0, 0, 1 ] ], [ [ 1, -3/4, -5/2 ] ] ]
```

5 ▶ `BaseSteinitzVectors( bas, mat )` F

find vectors extending mat to a basis spanning the span of *bas*. Both *bas* and *mat* must be matrices of full (row) rank. It returns a record with the following components:

`subspace`
: is a basis of the space spanned by *mat* in upper triangular form with leading ones at all echelon steps and zeroes above these ones.

`factorspace`
: is a list of extending vectors in upper triangular form.

`factorzero`
: is a zero vector.

`heads`
: is a list of integers which can be used to decompose vectors in the basis vectors. The ith entry indicating the vector that gives an echelon step at position i. A negative number indicates an echelon step in the subspace, a positive number an echelon step in the complement, the absolute value gives the position of the vector in the lists `subspace` and `factorspace`.

```
gap> BaseSteinitzVectors(IdentityMat(3,1),[[11,13,15]]);
rec( factorspace := [ [ 0, 1, 15/13 ], [ 0, 0, 1 ] ],
  factorzero := [ 0, 0, 0 ], subspace := [ [ 1, 13/11, 15/11 ] ],
  heads := [ -1, 1, 2 ] )
```

See also chapter 25

24.11 Triangular Matrices

1 ▶ `DiagonalOfMat( mat )` O

returns the diagonal of *mat* as a list.

```
gap> DiagonalOfMat([[1,2],[3,4]]);
[ 1, 4 ]
```

2 ▶ `UpperSubdiagonal( mat, pos )` O

returns a mutable list containing the entries of the *pos*th upper subdiagonal of *mat*.

```
gap> UpperSubdiagonal(mat,1);
[ 2, 6 ]
```

3 ▶ `DepthOfUpperTriangularMatrix( mat )` A

If *mat* is an upper triangular matrix this attribute returns the index of the first nonzero diagonal.

```
gap> DepthOfUpperTriangularMatrix([[0,1,2],[0,0,1],[0,0,0]]);
1
gap> DepthOfUpperTriangularMatrix([[0,0,2],[0,0,0],[0,0,0]]);
2
```

24.12 Matrices as Linear Mappings

1 ▶ `CharacteristicPolynomial( mat )` A
▶ `CharacteristicPolynomial( [F, ]mat[, ind] )` O

For a square matrix *mat*, `CharacteristicPolynomial` returns the **characteristic polynomial** of *mat*, that is, the `StandardAssociate` of the determinant of the matrix $mat - X \cdot I$, where X is an indeterminate and I is the appropriate identity matrix.

If a field F is given as first argument then the characteristic polynomial of the F-linear mapping induced by *mat* is computed. If F contains the entries of *mat* then this is of course the same polynomial as the one computed by the one argument version; if F is a proper subfield of the default field (see 24.3.2) of *mat* then the characteristic polynomial is computed using `BlownUpMat` (see 24.12.3).

The returned polynomials are expressed in the indeterminate number *ind*. If *ind* is not given, it defaults to 1.

The characteristic polynomial is a multiple of the minimal polynomial (see 64.8.1).

```
gap> CharacteristicPolynomial( [ [ 1, 1 ], [ 0, 1 ] ] );
x^2-2*x+1
gap> mat := [[0,1],[E(4)-1,E(4)]];;
gap> CharacteristicPolynomial( mat );
x^2+-E(4)*x+1-E(4)
gap> CharacteristicPolynomial( Rationals, mat );
x^4+3*x^2+2*x+2
gap> mat:= [ [ E(4), 1 ], [ 0, -E(4) ] ];;
gap> CharacteristicPolynomial( mat );
x^2+1
gap> CharacteristicPolynomial( Rationals, mat );
x^4+2*x^2+1
```

2 ▶ `JordanDecomposition( mat )` A

`JordanDecomposition( mat )` returns a list `[S,N]` such that `S` is a semisimple matrix and `N` is nilpotent. Furthermore, `S` and `N` commute and *mat*`=S+N`.

```
gap> mat:=[[1,2,3],[4,5,6],[7,8,9]];;
gap> JordanDecomposition(mat);
[ [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ],
  [ [ 0, 0, 0 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ] ]
```

3 ▶ `BlownUpMat( B, mat )` F

Let B be a basis of a field extension F/K, and *mat* a matrix whose entries are all in F. (This is not checked.) `BlownUpMat` returns a matrix over K that is obtained by replacing each entry of *mat* by its regular representation w.r.t. B.

More precisely, regard *mat* as the matrix of a linear transformation on the row space F^n w.r.t. the F-basis with vectors $(v_1, ldots, v_n)$, say, and suppose that the basis B consists of the vectors $(b_1, \ldots, b_m)$; then the returned matrix is the matrix of the linear transformation on the row space K^{mn} w.r.t. the K-basis whose vectors are $(b_1 v_1, \ldots b_m v_1, \ldots, b_m v_n)$.

Note that the linear transformations act on **row** vectors, i.e., each row of the matrix is a concatenation of vectors of B-coefficients.

4 ▶ `BlownUpVector( B, vector )` F

Let B be a basis of a field extension F/K, and *vector* a row vector whose entries are all in F. `BlownUpVector` returns a row vector over K that is obtained by replacing each entry of *vector* by its coefficients w.r.t. B.

So `BlownUpVector` and `BlownUpMat` (see 24.12.3) are compatible in the sense that for a matrix *mat* over F, `BlownUpVector( B, mat * vector )` is equal to `BlownUpMat( B, mat ) * BlownUpVector( B, vector )`.

```
gap> B:= Basis( CF(4), [ 1, E(4) ] );;
gap> mat:= [ [ 1, E(4) ], [ 0, 1 ] ];;  vec:= [ 1, E(4) ];;
gap> bmat:= BlownUpMat( B, mat );;  bvec:= BlownUpVector( B, vec );;
gap> Display( bmat );  bvec;
[ [   1,   0,   0,   1 ],
  [   0,   1,  -1,   0 ],
  [   0,   0,   1,   0 ],
  [   0,   0,   0,   1 ] ]
[ 1, 0, 0, 1 ]
gap> bvec * bmat = BlownUpVector( B, vec * mat );
true
```

5 ▶ `CompanionMat( poly )` F

computes a companion matrix of the polynomial *poly*. This matrix has *poly* as its minimal polynomial.

24.13 Matrices over Finite Fields

Just as for row vectors, (see section 23.2), GAP has a special representation for matrices over small finite fields.

To be eligible to be represented in this way, each row of a matrix must be able to be represented as a compact row vector of the same length over **the same** finite field.

```
gap> v := Z(2)*[1,0,0,1,1];
[ Z(2)^0, 0*Z(2), 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> ConvertToVectorRep(v,2);
2
gap> v;
<a GF2 vector of length 5>
gap> m := [v];; ConvertToMatrixRep(m,GF(2));; m;
<a 1x5 matrix over GF2>
gap> m := [v,v];; ConvertToMatrixRep(m,GF(2));; m;
<a 2x5 matrix over GF2>
gap> m := [v,v,v];; ConvertToMatrixRep(m,GF(2));; m;
<a 3x5 matrix over GF2>
gap> v := Z(3)*[1..8];
[ Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ]
gap> ConvertToVectorRep(v);
3
gap> m := [v];; ConvertToMatrixRep(m,GF(3));; m;
[ [ Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0, 0*Z(3), Z(3), Z(3)^0 ] ]
gap> RepresentationsOfObject(m);
[ "IsPositionalObjectRep", "Is8BitMatrixRep" ]
gap> m := [v,v,v,v];; ConvertToMatrixRep(m,GF(3));; m;
< mutable compressed matrix 4x8 over GF(3) >
```

All compressed matrices over GF(2) are viewed as `<a` *nxm* `matrix over GF2>`, while over fields GF(q) for q between 3 and 256, matrices with 25 or more entries are viewed in this way, and smaller ones as lists of lists.

Matrices can be converted to this special representation via the following functions.

1 ▶ `ImmutableMatrix(` *field*`,` *matrix*`, [`*change*`] )` F

returns an immutable matrix equal to *matrix* which is in the most compact representation possible over *field*. The input matrix *matrix* or its rows might change the representation, however the result of `ConvertedMatrix` is not necessarily **identical** to *matrix* if a conversion is not possible. If *change* is `true`, the rows of `matrix` (or `matrix` itself) may be changed to become immutable (otherwise they are copied first).

2 ▶ `ConvertToMatrixRep(` *list* `)` F
▶ `ConvertToMatrixRep(` *list*`,` *field* `)` F
▶ `ConvertToMatrixRep(` *list*`,` *fieldsize* `)` F
▶ `ConvertToMatrixRepNC(` *list* `)` F
▶ `ConvertToMatrixRepNC(` *list*`,` *field* `)` F
▶ `ConvertToMatrixRepNC(` *list*`,` *fieldsize* `)` F

`ConvertToMatrixRep(` *list* `)` converts *list* to an internal matrix representation if possible. `ConvertToMatrixRep(` *list* `,` *field* `)` converts *list* to an internal matrix representation appropriate for a matrix over *field*.

It is forbidden to call this function unless all elements of *list* are vectors with entries in *field*. Violation of this condition can lead to unpredictable behaviour or a system crash. (Setting the assertion level to at least 2 might catch some violations before a crash, see 7.5.1.)

Instead of a *field* also its size *fieldsize* may be given.

list may already be a compressed matrix. In this case, if no *field* or *fieldsize* is given, then nothing happens.

list itself may be mutable, but its entries must be immutable.

The return value is the size of the field over which the matrix ends up written, if it is written in a compressed representation.

In general, it is better to call `ImmutableMatrix` (see 24.13.1) instead since this function can also deal with mutable rows or rows locked in a wrong representation.

Note that the main advantage of this special representation of matrices is in low dimensions, where various overheads can be reduced. In higher dimensions, a list of compressed vectors will be almost as fast. Note also that list access and assignment will be somewhat slower for compressed matrices than for plain lists.

In order to form a row of a compressed matrix a vector must accept certain restrictions. Specifically, it cannot change its length or change the field over which it is compressed. The main consequences of this are: that only elements of the appropriate field can be assigned to entries of the vector, and only to positions between 1 and the original length; that the vector cannot be shared between two matrices compressed over different fields.

This is enforced by the filter `IsLockedRepresentationVector`. When a vector becomes part of a compressed matrix, this filter is set for it. Assignment, `Unbind`, `ConvertToVectorRep` and `ConvertToMatrixRep` are all prevented from altering a vector with this filter.

```
gap> v := [Z(2),Z(2)];; ConvertToVectorRep(v,GF(2));; v;
<a GF2 vector of length 2>
gap> m := [v,v];
[ <a GF2 vector of length 2>, <a GF2 vector of length 2> ]
gap> ConvertToMatrixRep(m,GF(2));
2
gap> m2 := [m[1], [Z(4),Z(4)]]; # now try and mix in some GF(4)
[ <a GF2 vector of length 2>, [ Z(2^2), Z(2^2) ] ]
gap> ConvertToMatrixRep(m2); # but m2[1] is locked
#I  ConvertToVectorRep: locked vector not converted to different field
fail
gap> m2 := [ShallowCopy(m[1]), [Z(4),Z(4)]]; # a fresh copy of row 1
[ <a GF2 vector of length 2>, [ Z(2^2), Z(2^2) ] ]
gap> ConvertToMatrixRep(m2); # now it works
4
gap> m2;
[ [ Z(2)^0, Z(2)^0 ], [ Z(2^2), Z(2^2) ] ]
gap> RepresentationsOfObject(m2);
[ "IsPositionalObjectRep", "Is8BitMatrixRep" ]
```

Arithmetic operations (see 21.11 and the following sections) preserve the compression status of matrices in the sense that if all arguments are compressed matrices written over the same field and the result is a matrix then also the result is a compressed matrix written over this field.

There are also two operations that are only available for matrices written over finite fields.

3 ▶ `ProjectiveOrder(` *mat* `)` A

Returns an integer n and a finite field element e such that A^n = eI. *mat* must be a matrix defined over a finite field.

```
gap> ProjectiveOrder([[1,4],[5,2]]*Z(11)^0);
[ 5, Z(11)^5 ]
```

4 ▶ `SimultaneousEigenvalues(` *matlist*, *expo* `)` F

The matrices in *matlist* must be matrices over GF(q) for some prime q. Together, they must generate an abelian p-group of exponent *expo*. Then the eigenvalues of *mat* in the splitting field `GF(`q`^`r`)` for some r are

powers of an element ξ in the splitting field, which is of order *expo*. `SimultaneousEigenvalues` returns a matrix of integers mod *expo*, say $(a_{i,j})$, such that the power $\xi^{a_{i,j}}$ is an eigenvalue of the i-th matrix in *matlist* and the eigenspaces of the different matrices to the eigenvalues $\xi^{a_{i,j}}$ for fixed j are equal.

Finally, there are two operations that deal with matrices over a ring, but only care about the residues of their entries modulo some ring element. In the case of the integers and a prime number p, say, this is effectively computation in a matrix over the prime field in characteristic p.

5 ► `InverseMatMod( mat, obj )` O

For a square matrix *mat*, `InverseMatMod` returns a matrix *inv* such that *inv* `*` *mat* is congruent to the identity matrix modulo *obj*, if such a matrix exists, and `fail` otherwise.

```
gap> mat:= [ [ 1, 2 ], [ 3, 4 ] ];;  inv:= InverseMatMod( mat, 5 );
[ [ 3, 1 ], [ 4, 2 ] ]
gap> mat * inv;
[ [ 11, 5 ], [ 25, 11 ] ]
```

6 ► `NullspaceModQ( E, q )` F

E must be a matrix of integers and q a prime power. Then `NullspaceModQ` returns the set of all vectors of integers modulo q, which solve the homogeneous equation system given by E modulo q.

```
gap> mat:= [ [ 1, 3 ], [ 1, 2 ], [ 1, 1 ] ];;  NullspaceModQ( mat, 5 );
[ [ 0, 0, 0 ], [ 1, 3, 1 ], [ 2, 1, 2 ], [ 4, 2, 4 ], [ 3, 4, 3 ] ]
```

24.14 Special Multiplication Algorithms for Matrices over GF(2)

When multiplying two compressed matrices M and N over GF(2) of dimensions $a \times b$ and $b \times c$, say, where a, b and c are all greater than or equal to 128, GAP by default uses a more sophisticated matrix multiplication algorithm, in which linear combinations of groups of 8 rows of M are remembered and re-used in constructing various rows of the product. This is called level 8 grease. To optimise memory access patterns, these combinations are stored for $(b + 255)/256$ sets of 8 rows at once. This number is called the blocking level.

These levels of grease and blocking are found experimentally to give good performance across a range of processors and matrix sizes, but other levels may do even better in some cases. You can control the levels exactly using the functions below:

1 ► `PROD_GF2MAT_GF2MAT_SIMPLE( m1, m2 )` F

This function performs the standard unblocked and ungreased matrix multiplication for matrices of any size.

2 ► `PROD_GF2MAT_GF2MAT_ADVANCED( m1, m2, g, b )` F

This function computes the product of *m1* and *m2*, which must be compressed matrices over GF(2) of compatible dimensions, using level g grease and level b blocking.

We plan to include greased blocked matrix multiplication for other finite fields, and greased blocked algorithms for inversion and other matrix operations in a future release.

24.15 Block Matrices

Block matrices are a special representation of matrices which can save a lot of memory if large matrices have a block structure with lots of zero blocks. GAP uses the representation `IsBlockMatrixRep` to store block matrices.

1 ▶ `AsBlockMatrix(` *m*, *nrb*, *ncb* `)` F

returns a block matrix with *nrb* row blocks and *ncb* column blocks which is equal to the ordinary matrix *m*.

2 ▶ `BlockMatrix(` *blocks*, *nrb*, *ncb* `)` F
▶ `BlockMatrix(` *blocks*, *nrb*, *ncb*, *rpb*, *cpb*, *zero* `)` F

`BlockMatrix` returns an immutable matrix in the sparse representation `IsBlockMatrixRep`. The nonzero blocks are described by the list *blocks* of triples, the matrix has *nrb* row blocks and *ncb* column blocks.

If *blocks* is empty (i.e., if the matrix is a zero matrix) then the dimensions of the blocks must be entered as *rpb* and *cpb*, and the zero element as *zero*.

Note that all blocks must be ordinary matrices (see 24.1.2), and also the block matrix is an ordinary matrix.

```
gap> M := BlockMatrix([[1,1,[[1, 2],[ 3, 4]]],
>                      [1,2,[[9,10],[11,12]]],
>                      [2,2,[[5, 6],[ 7, 8]]]],2,2);
<block matrix of dimensions (2*2)x(2*2)>
gap> Display(M);
[ [   1,   2,   9,  10 ],
  [   3,   4,  11,  12 ],
  [   0,   0,   5,   6 ],
  [   0,   0,   7,   8 ] ]
```

3 ▶ `MatrixByBlockMatrix(` *blockmat* `)` A

returns a plain ordinary matrix that is equal to the block matrix *blockmat*.

25 Integral matrices and lattices

25.1 Linear equations over the integers and Integral Matrices

1 ▶ `NullspaceIntMat( mat )` A

If *mat* is a matrix with integral entries, this function returns a list of vectors that forms a basis of the integral nullspace of *mat*, i.e. of those vectors in the nullspace of *mat* that have integral entries.

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> NullspaceMat(mat);
[ [ -7/4, 9/2, -15/4, 1, 0 ], [ -3/4, -3/2, 1/4, 0, 1 ] ]
gap> NullspaceIntMat(mat);
[ [ 1, 18, -9, 2, -6 ], [ 0, 24, -13, 3, -7 ] ]
```

2 ▶ `SolutionIntMat( mat, vec )` O

If *mat* is a matrix with integral entries and *vec* a vector with integral entries, this function returns a vector x with integer entries that is a solution of the equation x `*` *mat* `=` *vec*. It returns `fail` if no such vector exists.

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> SolutionMat(mat,[95,115,182]);
[ 47/4, -17/2, 67/4, 0, 0 ]
gap> SolutionIntMat(mat,[95,115,182]);
[ 2285, -5854, 4888, -1299, 0 ]
```

3 ▶ `SolutionNullspaceIntMat( mat, vec )` O

This function returns a list of length two, its first entry being the result of a call to `SolutionIntMat` with same arguments, the second the result of `NullspaceIntMat` applied to the matrix *mat*. The calculation is performed faster than if two separate calls would be used.

```
gap> mat:=[[1,2,7],[4,5,6],[7,8,9],[10,11,19],[5,7,12]];;
gap> SolutionNullspaceIntMat(mat,[95,115,182]);
[ [ 2285, -5854, 4888, -1299, 0 ],
  [ [ 1, 18, -9, 2, -6 ], [ 0, 24, -13, 3, -7 ] ] ]
```

4 ▶ `BaseIntMat( mat )` A

If *mat* is a matrix with integral entries, this function returns a list of vectors that forms a basis of the integral row space of *mat*, i.e. of the set of integral linear combinations of the rows of *mat*.

```
gap> mat:=[[1,2,7],[4,5,6],[10,11,19]];;
gap> BaseIntMat(mat);
[ [ 1, 2, 7 ], [ 0, 3, 7 ], [ 0, 0, 15 ] ]
```

5 ▶ `BaseIntersectionIntMats( m, n )` A

If m and n are matrices with integral entries, this function returns a list of vectors that forms a basis of the intersection of the integral row spaces of m and n.

```
gap> nat:=[[5,7,2],[4,2,5],[7,1,4]];;
gap> BaseIntMat(nat);
[ [ 1, 1, 15 ], [ 0, 2, 55 ], [ 0, 0, 64 ] ]
gap> BaseIntersectionIntMats(mat,nat);
[ [ 1, 5, 509 ], [ 0, 6, 869 ], [ 0, 0, 960 ] ]
```

6 ► `ComplementIntMat(` *full*, *sub* `)` A

Let *full* be a list of integer vectors generating an Integral module M and *sub* a list of vectors defining a submodule S. This function computes a free basis for M that extends S. I.e., if the dimension of S is n it determines a basis $B = \{\underline{b}_1, \ldots, \underline{b}_m\}$ for M, as well as n integers x_i such that the n vectors $\underline{s}_i := x_i \cdot \underline{b}_i\}$ form a basis for S.

It returns a record with the following components:

`complement`
: the vectors $\underline{b}_{n+1}$ up to $\underline{b}_m$ (they generate a complement to S).

`sub`
: the vectors s_i (a basis for S).

`moduli`
: the factors x_i.

```
gap> m:=IdentityMat(3);;
gap> n:=[[1,2,3],[4,5,6]];;
gap> ComplementIntMat(m,n);
rec( complement := [ [ 0, 0, 1 ] ], sub := [ [ 1, 2, 3 ], [ 0, 3, 6 ] ],
  moduli := [ 1, 3 ] )
```

25.2 Normal Forms over the Integers

1 ► `TriangulizedIntegerMat(` *mat* `)` O

Computes an upper triangular form of a matrix with integer entries. It returns a immutable matrix in upper triangular form.

2 ► `TriangulizedIntegerMatTransform(` *mat* `)` O

Computes an upper triangular form of a matrix with integer entries. It returns a record with a component `normal` (an immutable matrix in upper triangular form) and a component `rowtrans` that gives the transformations done to the original matrix to bring it into upper triangular form.

3 ► `TriangulizeIntegerMat(` *mat* `)` O

Changes *mat* to be in upper triangular form. (The result is the same as that of `TriangulizedIntegerMat`, but *mat* will be modified, thus using less memory.) If *mat* is immutable an error will be triggered.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> TriangulizedIntegerMat(m);
[ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
gap> n:=TriangulizedIntegerMatTransform(m);
rec( normal := [ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ 1, 0, 0 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ], rank := 3,
  signdet := 1, rowtrans := [ [ 1, 0, 0 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ] )
gap> n.rowtrans*m=n.normal;
true
```

```
gap> TriangulizeIntegerMat(m); m;
[ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
```

The Hermite Normal Form (HNF), H of an integer matrix, A is a row equivalent upper triangular form such that all off-diagonal entries are reduced modulo the diagonal entry of the column they are in. There exists a unique unimodular matrix Q such that $QA = H$.

4 ▶ `HermiteNormalFormIntegerMat(` *mat* `)` O

This operation computes the Hermite normal form of a matrix *mat* with integer entries. It returns a immutable matrix in HNF.

5 ▶ `HermiteNormalFormIntegerMatTransform(` *mat* `)` O

This operation computes the Hermite normal form of a matrix *mat* with integer entries. It returns a record with components `normal` (a matrix H) and `rowtrans` (a matrix Q) such that $QA = H$

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> HermiteNormalFormIntegerMat(m);
[ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ]
gap> n:=HermiteNormalFormIntegerMatTransform(m);
rec( normal := [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ], rank := 3,
  signdet := 1,
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ] )
gap> n.rowtrans*m=n.normal;
true
```

The Smith Normal Form,S, of an integer matrix A is the unique equivalent diagonal form with S_i dividing S_j for $i < j$. There exist unimodular integer matrices P, Q such that $PAQ = S\cdot$

6 ▶ `SmithNormalFormIntegerMat(` *mat* `)` O

This operation computes the Smith normal form of a matrix *mat* with integer entries. It returns a new immutable matrix in the Smith normal form.

7 ▶ `SmithNormalFormIntegerMatTransforms(` *mat* `)` O

This operation computes the Smith normal form of a matrix *mat* with integer entries. It returns a record with components `normal` (a matrix S), `rowtrans` (a matrix P), and `coltrans` (a matrix Q) such that $PAQ = S$.

8 ▶ `DiagonalizeIntMat(` *mat* `)` O

This function changes *mat* to its SNF. (The result is the same as that of `SmithNormalFormIntegerMat`, but *mat* will be modified, thus using less memory.) If *mat* is immutable an error will be triggered.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> SmithNormalFormIntegerMat(m);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
gap> n:=SmithNormalFormIntegerMatTransforms(m);
rec( normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  colQ := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ], rank := 3,
  signdet := 1,
```

```
    rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
    coltrans := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ] )
gap> n.rowtrans*m*n.coltrans=n.normal;
true
gap> DiagonalizeIntMat(m);m;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
```

All these routines build on the following "workhorse" routine:

9 ► `NormalFormIntMat(` *mat*, *options* `)` O

This general operation for computation of various Normal Forms is probably the most efficient.

Options bit values:

0/1 Triangular Form / Smith Normal Form.

2 Reduce off diagonal entries.

4 Row Transformations.

8 Col Transformations.

16 Destructive (the original matrix may be destroyed)

Compute a Triangular, Hermite or Smith form of the $n \times m$ integer input matrix A. Optionally, compute $n \times n$ and $m \times m$ unimodular transforming matrices Q, P which satisfy $QA = H$ or $QAP = S$.

Note option is a value ranging from 0 - 15 but not all options make sense (eg reducing off diagonal entries with SNF option selected already). If an option makes no sense it is ignored.

Returns a record with component `normal` containing the computed normal form and optional components `rowtrans` and/or `coltrans` which hold the respective transformation matrix. Also in the record are components holding the sign of the determinant, signdet, and the Rank of the matrix, rank.

```
gap> m:=[[1,15,28],[4,5,6],[7,8,9]];;
gap> NormalFormIntMat(m,0);  # Triangular, no transforms
rec( normal := [ [ 1, 15, 28 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ], rank := 3,
  signdet := 1 )
gap> NormalFormIntMat(m,6);  # Hermite Normal Form with row transforms
rec( normal := [ [ 1, 0, 1 ], [ 0, 1, 1 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ], rank := 3,
  signdet := 1,
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ] )
gap> NormalFormIntMat(m,13); # Smith Normal Form with both transforms
rec( normal := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ],
  rowC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  rowQ := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  colC := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  colQ := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ], rank := 3,
  signdet := 1,
  rowtrans := [ [ -2, 62, -35 ], [ 1, -30, 17 ], [ -3, 97, -55 ] ],
  coltrans := [ [ 1, 0, -1 ], [ 0, 1, -1 ], [ 0, 0, 1 ] ] )
gap> last.rowtrans*m*last.coltrans;
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 3 ] ]
```

10 ► `AbelianInvariantsOfList(` *list* `)` A

Given a list of positive integers, this routine returns a list of prime powers, such that the prime power factors of the entries in the list are returned in sorted form.

```
gap> AbelianInvariantsOfList([4,6,2,12]);
[ 2, 2, 3, 3, 4, 4 ]
```

25.3 Determinant of an integer matrix

1 ▶ `DeterminantIntMat( mat )` O

Computes the determinant of an integer matrix using the same strategy as `NormalFormIntMat` (see 25.2.9). This method is faster in general for matrices greater than 20×20 but quite a lot slower for smaller matrices. It therefore passes the work to the more general `DeterminantMat` (see 24.3.4) for these smaller matrices.

25.4 Decompositions

For computing the decomposition of a vector of integers into the rows of a matrix of integers, with integral coefficients, one can use p-adic approximations, as follows.

Let A be a square integral matrix, and p an odd prime. The reduction of A modulo p is $\overline{A}$, its entries are chosen in the interval $[-\frac{p-1}{2}, \frac{p-1}{2}]$. If $\overline{A}$ is regular over the field with p elements, we can form $A' = \overline{A}^{-1}$. Now we consider the integral linear equation system $xA = b$, i.e., we look for an integral solution x. Define $b_0 = b$, and then iteratively compute

$$x_i = (b_i A') \bmod p, \quad b_{i+1} = \frac{1}{p}(b_i - x_i A), i = 0, 1, 2, \ldots .$$

By induction, we get

$$p^{i+1} b_{i+1} + \left(\sum_{j=0}^{i} p^j x_j \right) A = b.$$

If there is an integral solution x then it is unique, and there is an index l such that b_{l+1} is zero and $x = \sum_{j=0}^{l} p^j x_j$.

There are two useful generalizations of this idea. First, A need not be square; it is only necessary that there is a square regular matrix formed by a subset of columns of A. Second, A does not need to be integral; the entries may be cyclotomic integers as well, in this case one can replace each column of A by the columns formed by the coefficients w.r.t. an integral basis (which are integers). Note that this preprocessing must be performed compatibly for A and b.

GAP provides the following functions for this purpose (see also 24.13.5).

1 ▶ `Decomposition( A, B, depth )` F
▶ `Decomposition( A, B, "nonnegative" )` F

For a $m \times n$ matrix A of cyclotomics that has rank $m \leq n$, and a list B of cyclotomic vectors, each of length n, `Decomposition` tries to find integral solutions of the linear equation systems x `*` A `=` B`[i]`, by computing the p-adic series of hypothetical solutions.

`Decomposition( A, B, depth )`, where *depth* is a nonnegative integer, computes for each vector B`[i]` the initial part $\sum_{k=0}^{depth} x_k p^k$, with all x_k vectors of integers with entries bounded by $\pm\frac{p-1}{2}$. The prime p is 83 first; if the reduction of A modulo p is singular, the next prime is chosen automatically.

A list X is returned. If the computed initial part for x `*` A `=` B`[i]` **is** a solution, we have X`[i] =` x, otherwise X`[i] = fail`.

`Decomposition( A, B, "nonnegative" )` assumes that the solutions have only nonnegative entries, and that the first column of A consists of positive integers. This is satisfied, e.g., for the decomposition of ordinary characters into Brauer characters. In this case the necessary number *depth* of iterations can be computed;

the `i`-th entry of the returned list is `fail` if there **exists** no nonnegative integral solution of the system x `*` A `=` B`[i]`, and it is the solution otherwise.

Note that the result is a list of `fail` if A has not full rank, even if there might be a unique integral solution for some equation system.

2 ▶ `LinearIndependentColumns(` *mat* `)` F

Called with a matrix *mat*, `LinearIndependentColumns` returns a maximal list of column positions such that the restriction of *mat* to these columns has the same rank as *mat*.

3 ▶ `PadicCoefficients(` *A*`,` *Amodpinv*`,` *b*`,` *prime*`,` *depth* `)` F

Let A be an integral matrix, *prime* a prime integer, *Amodpinv* an inverse of A modulo *prime*, b an integral vector, and *depth* a nonnegative integer. `PadicCoefficients` returns the list $[x_0, x_1, \ldots, x_l, b_{l+1}]$ describing the *prime*-adic approximation of b (see above), where $l =$ *depth* or l is minimal with the property that $b_{l+1} = 0$.

4 ▶ `IntegralizedMat(` *A* `)` F
▶ `IntegralizedMat(` *A*`,` *inforec* `)` F

`IntegralizedMat` returns for a matrix A of cyclotomics a record *intmat* with components `mat` and `inforec`. Each family of algebraic conjugate columns of A is encoded in a set of columns of the rational matrix *intmat*`.mat` by replacing cyclotomics in A by their coefficients w.r.t. an integral basis. *intmat*`.inforec` is a record containing the information how to encode the columns.

If the only argument is A, the value of the component `inforec` is computed that can be entered as second argument *inforec* in a later call of `IntegralizedMat` with a matrix B that shall be encoded compatibly with A.

5 ▶ `DecompositionInt(` *A*`,` *B*`,` *depth* `)` F

`DecompositionInt` does the same as `Decomposition` (see 25.4.1), except that A and B must be integral matrices, and *depth* must be a nonnegative integer.

25.5 Lattice Reduction

1 ▶ `LLLReducedBasis( [`*L*`, ]`*vectors*`[,` *y*`][, "linearcomb"][,` *lllout*`] )` F

provides an implementation of the LLL algorithm by Lenstra, Lenstra and Lovász (see [LLL82], [Poh87]). The implementation follows the description on pages 94f. in [Coh93].

`LLLReducedBasis` returns a record whose component `basis` is a list of LLL reduced linearly independent vectors spanning the same lattice as the list *vectors*. L must be a lattice, with scalar product of the vectors v and w given by `ScalarProduct(` L`,` v`,` w `)`. If no lattice is specified then the scalar product of vectors given by `ScalarProduct(` v`,` w `)` is used.

In the case of the option `"linearcomb"`, the result record contains also the components `relations` and `transformation`, with the following meaning. `relations` is a basis of the relation space of *vectors*, i.e., of vectors x such that x `*` *vectors* is zero. `transformation` gives the expression of the new lattice basis in terms of the old, i.e., `transformation *` *vectors* equals the `basis` component of the result.

Another optional argument is y, the "sensitivity" of the algorithm, a rational number between $\frac{1}{4}$ and 1 (the default value is $\frac{3}{4}$).

The optional argument *lllout* is a record with the components `mue` and `B`, both lists of length k, with the meaning that if *lllout* is present then the first k vectors in *vectors* form an LLL reduced basis of the lattice they generate, and *lllout*`.mue` and *lllout*`.B` contain their scalar products and norms used internally in the algorithm, which are also present in the output of `LLLReducedBasis`. So *lllout* can be used for "incremental" calls of `LLLReducedBasis`.

The function `LLLReducedGramMat` (see 25.5.2) computes an LLL reduced Gram matrix.

```
gap> vectors:= [ [ 9, 1, 0, -1, -1 ], [ 15, -1, 0, 0, 0 ],
>                [ 16, 0, 1, 1, 1 ], [ 20, 0, -1, 0, 0 ],
>                [ 25, 1, 1, 0, 0 ] ];;
gap> LLLReducedBasis( vectors, "linearcomb" );; Display( last );
rec(
  basis := [ [ 1, 1, 1, 1, 1 ], [ 1, 1, -2, 1, 1 ], [ -1, 3, -1, -1, -1 ],
      [ -3, 1, 0, 2, 2 ] ],
  relations := [ [ -1, 0, -1, 0, 1 ] ],
  transformation :=
   [ [ 0, -1, 1, 0, 0 ], [ -1, -2, 0, 2, 0 ], [ 1, -2, 0, 1, 0 ],
      [ -1, -2, 1, 1, 0 ] ],
  mue := [ [  ], [ 2/5 ], [ -1/5, 1/3 ], [ 2/5, 1/6, 1/6 ] ],
  B := [ 5, 36/5, 12, 50/3 ] )
```

2 ▶ `LLLReducedGramMat( `G` )` F

▶ `LLLReducedGramMat( `G`, `y` )` F

`LLLReducedGramMat` provides an implementation of the LLL algorithm by Lenstra, Lenstra and Lovász (see [LLL82], [Poh87]). The implementation follows the description on pages 94f. in [Coh93].

Let G the Gram matrix of the vectors $(b_1, b_2, \ldots, b_n)$; this means G is either a square symmetric matrix or lower triangular matrix (only the entries in the lower triangular half are used by the program).

`LLLReducedGramMat` returns a record whose component `remainder` is the Gram matrix of the LLL reduced basis corresponding to $(b_1, b_2, \ldots, b_n)$. If G is a lower triangular matrix then also the `remainder` component of the result record is a lower triangular matrix.

The result record contains also the components `relations` and `transformation`, which have the following meaning.

`relations` is a basis of the space of vectors $(x_1, x_2, \ldots, x_n)$ such that $\sum_{i=1}^{n} x_i b_i$ is zero, and `transformation` gives the expression of the new lattice basis in terms of the old, i.e., `transformation` is the matrix T such that $T \cdot G \cdot T^{tr}$ is the `remainder` component of the result.

The optional argument y denotes the "sensitivity" of the algorithm, it must be a rational number between $\frac{1}{4}$ and 1; the default value is $y = \frac{3}{4}$.

The function `LLLReducedBasis` (see 25.5.1) computes an LLL reduced basis.

```
gap> g:= [ [ 4, 6, 5, 2, 2 ], [ 6, 13, 7, 4, 4 ],
>    [ 5, 7, 11, 2, 0 ], [ 2, 4, 2, 8, 4 ], [ 2, 4, 0, 4, 8 ] ];;
gap> LLLReducedGramMat( g );; Display( last );
rec(
  remainder := [ [ 4, 2, 1, 2, -1 ], [ 2, 5, 0, 2, 0 ], [ 1, 0, 5, 0, 2 ],
      [ 2, 2, 0, 8, 2 ], [ -1, 0, 2, 2, 7 ] ],
  relations := [  ],
  transformation :=
   [ [ 1, 0, 0, 0, 0 ], [ -1, 1, 0, 0, 0 ], [ -1, 0, 1, 0, 0 ],
      [ 0, 0, 0, 1, 0 ], [ -2, 0, 1, 0, 1 ] ],
  mue := [ [  ], [ 1/2 ], [ 1/4, -1/8 ], [ 1/2, 1/4, -2/25 ],
      [ -1/4, 1/8, 37/75, 8/21 ] ],
  B := [ 4, 4, 75/16, 168/25, 32/7 ] )
```

25.6 Orthogonal Embeddings

1 ▶ `OrthogonalEmbeddings(` *gram*`[, "positive"][,` *maxdim*`] )` F

computes all possible orthogonal embeddings of a lattice given by its Gram matrix *gram*, which must be a regular matrix. In other words, all solutions X of the problem

$$X^{tr} \cdot X = gram$$

are calculated (see [Ple90]). Usually there are many solutions X but all their rows are chosen from a small set of vectors, so `OrthogonalEmbeddings` returns the solutions in an encoded form, namely as a record with components

`vectors`
: the list $L = [x_1, x_2, \ldots, x_n]$ of vectors that may be rows of a solution; these are exactly those vectors that fulfill the condition $x_i \cdot gram^{-1} \cdot x_i^{tr} \leq 1$ (see 25.6.2), and we have $gram = \sum_{i=1}^{n} x_i^{tr} \cdot x_i$,

`norms`
: the list of values $x_i \cdot gram^{-1} \cdot x_i^{tr}$, and

`solutions`
: a list S of lists; the i-th solution matrix is L `S[i]` , so the dimension of the i-th solution is the length of S`[i]`.

The optional argument `"positive"` will cause `OrthogonalEmbeddings` to compute only vectors x_i with nonnegative entries. In the context of characters this is allowed (and useful) if *gram* is the matrix of scalar products of ordinary characters.

When `OrthogonalEmbeddings` is called with the optional argument *maxdim* (a positive integer), only solutions up to dimension *maxdim* are computed; this will accelerate the algorithm in some cases.

```
gap> b:= [ [ 3, -1, -1 ], [ -1, 3, -1 ], [ -1, -1, 3 ] ];;
gap> c:=OrthogonalEmbeddings( b );; Display( c );
rec(
  vectors := [ [ -1, 1, 1 ], [ 1, -1, 1 ], [ -1, -1, 1 ], [ -1, 1, 0 ],
      [ -1, 0, 1 ], [ 1, 0, 0 ], [ 0, -1, 1 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  norms := [ 1, 1, 1, 1/2, 1/2, 1/2, 1/2, 1/2, 1/2 ],
  solutions := [ [ 1, 2, 3 ], [ 1, 6, 6, 7, 7 ], [ 2, 5, 5, 8, 8 ],
      [ 3, 4, 4, 9, 9 ], [ 4, 5, 6, 7, 8, 9 ] ] )
gap> c.vectors{ c.solutions[1] };
[ [ -1, 1, 1 ], [ 1, -1, 1 ], [ -1, -1, 1 ] ]
```

gram may be the matrix of scalar products of some virtual characters. From the characters and the embedding given by the matrix X, `Decreased` (see 70.10.7) may be able to compute irreducibles, see 70.10.

2 ▶ `ShortestVectors(` G, m`[, "positive"] )` F

Let G be a regular matrix of a symmetric bilinear form, and m a nonnegative integer. `ShortestVectors` computes the vectors x that satisfy $x \cdot G \cdot x^{tr} \leq m$, and returns a record describing these vectors. The result record has the components

`vectors`
: list of the nonzero vectors x, but only one of each pair $(x, -x)$,

`norms`
: list of norms of the vectors according to the Gram matrix G.

If the optional argument `"positive"` is entered, only those vectors x with nonnegative entries are computed.

```
gap> g:= [ [ 2, 1, 1 ], [ 1, 2, 1 ], [ 1, 1, 2 ] ];;
gap> ShortestVectors(g,4);; Display( last );
rec(
  vectors := [ [ -1, 1, 1 ], [ 0, 0, 1 ], [ -1, 0, 1 ], [ 1, -1, 1 ],
      [ 0, -1, 1 ], [ -1, -1, 1 ], [ 0, 1, 0 ], [ -1, 1, 0 ], [ 1, 0, 0 ] ],
  norms := [ 4, 2, 2, 4, 2, 4, 2, 2, 2 ] )
```

26 Strings and Characters

1 ▶ `IsChar( obj )` C
▶ `IsCharCollection( obj )` C

A **character** is simply an object in GAP that represents an arbitrary character from the character set of the operating system. Character literals can be entered in GAP by enclosing the character in **singlequotes** '.

```
gap> x:= 'a';  IsChar( x );
'a'
true
gap> '*';
'*'
```

2 ▶ `IsString( obj )` C

A **string** is a dense list (see 21.1.1, 21.1.2) of characters (see 26); thus strings are always homogeneous (see 21.1.3).

A string literal can either be entered as the list of characters or by writing the characters between **doublequotes** ". GAP will always output strings in the latter format. However, the input via the double quote syntax enables GAP to store the string in an efficient compact internal representation. See 26.2.1 below for more details.

Each character, in particular those which cannot be typed directly from the keyboard, can also be typed in three digit octal notation. And for some special characters (like the newline character) there is a further possibility to type them, see section 26.1.

```
gap> s1 := ['H','e','l','l','o',' ','w','o','r','l','d','.'];
"Hello world."
gap> IsString( s1 );
true
gap> s2 := "Hello world.";
"Hello world."
gap> s1 = s2;
true
gap> s3 := "";  # the empty string
""
gap> s3 = [];
true
gap> IsString( [] );
true
gap> IsString( "123" );  IsString( 123 );
true
false
gap> IsString( [ '1', '2', '3' ] );
```

```
true
gap> IsString( [ '1', '2', , '4' ] );  # strings must be dense
false
gap> IsString( [ '1', '2', 3 ] );  # strings must only contain characters
false

gap> s := "\007";
"\007"
gap> Print(s); # rings bell in many terminals
```

Note that a string is just a special case of a list. So everything that is possible for lists (see 21) is also possible for strings. Thus you can access the characters in such a string (see 21.3), test for membership (see 28.5), ask for the length, concatenate strings (see 21.20.1), form substrings etc. You can even assign to a mutable string (see 21.4). Of course unless you assign a character in such a way that the list stays dense, the resulting list will no longer be a string.

```
gap> Length( s2 );
12
gap> s2[2];
'e'
gap> 'a' in s2;
false
gap> s2[2] := 'a';;  s2;
"Hallo world."
gap> s1{ [1..4] };
"Hell"
gap> Concatenation( s1{ [ 1 .. 6 ] }, s1{ [ 1 .. 4 ] } );
"Hello Hell"
```

If a string is displayed by `View`, for example as result of an evaluation (see 6.1), or by `ViewObj` and `PrintObj`, it is displayed with enclosing doublequotes. (But note that there is an ambiguity for the empty string which is also an empty list of arbitrary GAP objects; it is only printed like a string if it was input as empty string or converted to a string with 26.2.2.) The difference between `ViewObj` and `PrintObj` is that the latter prints **all** non-printable and non-ASCII characters in three digit octal notation, while `ViewObj` sends all printable characters to the output stream. The output of `PrintObj` can be read back into GAP.

Strings behave differently from other GAP objects with respect to `Print`, `PrintTo`, or `AppendTo`. These commands **interpret** a string in the sense that they essentially send the characters of the string directly to the output stream/file. (But depending on the type of the stream and the presence of some special characters used as hints for line breaks there may be sent some additional newline (or backslash and newline) characters.

```
gap> s4:= "abc\"def\nghi";;
gap> View( s4 );  Print( "\n" );
"abc\"def\nghi"
gap> ViewObj( s4 );  Print( "\n" );
"abc\"def\nghi"
gap> PrintObj( s4 );  Print( "\n" );
"abc\"def\nghi"
gap> Print( s4 );  Print( "\n" );
abc"def
ghi
gap> s := "German uses strange characters: \344\366\374\337\n";
"German uses strange characters: \n"
gap> Print(s);
```

```
German uses strange characters:
gap> PrintObj(s);
"German uses strange characters: \344\366\374\337\n"gap>
```

Note that only those line breaks are printed by `Print` that are contained in the string (`\n` characters, see 26.1), as is shown in the example below.

```
gap> s1;
"Hello world."
gap> Print( s1 );
Hello world.gap> Print( s1, "\n" );
Hello world.
gap> Print( s1, "\nnext line\n" );
Hello world.
next line
```

26.1 Special Characters

There are a number of **special character sequences** that can be used between the singlequotes of a character literal or between the doublequotes of a string literal to specify characters. They consist of two characters. The first is a backslash \. The second may be any character. If it is an octal digit (from `0` to `7`) there must be two more such digits. The meaning is given in the following list

`\n` **newline character**. This is the character that, at least on UNIX systems, separates lines in a text file. Printing of this character in a string has the effect of moving the cursor down one line and back to the beginning of the line.

`\"` **doublequote character**. Inside a string a doublequote must be escaped by the backslash, because it is otherwise interpreted as end of the string.

`\'` **singlequote character**. Inside a character a singlequote must escaped by the backslash, because it is otherwise interpreted as end of the character.

`\\` **backslash character**. Inside a string a backslash must be escaped by another backslash, because it is otherwise interpreted as first character of an escape sequence.

`\b` **backspace character**. Printing this character should have the effect of moving the cursor back one character. Whether it works or not is system dependent and should not be relied upon.

`\r` **carriage return character**. Printing this character should have the effect of moving the cursor back to the beginning of the same line. Whether this works or not is again system dependent.

`\c` **flush character**. This character is not printed. Its purpose is to flush the output queue. Usually GAP waits until it sees a *newline* before it prints a string. If you want to display a string that does not include this character use `\c`.

`\XYZ` with X, Y, Z three octal digits. This is translated to the character correponding to the number X**64+Y**8+Z modulo 256. This can be used to specify and store arbitrary binary data as a string in GAP.

other For any other character the backslash is simply ignored.

Again, if the line is displayed as result of an evaluation, those escape sequences are displayed in the same way that they are input.

Only `Print`, `PrintTo`, or `AppendTo` send the characters directly to the output stream.

```
gap> "This is one line.\nThis is another line.\n";
"This is one line.\nThis is another line.\n"
gap> Print( last );
This is one line.
This is another line.
```

Note in particular that it is not allowed to enclose a *newline* inside the string. You can use the special character sequence \n to write strings that include *newline* characters. If, however, an input string is too long to fit on a single line it is possible to **continue** it over several lines. In this case the last character of each input line, except the last line must be a backslash. Both backslash and *newline* are thrown away by GAP while reading the string. Note that the same continuation mechanism is available for identifiers and integers.

```
gap> "This is a very long string that does not fit on a line \
gap> and is therefore continued on the next line.";
"This is a very long string that does not fit on a line and is therefore conti\
nued on the next line."
```

Note that the output is also continued, but at a different place that is determined by the value of `SizeScreen` (see 6.11.1).

26.2 Internally Represented Strings

1 ▶ `IsStringRep( obj )` R

`IsStringRep` is a special (internal) representation of dense lists of characters. Dense lists of characters can be converted into this representation using `ConvertToStringRep`. Note that calling `IsString` does **not** change the representation.

2 ▶ `ConvertToStringRep( obj )` F

If *obj* is a dense internally represented list of characters then `ConvertToStringRep` changes the representation to `IsStringRep`. This is useful in particular for converting the empty list `[]`, which usually is in `IsPlistRep`, to `IsStringRep`. If *obj* is not a string then `ConvertToStringRep` signals an error.

3 ▶ `IsEmptyString( str )` F

`IsEmptyString` returns `true` if *str* is the empty string in the representation `IsStringRep`, and `false` otherwise. Note that the empty list `[]` and the empty string `""` have the same type, the recommended way to distinguish them is via `IsEmptyString`. For formatted printing, this distinction is sometimes necessary.

```
gap> l:= [];;  IsString( l );  IsEmptyString( l );  IsEmpty( l );
true
false
true
gap> l;  ConvertToStringRep( l );  l;
[ ]
""
gap> IsEmptyString( l );  IsEmptyString( "" );  IsEmptyString( "abc" );
true
true
false
gap> ll:= [ 'a', 'b' ];  IsStringRep( ll );  ConvertToStringRep( ll );
"ab"
false
```

```
gap> ll;  IsStringRep( ll );
"ab"
true
```

4 ▶ `CharsFamily` V

Each character lies in the family `CharFamily`, each nonempty string lies in the collections family of this family. Note the subtle differences between the empty list `[]` and the empty string `""` when both are printed.

26.3 Recognizing Characters

1 ▶ `IsDigitChar(` *c* `)` F

checks whether the character *c* is a digit, i.e., occurs in the string `"0123456789"`.

2 ▶ `IsLowerAlphaChar(` *c* `)` F

checks whether the character *c* is a lowercase alphabet letter, i.e., occurs in the string `"abcdefghijklmnopqrstuvwxyz"`.

3 ▶ `IsUpperAlphaChar(` *c* `)` F

checks whether the character *c* is an uppercase alphabet letter, i.e., occurs in the string `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`.

4 ▶ `IsAlphaChar(` *c* `)` F

checks whether the character *c* is either a lowercase or an uppercase alphabet letter.

26.4 Comparisons of Strings

1 ▶ *string1* `=` *string2*
▶ *string1* `<>` *string2*

The equality operator `=` returns to `true` if the two strings *string1* and *string2* are equal and `false` otherwise. The inequality operator `<>` returns `true` if the two strings *string1* and *string2* are not equal and `false` otherwise.

```
gap> "Hello world.\n" = "Hello world.\n";
true
gap> "Hello World.\n" = "Hello world.\n";  # string comparison is case sensitive
false
gap> "Hello world." = "Hello world.\n";  # the first string has no <newline>
false
gap> "Goodbye world.\n" = "Hello world.\n";
false
gap> [ 'a', 'b' ] = "ab";
true
```

2 ▶ *string1* `<` *string2*

The ordering of strings is lexicographically according to the order implied by the underlying, system dependent, character set.

```
gap> "Hello world.\n" < "Hello world.\n";  # the strings are equal
false
gap> "Hello World." < "Hello world.";  # in ASCII capitals range before small letters
true
gap> "Hello world." < "Hello world.\n";  # prefixes are always smaller
true
gap> "Goodbye world.\n" < "Hello world.\n";  # 'G' comes before 'H', in ASCII at least
true
```

Strings can be compared via < with certain GAP objects that are not strings, see 4.11 for the details.

26.5 Operations to Produce or Manipulate Strings

1 ▶ `String(` *obj* `)` A
▶ `String(` *obj*`,` *length* `)` O

`String` returns a representation of *obj*, which may be an object of arbitrary type, as a string. This string should approximate as closely as possible the character sequence you see if you print *obj*.

If *length* is given it must be an integer. The absolute value gives the minimal length of the result. If the string representation of *obj* takes less than that many characters it is filled with blanks. If *length* is positive it is filled on the left, if *length* is negative it is filled on the right.

In the two argument case, the string returned is a new mutable string (in particular not a part of any other object); it can be modified safely, and `MakeImmutable` may be safely applied to it.

```
gap> String(123);String([1,2,3]);
"123"
"[ 1, 2, 3 ]"
```

2 ▶ `HexStringInt(` *int* `)` F

returns a string which represents the integer *int* with hexa-decimal digits (using `A-F` as digits `10-15`). The inverse translation can be achieved with 26.7.1.

3 ▶ `StringPP(` *int* `)` F

returns a string representing the prime factor decomposition of the integer *int*.

```
gap> StringPP(40320);
"2^7*3^2*5*7"
```

4 ▶ `WordAlp(` *alpha*`,` *nr* `)` F

returns a string that is the *nr*-th word over the alphabet list *alpha*, w.r.t. word length and lexicographical order. The empty word is `WordAlp(` *alpha*`, 0 )`.

```
gap> List([0..5],i->WordAlp("abc",i));
[ "", "a", "b", "c", "aa", "ab" ]
```

5 ▶ `LowercaseString(` *string* `)` F

returns a lowercase version of the string *string*, that is, a string in which each uppercase alphabet character is replaced by the corresponding lowercase character.

```
gap> LowercaseString("This Is UpperCase");
"this is uppercase"
```

6 ▶ `SplitString(` *string*`,` *seps*`[,` *wspace*`] )` O

This function accepts a string *string* and lists *seps* and, optionally, *wspace* of characters. Now string is split into substrings at each occurrence of a character in *seps* or *wspace*. The characters in *wspace* are interpreted

as white space characters. Substrings of characters in *wspace* are treated as one white space character and they are ignored at the beginning and end of a string.

Both arguments *seps* and *wspace* can be single characters.

Each string in the resulting list of substring does not contain any characters in *seps* or *wspace*.

A character that occurs both in *seps* and *wspace* is treated as a white space character.

A separator at the end of a string is interpreted as a terminator; in this case, the separator does not produce a trailing empty string. Also see 26.5.11.

```
gap> SplitString( "substr1:substr2::substr4", ":" );
[ "substr1", "substr2", "", "substr4" ]
gap> SplitString( "a;b;c;d;", ";" );
[ "a", "b", "c", "d" ]
gap> SplitString( "/home//user//dir/", "", "/" );
[ "home", "user", "dir" ]
```

7 ▶ `ReplacedString(` *string*`,` *old*`,` *new* `)` F

replaces occurrences of the string *old* in *string* by *new*, starting from the left and always replacing the first occurrence. To avoid infinite recursion, characters which have been replaced already, are not subject to renewed replacement.

```
gap> ReplacedString("abacab","a","zl");
"zlbzlczlb"
gap> ReplacedString("ababa", "aba","c");
"cba"
gap> ReplacedString("abacab","a","ba");
"babbacbab"
```

8 ▶ `NormalizeWhitespace(` *string* `)` F

This function changes the string *string* in place. The characters (space), `\n`, `\r` and `\t` are considered as **white space**. Leading and trailing white space characters in *string* are removed. Sequences of white space characters between other characters are replaced by a single space character.

See 26.5.9 for a non-destructive version.

```
gap> s := "   x y \n\n\t\r  z\n   \n";
"   x y \n\n\t\r  z\n   \n"
gap> NormalizeWhitespace(s);
gap> s;
"x y z"
```

9 ▶ `NormalizedWhitespace(` *str* `)` F

This function returns a copy of string *str* to which 26.5.8 was applied.

For the possibility to print GAP objects to strings, see 10.7.

10 ▶ `JoinStringsWithSeparator(` *list*`[,` *sep*`]` `)` F

joins *list* (a list of strings) after interpolating *sep* (or `","` if the second argument is omitted) between each adjacent pair of strings; *sep* should be a string.

Examples

```
gap> list := List([1..10], String);
[ "1", "2", "3", "4", "5", "6", "7", "8", "9", "10" ]
gap> JoinStringsWithSeparator(list);
"1,2,3,4,5,6,7,8,9,10"
gap> JoinStringsWithSeparator(["The", "quick", "brown", "fox"], " ");
"The quick brown fox"
gap> JoinStringsWithSeparator(["a", "b", "c", "d"], ",\n    ");
"a,\n    b,\n    c,\n    d"
gap> Print("    ", last, "\n");
    a,
    b,
    c,
    d
```

Recall, `last` is the last expression output by GAP.

11 ▶ `Chomp( str )` F

Like the similarly named Perl function, `Chomp` removes a trailing newline character (or carriage-return line-feed couplet) from a string argument *str* if present and returns the result. If *str* is not a string or does not have such trailing character(s) it is returned unchanged. This latter property means that `Chomp` is safe to use in cases where one is manipulating the result of another function which might sometimes return `fail`, for example.

```
gap> Chomp("The quick brown fox jumps over the lazy dog.\n");
"The quick brown fox jumps over the lazy dog."
gap> Chomp("The quick brown fox jumps over the lazy dog.\r\n");
"The quick brown fox jumps over the lazy dog."
gap> Chomp("The quick brown fox jumps over the lazy dog.");
"The quick brown fox jumps over the lazy dog."
gap> Chomp(fail);
fail
gap> Chomp(32);
32
```

Note: `Chomp` only removes a trailing newline character from *str*. If your string contains several newline characters and you really want to split *str* into lines at the newline characters (and remove those newline characters) then you should use `SplitString` (see 26.5.6), e.g.

```
gap> str := "The quick brown fox\njumps over the lazy dog.\n";
"The quick brown fox\njumps over the lazy dog.\n"
gap> SplitString(str, "", "\n");
[ "The quick brown fox", "jumps over the lazy dog." ]
gap> Chomp(str);
"The quick brown fox\njumps over the lazy dog."
```

26.6 Character Conversion

The following functions convert characters in their internal integer values and vice versa. Note that the number corresponding to a particular character might depend on the system used. While most systems use an extension of ASCII, in particular character values outside the range 32-126 might differ between architectures.

All functions in this section are internal and behaviour is undefined if invarid arguments are given.

1 ▶ `INT_CHAR(`*char*`)` F

returns an integer value in the range 0-255 that corresponds to *char*.

2 ▶ `CHAR_INT(`*int*`)` F

returns a character which corresponds to the integer value *int*, which must be in the range 0-255.

```
gap> c:=CHAR_INT(65);
'A'
gap> INT_CHAR(c);
65
```

3 ▶ `SINT_CHAR(`*char*`)` F

returns a signed integer value in the range −128–127 that corresponds to *char*.

4 ▶ `CHAR_SINT(`*int*`)` F

returns a character which corresponds to the signed integer value *int*, which must be in the range −128–127.

The signed and unsigned integer functions behave the same for values in the range from 0 to 127.

```
gap> SINT_CHAR(c);
65
gap> c:=CHAR_SINT(-20);;
gap> SINT_CHAR(c);
-20
gap> INT_CHAR(c);
236
gap> SINT_CHAR(CHAR_INT(255));
-1
```

26.7 Operations to Evaluate Strings

1 ▶ `Int(` *str* `)` A
▶ `Rat(` *str* `)` A
▶ `IntHexString(` *str* `)` F

return either an integer (`Int` and `IntHexString`), or a rational (`Rat`) as represented by the string *str*. `Int` returns `fail` if non-digit characters occur in *str*. For `Rat`, the argument string may start with the sign character `'-'`, followed by either a sequence of digits or by two sequences of digits that are separated by one of the characters `'/'` or `'.'`, where the latter stands for a decimal dot. (The methods only evaluate numbers but do **not** perform arithmetic!)

`IntHexString` evaluates an integer written with hexa-decimal digits. Here the letters *a-f* or *A-F* are used as **digits** *10-15*. An error occurs when a wrong character is found in the string. This function can be used (together with 26.5.2) for efficiently storing and reading large integers from respectively into GAP. Note that the translation between integers and their hexa-decimal representation costs linear computation time in terms of the number of digits, while translation from and into decimal representation needs substantial computations. If *str* is not in compact string representation then 26.2.2 is applied to it as side effect.

```
gap> Int("12345")+1;
12346
gap> Int("123/45");
fail
gap> Int("1+2");
fail
gap> Int("-12");
-12
gap> Rat("123/45");
41/15
gap> Rat( "123.45" );
2469/20
gap> IntHexString("-abcdef0123456789");
-12379813738877118345
gap> HexStringInt(last);
"-ABCDEF0123456789"
```

2 ▶ `Ordinal( n )` F

returns the ordinal of the integer n as a string.

```
gap> Ordinal(2);  Ordinal(21);  Ordinal(33);  Ordinal(-33);
"2nd"
"21st"
"33rd"
"-33rd"
```

3 ▶ `EvalString( expr )` F

passes *expr* (a string) through an input text stream so that GAP interprets it, and returns the result. The following trivial example demonstrates its use.

```
gap> a:=10;
10
gap> EvalString("a^2");
100
```

`EvalString` is intended for **single** expressions. A sequence of commands may be interpreted by using the functions `InputTextString` (see 10.7.1) and `ReadAsFunction` (see 10.3.2) together; see 10.3 for an example.

26.8 Calendar Arithmetic

All calendar functions use the Gregorian calendar.

1 ▶ `DaysInYear( year )` F

returns the number of days in a year.

2 ▶ `DaysInMonth( month, year )` F

returns the number of days in month number *month* of *year* (and `fail` if `month` is integer not in valid range.

```
gap> DaysInYear(1998);
365
gap> DaysInMonth(3,1998);
31
```

3 ▶ `DMYDay(` *day* `)` F

converts a number of days, starting 1-Jan-1970 to a list [*day*,*month*,*year*] in Gregorian calendar counting.

4 ▶ `DayDMY(` *dmy* `)` F

returns the number of days from 01-Jan-1970 to the day given by *dmy*. *dmy* must be a list of the form [*day*,*month*,*year*] in Gregorian calendar counting. The result is `fail` on input outside valid ranges.

Note that this makes not much sense for early dates like: before 1582 (no Gregorian calendar at all), or before 1753 in many English countries or before 1917 in Russia.

5 ▶ `WeekDay(` *date* `)` F

returns the weekday of a day given by *date*. *date* can be a number of days since 1-Jan-1970 or a list [*day*,*month*,*year*].

6 ▶ `StringDate(` *date* `)` F

converts *date* to a readable string. *date* can be a number of days since 1-Jan-1970 or a list [*day*,*month*,*year*].

```
gap> DayDMY([1,1,1970]);DayDMY([2,1,1970]);
0
1
gap> DMYDay(12345);
[ 20, 10, 2003 ]
gap> WeekDay([11,3,1998]);
"Wed"
gap> StringDate([11,3,1998]);
"11-Mar-1998"
```

7 ▶ `HMSMSec(` *msec* `)` F

converts a number *msec* of milliseconds into a list [*hour*,*min*,*sec*,*milli*].

8 ▶ `SecHMSM(` *hmsm* `)` F

is the reverse of `HMSMSec`.

9 ▶ `StringTime(` *time* `)` F

converts *time* (given as a number of milliseconds or a list [*hour*, *min*, *sec*, *milli*]) to a readable string.

```
gap> HMSMSec(Factorial(10));
[ 1, 0, 28, 800 ]
gap> SecHMSM([1,10,5,13]);
4205013
gap> StringTime([1,10,5,13]);
" 1:10:05.013"
```

10 ▶ `SecondsDMYhms(` *DMYhms* `)` F

returns the number of seconds from 01-Jan-1970, 00:00:00, to the time given by *DMYhms*. *DMYhms* must be a list of the form [*day*,*month*,*year*,*hour*,*minute*,*second*]. The remarks on the Gregorian calendar in the section on 26.8.4 apply here as well. The last three arguments must lie in the appropriate ranges.

11 ▶ `DMYhmsSeconds( secs )` F

This is the inverse function to 26.8.10.

```
gap> SecondsDMYhms([ 9, 9, 2001, 1, 46, 40 ]);
1000000000
gap> DMYhmsSeconds(-1000000000);
[ 24, 4, 1938, 22, 13, 20 ]
```

27 Records

Records are next to lists the most important way to collect objects together. A record is a collection of **components**. Each component has a unique **name**, which is an identifier that distinguishes this component, and a **value**, which is an object of arbitrary type. We often abbreviate **value of a component** to **element**. We also say that a record **contains** its elements. You can access and change the elements of a record using its name.

Record literals are written by writing down the components in order between "`rec(`" and "`)`", and separating them by commas "`,`". Each component consists of the name, the assignment operator `:=`, and the value. The **empty record**, i.e., the record with no components, is written as `rec()`.

```
gap> rec( a := 1, b := "2" ); # a record with two components
rec( a := 1, b := "2" )
gap> rec( a := 1, b := rec( c := 2 ) ); # record may contain records
rec( a := 1, b := rec( c := 2 ) )
```

We may use the `Display` function to illustrate the hierarchy of the record components.

```
gap> Display( last );
rec(
  a := 1,
  b := rec(
      c := 2 ) )
```

Records usually contain elements of various types, i.e., they are usually not homogeneous like lists.

1 ▶ `IsRecord(` *obj* `)` C
▶ `IsRecordCollection(` *obj* `)` C
▶ `IsRecordCollColl(` *obj* `)` C

```
gap> IsRecord( rec( a := 1, b := 2 ) );
true
gap> IsRecord( IsRecord );
false
```

2 ▶ `RecNames(` *rec* `)` A

returns a list of strings corresponding to the names of the record components of the record *rec*.

```
gap> r := rec( a := 1, b := 2 );;
gap> RecNames( r );
[ "a", "b" ]
```

Note that you cannot use the string result in the ordinary way to access or change a record component. You must use the *rec*`.(`*name*`)` construct (see 27.1 and 27.2).

27.1 Accessing Record Elements

1 ▶ *rec*.*name* O

The above construct evaluates to the value of the record component with the name *name* in the record *rec*. Note that the *name* is not evaluated, i.e. it is taken literal.

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a;
1
gap> r.b;
2
```

2 ▶ *rec*.(*name*) O

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then evaluates to the element of the record *rec* whose name is, as a string, equal to *name*.

```
gap> old := rec( a := 1, b := 2 );;
gap> new := rec();
rec(  )
gap> for i in RecNames( old ) do
>  new.(i) := old.(i);
> od;
gap> Display( new );
rec(
  a := 1,
  b := 2 )
```

27.2 Record Assignment

1 ▶ *rec*.*name* := *obj* O

The record assignment assigns the object *obj*, which may be an object of arbitrary type, to the record component with the name *name*, which must be an identifier, of the record *rec*. That means that accessing the element with name *name* of the record *rec* will return *obj* after this assignment. If the record *rec* has no component with the name *name*, the record is automatically extended to make room for the new component.

```
gap> r := rec( a := 1, b := 2 );;
gap> r.a := 10;;
gap> Display( r );
rec(
  a := 10,
  b := 2 )
gap> r.c := 3;;
gap> Display( r );
rec(
  a := 10,
  b := 2,
  c := 3 )
```

Note that assigning to a record changes the record.

The function `IsBound` can be used to test if a record has a component with a certain name, the function `Unbind` (see 4.8.1) can be used to remove a component with a certain name again.

```
gap> IsBound(r.a);
true
gap> IsBound(r.d);
false
gap> Unbind(r.b);
gap> Display( r );
rec(
  a := 10,
  c := 3 )
```

2 ▶ *rec*.(*name*) := *obj* O

This construct is similar to the above construct. The difference is that the second operand *name* is evaluated. It must evaluate to a string or an integer otherwise an error is signalled. The construct then assigns *obj* to the record component of the record *rec* whose name is, as a string, equal to *name*.

27.3 Identical Records

With the record assignment (see 27.2) it is possible to change a record. This section describes the semantic consequences of this fact which are essentially the same as for lists (see 21.6).

```
r := rec( a := 1 );
r := rec( a := 1, b := 2 );
```

The second assignment does not change the first record, instead it assigns a new record to the variable `r`. On the other hand, in the following example the record is changed by the second assignment.

```
r := rec( a := 1 );
r.b := 2;
```

To understand the difference first think of a variable as a name for an object. The important point is that a record can have several names at the same time. An assignment *var* `:=` *record* means in this interpretation that *var* is a name for the object *record*. At the end of the following example `r2` still has the value `rec( a := 1 )` as this record has not been changed and nothing else has been assigned to `r2`.

```
r1 := rec( a := 1 );
r2 := r1;
r1 := rec( a := 1, b := 2 );
```

But after the following example the record for which `r2` is a name has been changed and thus the value of `r2` is now `rec( a := 1, b := 2 )`.

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

We shall say that two records are **identical** if changing one of them by a record assignment also changes the other one. This is slightly incorrect, because if **two** records are identical, there are actually only two names for **one** record. However, the correct usage would be very awkward and would only add to the confusion. Note that two identical records must be equal, because there is only one records with two different names. Thus identity is an equivalence relation that is a refinement of equality.

Let us now consider under which circumstances two records are identical.

If you enter a record literal then the record denoted by this literal is a new record that is not identical to any other record. Thus in the following example `r1` and `r2` are not identical, though they are equal of course.

```
r1 := rec( a := 1 );
r2 := rec( a := 1 );
```

Also in the following example, no records in the list `l` are identical.

```
l := [];
for i in [1..10] do
  l[i] := rec( a := 1 );
od;
```

If you assign a record to a variable no new record is created. Thus the record value of the variable on the left hand side and the record on the right hand side of the assignment are identical. So in the following example `r1` and `r2` are identical records.

```
r1 := rec( a := 1 );
r2 := r1;
```

If you pass a record as argument, the old record and the argument of the function are identical. Also if you return a record from a function, the old record and the value of the function call are identical. So in the following example `r1` and `r2` are identical record

```
r1 := rec( a := 1 );
f := function ( r ) return r; end;
r2 := f( r1 );
```

The functions `StructuralCopy` and `ShallowCopy` (see 12.7.2 and 12.7.1) accept a record and return a new record that is equal to the old record but that is **not** identical to the old record. The difference between `StructuralCopy` and `ShallowCopy` is that in the case of `ShallowCopy` the corresponding components of the new and the old records will be identical, whereas in the case of `StructuralCopy` they will only be equal. So in the following example `r1` and `r2` are not identical records.

```
r1 := rec( a := 1 );
r2 := Copy( r1 );
```

If you change a record it keeps its identity. Thus if two records are identical and you change one of them, you also change the other, and they are still identical afterwards. On the other hand, two records that are not identical will never become identical if you change one of them. So in the following example both `r1` and `r2` are changed, and are still identical.

```
r1 := rec( a := 1 );
r2 := r1;
r1.b := 2;
```

27.4 Comparisons of Records

1 ▶ *rec1* `=` *rec2* O
▶ *rec1* `<>` *rec2* O

Two records are considered equal, if for each component of one record the other record has a component of the same name with an equal value and vice versa.

```
gap> rec( a := 1, b := 2 ) = rec( b := 2, a := 1 );
true
gap> rec( a := 1, b := 2 ) = rec( a := 2, b := 1 );
false
gap> rec( a := 1 ) = rec( a := 1, b := 2 );
false
gap> rec( a := 1 ) = 1;
false
```

2 ► *rec1* `<` *rec2* O
► *rec1* `<=` *rec2* O

To compare records we imagine that the components of both records are sorted according to their names. Then the records are compared lexicographically with unbound elements considered smaller than anything else. Precisely one record *rec1* is considered less than another record *rec2* if *rec2* has a component with name *name2* and either *rec1* has no component with this name or *rec1*`.`*name2* `<` *rec2*`.`*name2* and for each component of *rec1* with name *name1* `<` *name2* *rec2* has a component with this name and *rec1*`.`*name1* `=` *rec2*`.`*name1*.

```
gap> rec( a := 1, b := 2 ) < rec( b := 2, a := 1 ); # they are equal
false
gap> rec( a := 1 ) < rec( a := 1, b := 2 ); # unbound is less than 2
true
gap> # note in the following examples that the 'a' elements are compared first
gap> rec( a := 1, b := 2 ) < rec( a := 2, b := 0 ); # 1 is less than 2
true
gap> rec( a := 1 ) < rec( a := 0, b := 2 ); # 0 is less than 1
false
gap> rec( b := 1 ) < rec( b := 0, a := 2 ); # unbound is less than 2
true
```

27.5 IsBound and Unbind for Records

► 'IsBound(*rec*`.`*name*)' O

`IsBound` returns `true` if the record *rec* has a component with the name *name* (which must be an identifier) and `false` otherwise. *rec* must evaluate to a record, otherwise an error is signalled.

```
gap> r := rec( a := 1, b := 2 );;
gap> IsBound( r.a );
true
gap> IsBound( r.c );
false
```

► 'Unbind(*rec*`.`*name*)' O

`Unbind` deletes the component with the name *name* in the record *rec*. That is, after execution of `Unbind`, *rec* no longer has a record component with this name. Note that it is not an error to unbind a nonexisting record component. *rec* must evaluate to a record, otherwise an error is signalled.

```
gap> r := rec( a := 1, b := 2 );;
gap> Unbind( r.a ); r;
rec( b := 2 )
gap> Unbind( r.c ); r;
rec( b := 2 )
```

Note that `IsBound` and `Unbind` are special in that they do not evaluate their argument, otherwise `IsBound` would always signal an error when it is supposed to return `false` and there would be no way to tell `Unbind` which component to remove.

27.6 Record Access Operations

Internally, record accesses are done using the operations listed in this section. For the records implemented in the kernel, kernel methods are provided for all these operations but otherwise it is possible to install methods for these operations for any object. This permits objects to simulate record behavior.

To save memory, records do not store a list of all component names, but only numbers identifying the components. There numbers are called **RNams**. GAP keeps a list of all RNams that are used and provides functions to translate RNams to strings that give the component names and vice versa.

1 ▶ `NameRNam(`*nr*`)` F

returns a string representing the component name corresponding to the RNam *nr*.

2 ▶ `RNamObj(`*str*`)` F
▶ `RNamObj(`*int*`)` F

returns a number (the RNam) corresponding to the string *str*. It is also possible to pass a positive integer *int* in which case the decimal expansion of *int* is used as a string.

```
gap> NameRNam(798);
"BravaisSupergroups"
gap> RNamObj("blubberflutsch");
2075
gap> NameRNam(last);
"blubberflutsch"
```

The correspondence between Strings and RNams is not predetermined ab initio, but RNams are assigned to component names dynamically on a "first come, first serve" basis. Therefore, depending on the version of the library you are using and on the assignments done so far, the **same** component name may be represented by **different** RNams in different runs of GAP.

The following operations are called for record accesses to arbitrary objects. If applicable methods are installed, they are called when the object is accessed as a record.

3 ▶ `\.(`*obj*`,`*rnam*`)` O
▶ `IsBound\.(`*obj*`,`*rnam*`)` O
▶ `\.\:\=(`*obj*`,`*rnam*`)` O
▶ `Unbind\.(`*obj*`,`*rnam*`)` O

These operations implement component access, test for element boundness, component assignment and removal of the component represented by the RNam *rnam*.

The component identifier *rnam* is always declared as `IsPosInt`.

28 Collections

A **collection** in GAP consists of elements in the same family (see 13.1). The most important kinds of collections are **homogeneous lists** (see 21) and **domains** (see 12.4). Note that a list is never a domain, and a domain is never a list. A list is a collection if and only if it is nonempty and homogeneous.

Basic operations for collections are `Size` (see 28.3.6) and `Enumerator` (see 28.2.2); for **finite** collections, `Enumerator` admits to delegate the other operations for collections (see 28.3 and 28.4) to functions for lists (see 21). Obviously, special methods depending on the arguments are needed for the computation of e.g. the intersection of two **infinite** domains.

1 ▶ `IsCollection(` *obj* `)` C

tests whether an object is a collection.

Some of the functions for lists and collections have been described in the chapter about lists, mainly in Section 21.20. In this chapter, we describe those functions for which the "collection aspect" seems to be more important than the "list aspect". As in Chapter 21, an argument that is a list will be denoted by *list*, and an argument that is a collection will be denoted by C.

28.1 Collection Families

1 ▶ `CollectionsFamily(` *Fam* `)` A

For a family *Fam*, `CollectionsFamily` returns the family of all collections that consist of elements in *Fam*.

Note that families (see 13.1) are used to describe relations between objects. Important such relations are that between an element *elm* and each collection of elements that lie in the same family as *elm*, and that between two collections whose elements lie in the same family. Therefore, all collections of elements in the family *Fam* form the new family `CollectionsFamily(` *Fam* `)`.

2 ▶ `IsCollectionFamily(` *Fam* `)` C

is `true` if *Fam* is a family of collections, and `false` otherwise.

3 ▶ `ElementsFamily(` *Fam* `)` A

returns the family from which the collections family *Fam* was created by `CollectionsFamily`. The way a collections family is created, it always has its elements family stored. If *Fam* is not a collections family (see 28.1.2) then an error is signalled.

```
gap> fam:= FamilyObj( (1,2) );;
gap> collfam:= CollectionsFamily( fam );;
gap> fam = collfam;  fam = ElementsFamily( collfam );
false
true
gap> collfam = FamilyObj( [ (1,2,3) ] );  collfam = FamilyObj( Group( () ) );
true
true
gap> collfam = CollectionsFamily( collfam );
```

```
false
```

4 ▶ `CategoryCollections( filter )` F

Let *filter* be a filter that is `true` for all elements of a family *Fam*, by construction of *Fam*. Then `CategoryCollections` returns a category that is `true` for all elements in `CollectionsFamily( Fam )`.

For example, the construction of `PermutationsFamily` guarantees that each of its elements lies in the filter `IsPerm`, and each collection of permutations lies in the category `CategoryCollections( IsPerm )`.

Note that this works only if the collections category is created **before** the collections family. So it is necessary to construct interesting collections categories immediately after the underlying category has been created.

28.2 Lists and Collections

1 ▶ `IsListOrCollection( obj )` C

Several functions are defined for both lists and collections, for example `Intersection` (see 28.4.2), `Iterator` (see 28.7.1), and `Random` (see 28.6.1). `IsListOrCollection` is a supercategory of `IsList` and `IsCollection` (that is, all lists and collections lie in this category), which is used to describe the arguments of functions such as the ones listed above.

The following functions take a **list or collection** as argument, and return a corresponding **list**. They differ in whether or not the result is mutable or immutable (see 12.6), guaranteed to be sorted, or guaranteed to admit list access in constant time (see 21.1.5).

2 ▶ `Enumerator( C )` A
▶ `Enumerator( list )` A

`Enumerator` returns an immutable list *enum*. If the argument is a list *list* (which may contain holes), then `Length( enum )` is `Length( list )`, and *enum* contains the elements (and holes) of *list* in the same order. If the argument is a collection C that is not a list, then `Length( enum )` is the number of different elements of C, and *enum* contains the different elements of C in an unspecified order, which may change for repeated calls of `Enumerator`. *enum*`[`*pos*`]` may not execute in constant time (see 21.1.5), and the size of *enum* in memory is as small as is feasible.

For lists *list*, the default method is `Immutable`. For collections C that are not lists, there is no default method.

3 ▶ `EnumeratorSorted( C )` A
▶ `EnumeratorSorted( list )` A

`EnumeratorSorted` returns an immutable list *enum*. The argument must be a collection C or a list *list* which may contain holes but whose elements lie in the same family (see 13.1). `Length( enum )` is the number of different elements of C resp. *list*, and *enum* contains the different elements in sorted order, w.r.t. `<`. *enum*`[`*pos*`]` may not execute in constant time (see 21.1.5), and the size of *enum* in memory is as small as is feasible.

```
gap> Enumerator( [ 1, 3,, 2 ] );
[ 1, 3,, 2 ]
gap> enum:= Enumerator( Rationals );;  elm:= enum[ 10^6 ];
-69/907
gap> Position( enum, elm );
1000000
gap> IsMutable( enum );  IsSortedList( enum );
false
false
gap> IsConstantTimeAccessList( enum );
```

```
false
gap> EnumeratorSorted( [ 1, 3,, 2 ] );
[ 1, 2, 3 ]
```

4 ▶ `EnumeratorByFunctions( D, record )` F
▶ `EnumeratorByFunctions( Fam, record )` F

`EnumeratorByFunctions` returns an immutable, dense, and duplicate-free list *enum* for which `IsBound`, element access, `Length`, and `Position` are computed via prescribed functions.

Let *record* be a record with at least the following components.

`ElementNumber`
: a function taking two arguments *enum* and *pos*, which returns *enum*`[ pos ]` (see 21.2); it can be assumed that the argument *pos* is a positive integer, but *pos* may be larger than the length of *enum* (in which case an error must be signalled); note that the result must be immutable since *enum* itself is immutable,

`NumberElement`
: a function taking two arguments *enum* and *elm*, which returns `Position( enum, elm )` (see 21.16.1); it cannot be assumed that *elm* is really contained in *enum* (and `fail` must be returned if not); note that for the three argument version of `Position`, the method that is available for duplicate-free lists suffices.

Further (data) components may be contained in *record* which can be used by these function.

If the first argument is a domain D then *enum* lists the elements of D (in general *enum* is **not** sorted), and methods for `Length`, `IsBound`, and `PrintObj` may use D.

If one wants to describe the result without creating a domain then the elements are given implicitly by the functions in *record*, and the first argument must be a family *Fam* which will become the family of *enum*; if *enum* is not homogeneous then *Fam* must be `ListsFamily`, otherwise it must be the collections family of any element in *enum*. In this case, additionally the following component in *record* is needed.

`Length`
: a function taking the argument *enum*, which returns the length of *enum* (see 21.17.5).

The following components are optional; they are used if they are present but default methods are installed for the case that they are missing.

`IsBound\[\]`
: a function taking two arguments *enum* and k, which returns `IsBound(` *enum*`[ k ] )` (see 21.2); if this component is missing then `Length` is used for computing the result,

`Membership`
: a function taking two arguments *elm* and *enum*, which returns `true` is *elm* is an element of *enum*, and `false` otherwise (see 21.2); if this component is missing then `NumberElement` is used for computing the result,

`AsList`
: a function taking one argument *enum*, which returns a list with the property that the access to each of its elements will take roughly the same time (see 21.1.5); if this component is missing then `ConstantTimeAccessList` is used for computing the result,

`ViewObj` and `PrintObj`
: two functions that print what one wants to be printed when `View(` *enum* `)` or `Print(` *enum* `)` is called (see 6.2), if the `ViewObj` component is missing then the `PrintObj` method is used as a default.

If the result is known to have additional properties such as being strictly sorted (see 21.17.4) then it can be useful to set these properties after the construction of the enumerator, before it is used for the first time. And in the case that a new sorted enumerator of a domain is implemented via `EnumeratorByFunctions`, and this construction is installed as a method for the operation `Enumerator` (see 28.2.2), then it should be installed also as a method for `EnumeratorSorted` (see 28.2.3).

Note that it is **not** checked that `EnumeratorByFunctions` really returns a dense and duplicate-free list. `EnumeratorByFunctions` does **not** make a shallow copy of *record*, this record is changed in place (see 3.8 in "Programming in GAP").

It would be easy to implement a slightly generalized setup for enumerators that need not be duplicate-free (where the three argument version of `Position` is supported), but the resulting overhead for the methods seems not to be justified.

▶ `List(` *C* `)`
▶ `List(` *list* `)`

This function is described in 21.20.17, together with the probably more frequently used version which takes a function as second argument and returns the list of function values of the list elements.

```
gap> l:= List( Group( (1,2,3) ) );
[ (), (1,3,2), (1,2,3) ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
true
false
true
```

5 ▶ `SortedList(` *C* `)` O
▶ `SortedList(` *list* `)` O

`SortedList` returns a new mutable and dense list *new*. The argument must be a collection *C* or a list *list* which may contain holes but whose elements lie in the same family (see 13.1). `Length(` *new* `)` is the number of elements of *C* resp. *list*, and *new* contains the elements in sorted order, w.r.t. `<=`. *new*`[`*pos*`]` executes in constant time (see 21.1.5), and the size of *new* in memory is proportional to its length.

```
gap> l:= SortedList( Group( (1,2,3) ) );
[ (), (1,2,3), (1,3,2) ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
true
true
true
gap> SortedList( [ 1, 2, 1,, 3, 2 ] );
[ 1, 1, 2, 2, 3 ]
```

6 ▶ `SSortedList(` *C* `)` O
▶ `SSortedList(` *list* `)` O
▶ `Set(` *C* `)` O

`SSortedList` ("strictly sorted list") returns a new dense, mutable, and duplicate free list *new*. The argument must be a collection *C* or a list *list* which may contain holes but whose elements lie in the same family (see 13.1). `Length(` *new* `)` is the number of different elements of *C* resp. *list*, and *new* contains the different elements in strictly sorted order, w.r.t. `<`. *new*`[`*pos*`]` executes in constant time (see 21.1.5), and the size of *new* in memory is proportional to its length.

`Set` is simply a synonym for `SSortedList`.

```
gap> l:= SSortedList( Group( (1,2,3) ) );
[ (), (1,2,3), (1,3,2) ]
gap> IsMutable( l );  IsSSortedList( l );  IsConstantTimeAccessList( l );
true
true
true
gap> SSortedList( [ 1, 2, 1,, 3, 2 ] );
[ 1, 2, 3 ]
```

7 ▶ `AsList( `*C*` )` A
▶ `AsList( `*list*` )` A

`AsList` returns a immutable list *imm*. If the argument is a list *list* (which may contain holes), then `Length(` *imm* `)` is `Length(` *list* `)`, and *imm* contains the elements (and holes) of *list* in the same order. If the argument is a collection C that is not a list, then `Length(` *imm* `)` is the number of different elements of C, and *imm* contains the different elements of C in an unspecified order, which may change for repeated calls of `AsList`. *imm*`[`*pos*`]` executes in constant time (see 21.1.5), and the size of *imm* in memory is proportional to its length.

If you expect to do many element tests in the resulting list, it might be worth to use a sorted list instead, using `AsSSortedList`.

```
gap> l:= AsList( [ 1, 3, 3,, 2 ] );
[ 1, 3, 3,, 2 ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
false
false
true
gap> AsList( Group( (1,2,3), (1,2) ) );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

8 ▶ `AsSortedList( `*C*` )` A
▶ `AsSortedList( `*list*` )` A

`AsSortedList` returns a dense and immutable list *imm*. The argument must be a collection C or a list *list* which may contain holes but whose elements lie in the same family (see 13.1). `Length(` *imm* `)` is the number of elements of C resp. *list*, and *imm* contains the elements in sorted order, w.r.t. `<=`. *new*`[`*pos*`]` executes in constant time (see 21.1.5), and the size of *imm* in memory is proportional to its length.

The only difference to the operation `SortedList` (see 28.2.5) is that `AsSortedList` returns an **immutable** list.

```
gap> l:= AsSortedList( [ 1, 3, 3,, 2 ] );
[ 1, 2, 3, 3 ]
gap> IsMutable( l );  IsSortedList( l );  IsConstantTimeAccessList( l );
false
true
true
gap> IsSSortedList( l );
false
```

9 ▶ `AsSSortedList( `*C*` )` A
▶ `AsSSortedList( `*list*` )` A
▶ `AsSet( `*C*` )` A

`AsSSortedList` ("as strictly sorted list") returns a dense, immutable, and duplicate free list *imm*. The argument must be a collection C or a list *list* which may contain holes but whose elements lie in the same

family (see 13.1). `Length( imm )` is the number of different elements of C resp. *list*, and *imm* contains the different elements in strictly sorted order, w.r.t. `<`. *imm*`[`*pos*`]` executes in constant time (see 21.1.5), and the size of *imm* in memory is proportional to its length.

Because the comparisons required for sorting can be very expensive for some kinds of objects, you should use `AsList` instead if you do not require the result to be sorted.

The only difference to the operation `SSortedList` (see 28.2.6) is that `AsSSortedList` returns an **immutable** list.

`AsSet` is simply a synonym for `AsSSortedList`.

In general a function that returns a set of elements is free, in fact encouraged, to return a domain instead of the proper set of its elements. This allows one to keep a given structure, and moreover the representation by a domain object is usually more space efficient. `AsSSortedList` must of course **not** do this, its only purpose is to create the proper set of elements.

```
gap> l:= AsSSortedList( l );
[ 1, 2, 3 ]
gap> IsMutable( l );  IsSSortedList( l );  IsConstantTimeAccessList( l );
false
true
true
gap> AsSSortedList( Group( (1,2,3), (1,2) ) );
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

10 ▶ `Elements( C )` F

`Elements` does the same as `AsSSortedList` (see 28.2.9), that is, the return value is a strictly sorted list of the elements in the list or collection C.

`Elements` is only supported for backwards compatibility. In many situations, the sortedness of the "element list" for a collection is in fact not needed, and one can save a lot of time by asking for a list that is **not** necessarily sorted, using `AsList` (see 28.2.7). If one is really interested in the strictly sorted list of elements in C then one should use `AsSet` or `AsSSortedList` instead.

28.3 Attributes and Properties for Collections

1 ▶ `IsEmpty( C )` P
▶ `IsEmpty( list )` P

`IsEmpty` returns `true` if the collection C resp. the list *list* is **empty** (that is it contains no elements), and `false` otherwise.

2 ▶ `IsFinite( C )` P

`IsFinite` returns `true` if the collection C is finite, and `false` otherwise.

The default method for `IsFinite` checks the size (see 28.3.6) of C.

Methods for `IsFinite` may call `Size`, but methods for `Size` must **not** call `IsFinite`.

3 ▶ `IsTrivial( C )` P

`IsTrivial` returns `true` if the collection C consists of exactly one element.

4 ▶ `IsNonTrivial( C )` P

`IsNonTrivial` returns `true` if the collection C is empty or consists of at least two elements (see 28.3.3).

```
gap> IsEmpty( [] );  IsEmpty( [ 1 .. 100 ] );  IsEmpty( Group( (1,2,3) ) );
true
false
false
gap> IsFinite( [ 1 .. 100 ] );  IsFinite( Integers );
true
false
gap> IsTrivial( Integers );  IsTrivial( Group( () ) );
false
true
gap> IsNonTrivial( Integers );  IsNonTrivial( Group( () ) );
true
false
```

5 ▶ `IsWholeFamily( C )` P

`IsWholeFamily` returns `true` if the collection C contains the whole family (see 13.1) of its elements.

```
gap> IsWholeFamily( Integers )
>    ;  # all rationals and cyclotomics lie in the family
false
gap> IsWholeFamily( Integers mod 3 )
>    ;  # all finite field elements in char. 3 lie in this family
false
gap> IsWholeFamily( Integers mod 4 );
true
gap> IsWholeFamily( FreeGroup( 2 ) );
true
```

6 ▶ `Size( C )` A
▶ `Size( list )` A

`Size` returns the size of the collection C, which is either an integer or `infinity`. The argument may also be a list *list*, in which case the result is the length of *list* (see 21.17.5).

The default method for `Size` checks the length of an enumerator of C.

Methods for `IsFinite` may call `Size`, but methods for `Size` must not call `IsFinite`.

```
gap> Size( [1,2,3] );  Size( Group( () ) );  Size( Integers );
3
1
infinity
```

7 ▶ `Representative( C )` A

`Representative` returns a **representative** of the collection C.

Note that `Representative` is free in choosing a representative if there are several elements in C. It is not even guaranteed that `Representative` returns the same representative if it is called several times for one collection. The main difference between `Representative` and `Random` (see 28.6.1) is that `Representative` is free to choose a value that is cheap to compute, while `Random` must make an effort to randomly distribute its answers.

If C is a domain then there are methods for `Representative` that try to fetch an element from any known generator list of C, see 30. Note that `Representative` does not try to **compute** generators of C, thus `Representative` may give up and signal an error if C has no generators stored at all.

8 ▶ `RepresentativeSmallest( C )` A

returns the smallest element in the collection C, w.r.t. the ordering `<`. While the operation defaults to comparing all elements, better methods are installed for some collections.

```
gap> Representative( Rationals );
1
gap> Representative( [ -1, -2 .. -100 ] );
-1
gap> RepresentativeSmallest( [ -1, -2 .. -100 ] );
-100
```

28.4 Operations for Collections

1 ▶ `IsSubset( C1, C2 )` O

`IsSubset` returns `true` if *C2*, which must be a collection, is a **subset** of *C1*, which also must be a collection, and `false` otherwise.

C2 is considered a subset of *C1* if and only if each element of *C2* is also an element of *C1*. That is `IsSubset` behaves as if implemented as `IsSubsetSet( AsSSortedList( C1 ), AsSSortedList( C2 ) )`, except that it will also sometimes, but not always, work for infinite collections, and that it will usually work much faster than the above definition. Either argument may also be a proper set (see 21.19).

```
gap> IsSubset( Rationals, Integers );
true
gap> IsSubset( Integers, [ 1, 2, 3 ] );
true
gap> IsSubset( Group( (1,2,3,4) ), [ (1,2,3) ] );
false
```

2 ▶ `Intersection( C1, C2 ... )` F
▶ `Intersection( list )` F
▶ `Intersection2( C1, C2 )` O

In the first form `Intersection` returns the intersection of the collections *C1*, *C2*, etc. In the second form *list* must be a **nonempty** list of collections and `Intersection` returns the intersection of those collections. Each argument or element of *list* respectively may also be a homogeneous list that is not a proper set, in which case `Intersection` silently applies `Set` (see 28.2.6) to it first.

The result of `Intersection` is the set of elements that lie in every of the collections *C1*, *C2*, etc. If the result is a list then it is mutable and new, i.e., not identical to any of *C1*, *C2*, etc.

Methods can be installed for the operation `Intersection2` that takes only two arguments. `Intersection` calls `Intersection2`.

Methods for `Intersection2` should try to maintain as much structure as possible, for example the intersection of two permutation groups is again a permutation group.

```
gap> Intersection( CyclotomicField(9), CyclotomicField(12) )
>       # this is one of the rare cases where the intersection of two infinite
>    ;  # domains works (‘CF’ is a shorthand for ‘CyclotomicField’)
CF(3)
gap> D12 := Group( (2,6)(3,5), (1,2)(3,6)(4,5) );;
```

```
gap> Intersection( D12, Group( (1,2), (1,2,3,4,5) ) );
Group([ (1,5)(2,4) ])
gap> Intersection( D12, [ (1,3)(4,6), (1,2)(3,4) ] )
>    ;  # note that the second argument is not a proper set
[ (1,3)(4,6) ]
gap> Intersection( D12, [ (), (1,2)(3,4), (1,3)(4,6), (1,4)(5,6) ] )
>       # although the result is mathematically a group it is returned as a
>    ;  # proper set because the second argument is not regarded as a group
[ (), (1,3)(4,6) ]
gap> Intersection( Group( () ), [1,2,3] );
[  ]
gap> Intersection( [2,4,6,8,10], [3,6,9,12,15], [5,10,15,20,25] )
>    ;  # two or more lists or collections as arguments are legal
[  ]
gap> Intersection( [ [1,2,4], [2,3,4], [1,3,4] ] )
>    ;  # or one list of lists or collections
[ 4 ]
```

3 ► `Union(` *C1*`,` *C2* `...` `)` F
► `Union(` *list* `)` F
► `Union2(` *C1*`,` *C2* `)` O

In the first form `Union` returns the union of the collections *C1*, *C2*, etc. In the second form *list* must be a list of collections and `Union` returns the union of those collections. Each argument or element of *list* respectively may also be a homogeneous list that is not a proper set, in which case `Union` silently applies `Set` (see 28.2.6) to it first.

The result of `Union` is the set of elements that lie in any of the collections *C1*, *C2*, etc. If the result is a list then it is mutable and new, i.e., not identical to any of *C1*, *C2*, etc.

Methods can be installed for the operation `Union2` that takes only two arguments. `Union` calls `Union2`.

```
gap> Union( [ (1,2,3), (1,2,3,4) ], Group( (1,2,3), (1,2) ) );
[ (), (2,3), (1,2), (1,2,3), (1,2,3,4), (1,3,2), (1,3) ]
gap> Union( [2,4,6,8,10], [3,6,9,12,15], [5,10,15,20,25] )
>    ;  # two or more lists or collections as arguments are legal
[ 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 20, 25 ]
gap> Union( [ [1,2,4], [2,3,4], [1,3,4] ] )
>    ;  # or one list of lists or collections
[ 1, 2, 3, 4 ]
gap> Union( [ ] );
[  ]
```

4 ► `Difference(` *C1*`,` *C2* `)` O

`Difference` returns the set difference of the collections *C1* and *C2*. Either argument may also be a homogeneous list that is not a proper set, in which case `Difference` silently applies `Set` (see 28.2.6) to it first.

The result of `Difference` is the set of elements that lie in *C1* but not in *C2*. Note that *C2* need not be a subset of *C1*. The elements of *C2*, however, that are not elements of *C1* play no role for the result. If the result is a list then it is mutable and new, i.e., not identical to *C1* or *C2*.

```
gap> Difference( [ (1,2,3), (1,2,3,4) ], Group( (1,2,3), (1,2) ) );
[ (1,2,3,4) ]
```

28.5 Membership Test for Collections

1 ▶ *obj* `in` *C*
▶ `\in(` *obj*`,` *C* `)` O

returns `true` if the object *obj* lies in the collection *C*, and `false` otherwise.

The infix version of the command calls the operation `\in`, for which methods can be installed.

```
gap> 13 in Integers;  [ 1, 2 ] in Integers;
true
false
gap> g:= Group( (1,2) );;  (1,2) in g;  (1,2,3) in g;
true
false
```

28.6 Random Elements

1 ▶ `Random(` *C* `)` O
▶ `Random(` *list* `)` O

`Random` returns a (pseudo-)random element of the collection *C* respectively the list *list*.

The distribution of elements returned by `Random` depends on the argument. For a list *list*, all elements are equally likely. The same holds usually for finite collections *C* that are not lists. For infinite collections *C* some reasonable distribution is used.

See the chapters of the various collections to find out which distribution is being used.

For some collections ensuring a reasonable distribution can be difficult and require substantial runtime. If speed at the cost of equal distribution is desired, the operation `PseudoRandom` should be used instead.

Note that `Random` is of course **not** an attribute.

```
gap> Random(Rationals);
-4
gap> g:= Group( (1,2,3) );;  Random( g );  Random( g );
(1,3,2)
(1,3,2)
```

2 ▶ `StateRandom( )` F
▶ `RestoreStateRandom(` *obj* `)` F

For debugging purposes, it can be desirable to reset the random number generator to a state it had before. `StateRandom` returns a GAP object that represents the current state of the random number generator used by `RandomList`.

By calling `RestoreStateRandom` with this object as argument, the random number is reset to this same state.

(The same result can be obtained by accessing the two global variables `R_N` and `R_X`.)

(The format of the object used to represent the random generator seed is not guaranteed to be stable betweed different machines or versions of GAP.

```
gap> seed:=StateRandom();;
gap> List([1..10],i->Random(Integers));
[ -3, 2, 5, 1, 0, -2, 4, 3, 5, 3 ]
gap> List([1..10],i->Random(Integers));
[ 1, -2, 1, -1, -2, 4, -1, -3, -1, 1 ]
gap> RestoreStateRandom(seed);
gap> List([1..10],i->Random(Integers));
[ -3, 2, 5, 1, 0, -2, 4, 3, 5, 3 ]
```

3 ► `PseudoRandom(` *C* `)` O
► `PseudoRandom(` *list* `)` O

`PseudoRandom` returns a pseudo random element of the collection *C* respectively the list *list*, which can be roughly described as follows. For a list *list*, `PseudoRandom` returns the same as `Random`. For collections *C* that are not lists, the elements returned by `PseudoRandom` are **not** necessarily equally distributed, even for finite collections *C*; the idea is that `Random` (see 28.6.1) returns elements according to a reasonable distribution, `PseudoRandom` returns elements that are cheap to compute but need not satisfy this strong condition, and `Representative` (see 28.3.7) returns arbitrary elements, probably the same element for each call.

The method used by GAP to obtain random elements may depend on the type object.

Many random methods in the library are eventually based on the function `RandomList`. As `RandomList` is restricted to lists of up to 2^{28} elements, this may create problems for very large collections. Also note that the method used by `RandomList` is intended to provide a fast algorithm rather than to produce high quality randomness for statistical purposes.

If you implement your own `Random` methods we recommend that they initialize their seed to a defined value when they are loaded to permit to reproduce calculations even if they involved random elements.

4 ► `RandomList(` *list* `)` F

For a dense list *list* of up to 2^{28} elements, `RandomList` returns a (pseudo-)random element with equal distribution.

The algorithm used is an additive number generator (Algorithm A in section 3.2.2 of [Knu98] with lag 30)

This random number generator is (deliberately) initialized to the same values when GAP is started, so different runs of GAP with the same input will always produce the same result, even if random calculations are involved.

See `StateRandom` for a description on how to reset the random number generator to a previous state.

28.7 Iterators

1 ► `Iterator(` *C* `)` O
► `Iterator(` *list* `)` O

Iterators provide a possibility to loop over the elements of a (countable) collection *C* or a list *list*, without repetition. For many collections *C*, an iterator of *C* need not store all elements of *C*, for example it is possible to construct an iterator of some infinite domains, such as the field of rational numbers.

`Iterator` returns a mutable **iterator** *iter* for its argument. If this is a list *list* (which may contain holes), then *iter* iterates over the elements (but not the holes) of *list* in the same order (see 28.7.6 for details). If this is a collection *C* but not a list then *iter* iterates over the elements of *C* in an unspecified order, which may change for repeated calls of `Iterator`. Because iterators returned by `Iterator` are mutable (see 12.6), each call of `Iterator` for the same argument returns a **new** iterator. Therefore `Iterator` is not an attribute (see 13.5).

The only operations for iterators are `IsDoneIterator`, `NextIterator`, and `ShallowCopy`. In particular, it is only possible to access the next element of the iterator with `NextIterator` if there is one, and this can be checked with `IsDoneIterator` (see 28.7.5). For an iterator *iter*, `ShallowCopy(` *iter* `)` is a mutable iterator *new* that iterates over the remaining elements independent of *iter*; the results of `IsDoneIterator` for *iter* and *new* are equal, and if *iter* is mutable then also the results of `NextIterator` for *iter* and *new* are equal; note that `=` is not defined for iterators, so the equality of two iterators cannot be checked with `=`.

When `Iterator` is called for a **mutable** collection C then it is not defined whether *iter* respects changes to C occurring after the construction of *iter*, except if the documentation explicitly promises a certain behaviour. The latter is the case if the argument is a mutable list *list* (see 28.7.6 for subtleties in this case).

It is possible to have `for`-loops run over mutable iterators instead of lists.

In some situations, one can construct iterators with a special succession of elements, see 59.5.6 for the possibility to loop over the elements of a vector space w.r.t. a given basis.

For lists, `Iterator` is implemented by `IteratorList(` *list* `)`. For collections that are not lists, the default method is `IteratorList( Enumerator(` C `) )`. Better methods depending on C should be provided if possible.

For random access to the elements of a (possibly infinite) collection, **enumerators** are used. See 21.23 for the facility to compute a list from C, which provides a (partial) mapping from C to the positive integers.

```
gap> iter:= Iterator( GF(5) );
<iterator>
gap> l:= [];;
gap> for i in iter do Add( l, i ); od; l;
[ 0*Z(5), Z(5)^0, Z(5), Z(5)^2, Z(5)^3 ]
gap> iter:= Iterator( [ 1, 2, 3, 4 ] );;  l:= [];;
gap> for i in iter do
>      new:= ShallowCopy( iter );
>      for j in new do Add( l, j ); od;
>    od; l;
[ 2, 3, 4, 3, 4, 4 ]
```

2 ▶ `IteratorSorted(` C `)` O
▶ `IteratorSorted(` *list* `)` O

`IteratorSorted` returns a mutable iterator. The argument must be a collection C or a list *list* that is not necessarily dense but whose elements lie in the same family (see 13.1). It loops over the different elements in sorted order.

For collections C that are not lists, the generic method is `IteratorList( EnumeratorSorted(` C `) )`.

3 ▶ `IsIterator(` *obj* `)` C

Every iterator lies in the category `IsIterator`.

4 ▶ `IsDoneIterator(` *iter* `)` O

If *iter* is an iterator for the list or collection C then `IsDoneIterator(` *iter* `)` is `true` if all elements of C have been returned already by `NextIterator(` *iter* `)`, and `false` otherwise.

5 ▶ `NextIterator(` *iter* `)` O

Let *iter* be a mutable iterator for the list or collection C. If `IsDoneIterator(` *iter* `)` is `false` then `NextIterator` is applicable to *iter*, and the result is the next element of C, according to the succession defined by *iter*.

If `IsDoneIterator( iter )` is `true` then it is not defined what happens if `NextIterator` is called for *iter*; that is, it may happen that an error is signalled or that something meaningless is returned, or even that GAP crashes.

6 ▶ `IteratorList( list )` F

`IteratorList` returns a new iterator that allows iteration over the elements of the list *list* (which may have holes) in the same order.

If *list* is mutable then it is in principle possible to change *list* after the call of `IteratorList`. In this case all changes concerning positions that have not yet been reached in the iteration will also affect the iterator. For example, if *list* is enlarged then the iterator will iterate also over the new elements at the end of the changed list.

Note that changes of *list* will also affect all shallow copies of *list*.

7 ▶ `TrivialIterator( elm )` F

is a mutable iterator for the collection `[ elm ]` that consists of exactly one element *elm* (see 28.3.3).

```
gap> iter:= Iterator( [ 1, 2, 3, 4 ] );
<iterator>
gap> sum:= 0;;
gap> while not IsDoneIterator( iter ) do
>      sum:= sum + NextIterator( iter );
>    od;
gap> IsDoneIterator( iter ); sum;
true
10
gap> ir:= Iterator( Rationals );;
gap> l:= [];; for i in [1..20] do Add( l, NextIterator( ir ) ); od; l;
[ 0, 1, -1, 1/2, 2, -1/2, -2, 1/3, 2/3, 3/2, 3, -1/3, -2/3, -3/2, -3, 1/4,
  3/4, 4/3, 4, -1/4 ]
gap> for i in ir do
>      if DenominatorRat( i ) > 10 then break; fi;
>    od;
gap> i;
1/11
```

8 ▶ `IteratorByFunctions( record )` F

`IteratorByFunctions` returns a (mutable) iterator *iter* for which `NextIterator`, `IsDoneIterator`, and `ShallowCopy` are computed via prescribed functions.

Let *record* be a record with at least the following components.

`NextIterator`
: a function taking one argument *iter*, which returns the next element of *iter* (see 28.7.5); for that, the components of *iter* are changed,

`IsDoneIterator`
: a function taking one argument *iter*, which returns `IsDoneIterator( iter )` (see 28.7.4);

`ShallowCopy`
: a function taking one argument *iter*, which returns a record for which `IteratorByFunctions` can be called in order to create a new iterator that is independent of *iter* but behaves like *iter* w.r.t. the operations `NextIterator` and `IsDoneIterator`.

Further (data) components may be contained in *record* which can be used by these function.

`IteratorByFunctions` does **not** make a shallow copy of *record*, this record is changed in place (see 3.8 in "Programming in GAP").

29 Orderings

In GAP an ordering is a relation defined on a family, which is reflexive, anti-symmetric and transitive.

1 ▶ `IsOrdering( ` *ord* `)` C

returns `true` if and only if the object *ord* is an ordering.

2 ▶ `OrderingsFamily( ` *fam* `)` A

for a family *fam*, returns the family of all orderings on elements of *fam*.

29.1 Building new orderings

1 ▶ `OrderingByLessThanFunctionNC( ` *fam*`, ` *lt* `)` O
▶ `OrderingByLessThanFunctionNC( ` *fam*`, ` *lt*`, ` *l* `)` O

In the first form, `OrderingByLessThanFunctionNC` returns the ordering on the elements of the elements of the family *fam* according to the `LessThanFunction` given by *lt*, where *lt* is a function that takes two arguments in *fam* and returns `true` or `false`.

In the second form, for a family *fam*, a function *lt* that takes two arguments in *fam* and returns `true` or `false`, and a list *l* of properties of orderings, `OrderingByLessThanFunctionNC` returns the ordering on the elements of *fam* with `LessThanFunction` given by *lt* and with the properties from *l* set to `true`.

2 ▶ `OrderingByLessThanOrEqualFunctionNC( ` *fam*`, ` *lteq* `)` O
▶ `OrderingByLessThanOrEqualFunctionNC( ` *fam*`, ` *lteq*`, ` *l* `)` O

In the first form, `OrderingByLessThanOrEqualFunctionNC` returns the ordering on the elements of the elements of the family *fam* according to the `LessThanOrEqualFunction` given by *lteq*, where *lteq* is a function that takes two arguments in *fam* and returns `true` or `false`.

In the second form, for a family *fam*, a function *lteq* that takes two arguments in *fam* and returns `true` or `false`, and a list *l* of properties of orderings, `OrderingByLessThanOrEqualFunctionNC` returns the ordering on the elements of *fam* with `LessThanOrEqualFunction` given by *lteq* and with the properties from *l* set to `true`.

Notice that these functions do not check whether *fam* and *lt* or *lteq* are compatible, and whether the properties listed in *l* are indeed true.

```
gap> f := FreeSemigroup("a","b");;
gap> a := GeneratorsOfSemigroup(f)[1];;
gap> b := GeneratorsOfSemigroup(f)[2];;
gap> lt := function(x,y) return Length(x)<Length(y); end;
function( x, y ) ... end
gap> fam := FamilyObj(a);;
gap> ord := OrderingByLessThanFunctionNC(fam,lt);
Ordering
```

29.2 Properties and basic functionality

1 ▶ `IsWellFoundedOrdering(` *ord* `)` P

for an ordering *ord*, returns `true` if and only if the ordering is well founded. An ordering *ord* is well founded if it admits no infinite descending chains. Normally this property is set at the time of creation of the ordering and there is no general method to check whether a certain ordering is well founded.

2 ▶ `IsTotalOrdering(` *ord* `)` P

for an ordering *ord*, returns true if and only if the ordering is total. An ordering *ord* is total if any two elements of the family are comparable under *ord*. Normally this property is set at the time of creation of the ordering and there is no general method to check whether a certain ordering is total.

3 ▶ `IsIncomparableUnder(` *ord*, *el1*, *el2* `)` O

for an ordering *ord* on the elements of the family of *el1* and *el2*, returns `true` if $el1 \neq el2$ and `IsLessThanUnder`(*ord*,*el1*,*el2*), `IsLessThanUnder`(*ord*,*el2*,*el1*) are both false; and returns `false` otherwise.

4 ▶ `FamilyForOrdering(` *ord* `)` A

for an ordering *ord*, returns the family of elements that the ordering *ord* compares.

5 ▶ `LessThanFunction(` *ord* `)` A

for an ordering *ord*, returns a function f which takes two elements *el1*, *el2* in the `FamilyForOrdering`(*ord*) and returns `true` if *el1* is strictly less than *el2* (with respect to *ord*) and returns `false` otherwise.

6 ▶ `LessThanOrEqualFunction(` *ord* `)` A

for an ordering *ord*, returns a function that takes two elements *el1*, *el2* in the `FamilyForOrdering`(*ord*) and returns `true` if *el1* is less than **or equal to** *el2* (with respect to *ord*) and returns `false` otherwise.

7 ▶ `IsLessThanUnder(` *ord*, *el1*, *el2* `)` O

for an ordering *ord* on the elements of the family of *el1* and *el2*, returns `true` if *el1* is (strictly) less than *el2* with respect to *ord*, and `false` otherwise.

8 ▶ `IsLessThanOrEqualUnder(` *ord*, *el1*, *el2* `)` O

for an ordering *ord* on the elements of the family of *el1* and *el2*, returns `true` if *el1* is less than or equal to *el2* with respect to *ord*, and `false` otherwise.

```
gap> IsLessThanUnder(ord,a,a*b);
true
gap> IsLessThanOrEqualUnder(ord,a*b,a*b);
true
gap> IsIncomparableUnder(ord,a,b);
true
gap> FamilyForOrdering(ord) = FamilyObj(a);
true
```

29.3 Orderings on families of associative words

We now consider orderings on families of associative words.

1 ▶ `IsOrderingOnFamilyOfAssocWords( ` *ord* ` )` P

for an ordering *ord*, returns true if *ord* is an ordering over a family of associative words.

Examples of families of associative words are the families of elements of a free semigroup or a free monoid; these are the two cases that we consider mostly. Associated with those families is an alphabet, which is the semigroup (resp. monoid) generating set of the correspondent free semigroup (resp. free monoid). For definitions of the orderings considered see Sims [Sim94].

2 ▶ `IsTranslationInvariantOrdering( ` *ord* ` )` P

for an ordering *ord* on a family of associative words, returns `true` if and only if the ordering is translation invariant. This is a property of orderings on families of associative words. An ordering *ord* over a family *fam*, with alphabet X is translation invariant if `IsLessThanUnder(`*ord*`, ` u`, ` v`)` implies that for any $a, b \in X^*$ `IsLessThanUnder(`*ord*`, ` $a * u * b, a * v * b$`)`.

3 ▶ `IsReductionOrdering( ` *ord* ` )` P

for an ordering *ord* on a family of associative words, returns `true` if and only if the ordering is a reduction ordering. An ordering *ord* is a reduction ordering if it is founded and translation invariant.

4 ▶ `OrderingOnGenerators( ` *ord* ` )` A

for an ordering *ord* on a family of associative words, returns a list *alphabet* in which the generators are considered. This could be indeed the ordering of the generators in the ordering, but, for example, if a weight is associated to each generator then this is not true anymore. See the example for `WeightLexOrdering` (29.3.8).

5 ▶ `LexicographicOrdering( ` *fam* ` )` O
▶ `LexicographicOrdering( ` *fam*`, ` *gensord* ` )` O
▶ `LexicographicOrdering( ` *fam*`, ` *alphabet* ` )` O
▶ `LexicographicOrdering( ` *f* ` )` O
▶ `LexicographicOrdering( ` *f*`, ` *alphabet* ` )` O
▶ `LexicographicOrdering( ` *f*`, ` *gensord* ` )` O

In the first form, for a family *fam* of associative words, `LexicographicOrdering` returns the lexicographic ordering on the elements of *fam*.

In the second form, for a family *fam* of associate words and a list *alphabet* which is the actual list of generators in the desired order, `LexicographicOrdering` returns the lexicographic ordering on the elements of *fam* with the ordering on the alphabet as given.

In the third form, for a family *fam* of associative words and a list *gensorder* of the length of the alphabet, `LexicographicOrdering` returns the lexicographic ordering on the elements of *fam* with the order on the alphabet given by *gensord*.

In the fourth form, for a free semigroup of a free monoid f, `LexicographicOrdering` returns the lexicographic ordering on the family of the elements of f with the order in the alphabet being the default one.

In the fifth form, for a free semigroup or a free monoid f and a list *alphabet* which is the actual list of generators in the desired order, `LexicographicOrdering` returns the lexicographic ordering on the elements of f with the ordering on the alphabet as given.

In the sixth form, for a free semigroup of a free monoid f, and a list *gensorder*, `LexicographicOrdering` returns the lexicographic ordering on the elements of f with the order on the alphabet given by *gensord*.

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> lex := LexicographicOrdering(f,[2,3,1]);
Ordering
gap> IsLessThanUnder(lex,f.2*f.3,f.3);
true
gap> IsLessThanUnder(lex,f.3,f.2);
false
```

6 ▶ `ShortLexOrdering(` *fam* `)` O
▶ `ShortLexOrdering(` *fam*`,` *alphabet* `)` O
▶ `ShortLexOrdering(` *fam*`,` *gensord* `)` O
▶ `ShortLexOrdering(` *f* `)` O
▶ `ShortLexOrdering(` *f*`,` *alphabet* `)` O
▶ `ShortLexOrdering(` *f*`,` *gensord* `)` O

In the first form, for a family *fam* of associative words, `ShortLexOrdering` returns the ShortLex ordering on the elements of *fam* with the order in the alphabet being the default one.

In the second form, for a family *fam* of associate words and a list *alphabet* which is the actual list of generators in the desired order, `ShortLexOrdering` returns the ShortLex ordering on the elements of *fam* with the ordering on the alphabet as given.

In the third form, for a family *fam* of associative words and a list *gensorder* of the length of the alphabet, `ShortLexOrdering` returns the ShortLex ordering on the elements of *fam* with the order on the alphabet given by *gensord*.

In the fourth form, for a free semigroup of a free monoid f, `ShortLexOrdering` returns the ShortLex ordering on the family of the elements of f with the order in the alphabet being the default one.

In the fifth form, for a free semigroup or a free monoid f and a list *alphabet* which is the actual list of generators in the desired order, `ShortLexOrdering` returns the ShortLex ordering on the elements of f with the ordering on the alphabet as given.

In the sixth form, for a free semigroup of a free monoid f, and a list *gensorder*, `ShortLexOrdering` returns the ShortLex ordering on the elements of f with the order on the alphabet given by *gensord*.

7 ▶ `IsShortLexOrdering(` *ord* `)` P

for an ordering *ord* of a family of associative words, returns `true` if and only if *ord* is a ShortLex ordering.

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> sl := ShortLexOrdering(f,[2,3,1]);
Ordering
gap> IsLessThanUnder(sl,f.1,f.2);
false
gap> IsLessThanUnder(sl,f.3,f.2);
false
gap> IsLessThanUnder(sl,f.3,f.1);
true
```

8 ▶ `WeightLexOrdering(` *fam*`,` *alphabet*`,` *wt* `)` O
▶ `WeightLexOrdering(` *fam*`,` *gensord*`,` *wt* `)` O
▶ `WeightLexOrdering(` *f*`,` *alphabet*`,` *wt* `)` O
▶ `WeightLexOrdering(` *f*`,` *gensord*`,` *wt* `)` O

In the first form, for a family *fam* of associative words and a list *wt*, `WeightLexOrdering` returns the WeightLex ordering on the elements of *fam* with the order in the alphabet being the default one and the weights of the letters in the alphabet being given by *wt*.

In the second form, for a family *fam* of associative words, a list *wt* and a list *gensorder* of the length of the alphabet, `WeightLexOrdering` returns the WeightLex ordering on the elements of *fam* with the order on the alphabet given by *gensord* and the weights of the letters in the alphabet being given by *wt*.

In the third form, for a free semigroup of a free monoid *f* and a list *wt*, `WeightLexOrdering` returns the WeightLex ordering on the family of the elements of *f* with the order in the alphabet being the default one and the weights of the letters in the alphabet being given by *wt*.

In the fourth form, for a free semigroup of a free monoid *f*, a list *wt* and a list *gensorder* of the length of the alphabet, `WeightLexOrdering` returns the WeightLex ordering on the elements of *f* with the order on the alphabet given by *gensord* and the weights of the letters in the alphabet being given by *wt*.

9 ▶ `IsWeightLexOrdering(` *ord* `)` P

for an ordering *ord* on a family of associative words, returns `true` if and only if *ord* is a WeightLex ordering.

10 ▶ `WeightOfGenerators(` *ord* `)` A

for a WeightLex ordering *ord*, returns a list *l* with length the size of the alphabet of the family. This list gives the weight of each of the letters of the alphabet which are used for WeightLex orderings with respect to the ordering given by `OrderingOnGenerators` (see 29.3.4).

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> wtlex := WeightLexOrdering(f,[f.2,f.3,f.1],[3,2,1]);
Ordering
gap> IsLessThanUnder(wtlex,f.1,f.2);
true
gap> IsLessThanUnder(wtlex,f.3,f.2);
true
gap> IsLessThanUnder(wtlex,f.3,f.1);
false
gap> OrderingOnGenerators(wtlex);
[ s2, s3, s1 ]
gap> WeightOfGenerators(wtlex);
[ 3, 2, 1 ]
```

11 ▶ `BasicWreathProductOrdering(` *fam* `)` O
▶ `BasicWreathProductOrdering(` *fam*, *alphabet* `)` O
▶ `BasicWreathProductOrdering(` *fam*, *gensord* `)` O
▶ `BasicWreathProductOrdering(` *f* `)` O
▶ `BasicWreathProductOrdering(` *f*, *alphabet* `)` O
▶ `BasicWreathProductOrdering(` *f*, *gensord* `)` O

In the first form, for a family of associative words, `BasicWreathProductOrdering` returns the basic wreath product ordering on the elements of *fam* with the order in the alphabet being the default one.

In the second form, for a family of associative words and a list *alphabet*, `BasicWreathProductOrdering` returns the basic wreath product ordering on the elements of *fam* with the order on the alphabet given by *alphabet*.

In the third form, for a family of associative words and a list *gensorder* of the length of the alphabet, `BasicWreathProductOrdering` returns the basic wreath product ordering on the elements of *fam* with the order on the alphabet given by *gensord*.

In the fourth form, for a free semigroup of a free monoid *f*, `BasicWreathProductOrdering` returns the basic wreath product ordering on the family of the elements of *f* with the order in the alphabet being the default one.

In the fifth form, for a free semigroup or a free monoid f, and a list *alphabet* of generators, `BasicWreathProductOrdering` returns the basic wreath product ordering on the family of the elements of f with the order on the alphabet given by *alphabet*.

In the sixth form, for a free semigroup or a free monoid f, and a list *gensorder*, `BasicWreathProductOrdering` returns the basic wreath product ordering on the family of the elements of f with the order on the alphabet given by *gensord*.

12 ▶ `IsBasicWreathProductOrdering( ord )` P

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> basic := BasicWreathProductOrdering(f,[2,3,1]);
Ordering
gap> IsLessThanUnder(basic,f.3,f.1);
true
gap> IsLessThanUnder(basic,f.3*f.2,f.1);
true
gap> IsLessThanUnder(basic,f.3*f.2*f.1,f.1*f.3);
false
```

13 ▶ `WreathProductOrdering( fam, levels )` O
▶ `WreathProductOrdering( fam, alphabet, levels )` O
▶ `WreathProductOrdering( fam, gensord, levels )` O
▶ `WreathProductOrdering( f, levels )` O
▶ `WreathProductOrdering( f, alphabet, levels )` O
▶ `WreathProductOrdering( f, gensord, levels )` O

returns the wreath product ordering of the family *fam* of associative words or a free semigroup/monoid f. The ordering on the generators may be omitted (in which case the default one is considered), or may be given either by a list *alphabet* consisting of the alphabet of the family in the appropriate ordering, or by a list *gensord* giving the permutation of the alphabet. It also needs a list *levels* giving the levels of each generator. Notice that this list gives the levels of the generators in the new ordering (not necessarily the default one), i.e. *levels*`[`i`]` is the level of the generator that comes i-th in the ordering of generators given by *alphabet* or *gensord*.

14 ▶ `IsWreathProductOrdering( ord )` P

15 ▶ `LevelsOfGenerators( ord )` A

for a wreath product ordering *ord*, returns the levels of the generators as given at creation (with respect to `OrderingOnGenerators`; see 29.3.4).

```
gap> f := FreeSemigroup(3);
<free semigroup on the generators [ s1, s2, s3 ]>
gap> wrp := WreathProductOrdering(f,[1,2,3],[1,1,2,]);
Ordering
gap> IsLessThanUnder(wrp,f.3,f.1);
false
gap> IsLessThanUnder(wrp,f.3,f.2);
false
gap> IsLessThanUnder(wrp,f.1,f.2);
true
gap> LevelsOfGenerators(wrp);
[ 1, 1, 2 ]
```

30 Domains and their Elements

Domain is GAP's name for structured sets. The ring of Gaussian integers $Z[i]$ is an example of a domain, the group D_{12} of symmetries of a regular hexahedron is another.

The GAP library predefines some domains. For example the ring of Gaussian integers is predefined as `GaussianIntegers` (see 58.5) and the field of rationals is predefined as `Rationals` (see 16). Most domains are constructed by functions, which are called **domain constructors** (see 30.3). For example the group D_{12} is constructed by the construction `Group( (1,2,3,4,5,6), (2,6)(3,5) )` (see 37.2.1) and the finite field with 16 elements is constructed by `GaloisField( 16 )` (see 57.3.1).

The first place where you need domains in GAP is the obvious one. Sometimes you simply want to deal with a domain. For example if you want to compute the size of the group D_{12}, you had better be able to represent this group in a way that the `Size` function can understand.

The second place where you need domains in GAP is when you want to be able to specify that an operation or computation takes place in a certain domain. For example suppose you want to factor 10 in the ring of Gaussian integers. Saying `Factors( 10 )` will not do, because this will return the factorization `[ 2, 5 ]` in the ring of integers. To allow operations and computations to happen in a specific domain, `Factors`, and many other functions as well, accept this domain as optional first argument. Thus `Factors( GaussianIntegers, 10 )` yields the desired result `[ 1+E(4), 1-E(4), 2+E(4), 2-E(4) ]`. (The imaginary unit $\exp(2\pi i/4)$ is written as `E(4)` in GAP.)

The most important facts about domains are stated in Chapter 7 of the GAP Tutorial.

There are only few **operations** especially for domains (see 30.9), operations such as `Intersection` and `Random` are defined for the more general situation of collections (see Chapter 28).

30.1 Operational Structure of Domains

Domains have an **operational structure**, that is, a collection of operations under which the domain is closed. For example, a group is closed under multiplication, taking the zeroth power of elements, and taking inverses of elements. The operational structure may be empty, examples of domains without additional structure are the underlying relations of general mappings (see 31.2).

The operations under which a domain is closed are a subset of the operations that the elements of a domain admit. It is possible that the elements admit more operations. For example, matrices can be multiplied and added. But addition plays no role in a group of matrices, and multiplication plays no role in a vector space of matrices. In particular, a matrix group is not closed under addition.

Note that the elements of a domain exist independently of this domain, usually they existed already before the domain was created. So it makes sense to say that a domain is **generated** by some elements with respect to certain operations.

Of course, different sets of operations yield different notions of generation. For example, the group generated by some matrices is different from the ring generated by these matrices, and these two will in general be different from the vector space generated by the same matrices, over a suitable field.

The other way round, the same set of elements may be obtained by generation w.r.t. different notions of generation. For example, one can get the group generated by two elements g and h also as the monoid

generated by the elements g, g^{-1}, h, h^{-1}; if both g and h have finite order then of course the group generated by g and h coincides with the monoid generated by g and h.

Additionally to the operational structure, a domain can have properties. For example, the multiplication of a group is associative, and the multiplication in a field is commutative.

Note that associativity and commutativity depend on the set of elements for which one considers the multiplication, i.e., it depends on the domain. For example, the multiplication in a full matrix ring over a field is not commutative, whereas its restriction to the set of diagonal matrices is commutative.

One important difference between the operational structure and the properties of a domain is that the operational structure is fixed when the domain is constructed, whereas properties can be discovered later. For example, take a domain whose operational structure is given by closure under multiplication. If it is discovered that the inverses of all its elements also do (by chance) lie in this domain, being closed under taking inverses is **not** added to the operational structure. But a domain with operational structure of multiplication, taking the identity, and taking inverses will be treated as a group as soon as the multiplication is found out to be associative for this domain.

The operational structures available in GAP form a hierarchy, which is explicitly formulated in terms of domain categories, see 30.6.

30.2 Equality and Comparison of Domains

Equality and **comparison** of domains are defined as follows.

Two domains are considered **equal** if and only if the sets of their elements as computed by `AsSSortedList` (see 28.2.9) are equal. Thus, in general `=` behaves as if each domain operand were replaced by its set of elements. Except that `=` will also sometimes, but not always, work for infinite domains, for which of course GAP cannot compute the set of elements. Note that this implies that domains with different algebraic structure may well be equal. As a special case of this, either operand of `=` may also be a proper set (see 21.19), i.e., a sorted list without holes or duplicates (see 28.2.9), and `=` will return `true` if and only if this proper set is equal to the set of elements of the argument that is a domain.

No general **ordering** of arbitrary domains via `<` is defined in GAP 4. This is because a well-defined `<` for domains or, more general, for collections, would have to be compatible with `=` and would need to be transitive and antisymmetric in order to be used to form ordered sets. In particular, `<` would have to be independent of the algebraic structure of its arguments because this holds for `=`, and thus there would be hardly a situation where one could implement an efficient comparison method. (Note that in the case that two domains are comparable with `<`, the result is in general **not** compatible with the set theoretical subset relation, which can be decided with `IsSubset`.)

30.3 Constructing Domains

For several operational structures (see 30.1), GAP provides functions to construct domains with this structure. For example, `Group` returns groups, `VectorSpace` returns vector spaces etc.

1 ► *Struct*`(` *arg1*`,` *arg2*`, ... )` F

The syntax of these functions may vary, dependent on the structure in question. Usually a domain is constructed as the closure of some elements under the given operations, that is, the domain is given by its **generators**. For example, a group can be constructed from a list of generating permutations or matrices or whatever is admissible as group elements, and a vector space over a given field F can be constructed from F and a list of appropriate vectors.

The idea of generation and generators in GAP is that the domain returned by a function such as `Group`, `Algebra`, or `FreeLeftModule` **contains** the given generators. This implies that the generators of a group must know how they are multiplied and inverted, the generators of a module must know how they are added

and how scalar multiplication works, and so on. Thus one cannot use for example permutations as generators of a vector space.

The function *Struct* first checks whether the arguments admit the construction of a domain with the desired structure. This is done by calling the operation

2 ▶ `IsGeneratorsOf`*Struct*`( [`*info*`, ]`*gens*`)` O

where *arglist* is the list of given generators and *info* an argument of *Struct*, for example the field of scalars in the case that a vector space shall be constructed. If the check failed then *Struct* returns `fail`, otherwise it returns the result of *Struct*`ByGenerators` (see below). (So if one wants to omit the check then one should call *Struct*`ByGenerators` directly.)

3 ▶ `GeneratorsOf`*Struct*`(` *D*`)` A

For a domain *D* with operational structure corresponding to *Struct*, the attribute `GeneratorsOf`*Struct* returns a list of corresponding generators of *D*. If these generators were not yet stored in *D* then *D* must know **some** generators if `GeneratorsOf`*Struct* shall have a chance to compute the desired result; for example, monoid generators of a group can be computed from known group generators (and vice versa). Note that several notions of generation may be meaningful for a given domain, so it makes no sense to ask for "the generators of a domain". Further note that the generators may depend on other information about *D*. For example the generators of a vector space depend on the underlying field of scalars; the vector space generators of a vector space over the field with four elements need not generate the same vector space when this is viewed as a space over the field with two elements.

4 ▶ *Struct*`ByGenerators(` `[`*info*`, ]`*gens* `)` O

Domain construction from generators *gens* is implemented by operations *Struct*`ByGenerators`, which are called by the simple functions *Struct*; methods can be installed only for the operations. Note that additional information *info* may be necessary to construct the domain; for example, a vector space needs the underlying field of scalars in addition to the list of vector space generators. The `GeneratorsOf`*Struct* value of the returned domain need **not** be equal to *gens*. But if a domain *D* is printed as *Struct*`([`*a*`, `*b*`, ...])` and if there is an attribute `GeneratorsOf`*Struct* then the list `GeneratorsOf`*Struct*`(` *D* `)` is guaranteed to be equal to `[` *a*`, `*b*`, ... ]`.

5 ▶ *Struct*`WithGenerators(` `[`*info*`, ]`*gens* `)` O

The only difference between *Struct*`ByGenerators` and *Struct*`WithGenerators` is that the latter guarantees that the `GeneratorsOf`*Struct* value of the result is equal to the given generators *gens*.

6 ▶ `Closure`*Struct*`(` *D*`, `*obj* `)` O

For constructing a domain as the closure of a given domain with an element or another domain, one can use the operation `Closure`*Struct*. It returns the smallest domain with operational structure corresponding to *Struct* that contains *D* as a subset and *obj* as an element.

30.4 Changing the Structure

The same set of elements can have different operational structures. For example, it may happen that a monoid *M* does in fact contain the inverses of all of its elements; if *M* has not been constructed as a group (see 30.6) then it is reasonable to ask for the group that is equal to *M*.

1 ▶ `As`*Struct*`(` `[`*info*`, ]`*D* `)` O

If *D* is a domain that is closed under the operational structure given by *Struct* then `As`*Struct* returns a domain *E* that consists of the same elements (that is, *D* `=` *E*) and that has this operational structure (that is, `Is`*Struct*`(` *E* `)` is `true`); if *D* is not closed under the structure given by *Struct* then `As`*Struct* returns `fail`.

If additional information besides generators are necessary to define D then the argument *info* describes the value of this information for the desired domain. For example, if we want to view D as a vector space over the field with two elements then we may call `AsVectorSpace( GF(2), D )`; this allows us to change the underlying field of scalars, for example if D is a vector space over the field with four elements. Again, if D is not equal to a domain with the desired structure and additional information then `fail` is returned.

In the case that no additional information *info* is related to the structure given by *Struct*, the operation `As`*Struct* is in fact an attribute (see 13.5).

See the index of the GAP Reference Manual for an overview of the available `As`*Struct* functions.

30.5 Changing the Representation

Often it is useful to answer questions about a domain via computations in a different but isomorphic domain. In the sense that this approach keeps the structure and changes the underlying set of elements, it can be viewed as a counterpart of keeping the set of elements and changing its structure (see 30.4).

One reason for doing so can be that computations with the elements in the given domain are not very efficient. For example, if one is given a solvable matrix group (see Chapter 42) then one can compute an isomorphism to a polycyclicly presented group G, say (see Chapter 43); the multiplication of two matrices –which is essentially determined by the dimension of the matrices– is much more expensive than the multiplication of two elements in G –which is essentially determined by the composition length of G.

1 ▶ `Isomorphism`*RepStruct*`( D )` A

If D is a domain that is closed under the operational structure given by *Struct* then `Isomorphism`*RepStruct* returns a mapping *hom* from D to a domain E having structure given by *Struct*, such that *hom* respects the structure *Struct* and *Rep* describes the representation of the elements in E. If no domain E with the required properties exists then `fail` is returned.

For example, `IsomorphismPermGroup` (see 41.2.1) takes a group as its argument and returns a group homomorphism (see 38) onto an isomorphic permutation group (see Chapter 41) provided the original group is finite; for infinite groups, `IsomorphismPermGroup` returns `fail`. Similarly, `IsomorphismPcGroup` (see 44.5.2) returns a group homomorphism from its argument to a polycyclicly presented group (see 44) if the argument is polycyclic, and `fail` otherwise.

See the index of the GAP Reference Manual for an overview of the available `Isomorphism`*RepStruct* functions.

30.6 Domain Categories

As mentioned in 30.1, the operational structure of a domain is fixed when the domain is constructed. For example, if D was constructed by `Monoid` then D is in general not regarded as a group in GAP, even if D is in fact closed under taking inverses. In this case, `IsGroup` returns `false` for D. The operational structure determines which operations are applicable for a domain, so for example `SylowSubgroup` is not defined for D and therefore will signal an error.

1 ▶ `Is`*Struct*`( D )`

The functions `Is`*Struct* implement the tests whether a domain D has the respective operational structure (upon construction). `Is`*Struct* is a filter (see 13) that involves certain categories (see 13.3) and usually also certain properties (see 13.7). For example, `IsGroup` is equivalent to `IsMagmaWithInverses and IsAssociative`, the first being a category and the second being a property.

Implications between domain categories describe the hierarchy of operational structures available in GAP. Here are some typical examples.

- `IsDomain` is implied by each domain category,
- `IsMagma` is implied by each category that describes the closure under multiplication `*`,

– `IsAdditiveMagma` is implied by each category that describes the closure under addition `+`,

– `IsMagmaWithOne` implies `IsMagma`; a **magma-with-one** is a magma such that each element (and thus also the magma itself) can be asked for its zeroth power,

– `IsMagmaWithInverses` implies `IsMagmaWithOne`; a **magma-with-inverses** is a magma such that each element can be asked for its inverse; important special cases are **groups**, which in addition are associative,

– a **ring** is a magma that is also an additive group,

– a **ring-with-one** is a ring that is also a magma-with-one,

– a **division ring** is a ring-with-one that is also closed under taking inverses of nonzero elements,

– a **field** is a commutative division ring.

Each operational structure *Struct* has associated with it a domain category `Is`*Struct*, and operations *Struct*`ByGenerators` for constructing a domain from generators, `GeneratorsOf`*Struct* for storing and accessing generators w.r.t. this structure, `Closure`*Struct* for forming the closure, and `As`*Struct* for getting a domain with the desired structure from one with weaker operational structure and for testing whether a given domain can be regarded as a domain with *Struct*.

The functions applicable to domains with the various structures are described in the corresponding chapters of the Reference Manual. For example, functions for rings, fields, groups, and vector spaces are described in Chapters 54, 56, 37, and 59, respectively. More general functions for arbitrary collections can be found in Chapter 28.

30.7 Parents

1 ▶ `Parent(` *D* `)` F
▶ `SetParent(` *D*`,` *P* `)` O
▶ `HasParent(` *D* `)` F

It is possible to assign to a domain D one other domain P containing D as a subset, in order to exploit this subset relation between D and P. Note that P need not have the same operational structure as D, for example P may be a magma and D a field.

The assignment is done by calling `SetParent`, and P is called the **parent** of D. If D has already a parent, calls to `SetParent` will be ignored.

If D has a parent P –this can be checked with `HasParent`– then P can be used to gain information about D. First, the call of `SetParent` causes `UseSubsetRelation` (see 30.13.1) to be called. Second, for a domain D with parent, information relative to the parent can be stored in D; for example, there is an attribute `NormalizerInParent` for storing `Normalizer(` P`,` D `)` in the case that D is a group. (More about such parent dependent attributes can be found in 6.2 in "Extending GAP".) Note that because of this relative information, one cannot change the parent; that is, one can set the parent only once, subsequent calls to `SetParent` for the same domain D are ignored. Further note that contrary to `UseSubsetRelation` (see 30.13.1), also knowledge about the parent P might be used that is discovered after the `SetParent` call.

A stored parent can be accessed using `Parent`. If D has no parent then `Parent` returns D itself, and `HasParent` will returns `false` also after a call to `Parent`. So `Parent` is **not** an attribute, the underlying attribute to store the parent is `ParentAttr`.

Certain functions that return domains with parent already set, for example `Subgroup`, are described in Section 30.8. Whenever a function has this property, the Reference Manual states this explicitly. Note that these functions **do not guarantee** a certain parent, for example `DerivedSubgroup` (see 37.12.3) for a perfect group G may return G itself, and if G had already a parent then this is not replaced by G. As a rule of thumb, GAP avoids to set a domain as its own parent, which is consistent with the behaviour of `Parent`, at least until a parent is set explicitly with `SetParent`.

```
gap> g:= Group( (1,2,3), (1,2) );; h:= Group( (1,2) );;
gap> HasParent( g );  HasParent( h );
false
false
gap> SetParent( h, g );
gap> Parent( g );  Parent( h );
Group([ (1,2,3), (1,2) ])
Group([ (1,2,3), (1,2) ])
gap> HasParent( g );  HasParent( h );
false
true
```

30.8 Constructing Subdomains

For many domains D, there are functions that construct certain subsets S of D as domains with parent (see 30.7) already set to D. For example, if G is a group that contains the elements in the list *gens* then `Subgroup( `G`, `*gens*` )` returns a group S that is generated by the elements in *gens* and with `Parent( `S` ) = `G.

1 ▶ `Sub`*struct*`( `D`, `*gens*` )` F

More general, if D is a domain whose algebraic structure is given by the function *Struct* (for example `Group`, `Algebra`, `Field`) then the function `Sub`*struct* (for example `Subgroup`, `Subalgebra`, `Subfield`) returns domains with structure *Struct* and parent set to the first argument.

2 ▶ `Sub`*struct*`NC( `D`, `*gens*` )` F

Each function `Sub`*struct* checks that the *Struct* generated by *gens* is in fact a subset of D. If one wants to omit this check then one can call `Sub`*struct*`NC` instead; the suffix `NC` stands for "no check".

3 ▶ `AsSub`*struct*`( `D`, `S` )` F

first constructs `As`*struct*`( [`*info*`, ]`S` )`, where *info* depends on D and S, and then sets the parent (see 30.7) of this new domain to D.

4 ▶ `IsSub`*struct*`( `D`, `S` )` F

There is no real need for functions that check whether a domain S is a `Sub`*struct* of a domain D, since this is equivalent to the checks whether S is a *Struct* and S is a subset of D. Note that in many cases, only the subset relation is what one really wants to check, and that appropriate methods for the operation `IsSubset` (see 28.4.1) are available for many special situations, such as the test whether a group is contained in another group, where only generators need to be checked.

If a function `IsSub`*struct* is available in GAP then it is implemented as first a call to `Is`*Struct* for the second argument and then a call to `IsSubset` for the two arguments.

30.9 Operations for Domains

For the meaning of the attributes `Characteristic`, `One`, `Zero` in the case of a domain argument, see 30.10.

1 ▶ `IsGeneralizedDomain( `D` )` C
▶ `IsDomain( `D` )` C

For some purposes, it is useful to deal with objects that are similar to domains but that are not collections in the sense of GAP because their elements may lie in different families; such objects are called **generalized domains**. An instance of generalized domains are "operation domains", for example any G-set for a

permutation group G consisting of some union of points, sets of points, sets of sets of points etc., under a suitable action.

`IsDomain` is a synonym for `IsGeneralizedDomain and IsCollection`.

2 ▶ `GeneratorsOfDomain( D )` A

For a domain D, `GeneratorsOfDomain` returns a list containing all elements of D, perhaps with repetitions. Note that if the domain D shall be generated by a list of some elements w.r.t. the empty operational structure (see 30.1), the only possible choice of elements is to take all elements of D. See 30.3 and 30.4 for the concepts of other notions of generation.

3 ▶ `Domain( [`*Fam*`, ]`*generators*` )` F
▶ `DomainByGenerators(` *Fam*`,` *generators*` )` O

`Domain` returns the domain consisting of the elements in the homogeneous list *generators*. If *generators* is empty then a family *Fam* must be entered as first argument, and the returned (empty) domain lies in the collections family of *Fam*.

`DomainByGenerators` is the operation called by `Domain`.

30.10 Attributes and Properties of Elements

The following attributes and properties for elements and domains correspond to the operational structure.

1 ▶ `Characteristic(` *obj* `)` A

`Characteristic` returns the **characteristic** of *obj*, where *obj* must either be an additive element, a domain or a family.

For a domain D, the characteristic is defined if D is closed under addition and has a zero element z `= Zero(` D `)` (see 30.10.3); in this case, `Characteristic(` D `)` is the smallest positive integer p such that p `*` x `=` z for all elements x in D, if such an integer exists, and the integer zero `0` otherwise.

If a family has a characteristic then this means that all domains of elements in this family have this characteristic. In this case, also each element in the family has this characteristic. (Note that also the zero element z of a finite field in characteristic p has characteristic p, although $n * z = z$ for any integer n.)

2 ▶ `OneImmutable(` *obj* `)` A
▶ `OneAttr(` *obj* `)` AM
▶ `One(` *obj* `)` AM
▶ `Identity(` *obj* `)` AM
▶ `OneMutable(` *obj* `)` O
▶ `OneOp(` *obj* `)` O
▶ `OneSameMutability(` *obj* `)` O
▶ `OneSM(` *obj* `)` O

`OneImmutable`, `OneMutable`, and `OneSameMutability` return the multiplicative neutral element of the multiplicative element *obj*.

They differ only w.r.t. the mutability of the result. `OneImmutable` is an attribute and hence returns an immutable result. `OneMutable` is guaranteed to return a new **mutable** object whenever a mutable version of the required element exists in GAP (see 12.6.1). `OneSameMutability` returns a result that is mutable if *obj* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

If *obj* is a multiplicative element then `OneSameMutability(` *obj* `)` is equivalent to *obj*`^0`.

`OneAttr`, `One` and `Identity` are synonyms of `OneImmutable`. `OneSM` is a synonym of `OneSameMutability`. `OneOp` is a synonym of `OneMutable`.

If *obj* is a domain or a family then `One` is defined as the identity element of all elements in *obj*, provided that all these elements have the same identity. For example, the family of all cyclotomics has the identity element `1`, but a collections family (see 28.1.1) may contain matrices of all dimensions and then it cannot have a unique identity element. Note that `One` is applicable to a domain only if it is a magma-with-one (see 33.1.2); use `MultiplicativeNeutralElement` (see 33.4.10) otherwise.

The identity of an object need not be distinct from its zero, so for example a ring consisting of a single element can be regarded as a ring-with-one (see 54). This is particularly useful in the case of finitely presented algebras, where any factor of a free algebra-with-one is again an algebra-with-one, no matter whether or not it is a zero algebra.

The default method of `One` for multiplicative elements calls `OneMutable` (note that methods for `OneMutable` must **not** delegate to `One`); so other methods to compute identity elements need to be installed only for `OneOp` and (in the case of copyable objects) `OneSameMutability`.

For domains, `One` may call `Representative` (see 28.3.7), but `Representative` is allowed to fetch the identity of a domain D only if `HasOne( D )` is `true`.

3 ► `ZeroImmutable( obj )` A
► `ZeroAttr( obj )` AM
► `Zero( obj )` AM
► `ZeroMutable( obj )` O
► `ZeroOp( obj )` O
► `ZeroSameMutability( obj )` O
► `ZeroSM( obj )` O

`ZeroImmutable`, `ZeroMutable`, and `ZeroSameMutability` all return the additive neutral element of the additive element *obj*.

They differ only w.r.t. the mutability of the result. `ZeroImmutable` is an attribute and hence returns an immutable result. `ZeroMutable` is guaranteed to return a new **mutable** object whenever a mutable version of the required element exists in GAP (see 12.6.1). `ZeroSameMutability` returns a result that is mutable if *obj* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

`ZeroSameMutability( obj )` is equivalent to `0 * obj`.

`ZeroAttr` and `Zero` are synonyms of `ZeroImmutable`. `ZeroSM` is a synonym of `ZeroSameMutability`. `ZeroOp` is a synonym of `ZeroMutable`.

If *obj* is a domain or a family then `Zero` is defined as the zero element of all elements in *obj*, provided that all these elements have the same zero. For example, the family of all cyclotomics has the zero element `0`, but a collections family (see 28.1.1) may contain matrices of all dimensions and then it cannot have a unique zero element. Note that `Zero` is applicable to a domain only if it is an additive magma-with-zero (see 53.1.5); use `AdditiveNeutralElement` (see 53.3.5) otherwise.

The default method of `Zero` for additive elements calls `ZeroMutable` (note that methods for `ZeroMutable` must **not** delegate to `Zero`); so other methods to compute zero elements need to be installed only for `ZeroMutable` and (in the case of copyable objects) `ZeroSameMutability`.

For domains, `Zero` may call `Representative` (see 28.3.7), but `Representative` is allowed to fetch the zero of a domain D only if `HasZero( D )` is `true`.

4 ► `MultiplicativeZeroOp( elt )` O

returns the element z in the family F of *elt* with the property that $z * m = z = m * z$ holds for all $m \in F$, if such an element is known.

Families of elements in the category IsMultiplicativeElementWithZero often arise from adjoining a new zero to an existing magma. See 33.2.12 for details.

5 ▶ `IsOne( elm )` P

is `true` if *elm* `= One(` *elm* `)`, and `false` otherwise.

6 ▶ `IsZero( elm )` P

is `true` if *elm* `= Zero(` *elm* `)`, and `false` otherwise.

7 ▶ `IsIdempotent( elt )` P

true iff *elt* is its own square. (Even if IsZero(*elt*) is also true.)

8 ▶ `InverseImmutable( elm )` A
▶ `InverseAttr( elm )` AM
▶ `Inverse( elm )` AM
▶ `InverseMutable( elm )` O
▶ `InverseOp( elm )` O
▶ `InverseSameMutability( elm )` O
▶ `InverseSM( elm )` O

`InverseImmutable`, `InverseMutable`, and `InverseSameMutability` all return the multiplicative inverse of an element *elm*, that is, an element *inv* such that *elm* `*` *inv* `=` *inv* `*` *elm* `= One(` *elm* `)` holds; if *elm* is not invertible then `fail` (see 20.1.1) is returned.

Note that the above definition implies that a (general) mapping is invertible in the sense of `Inverse` only if its source equals its range (see 31.13). For a bijective mapping f whose source and range differ, `InverseGeneralMapping` (see 31.1.3) can be used to construct a mapping g with the property that $f*g$ is the identity mapping on the source of f and $g*f$ is the identity mapping on the range of f.

The operations differ only w.r.t. the mutability of the result. `InverseImmutable` is an attribute and hence returns an immutable result. `InverseMutable` is guaranteed to return a new **mutable** object whenever a mutable version of the required element exists in GAP. `InverseSameMutability` returns a result that is mutable if *elm* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

`InverseSameMutability(` *elm* `)` is equivalent to *elm*`^-1`.

`InverseAttr` and `Inverse` are synonyms of `InverseImmutable`. `InverseSM` is a synonym of `InverseSameMutability`. `InverseOp` is a synonym of `InverseMutable`.

The default method of `InverseImmutable` calls `InverseMutable` (note that methods for `InverseMutable` must **not** delegate to `InverseImmutable`); other methods to compute inverses need to be installed only for `InverseMutable` and (in the case of copyable objects) `InverseSameMutability`.

9 ▶ `AdditiveInverseImmutable( elm )` A
▶ `AdditiveInverseAttr( elm )` AM
▶ `AdditiveInverse( elm )` AM
▶ `AdditiveInverseMutable( elm )` O
▶ `AdditiveInverseOp( elm )` O
▶ `AdditiveInverseSameMutability( elm )` O
▶ `AdditiveInverseSM( elm )` O

`AdditiveInverseImmutable`, `AdditiveInverseMutable`, and `AdditiveInverseSameMutability` all return the additive inverse of *elm*.

They differ only w.r.t. the mutability of the result. `AdditiveInverseImmutable` is an attribute and hence returns an immutable result. `AdditiveInverseMutable` is guaranteed to return a new **mutable** object whenever a mutable version of the required element exists in GAP (see 12.6.1). `AdditiveInverseSameMutability` returns a result that is mutable if *elm* is mutable and if a mutable version of the required element exists in GAP; for lists, it returns a result of the same immutability level as the argument. For instance, if the argument is a mutable matrix with immutable rows, it returns a similar object.

`AdditiveInverseSameMutability(` *elm* `)` is equivalent to `-`*elm*.

`AdditiveInverseAttr` and `AdditiveInverse` are synonyms of `AdditiveInverseImmutable`. `AdditiveInverseSM` is a synonym of `AdditiveInverseSameMutability`. `AdditiveInverseOp` is a synonym of `AdditiveInverseMutable`.

The default method of `AdditiveInverse` calls `AdditiveInverseMutable` (note that methods for `AdditiveInverseMutable` must **not** delegate to `AdditiveInverse`); so other methods to compute additive inverses need to be installed only for `AdditiveInverseMutable` and (in the case of copyable objects) `AdditiveInverseSameMutability`.

10 ▶ `Order(` *elm* `)` A

is the multiplicative order of *elm*. This is the smallest positive integer n such that *elm*`^`n `= One(` *elm* `)` if such an integer exists. If the order is infinite, `Order` may return the value `infinity`, but it also might run into an infinite loop trying to test the order.

30.11 Comparison Operations for Elements

Binary comparison operations have been introduced already in 4.11. The underlying operations for which methods can be installed are the following.

1 ▶ `\=(` *left-expr*`,` *right-expr* `)` O
▶ `\<(` *left-expr*`,` *right-expr* `)` O

Note that the comparisons via `<>`, `<=`, `>`, and `>=` are delegated to the operations `\=` and `\<`.

In general, objects in **different** families cannot be compared with `<`. For the reason and for exceptions from this rule, see 4.11.

For some objects a "normal form" is hard to compute and thus equality of elements of a domain might be expensive to test. Therefore GAP provides a (slightly technical) property with which an algorithm can test whether an efficient equality test is available for elements of a certain kind.

2 ▶ `CanEasilyCompareElements(` *obj* `)` P
▶ `CanEasilyCompareElementsFamily(` *fam* `)` F
▶ `CanEasilySortElements(` *obj* `)` P
▶ `CanEasilySortElementsFamily(` *fam* `)` F

`CanEasilyCompareElements` indicates whether the elements in the family *fam* of *obj* can be easily compared with `=`. (In some cases element comparisons are very hard, for example in cases where no normal forms for the elements exist.)

The default method for this property is to ask the family of *obj*, the default method for the family is to return `false`.

The ability to compare elements may depend on the successful computation of certain information. (For example for finitely presented groups it might depend on the knowledge of a faithful permutation representation.) This information might change over time and thus it might not be a good idea to store a value `false` too early in a family. Instead the function `CanEasilyCompareElementsFamily` should be called for the family of *obj* which returns `false` if the value of `CanEasilyCompareElements` is not known for the family without computing it. (This is in fact what the above mentioned family dispatch does.)

If a family knows ab initio that it can compare elements this property should be set as implied filter **and** filter for the family (the 3rd and 4th argument of `NewFamily` respectively). This guarantees that code which directly asks the family gets a right answer.

The property `CanEasilySortElements` and the function `CanEasilySortElementsFamily` behave exactly in the same way, except that they indicate that objects can be compared via `<`. This property implies `CanEasilyCompareElements`, as the ordering must be total.

30.12 Arithmetic Operations for Elements

Binary arithmetic operations have been introduced already in 4.12. The underlying operations for which methods can be installed are the following.

1 ▶ `\+(` *left-expr*`,` *right-expr* `)` O
▶ `\*(` *left-expr*`,` *right-expr* `)` O
▶ `\/(` *left-expr*`,` *right-expr* `)` O
▶ `\^(` *left-expr*`,` *right-expr* `)` O
▶ `\mod(` *left-expr*`,` *right-expr* `)` O

For details about special methods for `\mod`, consult the index entries for "mod".

2 ▶ `LeftQuotient(` *elm1*`,` *elm2* `)` O

returns the product *elm1*`^(-1) *` *elm2*. For some types of objects (for example permutations) this product can be evaluated more efficiently than by first inverting *elm1* and then forming the product with *elm2*.

3 ▶ `Comm(` *elm1*`,` *elm2* `)` O

returns the **commutator** of *elm1* and *elm2*. The commutator is defined as the product $elm1^{-1} * elm2^{-1} * elm1 * elm2$.

```
gap> a:= (1,3)(4,6);; b:= (1,6,5,4,3,2);;
gap> Comm( a, b );
(1,5,3)(2,6,4)
gap> LeftQuotient( a, b );
(1,2)(3,6)(4,5)
```

4 ▶ `LieBracket(` *elm1*`,` *elm2* `)` O

returns the element *elm1* `*` *elm2* `-` *elm2* `*` *elm1*.

The addition `\+` is assumed to be associative but **not** assumed to be commutative (see 53.3.1). The multiplication `\*` is **not** assumed to be commutative or associative (see 33.4.9, 33.4.7).

5 ▶ `Sqrt(` *obj* `)` O

`Sqrt` returns a square root of *obj*, that is, an object x with the property that $x \cdot x = obj$ holds. If such an x is not unique then the choice of x depends on the type of *obj*. For example, `ER` (see 18.4.2) is the `Sqrt` method for rationals (see 16.1.1).

30.13 Relations Between Domains

Domains are often constructed relative to other domains. The probably most usual case is to form a **subset** of a domain, for example the intersection (see 28.4.2) of two domains, or a Sylow subgroup of a given group (see 37.13.1).

In such a situation, the new domain can gain knowledge by exploiting that several attributes are maintained under taking subsets. For example, the intersection of an arbitrary domain with a finite domain is clearly finite, a Sylow subgroup of an abelian group is abelian, too, and so on.

Since usually the new domain has access to the knowledge of the old domain(s) only when it is created (see 30.8 for the exception), this is the right moment to take advantage of the subset relation.

Analogous relations occur when a **factor structure** is created from a domain and a subset, and when a domain **isomorphic** to a given one is created.

1 ► `UseSubsetRelation(` *super*`,` *sub* `)` O

Methods for this operation transfer possibly useful information from the domain *super* to its subset *sub*, and vice versa.

`UseSubsetRelation` is designed to be called automatically whenever substructures of domains are constructed. So the methods must be **cheap**, and the requirements should be as sharp as possible!

To achieve that **all** applicable methods are executed, all methods for this operation except the default method must end with `TryNextMethod()`. This default method deals with the information that is available by the calls of `InstallSubsetMaintenance` in the GAP library.

```
gap> g:= Group( (1,2), (3,4), (5,6) );; h:= Group( (1,2), (3,4) );;
gap> IsAbelian( g );  HasIsAbelian( h );
true
false
gap> UseSubsetRelation( g, h );;  HasIsAbelian( h );  IsAbelian( h );
true
true
```

2 ► `UseIsomorphismRelation(` *old*`,` *new* `)` O

Methods for this operation transfer possibly useful information from the domain *old* to the isomorphic domain *new*.

`UseIsomorphismRelation` is designed to be called automatically whenever isomorphic structures of domains are constructed. So the methods must be **cheap**, and the requirements should be as sharp as possible!

To achieve that **all** applicable methods are executed, all methods for this operation except the default method must end with `TryNextMethod()`. This default method deals with the information that is available by the calls of `InstallIsomorphismMaintenance` in the GAP library.

```
gap> g:= Group( (1,2) );;  h:= Group( [ [ -1 ] ] );;
gap> Size( g );  HasSize( h );
2
false
gap> UseIsomorphismRelation( g, h );;  HasSize( h );  Size( h );
true
2
```

3 ► `UseFactorRelation(` *numer*`,` *denom*`,` *factor* `)` O

Methods for this operation transfer possibly useful information from the domain *numer* or its subset *denom* to the domain *factor* that is isomorphic to the factor of *numer* by *denom*, and vice versa. *denom* may be

`fail`, for example if *factor* is just known to be a factor of *numer* but *denom* is not available as a GAP object; in this case those factor relations are used that are installed without special requirements for *denom*.

`UseFactorRelation` is designed to be called automatically whenever factor structures of domains are constructed. So the methods must be **cheap**, and the requirements should be as sharp as possible!

To achieve that **all** applicable methods are executed, all methods for this operation except the default method must end with `TryNextMethod()`. This default method deals with the information that is available by the calls of `InstallFactorMaintenance` in the GAP library.

```
gap> g:= Group( (1,2,3,4), (1,2) );; h:= Group( (1,2,3), (1,2) );;
gap> IsSolvableGroup( g );  HasIsSolvableGroup( h );
true
false
gap> UseFactorRelation( g, Subgroup( g, [ (1,2)(3,4), (1,3)(2,4) ] ), h );;
gap> HasIsSolvableGroup( h );  IsSolvableGroup( h );
true
true
```

The following functions are used to tell GAP under what conditions an attribute is maintained under taking subsets, or forming factor structures or isomorphic domains. This is used only when a new attribute is created, see 3.3 in "Programming in GAP". For the attributes already available, such as `IsFinite` and `IsCommutative`, the maintenances are already notified.

4 ▶ `InstallSubsetMaintenance( ` *opr*`, ` *super_req*`, ` *sub_req* ` )` F

opr must be a property or an attribute. The call of `InstallSubsetMaintenance` has the effect that for a domain D in the filter *super_req*, and a domain S in the filter *sub_req*, the call `UseSubsetRelation(` D`, ` S ` )` (see 30.13.1) sets a known value of *opr* for D as value of *opr* also for S. A typical example for which `InstallSubsetMaintenance` is applied is given by *opr* `= IsFinite`, *super_req* `= IsCollection and IsFinite`, and *sub_req* `= IsCollection`.

If *opr* is a property and the filter *super_req* lies in the filter *opr* then we can use also the following inverse implication. If D is in the filter whose intersection with *opr* is *super_req* and if S is in the filter *sub_req*, S is a subset of D, and the value of *opr* for S is `false` then the value of *opr* for D is also `false`.

5 ▶ `InstallIsomorphismMaintenance( ` *opr*`, ` *old_req*`, ` *new_req* ` )` F

opr must be a property or an attribute. The call of `InstallIsomorphismMaintenance` has the effect that for a domain D in the filter *old_req*, and a domain E in the filter *new_req*, the call `UseIsomorphismRelation(` D`, ` E ` )` (see 30.13.2) sets a known value of *opr* for D as value of *opr* also for E. A typical example for which `InstallIsomorphismMaintenance` is applied is given by *opr* `= Size`, *old_req* `= IsCollection`, and *new_req* `= IsCollection`.

6 ▶ `InstallFactorMaintenance( ` *opr*`, ` *numer_req*`, ` *denom_req*`, ` *factor_req* ` )` F

opr must be a property or an attribute. The call of `InstallFactorMaintenance` has the effect that for collections N, D, F in the filters *numer_req*, *denom_req*, and *factor_req*, respectively, the call `UseFactorRelation(` N`, ` D`, ` F ` )` (see 30.13.3) sets a known value of *opr* for N as value of *opr* also for F. A typical example for which `InstallFactorMaintenance` is applied is given by *opr* `= IsFinite`, *numer_req* `= IsCollection and IsFinite`, *denom_req* `= IsCollection`, and *factor_req* `= IsCollection`.

For the other direction, if *numer_req* involves the filter *opr* then a known `false` value of *opr* for F implies a `false` value for D provided that D lies in the filter obtained from *numer_req* by removing *opr*.

Note that an implication of a factor relation holds in particular for the case of isomorphisms. So one need **not** install an isomorphism maintained method when a factor maintained method is already installed. For example, `UseIsomorphismRelation` (see 30.13.2) will transfer a known `IsFinite` value because of the installed factor maintained method.

30.14 Useful Categories of Elements

This section and the following one are rather technical, and may be interesting only for those GAP users who want to implement new kinds of elements.

It deals with certain categories of elements that are useful mainly for the design of elements, from the viewpoint that one wants to form certain domains of these elements. For example, a domain closed under multiplication `*` (a so-called magma, see Chapter 33) makes sense only if its elements can be multiplied, and the latter is indicated by the category `IsMultiplicativeElement` for each element. Again note that the underlying idea is that a domain is regarded as **generated** by given elements, and that these elements carry information about the desired domain. For general information on categories and their hierarchies, see 13.3.

1 ► `IsExtAElement(` *obj* `)` C

An **external additive element** is an object that can be added via `+` with other elements (not necessarily in the same family, see 13.1).

2 ► `IsNearAdditiveElement(` *obj* `)` C

A **near-additive element** is an object that can be added via `+` with elements in its family (see 13.1); this addition is not necessarily commutative.

3 ► `IsAdditiveElement(` *obj* `)` C

An **additive element** is an object that can be added via `+` with elements in its family (see 13.1); this addition is commutative.

4 ► `IsNearAdditiveElementWithZero(` *obj* `)` C

A **near-additive element-with-zero** is an object that can be added via `+` with elements in its family (see 13.1), and that is an admissible argument for the operation `Zero` (see 30.10.3); this addition is not necessarily commutative.

5 ► `IsAdditiveElementWithZero(` *obj* `)` C

An **additive element-with-zero** is an object that can be added via `+` with elements in its family (see 13.1), and that is an admissible argument for the operation `Zero` (see 30.10.3); this addition is commutative.

6 ► `IsNearAdditiveElementWithInverse(` *obj* `)` C

A **near-additive element-with-inverse** is an object that can be added via `+` with elements in its family (see 13.1), and that is an admissible argument for the operations `Zero` (see 30.10.3) and `AdditiveInverse` (see 30.10.9); this addition is not necessarily commutative.

7 ► `IsAdditiveElementWithInverse(` *obj* `)` C

An **additive element-with-inverse** is an object that can be added via `+` with elements in its family (see 13.1), and that is an admissible argument for the operations `Zero` (see 30.10.3) and `AdditiveInverse` (see 30.10.9); this addition is commutative.

8 ► `IsExtLElement(` *obj* `)` C

An **external left element** is an object that can be multiplied from the left, via `*`, with other elements (not necessarily in the same family, see 13.1).

9 ► `IsExtRElement(` *obj* `)` C

An **external right element** is an object that can be multiplied from the right, via `*`, with other elements (not necessarily in the same family, see 13.1).

10 ► `IsMultiplicativeElement(` *obj* `)` C

A **multiplicative element** is an object that can be multiplied via `*` with elements in its family (see 13.1).

11 ▶ `IsMultiplicativeElementWithOne( obj )` C

A **multiplicative element-with-one** is an object that can be multiplied via `*` with elements in its family (see 13.1), and that is an admissible argument for the operation `One` (see 30.10.2).

12 ▶ `IsMultiplicativeElementWithZero( elt )` C

Elements in a family which can be the operands of the `*` and the operation MultiplicativeZero.

13 ▶ `IsMultiplicativeElementWithInverse( obj )` C

A **multiplicative element-with-inverse** is an object that can be multiplied via `*` with elements in its family (see 13.1), and that is an admissible argument for the operations `One` (see 30.10.2) and `Inverse` (see 30.10.8). (Note the word "admissible": an object in this category does not necessarily have an inverse, `Inverse` may return `fail`.)

14 ▶ `IsVector( obj )` C

A **vector** is an additive-element-with-inverse that can be multiplied from the left and right with other objects (not necessarily of the same type). Examples are cyclotomics, finite field elements, and of course row vectors (see below).

Note that not all lists of ring elements are regarded as vectors, for example lists of matrices are not vectors. This is because although the category `IsAdditiveElementWithInverse` is implied by the join of its collections category and `IsList`, the family of a list entry may not imply `IsAdditiveElementWithInverse` for all its elements.

15 ▶ `IsNearRingElement( obj )` C

`IsNearRingElement` is just a synonym for the join of `IsNearAdditiveElementWithInverse` and `IsMultiplicativeElement`.

16 ▶ `IsRingElement( obj )` C

`IsRingElement` is just a synonym for the join of `IsAdditiveElementWithInverse` and `IsMultiplicativeElement`.

17 ▶ `IsNearRingElementWithOne( obj )` C

`IsNearRingElementWithOne` is just a synonym for the join of `IsNearAdditiveElementWithInverse` and `IsMultiplicativeElementWithOne`.

18 ▶ `IsRingElementWithOne( obj )` C

`IsRingElementWithOne` is just a synonym for the join of `IsAdditiveElementWithInverse` and `IsMultiplicativeElementWithOne`.

19 ▶ `IsNearRingElementWithInverse( obj )` C

20 ▶ `IsRingElementWithInverse( obj )` C
▶ `IsScalar( obj )` C

`IsRingElementWithInverse` and `IsScalar` are just synonyms for the join of `IsAdditiveElementWithInverse` and `IsMultiplicativeElementWithInverse`.

More special categories of this kind are described in the contexts where they arise, they are `IsRowVector` (see 23), `IsMatrix` (see 24.1.1), `IsOrdinaryMatrix` (see 24.1.2), and `IsLieMatrix` (see 24.1.3).

30.15 Useful Categories for all Elements of a Family

The following categories of elements are to be understood mainly as categories for all objects in a family, they are usually used as third argument of `NewFamily` (see 3.6 in "Programming in GAP"). The purpose of each of the following categories is then to guarantee that each collection of its elements automatically lies in its collections category (see 28.1.4).

For example, the multiplication of permutations is associative, and it is stored in the family of permutations that each permutation lies in `IsAssociativeElement`. As a consequence, each magma consisting of permutations (more precisely: each collection that lies in the family `CollectionsFamily( PermutationsFamily )`, see 28.1.1) automatically lies in `CategoryCollections( IsAssociativeElement )`. A magma in this category is always known to be associative, via a logical implication (see 2.7 in "Programming in GAP").

Similarly, if a family knows that all its elements are in the categories `IsJacobianElement` and `IsZeroSquaredElement`, then each algebra of these elements is automatically known to be a Lie algebra (see 60).

1 ▶ `IsAssociativeElement(` *obj* `)` C
▶ `IsAssociativeElementCollection(` *obj* `)` C
▶ `IsAssociativeElementCollColl(` *obj* `)` C

An element *obj* in the category `IsAssociativeElement` knows that the multiplication of any elements in the family of *obj* is associative. For example, all permutations lie in this category, as well as those ordinary matrices (see 24.1.2) whose entries are also in `IsAssociativeElement`.

2 ▶ `IsAdditivelyCommutativeElement(` *obj* `)` C
▶ `IsAdditivelyCommutativeElementCollection(` *obj* `)` C
▶ `IsAdditivelyCommutativeElementCollColl(` *obj* `)` C
▶ `IsAdditivelyCommutativeElementFamily(` *obj* `)` C

An element *obj* in the category `IsAdditivelyCommutativeElement` knows that the addition of any elements in the family of *obj* is commutative. For example, each finite field element and each rational number lies in this category.

3 ▶ `IsCommutativeElement(` *obj* `)` C
▶ `IsCommutativeElementCollection(` *obj* `)` C
▶ `IsCommutativeElementCollColl(` *obj* `)` C

An element *obj* in the category `IsCommutativeElement` knows that the multiplication of any elements in the family of *obj* is commutative. For example, each finite field element and each rational number lies in this category.

4 ▶ `IsFiniteOrderElement(` *obj* `)` C
▶ `IsFiniteOrderElementCollection(` *obj* `)` C
▶ `IsFiniteOrderElementCollColl(` *obj* `)` C

An element *obj* in the category `IsFiniteOrderElement` knows that it has finite multiplicative order. For example, each finite field element and each permutation lies in this category. However the value may be `false` even if *obj* has finite order, but if this was not known when *obj* was constructed.

Although it is legal to set this filter for any object with finite order, this is really useful only in the case that all elements of a family are known to have finite order.

5 ▶ `IsJacobianElement(` *obj* `)` C
▶ `IsJacobianElementCollection(` *obj* `)` C
▶ `IsJacobianElementCollColl(` *obj* `)` C

An element *obj* in the category `IsJacobianElement` knows that the multiplication of any elements in the family F of *obj* satisfies the Jacobi identity, that is, $x*y*z+z*x*y+y*z*x$ is zero for all x, y, z in F.

For example, each Lie matrix (see 24.1.3) lies in this category.

6 ▶ `IsZeroSquaredElement( obj )` C
▶ `IsZeroSquaredElementCollection( obj )` C
▶ `IsZeroSquaredElementCollColl( obj )` C

An element *obj* in the category `IsZeroSquaredElement` knows that *obj*`^2 = Zero( obj )`. For example, each Lie matrix (see 24.1.3) lies in this category.

Although it is legal to set this filter for any zero squared object, this is really useful only in the case that all elements of a family are known to have square zero.

31 Mappings

A **mapping** in GAP is what is called a "function" in mathematics. GAP also implements **generalized mappings** in which one element might have several images, these can be imagined as subsets of the cartesian product and are often called "relations".

Most operations are declared for general mappings and therefore this manual often refers to "(general) mappings", unless you deliberately need the generalization you can ignore the "general" bit and just read it as "mappings".

A **general mapping** F in GAP is described by its source S, its range R, and a subset Rel of the direct product $S \times R$, which is called the underlying relation of F. S, R, and Rel are generalized domains (see Chapter 12.4). The corresponding attributes for general mappings are `Source`, `Range`, and `UnderlyingRelation`.

Note that general mappings themselves are **not** domains. One reason for this is that two general mappings with same underlying relation are regarded as equal only if also the sources are equal and the ranges are equal. Other, more technical, reasons are that general mappings and domains have different basic operations, and that general mappings are arithmetic objects (see 31.5); both should not apply to domains.

Each element of an underlying relation of a general mapping lies in the category of tuples (see 31).

For each $s \in S$, the set $\{r \in R | (s, r) \in Rel\}$ is called the set of **images** of s. Analogously, for $r \in R$, the set $\{s \in S | (s, r) \in Rel\}$ is called the set of **preimages** of r.

The **ordering** of general mappings via `<` is defined by the ordering of source, range, and underlying relation. Specifically, if the Source and Range domains of *map1* and *map2* are the same, then one considers the union of the preimages of *map1* and *map2* as a strictly ordered set. The smaller of *map1* and *map2* is the one whose image is smaller on the first point of this sequence where they differ.

For mappings which preserve an algebraic structure a **kernel** is defined. Depending on the structure preserved the operation to compute this kernel is called differently, see section 31.6.

Some technical details of general mappings are described in section 31.12.

1 ▶ `IsTuple(` *obj* `)` C

`IsTuple` is a subcategory of the meet of `IsDenseList` (see 21.1.2), `IsMultiplicativeElementWithInverse` (see 30.14.13), and `IsAdditiveElementWithInverse` (see 30.14.7), where the arithmetic operations (addition, zero, additive inverse, multiplication, powering, one, inverse) are defined componentwise.

Note that each of these operations will cause an error message if its result for at least one component cannot be formed.

The sum and the product of a tuple and a list in `IsListDefault` is the list of sums and products, respectively. The sum and the product of a tuple and a non-list is the tuple of componentwise sums and products, respectively.

31.1 Creating Mappings

1 ▶ `GeneralMappingByElements(` *S*, *R*, *elms* `)` F

is the general mapping with source S and range R, and with underlying relation consisting of the tuples collection *elms*.

2 ▶ `MappingByFunction(` *S*, *R*, *fun* `)` F
▶ `MappingByFunction(` *S*, *R*, *fun*, *invfun* `)` F
▶ `MappingByFunction(` *S*, *R*, *fun*, '`false`, *prefun* `)'` F

`MappingByFunction` returns a mapping *map* with source S and range R, such that each element s of S is mapped to the element *fun*`(` *s* `)`, where *fun* is a GAP function.

If the argument *invfun* is bound then *map* is a bijection between S and R, and the preimage of each element r of R is given by *invfun*`(` *r* `)`, where *invfun* is a GAP function.

In the third variant, a function *prefun* is given that can be used to compute a single preimage. In this case, the third entry must be `false`.

`MappingByFunction` creates a mapping which `IsNonSPGeneralMapping`

3 ▶ `InverseGeneralMapping(` *map* `)` A

The **inverse general mapping** of a general mapping *map* is the general mapping whose underlying relation (see 31.2.9) contains a pair (r, s) if and only if the underlying relation of *map* contains the pair (s, r).

See the introduction to Chapter 31 for the subtleties concerning the difference between `InverseGeneralMapping` and `Inverse`.

Note that the inverse general mapping of a mapping *map* is in general only a general mapping. If *map* knows to be bijective its inverse general mapping will know to be a mapping. In this case also `Inverse(` *map* `)` works.

4 ▶ `CompositionMapping(` *map1*, *map2*, ... `)` F

`CompositionMapping` allows one to compose arbitrarily many general mappings, and delegates each step to `CompositionMapping2`.

Additionally, the properties `IsInjective` and `IsSingleValued` are maintained; if the source of the $i+1$-th general mapping is identical to the range of the i-th general mapping, also `IsTotal` and `IsSurjective` are maintained. (So one should not call `CompositionMapping2` directly if one wants to maintain these properties.)

Depending on the types of *map1* and *map2*, the returned mapping might be constructed completely new (for example by giving domain generators and their images, this is for example the case if both mappings preserve the same alagebraic structures and GAP can decompose elements of the source of *map2* into generators) or as an (iterated) composition (see 31.1.6).

5 ▶ `CompositionMapping2(` *map2*, *map1* `)` O

`CompositionMapping2` returns the composition of *map2* and *map1*, this is the general mapping that maps an element first under *map1*, and then maps the images under *map2*.

(Note the reverse ordering of arguments in the composition via `*`.

6 ▶ `IsCompositionMappingRep(` *map* `)` R

Mappings in this representation are stored as composition of two mappings, (pre)images of elements are computed in a two-step process. The constituent mappings of the composition can be obtained via `ConstituentsCompositionMapping`.

7 ▶ `ConstituentsCompositionMapping(` *map* `)` F

If *map* is stored in the representation `IsCompositionMappingRep` as composition of two mappings *map1* and *map2*, this function returns the two constituent mappings in a list [*map1*,*map2*].

8 ▶ `ZeroMapping(` *S*, *R* `)` O

A zero mapping is a total general mapping that maps each element of its source to the zero element of its range.

(Each mapping with empty source is a zero mapping.)

9 ▶ `IdentityMapping(` *D* `)` A

is the bijective mapping with source and range equal to the collection D, which maps each element of D to itself.

10 ▶ `Embedding(` *S*, *T* `)` O
▶ `Embedding(` *S*, *i* `)` O

returns the embedding of the domain S in the domain T, or in the second form, some domain indexed by the positive integer i. The precise natures of the various methods are described elsewhere: for Lie algebras, see `LieFamily` (61.1.3); for group products, see 47.5 for a general description, or for examples see 47.1 for direct products, 47.2 for semidirect products, or 47.4 for wreath products; or for magma rings see 63.3.

11 ▶ `Projection(` *S*, *T* `)` O
▶ `Projection(` *S*, *i* `)` O
▶ `Projection(` *S* `)` O

returns the projection of the domain S onto the domain T, or in the second form, some domain indexed by the positive integer i, or in the third form some natural subdomain of S. Various methods are defined for group products; see 47.5 for a general description, or for examples see 47.1 for direct products, 47.2 for semidirect products, 47.3 for subdirect products, or 47.4 for wreath products.

12 ▶ `RestrictedMapping(` *map*, *subdom* `)` O

If *subdom* is a subdomain of the source of the general mapping *map*, this operation returns the restriction of *map* to *subdom*.

31.2 Properties and Attributes of (General) Mappings

1 ▶ `IsTotal(` *map* `)` P

is `true` if each element in the source S of the general mapping *map* has images, i.e., $s^{map} \neq \emptyset$ for all $s \in S$, and `false` otherwise.

2 ▶ `IsSingleValued(` *map* `)` P

is `true` if each element in the source S of the general mapping *map* has at most one image, i.e., $|s^{map}| \leq 1$ for all $s \in S$, and `false` otherwise.

Equivalently, `IsSingleValued(` *map* `)` is `true` if and only if the preimages of different elements in R are disjoint.

3 ▶ `IsMapping(` *map* `)` P

A **mapping** *map* is a general mapping that assigns to each element `elm` of its source a unique element `Image(` *map*, *elm* `)` of its range.

Equivalently, the general mapping *map* is a mapping if and only if it is total and single-valued (see 31.2.1, 31.2.2).

4 ▶ `IsInjective( map )` P

is `true` if the images of different elements in the source S of the general mapping *map* are disjoint, i.e., $x^{map} \cap y^{map} = \emptyset$ for $x \neq y \in S$, and `false` otherwise.

Equivalently, `IsInjective( map )` is `true` if and only if each element in the range of *map* has at most one preimage in S.

5 ▶ `IsSurjective( map )` P

is `true` if each element in the range R of the general mapping *map* has preimages in the source S of *map*, i.e., $\{s \in S \mid x \in s^{map}\} \neq \emptyset$ for all $x \in R$, and `false` otherwise.

6 ▶ `IsBijective( map )` P

A general mapping *map* is **bijective** if and only if it is an injective and surjective mapping (see 31.2.3, 31.2.4, 31.2.5).

7 ▶ `Range( map )` A

8 ▶ `Source( map )` A

9 ▶ `UnderlyingRelation( map )` A

The **underlying relation** of a general mapping *map* is the domain of pairs (s, r), with s in the source and r in the range of *map* (see 31.2.8, 31.2.7), and $r \in$`ImagesElm( map, s )`.

Each element of the underlying relation is a tuple (see 31).

10 ▶ `UnderlyingGeneralMapping( map )` A

attribute for underlying relations of general mappings

31.3 Images under Mappings

1 ▶ `ImagesSource( map )` A

is the set of images of the source of the general mapping *map*.

`ImagesSource` delegates to `ImagesSet`, it is introduced only to store the image of *map* as attribute value.

2 ▶ `ImagesRepresentative( map, elm )` O

If *elm* is an element of the source of the general mapping *map* then `ImagesRepresentative` returns either a representative of the set of images of *elm* under *map* or `fail`, the latter if and only if *elm* has no images under *map*.

Anything may happen if *elm* is not an element of the source of *map*.

3 ▶ `ImagesElm( map, elm )` O

If *elm* is an element of the source of the general mapping *map* then `ImagesElm` returns the set of all images of *elm* under *map*.

Anything may happen if *elm* is not an element of the source of *map*.

4 ▶ `ImagesSet( map, elms )` O

If *elms* is a subset of the source of the general mapping *map* then `ImagesSet` returns the set of all images of *elms* under *map*.

Anything may happen if *elms* is not a subset of the source of *map*.

5 ▶ `ImageElm( map, elm )` O

If *elm* is an element of the source of the total and single-valued mapping *map* then `ImageElm` returns the unique image of *elm* under *map*.

Anything may happen if *elm* is not an element of the source of *map*.

6 ▶ `Image( map )` F
▶ `Image( map, elm )` F
▶ `Image( map, coll )` F

`Image( map )` is the image of the general mapping *map*, i.e., the subset of elements of the range of *map* that are actually values of *map*. Note that in this case the argument may also be multi-valued.

`Image( map, elm )` is the image of the element *elm* of the source of the mapping *map* under *map*, i.e., the unique element of the range to which *map* maps *elm*. This can also be expressed as *elm* `^` *map*. Note that *map* must be total and single valued, a multi valued general mapping is not allowed (see 31.3.7).

`Image( map, coll )` is the image of the subset *coll* of the source of the mapping *map* under *map*, i.e., the subset of the range to which *map* maps elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. Note that in this case *map* may also be multi-valued. (If *coll* and the result are lists then the positions of entries do in general **not** correspond.)

`Image` delegates to `ImagesSource` when called with one argument, and to `ImageElm` resp. `ImagesSet` when called with two arguments.

If the second argument is not an element or a subset of the source of the first argument, an error is signalled.

7 ▶ `Images( map )` F
▶ `Images( map, elm )` F
▶ `Images( map, coll )` F

`Images( map )` is the image of the general mapping *map*, i.e., the subset of elements of the range of *map* that are actually values of *map*.

`Images( map, elm )` is the set of images of the element *elm* of the source of the general mapping *map* under *map*, i.e., the set of elements of the range to which *map* maps *elm*.

`Images( map, coll )` is the set of images of the subset *coll* of the source of the general mapping *map* under *map*, i.e., the subset of the range to which *map* maps elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. (If *coll* and the result are lists then the positions of entries do in general **not** correspond.)

`Images` delegates to `ImagesSource` when called with one argument, and to `ImagesElm` resp. `ImagesSet` when called with two arguments.

If the second argument is not an element or a subset of the source of the first argument, an error is signalled.

31.4 Preimages under Mappings

1 ▶ `PreImagesRange( map )` A

is the set of preimages of the range of the general mapping *map*.

`PreImagesRange` delegates to `PreImagesSet`, it is introduced only to store the preimage of *map* as attribute value.

2 ▶ `PreImagesElm( map, elm )` O

If *elm* is an element of the range of the general mapping *map* then `PreImagesElm` returns the set of all preimages of *elm* under *map*.

Anything may happen if *elm* is not an element of the range of *map*.

3 ▶ `PreImageElm( map, elm )` O

If *elm* is an element of the range of the injective and surjective general mapping *map* then `PreImageElm` returns the unique preimage of *elm* under *map*.

Anything may happen if *elm* is not an element of the range of *map*.

4 ▶ `PreImagesRepresentative( map, elm )` O

If *elm* is an element of the range of the general mapping *map* then `PreImagesRepresentative` returns either a representative of the set of preimages of *elm* under *map* or `fail`, the latter if and only if *elm* has no preimages under *map*.

Anything may happen if *elm* is not an element of the range of *map*.

5 ▶ `PreImagesSet( map, elms )` O

If *elms* is a subset of the range of the general mapping *map* then `PreImagesSet` returns the set of all preimages of *elms* under *map*.

Anything may happen if *elms* is not a subset of the range of *map*.

6 ▶ `PreImage( map )` F
▶ `PreImage( map, elm )` F
▶ `PreImage( map, coll )` F

`PreImage( map )` is the preimage of the general mapping *map*, i.e., the subset of elements of the source of *map* that actually have values under *map*. Note that in this case the argument may also be non-injective or non-surjective.

`PreImage( map, elm )` is the preimage of the element *elm* of the range of the injective and surjective mapping *map* under *map*, i.e., the unique element of the source which is mapped under *map* to *elm*. Note that *map* must be injective and surjective (see 31.4.7).

`PreImage( map, coll )` is the preimage of the subset *coll* of the range of the general mapping *map* under *map*, i.e., the subset of the source which is mapped under *map* to elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. Note that in this case *map* may also be non-injective or non-surjective. (If *coll* and the result are lists then the positions of entries do in general **not** correspond.)

`PreImage` delegates to `PreImagesRange` when called with one argument, and to `PreImageElm` resp. `PreImagesSet` when called with two arguments.

If the second argument is not an element or a subset of the range of the first argument, an error is signalled.

7 ▶ `PreImages( map )` F
▶ `PreImages( map, elm )` F
▶ `PreImages( map, coll )` F

`PreImages( map )` is the preimage of the general mapping *map*, i.e., the subset of elements of the source of *map* that have actually values under *map*.

`PreImages( map, elm )` is the set of preimages of the element *elm* of the range of the general mapping *map* under *map*, i.e., the set of elements of the source which *map* maps to *elm*.

`PreImages( map, coll )` is the set of images of the subset *coll* of the range of the general mapping *map* under *map*, i.e., the subset of the source which *map* maps to elements of *coll*. *coll* may be a proper set or a domain. The result will be either a proper set or a domain. (If *coll* and the result are lists then the positions of entries do in general **not** correspond.)

`PreImages` delegates to `PreImagesRange` when called with one argument, and to `PreImagesElm` resp. `PreImagesSet` when called with two arguments.

If the second argument is not an element or a subset of the range of the first argument, an error is signalled.

31.5 Arithmetic Operations for General Mappings

General mappings are arithmetic objects. One can form groups and vector spaces of general mappings provided that they are invertible or can be added and admit scalar multiplication, respectively.

For two general mappings with same source, range, preimage, and image, the **sum** is defined pointwise, i.e., the images of a point under the sum is the set of all sums with first summand in the images of the first general mapping and second summand in the images of the second general mapping.

Scalar multiplication of general mappings is defined likewise.

The **product** of two general mappings is defined as the composition. This multiplication is always associative. In addition to the composition via `*`, general mappings can be composed –in reversed order– via `CompositionMapping`.

General mappings are in the category of multiplicative elements with inverses. Similar to matrices, not every general mapping has an inverse or an identity, and we define the behaviour of `One` and `Inverse` for general mappings as follows. `One` returns `fail` when called for a general mapping whose source and range differ, otherwise `One` returns the identity mapping of the source. (Note that the source may differ from the preimage). `Inverse` returns `fail` when called for a non-bijective general mapping or for a general mapping whose source and range differ; otherwise `Inverse` returns the inverse mapping.

Besides the usual inverse of multiplicative elements, which means that `Inverse(` *g* `) * g = g * Inverse(` *g* `) = One(` *g* `)`, for general mappings we have the attribute `InverseGeneralMapping`. If F is a general mapping with source S, range R, and underlying relation Rel then `InverseGeneralMapping(` F `)` has source R, range S, and underlying relation $\{(r,s) \mid (s,r) \in Rel\}$. For a general mapping that has an inverse in the usual sense, i.e., for a bijection of the source, of course both concepts coincide.

`Inverse` may delegate to `InverseGeneralMapping`. `InverseGeneralMapping` must not delegate to `Inverse`, but a known value of `Inverse` may be fetched. So methods to compute the inverse of a general mapping should be installed for `InverseGeneralMapping`.

(Note that in many respects, general mappings behave similar to matrices, for example one can define left and right identities and inverses, which do not fit into the current concepts of GAP.)

31.6 Mappings which are Compatible with Algebraic Structures

From an algebraical point of view, the most important mappings are those which are compatible with a structure. For Magmas, Groups and Rings, GAP supports the following four types of such mappings:

1. General mappings that respect multiplication
2. General mappings that respect addition
3. General mappings that respect scalar mult.
4. General mappings that respect multiplicative and additive structure

(Very technical note: GAP defines categories `IsSPGeneralMapping` and `IsNonSPGeneralMapping`. The distinction between these is orthogonal to the Structure Compatibility described here and should not be confused.)

31.7 Magma Homomorphisms

1 ▶ `IsMagmaHomomorphism(` *mapp* `)` P

A `MagmaHomomorphism` is a total single valued mapping which respects multiplication.

2 ▶ `MagmaHomomorphismByFunctionNC(` *G*`,` *H*`,` *fn* `)` F

Creates the homomorphism from G to H without checking that *fn* is a homomorphism.

3 ▶ `NaturalHomomorphismByGenerators(` *f*`,` *s* `)` O

returns a mapping from the magma f with n generators to the magma s with n generators, which maps the ith generator of f to the ith generator of s.

31.8 Mappings that Respect Multiplication

1 ▶ `RespectsMultiplication( mapp )` P

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsMultiplication` returns `true` if S and R are magmas such that $(s_1, r_1), (s_2, r_2) \in F$ implies $(s_1 * s_2, r_1 * r_2) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsMultiplication` returns `true` if and only if the equation *s1*`^`*mapp* `*` *s2*`^`*mapp* `= (`*s1*`*`*s2*`)^`*mapp* holds for all *s1*, *s2* in S.

2 ▶ `RespectsOne( mapp )` P

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsOne` returns `true` if S and R are magmas-with-one such that $(\texttt{One}(S), \texttt{One}(R)) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsOne` returns `true` if and only if the equation `One(` *S* `)^`*mapp* `= One(` *R* `)` holds.

3 ▶ `RespectsInverses( mapp )` P

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsInverses` returns `true` if S and R are magmas-with-inverses such that, for $s \in S$ and $r \in R$, $(s, r) \in F$ implies $(s^{-1}, r^{-1}) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsInverses` returns `true` if and only if the equation `Inverse(` *s* `)^`*mapp* `= Inverse(` *s*`^`*mapp* `)` holds for all s in S.

Mappings that are defined on a group and respect multiplication and inverses are group homomorphisms. Chapter 38 explains them in more detail.

4 ▶ `IsGroupGeneralMapping( mapp )` P
▶ `IsGroupHomomorphism( mapp )` P

A `GroupGeneralMapping` is a mapping which respects multiplication and inverses. If it is total and single valued it is called a group homomorphism.

5 ▶ `KernelOfMultiplicativeGeneralMapping( mapp )` A

Let *mapp* be a general mapping. Then `KernelOfMultiplicativeGeneralMapping` returns the set of all elements in the source of *mapp* that have the identity of the range in their set of images.

(This is a monoid if *mapp* respects multiplication and one, and if the source of *mapp* is associative.)

6 ▶ `CoKernelOfMultiplicativeGeneralMapping( mapp )` A

Let *mapp* be a general mapping. Then `CoKernelOfMultiplicativeGeneralMapping` returns the set of all elements in the range of *mapp* that have the identity of the source in their set of preimages.

(This is a monoid if *mapp* respects multiplication and one, and if the range of *mapp* is associative.)

31.9 Mappings that Respect Addition

1 ▶ `RespectsAddition( mapp )` P

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsAddition` returns `true` if S and R are additive magmas such that $(s_1, r_1), (s_2, r_2) \in F$ implies $(s_1 + s_2, r_1 + r_2) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsAddition` returns `true` if and only if the equation *s1*`^`*mapp* `+` *s2*`^`*mapp* `=` `(`*s1*`+`*s2*`)^`*mapp* holds for all *s1*, *s2* in S.

2 ▶ `RespectsAdditiveInverses( mapp )` P

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsAdditiveInverses` returns `true` if S and R are additive-magmas-with-inverses such that $(s, r) \in F$ implies $(-s, -r) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsAdditiveInverses` returns `true` if and only if the equation `AdditiveInverse( `*s*` )^`*mapp* `= AdditiveInverse( `*s*`^`*mapp*` )` holds for all s in S.

3 ▶ `RespectsZero( mapp )` P

Let *mapp* be a general mapping with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsZero` returns `true` if S and R are additive-magmas-with-zero such that $(\texttt{Zero}(S), \texttt{Zero}(R)) \in F$, and `false` otherwise.

If *mapp* is single-valued then `RespectsZero` returns `true` if and only if the equation `Zero( `*S*` )^`*mapp* `= Zero( `*R*` )` holds.

4 ▶ `IsAdditiveGroupGeneralMapping( mapp )` P
▶ `IsAdditiveGroupHomomorphism( mapp )` P

5 ▶ `KernelOfAdditiveGeneralMapping( mapp )` A

Let *mapp* be a general mapping. Then `KernelOfAdditiveGeneralMapping` returns the set of all elements in the source of *mapp* that have the zero of the range in their set of images.

6 ▶ `CoKernelOfAdditiveGeneralMapping( mapp )` A

Let *mapp* be a general mapping. Then `CoKernelOfAdditiveGeneralMapping` returns the set of all elements in the range of *mapp* that have the zero of the source in their set of preimages.

31.10 Linear Mappings

Also see Sections 31.8 and 31.9.

1 ▶ `RespectsScalarMultiplication( mapp )` P

Let *mapp* be a general mapping, with underlying relation $F \subseteq S \times R$, where S and R are the source and the range of *mapp*, respectively. Then `RespectsScalarMultiplication` returns `true` if S and R are left modules with the left acting domain D of S contained in the left acting domain of R and such that $(s, r) \in F$ implies $(c * s, c * r) \in F$ for all $c \in D$, and `false` otherwise.

If *mapp* is single-valued then `RespectsScalarMultiplication` returns `true` if and only if the equation *c* `*` *s*`^`*mapp* `= (`*c* `*` *s*`)^`*mapp* holds for all c in D and s in S.

2 ▶ `IsLeftModuleGeneralMapping( mapp )` P
▶ `IsLeftModuleHomomorphism( mapp )` P

3 ▶ `IsLinearMapping( F, mapp )` O

For a field F and a general mapping *mapp*, `IsLinearMapping` returns `true` if *mapp* is an F-linear mapping, and `false` otherwise.

A mapping f is a linear mapping (or vector space homomorphism) if the source and range are vector spaces over the same division ring D, and if $f(a+b) = f(a) + f(b)$ and $f(s*a) = s*f(a)$ hold for all elements a, b in the source of f and $s \in D$.

See also `KernelOfMultiplicativeGeneralMapping` (31.8.5) and `CoKernelOfMultiplicativeGeneralMapping` (31.8.6).

31.11 Ring Homomorphisms

1 ▶ `IsRingGeneralMapping( `*mapp*` )` P
▶ `IsRingHomomorphism( `*mapp*` )` P

2 ▶ `IsRingWithOneGeneralMapping( `*mapp*` )` P
▶ `IsRingWithOneHomomorphism( `*mapp*` )` P

3 ▶ `IsAlgebraGeneralMapping( `*mapp*` )` P
▶ `IsAlgebraHomomorphism( `*mapp*` )` P

4 ▶ `IsAlgebraWithOneGeneralMapping( `*mapp*` )` P
▶ `IsAlgebraWithOneHomomorphism( `*mapp*` )` P

5 ▶ `IsFieldHomomorphism( `*mapp*` )` P

A general mapping is a field homomorphism if and only if it is a ring homomorphism with source a field.

31.12 General Mappings

1 ▶ `IsGeneralMapping( `*map*` )` C

Each general mapping lies in the category `IsGeneralMapping`. It implies the categories `IsMultiplicativeElementWithInverse` (see 30.14.13) and `IsAssociativeElement` (see 30.15.1); for a discussion of these implications, see 31.5.

2 ▶ `IsConstantTimeAccessGeneralMapping( `*map*` )` P

is `true` if the underlying relation of the general mapping *map* knows its `AsList` value, and `false` otherwise.

In the former case, *map* is allowed to use this list for calls to `ImagesElm` etc.

3 ▶ `IsEndoGeneralMapping( `*obj*` )` P

If a general mapping has this property then its source and range are equal.

31.13 Technical Matters Concerning General Mappings

`Source` and `Range` are basic operations for general mappings. `UnderlyingRelation` is secondary, its default method sets up a domain that delegates tasks to the general mapping. (Note that this allows one to handle also infinite relations by generic methods if source or range of the general mapping is finite.)

The distinction between basic operations and secondary operations for general mappings may be a little bit complicated. Namely, each general mapping must be in one of the two categories `IsNonSPGeneralMapping`, `IsSPGeneralMapping`. (The category `IsGeneralMapping` is defined as the disjoint union of these two.)

For general mappings of the first category, `ImagesElm` and `PreImagesElm` are basic operations. (Note that in principle it is possible to delegate from `PreImagesElm` to `ImagesElm`.) Methods for the secondary operations `(Pre)ImageElm`, `(Pre)ImagesSet`, and `(Pre)ImagesRepresentative` may use `(Pre)ImagesElm`, and methods for `(Pre)ImagesElm` must not call the secondary operations. In particular, there are no generic methods for `(Pre)ImagesElm`.

Methods for `(Pre)ImagesSet` must **not** use `PreImagesRange` and `ImagesSource`, e.g., compute the intersection of the set in question with the preimage of the range resp. the image of the source.

For general mappings of the second category (which means structure preserving general mappings), the situation is different. The set of preimages under a group homomorphism, for example, is either empty or can be described as a coset of the (multiplicative) kernel. So it is reasonable to have `(Pre)ImagesRepresentative` and `Multplicative(Co)Kernel` as basic operations here, and to make `(Pre)ImagesElm` secondary operations that may delegate to these.

In order to avoid infinite recursions, we must distinguish between the two different types of mappings.

(Note that the basic domain operations such as `AsList` for the underlying relation of a general mapping may use either `ImagesElm` or `ImagesRepresentative` and the appropriate cokernel. Conversely, if `AsList` for the underlying relation is known then `ImagesElm` resp. `ImagesRepresentative` may delegate to it, the general mapping gets the property `IsConstantTimeAccessGeneralMapping` for this; note that this is not allowed if only an enumerator of the underlying relation is known.)

Secondary operations are `IsInjective`, `IsSingleValued`, `IsSurjective`, `IsTotal`; they may use the basic operations, and must not be used by them.

1 ▶ `IsSPGeneralMapping(` *map* `)` C
▶ `IsNonSPGeneralMapping(` *map* `)` C

2 ▶ `IsGeneralMappingFamily(` *obj* `)` C

3 ▶ `FamilyRange(` *Fam* `)` A

is the elements family of the family of the range of each general mapping in the family *Fam*.

4 ▶ `FamilySource(` *Fam* `)` A

is the elements family of the family of the source of each general mapping in the family *Fam*.

5 ▶ `FamiliesOfGeneralMappingsAndRanges(` *Fam* `)` AM

is a list that stores at the odd positions the families of general mappings with source in the family *Fam*, at the even positions the families of ranges of the general mappings.

6 ▶ `GeneralMappingsFamily(` *sourcefam*`,` *rangefam* `)` F

All general mappings with same source family *FS* and same range family *FR* lie in the family `GeneralMappingsFamily(` *FS*`,` *FR* `)`.

7 ▶ `TypeOfDefaultGeneralMapping(` *source*`,` *range*`,` *filter* `)` F

is the type of mappings with `IsDefaultGeneralMappingRep` with source *source* and range *range* and additional categories *filter*.

Methods for the operations `ImagesElm`, `ImagesRepresentative`, `ImagesSet`, `ImageElm`, `PreImagesElm`, `PreImagesRepresentative`, `PreImagesSet`, and `PreImageElm` take two arguments, a general mapping *map* and an element or collection of elements *elm*. These methods must **not** check whether *elm* lies in the source or the range of *map*. In the case that *elm* does not, `fail` may be returned as well as any other GAP object, and even an error message is allowed. Checks of the arguments are done only by the functions `Image`, `Images`, `PreImage`, and `PreImages`, which then delegate to the operations listed above.

32 Relations

A **binary relation** R on a set X is a subset of $X \times X$. A binary relation can also be thought of as a (general) mapping from X to itself or as a directed graph where each edge represents a tuple of R.

In GAP, a relation is conceptually represented as a general mapping from X to itself. The category `IsBinaryRelation` is the same as the category `IsEndoGeneralMapping` (see 31.12.3). Attributes and properties of relations in GAP are supported for relations, via considering relations as a subset of $X \times X$, or as a directed graph; examples include finding the strongly connected components of a relation, via `StronglyConnectedComponents` (see 32.4.5), or enumerating the tuples of the relation.

32.1 General Binary Relations

1 ► `IsBinaryRelation(` *R* `)` C

is exactly the same category as (i.e. a synonym for) `IsEndoGeneralMapping` (see 31.12.3).

We have the following general constructors.

2 ► `BinaryRelationByElements(` *domain*`,` *elms* `)` F

is the binary relation on *domain* and with underlying relation consisting of the tuples collection *elms*. This construction is similar to `GeneralMappingByElements` (see 31.1.1) where the source and range are the same set.

3 ► `IdentityBinaryRelation(` *degree* `)` F
► `IdentityBinaryRelation(` *domain* `)` F

is the binary relation which consists of diagonal tuples i.e. tuples of the form (x, x). In the first form if a positive integer *degree* is given then the domain is the integers $\{1, \ldots, degree\}$. In the second form, the tuples are from the domain *domain*.

4 ► `EmptyBinaryRelation(` *degree* `)` F
► `EmptyBinaryRelation(` *domain* `)` F

is the relation with R empty. In the first form of the command with *degree* an integer, the domain is the points $\{1, \ldots, degree\}$. In the second form, the domain is that given by the argument *domain*.

32.2 Properties and Attributes of Binary Relations

1 ► `IsReflexiveBinaryRelation(` *R* `)` P

returns `true` if the binary relation R is reflexive, and `false` otherwise.

A binary relation R (as tuples) on a set X is **reflexive** if for all $x \in X$, $(x, x) \in R$. Alternatively, R as a mapping is reflexive if for all $x \in X$, x is an element of the image set $R(x)$.

A reflexive binary relation is necessarily a total endomorphic mapping (tested via `IsTotal`; see 31.2.1).

2 ▶ `IsSymmetricBinaryRelation( R )` P

returns `true` if the binary relation R is symmetric, and `false` otherwise.

A binary relation R (as tuples) on a set X is **symmetric** if $(x, y) \in R$ then $(y, x) \in R$. Alternatively, R as a mapping is symmetric if for all $x \in X$, the preimage set of x under R equals the image set $R(x)$.

3 ▶ `IsTransitiveBinaryRelation( R )` P

returns `true` if the binary relation R is transitive, and `false` otherwise.

A binary relation R (as tuples) on a set X is **transitive** if $(x, y), (y, z) \in R$ then $(x, z) \in R$. Alternatively, R as a mapping is transitive if for all $x \in X$, the image set $R(R(x))$ of the image set $R(x)$ of x is a subset of $R(x)$.

4 ▶ `IsAntisymmetricBinaryRelation( rel )` P

returns `true` if the binary relation *rel* is antisymmetric, and `false` otherwise.

A binary relation R (as tuples) on a set X is **antisymmetric** if $(x, y), (y, x) \in R$ implies $x = y$. Alternatively, R as a mapping is antisymmetric if for all $x \in X$, the intersection of the preimage set of x under R and the image set $R(x)$ is $\{x\}$.

5 ▶ `IsPreOrderBinaryRelation( rel )` P

returns `true` if the binary relation *rel* is a preorder, and `false` otherwise.

A **preorder** is a binary relation that is both reflexive and transitive.

6 ▶ `IsPartialOrderBinaryRelation( rel )` P

returns `true` if the binary relation *rel* is a partial order, and `false` otherwise.

A **partial order** is a preorder which is also antisymmetric.

7 ▶ `IsHasseDiagram( rel )` P

returns `true` if the binary relation *rel* is a Hasse Diagram of a partial order, i.e. was computed via `HasseDiagramBinaryRelation` (see 32.4.4).

8 ▶ `IsEquivalenceRelation( R )` P

returns `true` if the binary relation R is an equivalence relation, and `false` otherwise.

Recall, that a relation R on the set X is an **equivalence relation** if it is symmetric, transitive, and reflexive.

9 ▶ `Successors( R )` A

returns the list of images of a binary relation R. If the underlying domain of the relation is not `[1..n]` for some positive integer n, then an error is signalled.

The returned value of `Successors` is a list of lists where the lists are ordered as the elements according to the sorted order of the underlying set of R. Each list consists of the images of the element whose index is the same as the list with the underlying set in sorted order.

The `Successors` of a relation is the adjacency list representation of the relation.

10 ▶ `DegreeOfBinaryRelation( R )` A

returns the size of the underlying domain of the binary relation R. This is most natural when working with a binary relation on points.

11 ▶ `PartialOrderOfHasseDiagram( HD )` A

is the partial order associated with the Hasse Diagram *HD* i.e. the partial order generated by the reflexive and transitive closure of *HD*.

32.3 Binary Relations on Points

We have special construction methods when the underlying X of our relation is the set of integers $\{1, \ldots, n\}$.

1 ▶ `BinaryRelationOnPoints(` *list* `)` F
▶ `BinaryRelationOnPointsNC(` *list* `)` F

Given a list of n lists, each containing elements from the set $\{1, \ldots, n\}$, this function constructs a binary relation such that 1 is related to *list*`[1]`, 2 to *list*`[2]` and so on. The first version checks whether the list supplied is valid. The the `NC` version skips this check.

2 ▶ `RandomBinaryRelationOnPoints(` *degree* `)` F

creates a relation on points with degree *degree*.

3 ▶ `AsBinaryRelationOnPoints(` *trans* `)` F
▶ `AsBinaryRelationOnPoints(` *perm* `)` F
▶ `AsBinaryRelationOnPoints(` *rel* `)` F

return the relation on points represented by general relation *rel*, transformation *trans* or permutation *perm*. If *rel* is already a binary relation on points then *rel* is returned.

Transformations and permutations are special general endomorphic mappings and have a natural representation as a binary relation on points.

In the last form, an isomorphic relation on points is constructed where the points are indices of the elements of the underlying domain in sorted order.

32.4 Closure Operations and Other Constructors

1 ▶ `ReflexiveClosureBinaryRelation(` R `)` O

is the smallest binary relation containing the binary relation R which is reflexive. This closure inherents the properties symmetric and transitive from R. E.g. if R is symmetric then its reflexive closure is also.

2 ▶ `SymmetricClosureBinaryRelation(` R `)` O

is the smallest binary relation containing the binary relation R which is symmetric. This closure inherents the properties reflexive and transitive from R. E.g. if R is reflexive then its symmetric closure is also.

3 ▶ `TransitiveClosureBinaryRelation(` *rel* `)` O

is the smallest binary relation containing the binary relation R which is transitive. This closure inerents the properties reflexive and symmetric from R. E.g. if R is symmetric then its transitive closure is also.

`TransitiveClosureBinaryRelation` is a modified version of the Floyd-Warshall method of solving the all-pairs shortest-paths problem on a directed graph. Its asymptotic runtime is $O(n^3)$ where n is the size of the vertex set. It only assumes there is an arbitrary (but fixed) ordering of the vertex set.

4 ▶ `HasseDiagramBinaryRelation(` *partial-order* `)` O

is the smallest relation contained in the partial order *partial-order* whose reflexive and transitive closure is equal to *partial-order*.

5 ▶ `StronglyConnectedComponents(` R `)` O

returns an equivalence relation on the vertices of the binary relation R.

6 ▶ `PartialOrderByOrderingFunction(` *dom*`,` *orderfunc* `)` F

constructs a partial order whose elements are from the domain *dom* and are ordered using the ordering function *orderfunc*. The ordering function must be a binary function returning a boolean value. If the ordering function does not describe a partial order then `fail` is returned.

32.5 Equivalence Relations

An **equivalence relation** E over the set X is a relation on X which is reflexive, symmetric, and transitive. of the set X. A **partition** P is a set of subsets of X such that for all $R, S \in P$ $R \cap S$ is the empty set and $\cup P = X$. An equivalence relation induces a partition such that if $(x, y) \in E$ then x, y are in the same element of P.

Like all binary relations in GAP equivalence relations are regarded as general endomorphic mappings (and the operations, properties and attributes of general mappings are available). However, partitions provide an efficient way of representing equivalence relations. Moreover, only the non-singleton classes or blocks are listed allowing for small equivalence relations to be represented on infinite sets. Hence the main attribute of equivalence relations is `EquivalenceRelationPartition` which provides the partition induced by the given equivalence.

1 ▶ `EquivalenceRelationByPartition(` *domain*`,` *list* `)` F
▶ `EquivalenceRelationByPartitionNC(` *domain*`,` *list* `)` F

constructs the equivalence relation over the set *domain* which induces the partition represented by *list*. This representation includes only the non-trivial blocks (or equivalent classes). *list* is a list of lists, each of these lists contain elements of *domain* and are pairwise mutually exclusive.

The list of lists do not need to be in any order nor do the elements in the blocks (see `EquivalenceRelationPartition`). a list of elements of *domain* The partition *list* is a list of lists, each of these is a list of elements of *domain* that makes up a block (or equivalent class). The *domain* is the domain over which the relation is defined, and *list* is a list of lists, each of these is a list of elements of *domain* which are related to each other. *list* need only contain the nontrivial blocks and singletons will be ignored. The NC version will not check to see if the lists are pairwise mutually exclusive or that they contain only elements of the domain.

2 ▶ `EquivalenceRelationByRelation(` *rel* `)` F

returns the smallest equivalence relation containing the binary relation *rel*.

3 ▶ `EquivalenceRelationByPairs(` D`,` *elms* `)` F
▶ `EquivalenceRelationByPairsNC(` D`,` *elms* `)` F

return the smallest equivalence relation on the domain D such that every pair in *elms* is in the relation.

In the second form, it is not checked that *elms* are in the domain D.

4 ▶ `EquivalenceRelationByProperty(` *domain*`,` *property* `)` F

creates an equivalence relation on *domain* whose only defining datum is that of having the property *property*.

32.6 Attributes of and Operations on Equivalence Relations

1 ▶ `EquivalenceRelationPartition(` *equiv* `)` A

returns a list of lists of elements of the underlying set of the equivalence relation *equiv*. The lists are precisely the nonsingleton equivalence classes of the equivalence. This allows us to describe "small" equivalences on infinite sets.

2 ▶ `GeneratorsOfEquivalenceRelationPartition(` *equiv* `)` A

is a set of generating pairs for the equivalence relation *equiv*. This set is not unique. The equivalence *equiv* is the smallest equivalence relation over the underlying set X which contains the generating pairs.

3 ▶ `JoinEquivalenceRelations(` *equiv1*`,` *equiv2* `)` O
▶ `MeetEquivalenceRelations(` *equiv1*`,` *equiv2* `)` O

`JoinEquivalenceRelations(`*equiv1*`,`*equiv2*`)` returns the smallest equivalence relation containing both the equivalence relations *equiv1* and *equiv2*.

`MeetEquivalenceRelations(` *equiv1*`,`*equiv2* `)` returns the intersection of the two equivalence relations *equiv1* and *equiv2*.

32.7 Equivalence Classes

1 ▶ `IsEquivalenceClass(` *O* `)` C

returns `true` if the object *O* is an equivalence class, and `false` otherwise.

An **equivalence class** is a collection of elements which are mutually related to each other in the associated equivalence relation. Note, this is a special category of object and not just a list of elements.

2 ▶ `EquivalenceClassRelation(` *C* `)` A

returns the equivalence relation of which *C* is a class.

3 ▶ `EquivalenceClasses(` *rel* `)` A

returns a list of all equivalence classes of the equivalence relation *rel*. Note that it is possible for different methods to yield the list in different orders, so that for two equivalence relations $c1$ and $c2$ we may have $c1 = c2$ without having `EquivalenceClasses`$(c1) =$ `EquivalenceClasses`$(c2)$.

4 ▶ `EquivalenceClassOfElement(` *rel*`,` *elt* `)` O
▶ `EquivalenceClassOfElementNC(` *rel*`,` *elt* `)` O

return the equivalence class of *elt* in the binary relation *rel*, where *elt* is an element (i.e. a pair) of the domain of *rel*. In the second form, it is not checked that *elt* is in the domain over which *rel* is defined.

33 Magmas

This chapter deals with domains (see 30) that are closed under multiplication `*`. Following [Bou70], we call them **magmas** in GAP. Together with the domains closed under addition `+`, (see 53), they are the basic algebraic structures; every semigroup (see 49), monoid (see 50), group (see 37), ring (see 54), or field (see 56) is a magma. In the cases of a **magma-with-one** or **magma-with-inverses**, additional multiplicative structure is present, see 33.1. For functions to create free magmas, see 34.4.

33.1 Magma Categories

1 ▶ `IsMagma( obj )` C

A **magma** in GAP is a domain M with (not necessarily associative) multiplication `*`: $M \times M \to M$.

2 ▶ `IsMagmaWithOne( obj )` C

A **magma-with-one** in GAP is a magma M with an operation `^0` (or `One`) that yields the identity of M.

So a magma-with-one M does always contain a unique multiplicatively neutral element e, i.e., e `*` m `=` m `=` m `*` e holds for all $m \in M$ (see 33.4.10). This element e can be computed with the operation `One` (see 30.10.2) as `One(` M `)`, and e is also equal to `One(` *elm* `)` and to *elm*`^0` for each element *elm* in M.

Note that a magma may contain a multiplicatively neutral element but **not** an identity (see 30.10.2), and a magma containing an identity may **not** lie in the category `IsMagmaWithOne` (see 30.6).

3 ▶ `IsMagmaWithInversesIfNonzero( obj )` C

An object in this GAP category is a magma-with-one M with an operation `^-1`: $M \setminus Z \to M \setminus Z$ that maps each element m of $M \setminus Z$ to its inverse m`^-1` (or `Inverse(` m `)`, see 30.10.8), where Z is either empty or consists exactly of one element of M.

This category was introduced mainly to describe division rings, since the nonzero elements in a division ring form a group; So an object M in `IsMagmaWithInversesIfNonzero` will usually have both a multiplicative and an additive structure (see 53), and the set Z, if it is nonempty, contains exactly the zero element (see 30.10.3) of M.

4 ▶ `IsMagmaWithInverses( obj )` C

A **magma-with-inverses** in GAP is a magma-with-one M with an operation `^-1`: $M \to M$ that maps each element m of M to its inverse m`^-1` (or `Inverse(` m `)`, see 30.10.8).

Note that not every trivial magma is a magma-with-one, but every trivial magma-with-one is a magma-with-inverses. This holds also if the identity of the magma-with-one is a zero element. So a magma-with-inverses-if-nonzero can be a magma-with-inverses if either it contains no zero element or consists of a zero element that has itself as zero-th power.

33.2 Magma Generation

1 ▶ `Magma( gens )` F
▶ `Magma( Fam, gens )` F

returns the magma M that is generated by the elements in the list *gens*, that is, the closure of *gens* under multiplication `*`. The family *Fam* of M can be entered as first argument; this is obligatory if *gens* is empty (and hence also M is empty).

2 ▶ `MagmaWithOne( gens )` F
▶ `MagmaWithOne( Fam, gens )` F

returns the magma-with-one M that is generated by the elements in the list *gens*, that is, the closure of *gens* under multiplication `*` and `One`. The family *Fam* of M can be entered as first argument; this is obligatory if *gens* is empty (and hence M is trivial).

3 ▶ `MagmaWithInverses( gens )` F
▶ `MagmaWithInverses( Fam, gens )` F

returns the magma-with-inverses M that is generated by the elements in the list *gens*, that is, the closure of *gens* under multiplication `*`, `One`, and `Inverse`. The family *Fam* of M can be entered as first argument; this is obligatory if *gens* is empty (and hence M is trivial).

The underlying operations for which methods can be installed are the following.

4 ▶ `MagmaByGenerators( gens )` O
▶ `MagmaByGenerators( Fam, gens )` O

5 ▶ `MagmaWithOneByGenerators( gens )` O
▶ `MagmaWithOneByGenerators( Fam, generators )` O

6 ▶ `MagmaWithInversesByGenerators( generators )` O
▶ `MagmaWithInversesByGenerators( Fam, generators )` O

Substructures of a magma can be formed as follows.

7 ▶ `Submagma( D, gens )` F
▶ `SubmagmaNC( D, gens )` F

`Submagma` returns the magma generated by the elements in the list *gens*, with parent the domain D. `SubmagmaNC` does the same, except that it is not checked whether the elements of *gens* lie in D.

8 ▶ `SubmagmaWithOne( D, gens )` F
▶ `SubmagmaWithOneNC( D, gens )` F

`SubmagmaWithOne` returns the magma-with-one generated by the elements in the list *gens*, with parent the domain D. `SubmagmaWithOneNC` does the same, except that it is not checked whether the elements of *gens* lie in D.

9 ▶ `SubmagmaWithInverses( D, gens )` F
▶ `SubmagmaWithInversesNC( D, gens )` F

`SubmagmaWithInverses` returns the magma-with-inverses generated by the elements in the list *gens*, with parent the domain D. `SubmagmaWithInversesNC` does the same, except that it is not checked whether the elements of *gens* lie in D.

The following functions can be used to regard a collection as a magma.

10 ▶ `AsMagma( C )` A

For a collection C whose elements form a magma, `AsMagma` returns this magma. Otherwise `fail` is returned.

11 ▶ `AsSubmagma( D, C )` O

Let D be a domain and C a collection. If C is a subset of D that forms a magma then `AsSubmagma` returns this magma, with parent D. Otherwise `fail` is returned.

The following function creates a new magma which is the original magma with a zero adjoined.

12 ▶ `InjectionZeroMagma( M )` A

The canonical homomorphism i from the magma M into the magma formed from M with a single new element which is a multiplicative zero for the resulting magma.

The elements of the new magma form a family of elements in the category IsMultiplicativeElementWithZero, and the new magma is obtained as Range(i).

33.3 Magmas Defined by Multiplication Tables

The most elementary (but of course usually not recommended) way to implement a magma with only few elements is via a multiplication table.

1 ▶ `MagmaByMultiplicationTable( A )` F

For a square matrix A with n rows such that all entries of A are in the range $[1..n]$, `MagmaByMultiplicationTable` returns a magma M with multiplication `*` defined by A. That is, M consists of the elements $m_1, m_2, \ldots, m_n$, and $m_i * m_j = m_{A[i][j]}$.

The ordering of elements is defined by $m_1 < m_2 < \cdots < m_n$, so m_i can be accessed as `MagmaElement( M, i )`, see 33.3.4.

2 ▶ `MagmaWithOneByMultiplicationTable( A )` F

The only differences between `MagmaByMultiplicationTable` and `MagmaWithOneByMultiplicationTable` are that the latter returns a magma-with-one (see 33.2.2) if the magma described by the matrix A has an identity, and returns `fail` if not.

3 ▶ `MagmaWithInversesByMultiplicationTable( A )` F

`MagmaByMultiplicationTable` and `MagmaWithInversesByMultiplicationTable` differ only in that the latter returns magma-with-inverses (see 33.2.3) if each element in the magma described by the matrix A has an inverse, and returns `fail` if not.

4 ▶ `MagmaElement( M, i )` F

For a magma M and a positive integer i, `MagmaElement` returns the i-th element of M, w.r.t. the ordering `<`. If M has less than i elements then `fail` is returned.

5 ▶ `MultiplicationTable( elms )` A
▶ `MultiplicationTable( M )` A

For a list *elms* of elements that form a magma M, `MultiplicationTable` returns a square matrix A of positive integers such that $A[i][j] = k$ holds if and only if *elms*`[i] * `*elms*`[j] = `*elms*`[k]`. This matrix can be used to construct a magma isomorphic to M, using `MagmaByMultiplicationTable`.

For a magma M, `MultiplicationTable` returns the multiplication table w.r.t. the sorted list of elements of M.

```
gap> l:= [ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3) ];;
gap> a:= MultiplicationTable( l );
[ [ 1, 2, 3, 4 ], [ 2, 1, 4, 3 ], [ 3, 4, 1, 2 ], [ 4, 3, 2, 1 ] ]
gap> m:= MagmaByMultiplicationTable( a );
<magma with 4 generators>
gap> One( m );
m1
gap> elm:= MagmaElement( m, 2 );  One( elm );  elm^2;
m2
m1
m1
gap> Inverse( elm );
m2
gap> AsGroup( m );
<group of size 4 with 2 generators>
gap> a:= [ [ 1, 2 ], [ 2, 2 ] ];
[ [ 1, 2 ], [ 2, 2 ] ]
gap> m:= MagmaByMultiplicationTable( a );
<magma with 2 generators>
gap> One( m );  Inverse( MagmaElement( m, 2 ) );
m1
fail
```

33.4 Attributes and Properties for Magmas

1 ▶ `GeneratorsOfMagma( M )` A

is a list *gens* of elements of the magma M that generates M as a magma, that is, the closure of *gens* under multiplication is M.

2 ▶ `GeneratorsOfMagmaWithOne( M )` A

is a list *gens* of elements of the magma-with-one M that generates M as a magma-with-one, that is, the closure of *gens* under multiplication and `One` (see 30.10.2) is M.

3 ▶ `GeneratorsOfMagmaWithInverses( M )` A

is a list *gens* of elements of the magma-with-inverses M that generates M as a magma-with-inverses, that is, the closure of *gens* under multiplication and taking inverses (see 30.10.8) is M.

4 ▶ `Centralizer( M, elm )` O
▶ `Centralizer( M, S )` O
▶ `Centralizer( class )` O

For an element *elm* of the magma M this operation returns the **centralizer** of *elm*. This is the domain of those elements $m \in M$ that commute with *elm*.

For a submagma S it returns the domain of those elements that commute with **all** elements s of S.

If *class* is a class of objects of a magma (this magma then is stored as the `ActingDomain` of *class*) such as given by `ConjugacyClass` (see 37.10.1), `Centralizer` returns the centralizer of `Representative(`*class*`)` (which is a slight abuse of the notation).

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> Centralizer(g,(1,2,3));
Group([ (1,2,3) ])
gap> Centralizer(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3) ])
gap> Centralizer(g,Subgroup(g,[(1,2,3),(1,2)]));
Group(())
```

5 ▶ `Centre( M )` A
▶ `Center( M )` A

`Centre` returns the **centre** of the magma M, i.e., the domain of those elements $m \in M$ that commute with all elements of M. `Center` is just a synonym for `Centre`.

We have `Centre( M ) = Centralizer( M, M )`, see 33.4.4.

The centre of a magma is always commutative (see 33.4.9). (When one installs a new method for `Centre`, one should set the `IsCommutative` value of the result to `true`, in order to make this information available.)

6 ▶ `Idempotents( M )` A

The set of elements of M which are their own squares.

7 ▶ `IsAssociative( M )` P

A magma M is **associative** if for all elements $a, b, c \in M$ the equality `(a * b) * c = a * (b * c)` holds.

An associative magma is called a **semigroup** (see 49), an associative magma-with-one is called a **monoid** (see 50), and an associative magma-with-inverses is called a **group** (see 37).

8 ▶ `IsCentral( M, obj )` O

`IsCentral` returns `true` if the object *obj*, which must either be an element or a magma, commutes with all elements in the magma M.

9 ▶ `IsCommutative( M )` P
▶ `IsAbelian( M )` P

A magma M is **commutative** if for all elements $a, b \in M$ the equality `a * b = b * a` holds. `IsAbelian` is a synonym of `IsCommutative`.

Note that the commutativity of the **addition** `+` in an additive structure can be tested with `IsAdditivelyCommutative`, see 53.3.1.

10 ▶ `MultiplicativeNeutralElement( M )` A

returns the element e in the magma M with the property that `e * m = m = m * e` holds for all $m \in M$, if such an element exists. Otherwise `fail` is returned.

A magma that is not a magma-with-one can have a multiplicative neutral element e; in this case, e **cannot** be obtained as `One( M )`, see 30.10.2.

11 ▶ `MultiplicativeZero( M )` A

Returns the multiplicative zero of the magma which is the element z such that for all m in M, `z * m = m * z = z`.

12 ▶ `IsMultiplicativeZero( M, z )` O

returns true iff `z * m = m * z = z` for all m in M.

13 ▶ `SquareRoots( M, elm )` O

is the proper set of all elements r in the magma M such that r `*` r `=` *elm* holds.

14 ▶ `TrivialSubmagmaWithOne( M )` A

is the magma-with-one that has the identity of the magma-with-one M as only element.

Note that `IsAssociative` and `IsCommutative` always refer to the multiplication of a domain. If a magma M has also an additive structure, e.g., if M is a ring (see 54), then the addition `+` is always assumed to be associative and commutative, see 30.12.

34 Words

This chapter describes categories of **words** and **nonassociative words**, and operations for them. For information about **associative words**, which occur for example as elements in free groups, see Chapter 35.

34.1 Categories of Words and Nonassociative Words

1 ▶ `IsWord( obj )` C
▶ `IsWordWithOne( obj )` C
▶ `IsWordWithInverse( obj )` C

Given a free multiplicative structure M that is freely generated by a subset X, any expression of an element in M as an iterated product of elements in X is called a **word** over X.

Interesting cases of free multiplicative structures are those of free semigroups, free monoids, and free groups, where the multiplication is associative (see 33.4.7), which are described in Chapter 35, and also the case of free magmas, where the multiplication is nonassociative (see 34.1.3).

Elements in free magmas (see 34.4.1) lie in the category `IsWord`; similarly, elements in free magmas-with-one (see 34.4.2) lie in the category `IsWordWithOne`, and so on.

`IsWord` is mainly a "common roof" for the two **disjoint** categories `IsAssocWord` (see 35.1.1) and `IsNonassocWord` (see 34.1.3) of associative and nonassociative words. This means that associative words are **not** regarded as special cases of nonassociative words. The main reason for this setup is that we are interested in different external representations for associative and nonassociative words (see 34.5 and 35.7).

Note that elements in finitely presented groups and also elements in polycyclic groups in GAP are **not** in `IsWord` although they are usually called words, see Chapters 45 and 44.

Words are **constants** (see 12.6), that is, they are not copyable and not mutable.

The usual way to create words is to form them as products of known words, starting from **generators** of a free structure such as a free magma or a free group (see 34.4.1, 35.2.1).

Words are also used to implement free algebras, in the same way as group elements are used to implement group algebras (see 60.2 and Chapter 63).

```
gap> m:= FreeMagmaWithOne( 2 );;  gens:= GeneratorsOfMagmaWithOne( m );
[ x1, x2 ]
gap> w1:= gens[1] * gens[2] * gens[1];
((x1*x2)*x1)
gap> w2:= gens[1] * ( gens[2] * gens[1] );
(x1*(x2*x1))
gap> w1 = w2;  IsAssociative( m );
false
false
gap> IsWord( w1 );  IsAssocWord( w1 );  IsNonassocWord( w1 );
true
false
```

```
true
gap> s:= FreeMonoid( 2 );;  gens:= GeneratorsOfMagmaWithOne( s );
[ m1, m2 ]
gap> u1:= ( gens[1] * gens[2] ) * gens[1];
m1*m2*m1
gap> u2:= gens[1] * ( gens[2] * gens[1] );
m1*m2*m1
gap> u1 = u2;  IsAssociative( s );
true
true
gap> IsWord( u1 );  IsAssocWord( u1 );  IsNonassocWord( u1 );
true
true
false
gap> a:= (1,2,3);;  b:= (1,2);;
gap> w:= a*b*a;;  IsWord( w );
false
```

2 ▶ `IsWordCollection( obj )` C

`IsWordCollection` is the collections category (see 28.1.4) of `IsWord`.

```
gap> IsWordCollection( m );  IsWordCollection( s );
true
true
gap> IsWordCollection( [ "a", "b" ] );
false
```

3 ▶ `IsNonassocWord( obj )` C
▶ `IsNonassocWordWithOne( obj )` C

A **nonassociative word** in GAP is an element in a free magma or a free magma-with-one (see 34.4).

The default methods for `ViewObj` and `PrintObj` (see 6.2) show nonassociative words as products of letters, where the succession of multiplications is determined by round brackets.

In this sense each nonassociative word describes a "program" to form a product of generators. (Also associative words can be interpreted as such programs, except that the exact succession of multiplications is not prescribed due to the associativity.) The function `MappedWord` (see 34.3.1) implements a way to apply such a program. A more general way is provided by straight line programs (see 35.8).

Note that associative words (see Chapter 35) are **not** regarded as special cases of nonassociative words (see 34.1.1).

4 ▶ `IsNonassocWordCollection( obj )` C
▶ `IsNonassocWordWithOneCollection( obj )` C

`IsNonassocWordCollection` is the collections category (see 28.1.4) of `IsNonassocWord`, and `IsNonassocWordWithOneCollection` is the collections category of `IsNonassocWordWithOne`.

34.2 Comparison of Words

1 ► *w1* `=` *w2*

Two words are equal if and only if they are words over the same alphabet and with equal external representations (see 34.5 and 35.7). For nonassociative words, the latter means that the words arise from the letters of the alphabet by the same sequence of multiplications.

2 ► *w1* `<` *w2*

Words are ordered according to their external representation. More precisely, two words can be compared if they are words over the same alphabet, and the word with smaller external representation is smaller. For nonassociative words, the ordering is defined in 34.5; associative words are ordered by the shortlex ordering via `<` (see 35.7).

Note that the alphabet of a word is determined by its family (see 13.1), and that the result of each call to `FreeMagma`, `FreeGroup` etc. consists of words over a new alphabet. In particular, there is no "universal" empty word, every families of words in `IsWordWithOne` has its own empty word.

```
gap> m:= FreeMagma( "a", "b" );;
gap> x:= FreeMagma( "a", "b" );;
gap> mgens:= GeneratorsOfMagma( m );
[ a, b ]
gap> xgens:= GeneratorsOfMagma( x );
[ a, b ]
gap> a:= mgens[1];;  b:= mgens[2];;
gap> a = xgens[1];
false
gap> a*(a*a) = (a*a)*a;  a*b = b*a;  a*a = a*a;
false
false
true
gap> a < b;  b < a;  a < a*b;
true
false
true
```

34.3 Operations for Words

Two words can be multiplied via `*` only if they are words over the same alphabet (see 34.2).

1 ► `MappedWord(` *w*`,` *gens*`,` *imgs* `)` O

`MappedWord` returns the object that is obtained by replacing each occurrence in the word *w* of a generator in the list *gens* by the corresponding object in the list *imgs*. The lists *gens* and *imgs* must of course have the same length.

`MappedWord` needs to do some preprocessing to get internal generator numbers etc. When mapping many (several thousand) words, an explicit loop over the words syllables might be faster.

(For example, If the elements in *imgs* are all **associative words** (see Chapter 35) in the same family as the elements in *gens*, and some of them are equal to the corresponding generators in *gens*, then those may be omitted from *gens* and *imgs*. In this situation, the special case that the lists *gens* and *imgs* have only length 1 is handled more efficiently by `EliminatedWord` (see 35.4.6).)

```
gap> m:= FreeMagma( "a", "b" );;  gens:= GeneratorsOfMagma( m );;
gap> a:= gens[1];  b:= gens[2];
a
b
gap> w:= (a*b)*((b*a)*a)*b;
(((a*b)*((b*a)*a))*b)
gap> MappedWord( w, gens, [ (1,2), (1,2,3,4) ] );
(2,4,3)
gap> a:= (1,2);; b:= (1,2,3,4);;  (a*b)*((b*a)*a)*b;
(2,4,3)

gap> f:= FreeGroup( "a", "b" );;
gap> a:= GeneratorsOfGroup(f)[1];;  b:= GeneratorsOfGroup(f)[2];;
gap> w:= a^5*b*a^2/b^4*a;
a^5*b*a^2*b^-4*a
gap> MappedWord( w, [ a, b ], [ (1,2), (1,2,3,4) ] );
(1,3,4,2)
gap> (1,2)^5*(1,2,3,4)*(1,2)^2/(1,2,3,4)^4*(1,2);
(1,3,4,2)
gap> MappedWord( w, [ a ], [ a^2 ] );
a^10*b*a^4*b^-4*a^2
```

34.4 Free Magmas

The easiest way to create a family of words is to construct the free object generated by these words. Each such free object defines a unique alphabet, and its generators are simply the words of length one over this alphabet; These generators can be accessed via `GeneratorsOfMagma` in the case of a free magma, and via `GeneratorsOfMagmaWithOne` in the case of a free magma-with-one.

1 ▶ `FreeMagma(` *rank* `)` F
▶ `FreeMagma(` *rank*`,` *name* `)` F
▶ `FreeMagma(` *name1*`,` *name2*`, ... )` F
▶ `FreeMagma(` *names* `)` F
▶ `FreeMagma( infinity,` *name*`,` *init* `)` F

Called in the first form, `FreeMagma` returns a free magma on *rank* generators. Called in the second form, `FreeMagma` returns a free magma on *rank* generators, printed as *name*`1`, *name*`2` etc., that is, each name is the concatenation of the string *name* and an integer from `1` to *range*. Called in the third form, `FreeMagma` returns a free magma on as many generators as arguments, printed as *name1*, *name2* etc. Called in the fourth form, `FreeMagma` returns a free magma on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*`[`*i*`]`. Called in the fifth form, `FreeMagma` returns a free magma on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

2 ▶ `FreeMagmaWithOne(` *rank* `)` F
▶ `FreeMagmaWithOne(` *rank*`,` *name* `)` F
▶ `FreeMagmaWithOne(` *name1*`,` *name2*`, ... )` F
▶ `FreeMagmaWithOne(` *names* `)` F
▶ `FreeMagmaWithOne( infinity,` *name*`,` *init* `)` F

Called in the first form, `FreeMagmaWithOne` returns a free magma-with-one on *rank* generators. Called in the second form, `FreeMagmaWithOne` returns a free magma-with-one on *rank* generators, printed as *name*`1`, *name*`2` etc. Called in the third form, `FreeMagmaWithOne` returns a free magma-with-one on as many generators as arguments, printed as *name1*, *name2* etc. Called in the fourth form, `FreeMagmaWithOne` returns

a free magma-with-one on as many generators as the length of the list *names*, the i-th generator being printed as *names*[i]. Called in the fifth form, `FreeMagmaWithOne` returns a free magma on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

```
gap> FreeMagma( 3 );
<free magma on the generators [ x1, x2, x3 ]>
gap> FreeMagma( "a", "b" );
<free magma on the generators [ a, b ]>
gap> FreeMagma( infinity );
<free magma with infinity generators>
gap> FreeMagmaWithOne( 3 );
<free magma-with-one on the generators [ x1, x2, x3 ]>
gap> FreeMagmaWithOne( "a", "b" );
<free magma-with-one on the generators [ a, b ]>
gap> FreeMagmaWithOne( infinity );
<free magma-with-one with infinity generators>
```

Remember that the names of generators used for printing do not necessarily distinguish letters of the alphabet; so it is possible to create arbitrarily weird situations by choosing strange letter names.

```
gap> m:= FreeMagma( "x", "x" );  gens:= GeneratorsOfMagma( m );;
<free magma on the generators [ x, x ]>
gap> gens[1] = gens[2];
false
```

34.5 External Representation for Nonassociative Words

The external representation of nonassociative words is defined as follows. The i-th generator of the family of elements in question has external representation i, the identity (if exists) has external representation 0, the inverse of the i-th generator (if exists) has external representation $-i$. If v and w are nonassociative words with external representations e_v and e_w, respectively then the product $v * w$ has external representation $[e_v, e_w]$. So the external representation of any nonassociative word is either an integer or a nested list of integers and lists, where each list has length two.

One can create a nonassociative word from a family of words and the external representation of a nonassociative word using `ObjByExtRep`.

```
gap> m:= FreeMagma( 2 );;  gens:= GeneratorsOfMagma( m );
[ x1, x2 ]
gap> w:= ( gens[1] * gens[2] ) * gens[1];
((x1*x2)*x1)
gap> ExtRepOfObj( w );  ExtRepOfObj( gens[1] );
[ [ 1, 2 ], 1 ]
1
gap>  ExtRepOfObj( w*w );
[ [ [ 1, 2 ], 1 ], [ [ 1, 2 ], 1 ] ]
gap> ObjByExtRep( FamilyObj( w ), 2 );
x2
gap> ObjByExtRep( FamilyObj( w ), [ 1, [ 2, 1 ] ] );
(x1*(x2*x1))
```

35 Associative Words

35.1 Categories of Associative Words

Associative words are used to represent elements in free groups, semigroups and monoids in GAP (see 35.2). An associative word is just a sequence of letters, where each letter is an element of an alphabet (in the following called a **generator**) or its inverse. Associative words can be multiplied; in free monoids also the computation of an identity is permitted, in free groups also the computation of inverses (see 35.4).

1 ► `IsAssocWord( obj )` C
► `IsAssocWordWithOne( obj )` C
► `IsAssocWordWithInverse( obj )` C

`IsAssocWord` is the category of associative words in free semigroups, `IsAssocWordWithOne` is the category of associative words in free monoids (which admit the operation `One` to compute an identity), `IsAssocWordWithInverse` is the category of associative words in free groups (which have an inverse). See 34.1.1 for more general categories of words.

Different alphabets correspond to different families of associative words. There is no relation whatsoever between words in different families.

```
gap> f:= FreeGroup( "a", "b", "c" );
<free group on the generators [ a, b, c ]>
gap> gens:= GeneratorsOfGroup(f);
[ a, b, c ]
gap> w:= gens[1]*gens[2]/gens[3]*gens[2]*gens[1]/gens[1]*gens[3]/gens[2];
a*b*c^-1*b*c*b^-1
gap> w^-1;
b*c^-1*b^-1*c*b^-1*a^-1
```

Words are displayed as products of letters. The letters are usually printed like `f1`, `f2`, ..., but it is possible to give user defined names to them, which can be arbitrary strings. These names do not necessarily identify a unique letter (generator), it is possible to have several letters –even in the same family– that are displayed in the same way. Note also that **there is no relation between the names of letters and variable names**. In the example above, we might have typed

```
gap> a:= f.1;; b:= f.2;; c:= f.3;;
```

(**Interactively**, the function `AssignGeneratorVariables` (see 35.2.5) provides a shorthand for this.) This allows us to define `w` more conveniently:

```
gap> w := a*b/c*b*a/a*c/b;
a*b*c^-1*b*c*b^-1
```

Using homomorphisms it is possible to express elements of a group as words in terms of generators, see 37.5.

35.2 Free Groups, Monoids and Semigroups

Usually a family of associative words will be generated by constructing the free object generated by them.

1 ▶ `FreeGroup(` [*wfilt*`,`]*rank* `)` F
▶ `FreeGroup(` [*wfilt*`,`]*rank*`,` *name* `)` F
▶ `FreeGroup(` [*wfilt*`,`]*name1*`,` *name2*`, ... )` F
▶ `FreeGroup(` [*wfilt*`,`]*names* `)` F
▶ `FreeGroup(` [*wfilt*`,`]`infinity,` *name*`,` *init* `)` F

Called in the first form, `FreeGroup` returns a free group on *rank* generators. Called in the second form, `FreeGroup` returns a free group on *rank* generators, printed as *name*`1`, *name*`2` etc. Called in the third form, `FreeGroup` returns a free group on as many generators as arguments, printed as *name1*, *name2* etc. Called in the fourth form, `FreeGroup` returns a free group on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*`[`*i*`]`. Called in the fifth form, `FreeGroup` returns a free group on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either `IsSyllableWordsFamily` or `IsLetterWordsFamily` or `IsWLetterWordsFamily` or `IsBLetterWordsFamily`. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

(For interfacing to old code that omits the representation flag, use of the syllable representation is also triggered by setting the option `FreeGroupFamilyType` to the string "syllable".)

2 ▶ `IsFreeGroup(` *obj* `)` C

Any group consisting of elements in `IsAssocWordWithInverse` lies in the filter `IsFreeGroup`; this holds in particular for any group created with `FreeGroup` (see 35.2.1), or any subgroup of such a group.

Also see Chapter 45.

3 ▶ `FreeMonoid(` [*wfilt*`,`]*rank* `)` F
▶ `FreeMonoid(` [*wfilt*`,`]*rank*`,` *name* `)` F
▶ `FreeMonoid(` [*wfilt*`,`]*name1*`,` *name2*`, ... )` F
▶ `FreeMonoid(` [*wfilt*`,`]*names* `)` F
▶ `FreeMonoid(` [*wfilt*`,`]`infinity,` *name*`,` *init* `)` F

Called in the first form, `FreeMonoid` returns a free monoid on *rank* generators. Called in the second form, `FreeMonoid` returns a free monoid on *rank* generators, printed as *name*`1`, *name*`2` etc., that is, each name is the concatenation of the string *name* and an integer from `1` to *range*. Called in the third form, `FreeMonoid` returns a free monoid on as many generators as arguments, printed as *name1*, *name2* etc. Called in the fourth form, `FreeMonoid` returns a free monoid on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*`[`*i*`]`. Called in the fifth form, `FreeMonoid` returns a free monoid on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either `IsSyllableWordsFamily` or `IsLetterWordsFamily` or `IsWLetterWordsFamily` or `IsBLetterWordsFamily`. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

Also see Chapter 50.

4 ▶ `FreeSemigroup(` [*wfilt*`,`]*rank* `)` F
▶ `FreeSemigroup(` [*wfilt*`,`]*rank*`,` *name* `)` F
▶ `FreeSemigroup(` [*wfilt*`,`]*name1*`,` *name2*`, ... )` F
▶ `FreeSemigroup(` [*wfilt*`,`]*names* `)` F
▶ `FreeSemigroup(` [*wfilt*`,`]`infinity,` *name*`,` *init* `)` F

Called in the first form, `FreeSemigroup` returns a free semigroup on *rank* generators. Called in the second form, `FreeSemigroup` returns a free semigroup on *rank* generators, printed as *name*`1`, *name*`2` etc., that is,

each name is the concatenation of the string *name* and an integer from 1 to *range*. Called in the third form, `FreeSemigroup` returns a free semigroup on as many generators as arguments, printed as *name1*, *name2* etc. Called in the fourth form, `FreeSemigroup` returns a free semigroup on as many generators as the length of the list *names*, the *i*-th generator being printed as *names*`[`*i*`]`. Called in the fifth form, `FreeSemigroup` returns a free semigroup on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either `IsSyllableWordsFamily` or `IsLetterWordsFamily` or `IsWLetterWordsFamily` or `IsBLetterWordsFamily`. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

Also see Chapter 49 and 49.

Each free object defines a unique alphabet (and a unique family of words). Its generators are the letters of this alphabet, thus words of length one.

```
gap> FreeGroup( 5 );
<free group on the generators [ f1, f2, f3, f4, f5 ]>
gap> FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> FreeGroup( infinity );
<free group with infinity generators>
gap> FreeSemigroup( "x", "y" );
<free semigroup on the generators [ x, y ]>
gap> FreeMonoid( 7 );
<free monoid on the generators [ m1, m2, m3, m4, m5, m6, m7 ]>
```

Remember that names are just a help for printing and do not necessarily distinguish letters. It is possible to create arbitrarily weird situations by choosing strange names for the letters.

```
gap> f:= FreeGroup( "x", "x" );  gens:= GeneratorsOfGroup( f );;
<free group on the generators [ x, x ]>
gap> gens[1] = gens[2];
false
gap> f:= FreeGroup( "f1*f2", "f2^-1", "Group( [ f1, f2 ] )" );
<free group on the generators [ f1*f2, f2^-1, Group( [ f1, f2 ] ) ]>
gap> gens:= GeneratorsOfGroup( f );;
gap> gens[1]*gens[2];
f1*f2*f2^-1
gap> gens[1]/gens[3];
f1*f2*Group( [ f1, f2 ] )^-1
gap> gens[3]/gens[1]/gens[2];
Group( [ f1, f2 ] )*f1*f2^-1*f2^-1^-1
```

5 ► `AssignGeneratorVariables(` *G* `)` F

If *G* is a group, whose generators are represented by symbols (for example a free group, a finitely presented group or a pc group) this function assigns these generators to global variables with the same names.

The aim of this function is to make it easy in interactive use to work with (for example) a free group. It is a shorthand for a sequence of assignments of the form

```
var1:=GeneratorsOfGroup(G)[1];
var2:=GeneratorsOfGroup(G)[2];
...
varn:=GeneratorsOfGroup(G)[n];
```

However, since overwriting global variables can be very dangerous, **it is not permitted to use this function within a function**. (If – despite this warning – this is done, the result is undefined.)

If the assignment overwrites existing variables a warning is given, if any of the variables if write protected, or any of the generator names would not be a proper variable name, an error is raised.

35.3 Comparison of Associative Words

1 ▶ *w1* `=` *w2*

Two associative words are equal if they are words over the same alphabet and if they are sequences of the same letters. This is equivalent to saying that the external representations of the words are equal, see 35.7 and 34.2.

There is no "universal" empty word, every alphabet (that is, every family of words) has its own empty word.

```
gap> f:= FreeGroup( "a", "b", "b" );;
gap> gens:= GeneratorsOfGroup(f);
[ a, b, b ]
gap> gens[2] = gens[3];
false
gap> x:= gens[1]*gens[2];
a*b
gap> y:= gens[2]/gens[2]*gens[1]*gens[2];
a*b
gap> x = y;
true
gap> z:= gens[2]/gens[2]*gens[1]*gens[3];
a*b
gap> x = z;
false
```

2 ▶ *w1* `<` *w2*

The ordering of associative words is defined by length and lexicography (this ordering is called **short-lex** ordering), that is, shorter words are smaller than longer words, and words of the same length are compared w.r.t. the lexicographical ordering induced by the ordering of generators. Generators are sorted according to the order in which they were created. If the generators are invertible then each generator g is larger than its inverse `g^-1`, and `g^-1` is larger than every generator that is smaller than g.

```
gap> f:= FreeGroup( 2 );;  gens:= GeneratorsOfGroup( f );;
gap> a:= gens[1];;  b:= gens[2];;
gap> One(f) < a^-1;  a^-1 < a;  a < b^-1;  b^-1 < b; b < a^2;  a^2 < a*b;
true
true
true
true
true
true
```

3 ▶ `IsShortLexLessThanOrEqual(` *u*`,` *v* `)` F

returns `IsLessThanOrEqualUnder(`*ord*`,` *u*`,` *v*`)` where *ord* is the short less ordering for the family of u and v. (This is here for compatibility with GAP 4.2.)

4 ▶ `IsBasicWreathLessThanOrEqual(` *u*`,` *v* `)` F

returns `IsLessThanOrEqualUnder(`*ord*`,` *u*`,` *v*`)` where *ord* is the basic wreath product ordering for the family of u and v. (This is here for compatibility with GAP 4.2.)

35.4 Operations for Associative Words

The product of two given associative words is defined as the freely reduced concatenation of the words; so adjacent pairs of a generator and its inverse never occur in words. Besides the multiplication `*`, the arithmetical operators `One` (if the word lies in a family with identity) and (if the generators are invertible) `Inverse`, `/`,`^`, `Comm`, and `LeftQuotient` are applicable to associative words (see 30.12).

For the operation `MappedWord`, which is applicable to arbitrary words, see 34.3.1.

There are two internal representations of associative words: By letters and by syllables (see 35.6). Unless something else is specified, words are stored in the letter representation. Note, however, that operations to extract or act on parts of words (letter or syllables) can carry substantially different costs, depending on the representation the words are in.

1 ▶ `Length( w )` A

For an associative word w, `Length` returns the number of letters in w.

```
gap> f := FreeGroup("a","b");; gens := GeneratorsOfGroup(f);;
gap> a := gens[1];; b := gens[2];;w := a^5*b*a^2*b^-4*a;;
gap>  w; Length( w );  Length( a^17 );  Length( w^0 );
a^5*b*a^2*b^-4*a
13
17
0
```

2 ▶ `ExponentSumWord( w, gen )` O

For an associative word w and a generator *gen*, `ExponentSumWord` returns the number of times *gen* appears in w minus the number of times its inverse appears in w. If both *gen* and its inverse do not occur in w then 0 is returned. *gen* may also be the inverse of a generator.

```
gap> w;  ExponentSumWord( w, a );  ExponentSumWord( w, b );
a^5*b*a^2*b^-4*a
8
-3
gap> ExponentSumWord( (a*b*a^-1)^3, a );  ExponentSumWord( w, b^-1 );
0
3
```

3 ▶ `Subword( w, from, to )` O

For an associative word w and two positive integers *from* and *to*, `Subword` returns the subword of w that begins at position *from* and ends at position *to*. Indexing is done with origin 1.

```
gap> w;  Subword( w, 3, 7 );
a^5*b*a^2*b^-4*a
a^3*b*a
```

4 ▶ `PositionWord( w, sub, from )` O

Let w and *sub* be associative words, and *from* a positive integer. `PositionWord` returns the position of the first occurrence of *sub* as a subword of w, starting at position *from*. If there is no such occurrence, `fail` is returned. Indexing is done with origin 1.

In other words, `PositionWord( w, sub, from )` is the smallest integer i larger than or equal to *from* such that `Subword( w, i, i+Length( sub )-1 ) = sub`, see 35.4.3.

```
gap> w;  PositionWord( w, a/b, 1 );
a^5*b*a^2*b^-4*a
8
gap> Subword( w, 8, 9 );
a*b^-1
gap> PositionWord( w, a^2, 1 );
1
gap> PositionWord( w, a^2, 2 );
2
gap> PositionWord( w, a^2, 6 );
7
gap> PositionWord( w, a^2, 8 );
fail
```

5 ▶ `SubstitutedWord(` *w*`,` *from*`,` *to*`,` *by* `)` O
▶ `SubstitutedWord(` *w*`,` *sub*`,` *from*`,` *by* `)` O

Let w be an associative word.

In the first form, `SubstitutedWord` returns the associative word obtained by replacing the subword of w that begins at position *from* and ends at position *to* by the associative word *by*. *from* and *to* must be positive integers, indexing is done with origin 1. In other words, `SubstitutedWord(` *w*`,` *from*`,` *to*`,` *by* `)` is the product of the three words `Subword(` *w*`, 1,` *from*`-1 )`, *by*, and `Subword(` *w*`,` *to*`+1, Length(` *w* `)`, see 35.4.3.

In the second form, `SubstitutedWord` returns the associative word obtained by replacing the first occurrence of the associative word *sub* of w, starting at position *from*, by the associative word *by*; if there is no such occurrence, `fail` is returned.

```
gap> w;  SubstitutedWord( w, 3, 7, a^19 );
a^5*b*a^2*b^-4*a
a^22*b^-4*a
gap> SubstitutedWord( w, a, 6, b^7 );
a^5*b^8*a*b^-4*a
gap> SubstitutedWord( w, a*b, 6, b^7 );
fail
```

6 ▶ `EliminatedWord(` *w*`,` *gen*`,` *by* `)` O

For an associative word w, a generator *gen*, and an associative word *by*, `EliminatedWord` returns the associative word obtained by replacing each occurrence of *gen* in w by *by*.

```
gap> w;  EliminatedWord( w, a, a^2 );  EliminatedWord( w, a, b^-1 );
a^5*b*a^2*b^-4*a
a^10*b*a^4*b^-4*a^2
b^-11
```

35.5 Operations for Associative Words by their Syllables

For an associative word $w = x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k}$ over an alphabet containing $x_1, x_2, \ldots, x_k$, such that $x_i \neq x_{i+1}^{\pm 1}$ for $1 \leq i \leq k-1$, the subwords $x_i^{e_i}$ are uniquely determined; these powers of generators are called the **syllables** of w.

1 ▶ `NumberSyllables(` *w* `)` A

`NumberSyllables` returns the number of syllables of the associative word w.

2 ▶ `ExponentSyllable( w, i )` O

`ExponentSyllable` returns the exponent of the i-th syllable of the associative word w.

3 ▶ `GeneratorSyllable( w, i )` O

`GeneratorSyllable` returns the number of the generator that is involved in the i-th syllable of the associative word w.

4 ▶ `SubSyllables( w, from, to )` O

`SubSyllables` returns the subword of the associative word w that consists of the syllables from positions *from* to *to*, where *from* and *to* must be positive integers, and indexing is done with origin 1.

```
gap> w;  NumberSyllables( w );
a^5*b*a^2*b^-4*a
5
gap> ExponentSyllable( w, 3 );
2
gap> GeneratorSyllable( w, 3 );
1
gap> SubSyllables( w, 2, 3 );
b*a^2
```

There are two internal representations of associative words: By letters and by syllables (see 35.6). Unless something else is specified, words are stored in the letter representation. Note, however, that operations to extract or act on parts of words (letter or syllables) can carry substantially different costs, depending on the representation the words are in.

35.6 Representations for Associative Words

GAP provides two different internal kinds of representations of associative words. The first one are "syllable representations" in which words are stored in syllable (i.e. generator,exponent) form. (Older versions of GAP only used this representation.) The second kind are "letter representations" in which each letter in a word is represented by its index number. Negative numbers are used for inverses. Unless the syllable representation is specified explicitly when creating the free group/monoid or semigroup, a letter representation is used by default.

Depending on the task in mind, either of these two representations will perform better in time or in memory use and algorithms that are syllable or letter based (for example `GeneratorSyllable` and `Subword`) perform substantially better in the corresponding representation. For example when creating pc groups (see 44), it is advantageous to use a syllable representation while calculations in free groups usually benefit from using a letter representation.

1 ▶ `IsLetterAssocWordRep( obj )` R

A word in letter representation stores a list of generator/inverses numbers (as given by `LetterRepAssocWord`). Letter access is fast, syllable access is slow for such words.

2 ▶ `IsLetterWordsFamily( obj )` C

A letter word family stores words by default in letter form.

Internally, there are letter representations that use integers (4 Byte) to represent a generator and letter representations that use single bytes to represent a character. The latter are more memory efficient, but can only be used if there are less than 128 generators (in which case they are used by default).

3 ► `IsBLetterAssocWordRep( obj )` R
► `IsWLetterAssocWordRep( obj )` R

these two subrepresentations of `IsLetterAssocWordRep` indicate whether the word is stored as a list of bytes (in a string) or as a list of integers)

4 ► `IsBLetterWordsFamily( obj )` C
► `IsWLetterWordsFamily( obj )` C

These two subcategories of `IsLetterWordsFamily` specify the type of letter representation to be used.

5 ► `IsSyllableAssocWordRep( obj )` R

A word in syllable representation stores generator/exponents pairs (as given by `ExtRepOfObj`. Syllable access is fast, letter access is slow for such words.

6 ► `IsSyllableWordsFamily( obj )` C

A syllable word family stores words by default in syllable form.

There are also different versions of syllable representations, which compress a generator exponent pair in 8,16 or 32 bits or use a pair of integers. Internal mechanisms try to make this as memory efficient as possible.

7 ► `Is8BitsFamily( obj )` C
► `Is16BitsFamily( obj )` C
► `Is32BitsFamily( obj )` C
► `IsInfBitsFamily( obj )` C

Regardless of the internal representation used, it is possible to convert a word in a list of numbers in letter or syllable representation and vice versa:

8 ► `LetterRepAssocWord( w )` O
► `LetterRepAssocWord( w, gens )` O

The **letter representation** of an associated word is as a list of integers, each entry corresponding to a group generator. Inverses of the generators are represented by negative numbers. The generator numbers are as associated to the family.

This operation returns the letter representation of the associative word *w*.

In the second variant, the generator numbers correspond to the generator order given in the list *gens*.

(For words stored in syllable form the letter representation has to be comnputed.)

9 ► `AssocWordByLetterRep( Fam, lrep [, gens] )` O

takes a letter representation *lrep* (see `LetterRepAssocWord`, section 35.6.8) and returns an associative word in family *fam*. corresponding to this letter representation.

If *gens* is given, the numbers in the letter rerpresentation correspond to *gens*.

```
gap> w:=AssocWordByLetterRep( FamilyObj(a), [-1,2,1,-2,-2,-2,1,1,1,1]);
a^-1*b*a*b^-3*a^4
gap> LetterRepAssocWord( w^2 );
[ -1, 2, 1, -2, -2, -2, 1, 1, 1, 2, 1, -2, -2, -2, 1, 1, 1, 1 ]
```

The external representation (see section 35.7) can be used if a syllable representation is needed.

35.7 The External Representation for Associative Words

The external representation of the associative word w is defined as follows. If $w = g_{i_1}^{e_1} * g_{i_2}^{e_2} * \cdots * g_{i_k}^{e_k}$ is a word over the alphabet $g_1, g_2, \ldots$, i.e., g_i denotes the i-th generator of the family of w, then w has external representation $[i_1, e_1, i_2, e_2, \ldots, i_k, e_k]$. The empty list describes the identity element (if exists) of the family. Exponents may be negative if the family allows inverses. The external representation of an associative word is guaranteed to be freely reduced; for example, $g_1 * g_2 * g_2^{-1} * g_1$ has the external representation `[ 1, 2 ]`.

Regardless of the family preference for letter or syllable representations (see 35.6), `ExtRepOfObj` and `ObjByExtRep` can be used and interface to this "syllable"-like representation.

```
gap> w:= ObjByExtRep( FamilyObj(a), [1,5,2,-7,1,3,2,4,1,-2] );
a^5*b^-7*a^3*b^4*a^-2
gap> ExtRepOfObj( w^2 );
[ 1, 5, 2, -7, 1, 3, 2, 4, 1, 3, 2, -7, 1, 3, 2, 4, 1, -2 ]
```

35.8 Straight Line Programs

Straight line programs describe an efficient way for evaluating an abstract word at concrete generators, in a more efficient way than with `MappedWord` (see 34.3.1). For example, the associative word *ababbab* of length 7 can be computed from the generators a, b with only four multiplications, by first computing $c = ab$, then $d = cb$, and then cdc; Alternatively, one can compute $c = ab$, $e = bc$, and aee. In each step of these computations, one forms words in terms of the words computed in the previous steps.

A straight line program in GAP is represented by an object in the category `IsStraightLineProgram` (see 35.8.1) that stores a list of "lines" each of which has one of the following three forms.

1. a nonempty dense list l of integers,
2. a pair $[l, i]$ where l is a list of form 1. and i is a positive integer,
3. a list $[l_1, l_2, \ldots, l_k]$ where each l_i is a list of form 1.; this may occur only for the last line of the program.

The lists of integers that occur are interpreted as external representations of associative words (see 35.7); for example, the list $[1, 3, 2, -1]$ represents the word $g_1^3 g_2^{-1}$, with g_1 and g_2 the first and second abstract generator, respectively.

Straight line programs can be constructed using `StraightLineProgram` (see 35.8.2).

Defining attributes for straight line programs are `NrInputsOfStraightLineProgram` (see 35.8.4) and `LinesOfStraightLineProgram` (see 35.8.3). Another operation for straight line programs is `ResultOfStraightLineProgram` (see 35.8.5).

Special methods applicable to straight line programs are installed for the operations `Display`, `IsInternallyConsistent`, `PrintObj`, and `ViewObj`.

For a straight line program *prog*, the default `Display` method prints the interpretation of *prog* as a sequence of assignments of associative words; a record with components `gensnames` (with value a list of strings) and `listname` (a string) may be entered as second argument of `Display`, in this case these names are used, the default for `gensnames` is `[g1, g2, ...]`, the default for `listname` is r.

1 ▶ `IsStraightLineProgram(` *obj* `)` C

Each straight line program in GAP lies in the category `IsStraightLineProgram`.

2 ▶ `StraightLineProgram(` *lines*`[,` *nrgens*`] )` F
▶ `StraightLineProgram(` *string*`,` *gens* `)` F
▶ `StraightLineProgramNC(` *lines*`[,` *nrgens*`] )` F
▶ `StraightLineProgramNC(` *string*`,` *gens* `)` F

In the first form, *lines* must be a list of lists that defines a unique straight line program (see 35.8.1); in this case `StraightLineProgram` returns this program, otherwise an error is signalled. The optional argument

nrgens specifies the number of input generators of the program; if a line of form 1. (that is, a list of integers) occurs in *lines* except in the last position, this number is not determined by *lines* and therefore **must** be specified by the argument *nrgens*; if not then `StraightLineProgram` returns `fail`.

In the second form, *string* must be a string describing an arithmetic expression in terms of the strings in the list *gens*, where multiplication is denoted by concatenation, powering is denoted by `^`, and round brackets `(`, `)` may be used. Each entry in *gens* must consist only of (uppercase or lowercase) letters (i.e., letters in `IsAlphaChar`, see 26.3.4) such that no entry is an initial part of another one. Called with this input, `StraightLineProgramNC` returns a straight line program that evaluates to the word corresponding to *string* when called with generators corresponding to *gens*.

`StraightLineProgramNC` does the same as `StraightLineProgram`, except that the internal consistency of the program is not checked.

3 ▶ `LinesOfStraightLineProgram( prog )` A

For a straight line program *prog*, `LinesOfStraightLineProgram` returns the list of program lines. There is no default method to compute these lines if they are not stored.

4 ▶ `NrInputsOfStraightLineProgram( prog )` A

For a straight line program *prog*, `NrInputsOfStraightLineProgram` returns the number of generators that are needed as input.

If a line of form 1. (that is, a list of integers) occurs in the lines of *prog* except the last line then the number of generators is not determined by the lines, and must be set in the construction of the straight line program (see 35.8.2). So if *prog* contains a line of form 1. other than the last line and does **not** store the number of generators then `NrInputsOfStraightLineProgram` signals an error.

5 ▶ `ResultOfStraightLineProgram( prog, gens )` O

`ResultOfStraightLineProgram` evaluates the straight line program (see 35.8.1) *prog* at the group elements in the list *gens*.

The **result** of a straight line program with lines $p_1, p_2, \ldots, p_k$ when applied to *gens* is defined as follows.

(a) First a list r of intermediate results is initialized with a shallow copy of *gens*.

(b) For $i < k$, before the i-th step, let r be of length n. If p_i is the external representation of an associative word in the first n generators then the image of this word under the homomorphism that is given by mapping r to these first n generators is added to r; if p_i is a pair $[l, j]$, for a list l, then the same element is computed, but instead of being added to r, it replaces the j-th entry of r.

(c) For $i = k$, if p_k is the external representation of an associative word then the element described in (b) is the result of the program, if p_k is a pair $[l, j]$, for a list l, then the result is the element described by l, and if p_k is a list $[l_1, l_2, \ldots, l_k]$ of lists then the result is a list of group elements, where each l_i is treated as in (b).

Here are some examples.

```
gap> f:= FreeGroup( "x", "y" );;  gens:= GeneratorsOfGroup( f );;
gap> x:= gens[1];;  y:= gens[2];;
gap> prg:= StraightLineProgram( [ [] ] );
<straight line program>
gap> ResultOfStraightLineProgram( prg, [] );
[  ]
```

The above straight line program `prg` returns –for **any** list of input generators– an empty list.

```
gap> StraightLineProgram( [ [1,2,2,3], [3,-1] ] );
fail
gap> prg:= StraightLineProgram( [ [1,2,2,3], [3,-1] ], 2 );
<straight line program>
gap> LinesOfStraightLineProgram( prg );
[ [ 1, 2, 2, 3 ], [ 3, -1 ] ]
gap> prg:= StraightLineProgram( "(a^2b^3)^-1", [ "a", "b" ] );
<straight line program>
gap> LinesOfStraightLineProgram( prg );
[ [ [ 1, 2, 2, 3 ], 3 ], [ [ 3, -1 ], 4 ] ]
gap> res:= ResultOfStraightLineProgram( prg, gens );
y^-3*x^-2
gap> res = (x^2 * y^3)^-1;
true
gap> NrInputsOfStraightLineProgram( prg );
2
gap> Print( prg, "\n" );
StraightLineProgram( [ [ [ 1, 2, 2, 3 ], 3 ], [ [ 3, -1 ], 4 ] ], 2 )
gap> Display( prg );
# input:
r:= [ g1, g2 ];
# program:
r[3]:= r[1]^2*r[2]^3;
r[4]:= r[3]^-1;
# return value:
r[4]
gap> IsInternallyConsistent( StraightLineProgramNC( [ [1,2] ] ) );
true
gap> IsInternallyConsistent( StraightLineProgramNC( [ [1,2,3] ] ) );
false
gap> prg1:= StraightLineProgram( [ [1,1,2,2], [3,3,1,1] ], 2 );;
gap> prg2:= StraightLineProgram( [ [ [1,1,2,2], 2 ], [2,3,1,1] ] );;
gap> res1:= ResultOfStraightLineProgram( prg1, gens );
x*y^2*x*y^2*x*y^2*x
gap> res1 = (x*y^2)^3*x;
true
gap> res2:= ResultOfStraightLineProgram( prg2, gens );
x*y^2*x*y^2*x*y^2*x
gap> res2 = (x*y^2)^3*x;
true
gap> prg:= StraightLineProgram( [ [2,3], [ [3,1,1,4], [1,2,3,1] ] ], 2 );;
gap> res:= ResultOfStraightLineProgram( prg, gens );
[ y^3*x^4, x^2*y^3 ]
```

6 ▶ `StringOfResultOfStraightLineProgram(` *prog*, *gensnames*`[, "LaTeX"] )` F

`StringOfResultOfStraightLineProgram` returns a string that describes the result of the straight line program (see 35.8.1) *prog* as word(s) in terms of the strings in the list *gensnames*. If the result of *prog* is a single element then the return value of `StringOfResultOfStraightLineProgram` is a string consisting of the entries of *gensnames*, opening and closing brackets `(` and `)`, and powering by integers via `^`. If the result of *prog* is a list of elements then the return value of `StringOfResultOfStraightLineProgram` is a comma separated concatenation of the strings of the single elements, enclosed in square brackets `[`, `]`.

```
gap> prg:= StraightLineProgram( [ [ 1, 2, 2, 3 ], [ 3, -1 ] ], 2 );;
gap> StringOfResultOfStraightLineProgram( prg, [ "a", "b" ] );
"(a^2b^3)^-1"
gap> StringOfResultOfStraightLineProgram( prg, [ "a", "b" ], "LaTeX" );
"(a^{2}b^{3})^{-1}"
```

7 ▶ `CompositionOfStraightLinePrograms(` *prog2*`,` *prog1* `)` F

For two straight line programs *prog1* and *prog2*, `CompositionOfStraightLinePrograms` returns a straight line program *prog* with the properties that *prog1* and *prog* have the same number of inputs, and the result of *prog* when applied to given generators *gens* equals the result of *prog2* when this is applied to the output of *prog1* applied to *gens*.

(Of course the number of outputs of *prog1* must be the same as the number of inputs of *prog2*.)

```
gap> prg1:= StraightLineProgram( "a^2b", [ "a","b" ] );;
gap> prg2:= StraightLineProgram( "c^5", [ "c" ] );;
gap> comp:= CompositionOfStraightLinePrograms( prg2, prg1 );
<straight line program>
gap> StringOfResultOfStraightLineProgram( comp, [ "a", "b" ] );
"(a^2b)^5"
gap> prg:= StraightLineProgram( [ [2,3], [ [3,1,1,4], [1,2,3,1] ] ], 2 );;
gap> StringOfResultOfStraightLineProgram( prg, [ "a", "b" ] );
"[ b^3a^4, a^2b^3 ]"
gap> comp:= CompositionOfStraightLinePrograms( prg, prg );
<straight line program>
gap> StringOfResultOfStraightLineProgram( comp, [ "a", "b" ] );
"[ (a^2b^3)^3(b^3a^4)^4, (b^3a^4)^2(a^2b^3)^3 ]"
```

8 ▶ `IntegratedStraightLineProgram(` *listofprogs* `)` F

For a nonempty dense list *listofprogs* of straight line programs that have the same number n, say, of inputs (see 35.8.4) and for which the results (see 35.8.5) are single elements (i.e., **not** lists of elements), `IntegratedStraightLineProgram` returns a straight line program *prog* with n inputs such that for each n-tuple *gens* of generators, `ResultOfStraightLineProgram(` *prog*`,` *gens* `)` is equal to the list `List(` *listofprogs*`,` *p* `->` `ResultOfStraightLineProgram(` *p*`,` *gens* `)`.

```
gap> f:= FreeGroup( "x", "y" );;  gens:= GeneratorsOfGroup( f );;
gap> prg1:= StraightLineProgram( [ [ [ 1, 2 ], 1 ], [ 1, 2, 2, -1 ] ], 2 );;
gap> prg2:= StraightLineProgram( [ [ [ 2, 2 ], 3 ], [ 1, 3, 3, 2 ] ], 2 );;
gap> prg3:= StraightLineProgram( [ [ 2, 2 ], [ 1, 3, 3, 2 ] ], 2 );;
gap> prg:= IntegratedStraightLineProgram( [ prg1, prg2, prg3 ] );;
gap> ResultOfStraightLineProgram( prg, gens );
[ x^4*y^-1, x^3*y^4, x^3*y^4 ]
gap> prg:= IntegratedStraightLineProgram( [ prg2, prg3, prg1 ] );;
gap> ResultOfStraightLineProgram( prg, gens );
[ x^3*y^4, x^3*y^4, x^4*y^-1 ]
gap> prg:= IntegratedStraightLineProgram( [ prg3, prg1, prg2 ] );;
gap> ResultOfStraightLineProgram( prg, gens );
[ x^3*y^4, x^4*y^-1, x^3*y^4 ]
```

35.9 Straight Line Program Elements

When computing with very large (in terms of memory) elements, for example permutations of degree a few hundred thousands, it can be helpful (in terms of memory usage) to represent them via straight line programs in terms of an original generator set. (So every element takes only small extra storage for the straight line program.)

A straight line program element has a **seed** (a list of group elements) and a straight line program on the same number of generators as the length of this seed, its value is the value of the evaluated straight line program.

At the moment, the entries of the straight line program have to be simple lists (i.e. of the first form).

Straight line program elements are in the same categories and families as the elements of the seed, so they should work together with existing algorithms.

Note however, that due to the different way of storage some normally very cheap operations (such as testing for element equality) can become more expensive when dealing with straight line program elements. This is essentially the tradeoff for using less memory.

1 ▶ `IsStraightLineProgElm(` *obj* `)` R

A straight line program element is a group element given (for memory reasons) as a straight line program. Straight line program elements are positional objects, the first component is a record with a component `seeds`, the second component the straight line program. we need to rank higher than default methods

2 ▶ `StraightLineProgElm(` *seed*`,` *prog* `)` F

Creates a straight line program element for seed *seed* and program *prog*.

3 ▶ `StraightLineProgGens(` *gens*`[,` *base*`] )` F

returns a set of straight line program elements corresponding to the generators in *gens*. If *gens* is a set of permutations then *base* can be given which must be a base for the group generated by *gens*. (Such a base will be used to speed up equality tests.)

4 ▶ `EvalStraightLineProgElm(` *slpel* `)` F

evaluates a straight line program element *slpel* from its seeds.

5 ▶ `StretchImportantSLPElement(` *elm* `)` O

If *elm* is a straight line program element whose straight line representation is very long, this operation changes the representation of *elm* to a straight line program element, equal to *elm*, whose seed contains the evaluation of *elm* and whose straight line program has length 1.

For other objects nothing happens.

This operation permits to designate "important" elements within an algorithm (elements that wil be referred to often), which will be represented by guaranteed short straight line program elements.

```
gap> gens:=StraightLineProgGens([(1,2,3,4),(1,2)]);
[ <[ [ 2, 1 ] ]|(1,2,3,4)>, <[ [ 1, 1 ] ]|(1,2)> ]
gap> g:=Group(gens);;
gap> (gens[1]^3)^gens[2];
<[ [ 1, -1, 2, 3, 1, 1 ] ]|(1,2,4,3)>
gap> Size(g);
24
gap> Random(g);
<[ [ 1, -1, 2, -1, 1, 1, 2, -1, 1, -1, 2, 1, 1, 1, 2, 1, 1, -1, 2, 2, 1, 1 ],
  [ 3, -2, 2, -2, 1, -1, 2, -2, 1, 1 ], [ 4, 1, 2, -1, 4, 1 ] ]|(1,2,3,4)>
```

See also Section 41.12.

36 Rewriting Systems

Rewriting systems in GAP are a framework for dealing with the very general task of rewriting elements of a free (or **term**) algebra in some normal form. Although most rewriting systems currently in use are **string rewriting systems** (where the algebra has only one binary operation which is associative) the framework in GAP is general enough to encompass the task of rewriting algebras of any signature from groups to semirings.

Rewriting systems are already implemented in GAP for finitely presented semigroups and for pc groups. The use of these particular rewriting systems is described in the corresponding chapters. We describe here only the general framework of rewriting systems with a particular emphasis on material which would be helpful for a developer implementing a rewriting system.

We fix some definitions and terminology for the rest of this chapter. Let T be a term algebra in some signature. A **term rewriting system** for T is a set of ordered pairs of elements of T of the form *(l, r)*. Viewed as a set of relations, the rewriting system determines a presentation for a quotient algebra A of T.

When we take into account the fact that the relations are expressed as **ordered** pairs, we have a way of **reducing** the elements of T. Suppose an element u of T has a subword l and *(l,r)* is a rule of the rewriting system, then we can replace the subterm l of u by the term r and obtain a new word v. We say that we have **rewritten** u as v. Note that u and v represent the same element of A. If u can not be rewritten using any rule of the rewriting system we sat that u is **reduced**.

36.1 Operations on rewriting systems

1 ▶ `IsRewritingSystem(` *obj* `)` C

This is the category in which all rewriting systems lie.

2 ▶ `Rules(` *rws* `)` A

The rules comprising the rewriting system. Note that these may change through the life of the rewriting system, however they will always be a set of defining relations of the algebra described by the rewriting system.

3 ▶ `OrderOfRewritingSystem(` *rws* `)` A
▶ `OrderingOfRewritingSystem(` *rws* `)` A

return the ordering of the rewriting system *rws*.

4 ▶ `ReducedForm(` *rws*`,` *u* `)` O

Given an element u in the free (or term) algebra over which *rws* is defined, rewrite u by successive applications of the rules of *rws* until no further rewriting is possible, and return the resulting element of T.

5 ▶ `IsConfluent(` *rws* `)` P
▶ `IsConfluent(` *A* `)` P

return `true` if and only if the rewriting system *rws* is confluent. A rewriting system is **confluent** if, for every two words u and v in the free algebra T which represent the same element of the algebra A defined

by *rws*, `ReducedForm(`*rws*`,`*u*`) = ReducedForm(`*rws*`,`*v*`)` as words in the free algebra T. This element is the **unique normal form** of the element represented by u.

In its second form, if A is an algebra with a canonical rewriting system associated with it, `IsConfluent` checks whether that rewriting system is confluent.

Also see 44.4.7.

6 ► `ConfluentRws(` *rws* `)` A

Return a new rewriting system defining the same algebra as *rws* which is confluent.

7 ► `IsReduced(` *rws* `)` P

A rewriting system is reduced if for each rule (l, r), l and r are both reduced.

8 ► `ReduceRules(` *rws* `)` O

Reduce rules and remove redundant rules to make *rws* reduced.

9 ► `AddRule(` *rws*`,` *rule* `)` O

Add *rule* to a rewriting system *rws*.

10 ► `AddRuleReduced(` *rws*`,` *rule* `)` O

Add *rule* to rewriting system *rws*. Performs a reduction operation on the resulting system, so that if *rws* is reduced it will remain reduced.

11 ► `MakeConfluent(` *rws* `)` O

Add rules (and perhaps reduce) in order to make *rws* confluent

12 ► `GeneratorsOfRws(` *rws* `)` A

13 ► `AddGenerators(` *rws*`,` *gens* `)` O

36.2 Operations on elements of the algebra

In this section let u denote an element of the term algebra T representing *[u]* in the quotient algebra A.

1 ► `ReducedProduct(` *rws*`,` *u*`,` *v* `)` O

The result is w where *[w] = [u][v]* in A and w is in reduced form.

The remaining operations are defined similarly when they are defined (as determined by the signature of the term algebra).

2 ► `ReducedSum(` *rws*`,` *left*`,` *right* `)` O

3 ► `ReducedOne(` *rws* `)` O

4 ► `ReducedAdditiveInverse(` *rws*`,` *obj* `)` O

5 ► `ReducedComm(` *rws*`,` *left*`,` *right* `)` O

6 ► `ReducedConjugate(` *rws*`,` *left*`,` *right* `)` O

7 ► `ReducedDifference(` *rws*`,` *left*`,` *right* `)` O

8 ► `ReducedInverse(` *rws*`,` *obj* `)` O

9 ► `ReducedLeftQuotient(` *rws*`,` *left*`,` *right* `)` O

10 ► `ReducedPower(` *rws*`,` *obj*`,` *pow* `)` O

11 ► `ReducedQuotient(` *rws*`,` *left*`,` *right* `)` O

12 ► `ReducedScalarProduct(` *rws*`,` *left*`,` *right* `)` O

13 ► `ReducedZero(` *rws* `)` O

36.3 Properties of rewriting systems

The following properties may be used to identify the type of term algebra over which the rewriting system is defined.

1 ▶ `IsBuiltFromAdditiveMagmaWithInverses( obj )` P

2 ▶ `IsBuiltFromMagma( obj )` P

3 ▶ `IsBuiltFromMagmaWithOne( obj )` P

4 ▶ `IsBuiltFromMagmaWithInverses( obj )` P

5 ▶ `IsBuiltFromSemigroup( obj )` P

6 ▶ `IsBuiltFromGroup( obj )` P

36.4 Rewriting in Groups and Monoids

One application of rewriting is to reduce words in finitely presented groups and monoids. The rewriting system still has to be built for a finitely presented monoid (using `IsomorphismFpMonoid` for conversion). Rewriting then can take place for words in the underlying free monoid. (These can be obtained from monoid elements with the command `UnderlyingElement`.)

```
gap> f:=FreeGroup(3);;
gap> rels:=[f.1*f.2^2/f.3,f.2*f.3^2/f.1,f.3*f.1^2/f.2];;
gap> g:=f/rels;
<fp group on the generators [ f1, f2, f3 ]>
gap> mhom:=IsomorphismFpMonoid(g);
MappingByFunction( <fp group on the generators
[ f1, f2, f3 ]>, <fp monoid on the generators
[ f1^-1, f1, f2^-1, f2, f3^-1, f3
 ]>, function( x ) ... end, function( x ) ... end )
gap> mon:=Image(mhom);
<fp monoid on the generators [ f1^-1, f1, f2^-1, f2, f3^-1, f3 ]>
gap> k:=KnuthBendixRewritingSystem(mon);
Knuth Bendix Rewriting System for Monoid( [ f1^-1, f1, f2^-1, f2, f3^-1, f3
 ], ... ) with rules
[ [ f1^-1*f1, <identity ...> ], [ f1*f1^-1, <identity ...> ],
  [ f2^-1*f2, <identity ...> ], [ f2*f2^-1, <identity ...> ],
  [ f3^-1*f3, <identity ...> ], [ f3*f3^-1, <identity ...> ],
  [ f1*f2^2*f3^-1, <identity ...> ], [ f2*f3^2*f1^-1, <identity ...> ],
  [ f3*f1^2*f2^-1, <identity ...> ] ]
gap> MakeConfluent(k);
gap> a:=Product(GeneratorsOfMonoid(mon));
f1^-1*f1*f2^-1*f2*f3^-1*f3
gap> ReducedForm(k,UnderlyingElement(a));
<identity ...>
```

To rewrite a word in the finitely presented group, one has to convert it to a word in the monoid first, rewrite in the underlying free monoid and convert back (by forming first again an element of the fp monoid) to the finitely presented group.

```
gap> r:=PseudoRandom(g);;
gap> Length(r);
3704
gap> red:=ReducedForm(k,UnderlyingElement(melm));
f1^-1^3*f2^-1*f1^2
gap> melm:=ElementOfFpMonoid(FamilyObj(One(mon)),red);
f1^-1^3*f2^-1*f1^2
gap> gpelm:=PreImagesRepresentative(mhom,melm);
f1^-3*f2^-1*f1^2
gap> r=gpelm;
true
gap> CategoriesOfObject(red);
[ "IsExtLElement", "IsExtRElement", "IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne", "IsAssociativeElement", "IsWord" ]
gap> CategoriesOfObject(melm);
[ "IsExtLElement", "IsExtRElement", "IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne", "IsAssociativeElement",
  "IsElementOfFpMonoid" ]
gap> CategoriesOfObject(gpelm);
[ "IsExtLElement", "IsExtRElement", "IsMultiplicativeElement",
  "IsMultiplicativeElementWithOne", "IsMultiplicativeElementWithInverse",
  "IsAssociativeElement", "IsElementOfFpGroup" ]
```

Note, that the elements `red` (free monoid) `melm` (fp monoid) and `gpelm` (group) differ, though they are displayed identically.

Under Unix, it is possible to use the `kbmag` package to replace the built-in rewriting by this packages efficient C implementation. You can do this (after loading the `kbmag` package) by assigning the variable `KB_REW` to `KBMAG_REW`. Assignment to `GAPKB_REW` reverts to the built-in implementation. (See section 51.5.2.)

```
gap> LoadPackage("kbmag");
true
gap> KB_REW:=KBMAG_REW;;
```

36.5 Developing rewriting systems

The key point to note about rewriting systems is that they have properties such as `IsConfluent` and attributes such as `Rules`, however they are rarely stored, but rather computed afresh each time they are asked for, from data stored in the private members of the rewriting system object. This is because a rewriting system often evolves through a session, starting with some rules which define the algebra A as relations, and then adding more rules to make the system confluent. For example, in the case of Knuth-Bendix rewriting systems (see Chapter 51), the function `CreateKnuthBendixRewritingSystem` creating the rewriting system (in `kbsemi.gi`) uses

```
kbrws := Objectify(NewType(rwsfam,
  IsMutable and IsKnuthBendixRewritingSystem and
  IsKnuthBendixRewritingSystemRep),
  rec(family:= fam,
  reduced:=false,
  tzrules:=List(relwco,i->
   [LetterRepAssocWord(i[1]),LetterRepAssocWord(i[2])]),
  pairs2check:=CantorList(Length(r)),
  ordering:=wordord,
```

```
freefam:=freefam));
```

In particular, since we don't use the filter `IsAttributeStoringRep` in the `Objectify`, whenever `IsConfluent` is called, the appropriate method to determine confluence is called.

37 Groups

This chapter explains how to create groups and defines operations for groups, that is operations whose definition does not depend on the representation used. However methods for these operations in most cases will make use of the representation.

If not otherwise specified, in all examples in this chapter the group `g` will be the symmetric group S_4 acting on the letters $\{1, \ldots, 4\}$.

37.1 Group Elements

Groups in GAP are written multiplicatively. The elements from which a group can be generated must permit multiplication and multiplicative inversion (see 30.14).

```
gap> a:=(1,2,3);;b:=(2,3,4);;
gap> One(a);
()
gap> Inverse(b);
(2,4,3)
gap> a*b;
(1,3)(2,4)
gap> Order(a*b);
2
gap> Order( [ [ 1, 1 ], [ 0, 1 ] ] );
infinity
```

The next example may run into an infinite loop because the given matrix in fact has infinite order.

```
gap> Order( [ [ 1, 1 ], [ 0, 1 ] ] * Indeterminate( Rationals ) );
#I  Order: warning, order of <mat> might be infinite
```

Since groups are domains, the recommended command to compute the order of a group is `Size` (see 28.3.6). For convenience, group orders can also be computed with `Order`.

The operation `Comm` (see 30.12.3) can be used to compute the commutator of two elements, the operation `LeftQuotient` (see 30.12.2) computes the product $x^{-1}y$.

37.2 Creating Groups

When groups are created from generators, this means that the generators must be elements that can be multiplied and inverted (see also 30.3). For creating a free group on a set of symbols, see 35.2.1.

1 ▶ `Group(` *gen*`, ... )` F
▶ `Group(` *gens* `)` F
▶ `Group(` *gens*`,` *id* `)` F

`Group(` *gen*`, ... )` is the group generated by the arguments *gen*, ...

If the only argument *gens* is a list that is not a matrix then `Group( gens )` is the group generated by the elements of that list.

If there are two arguments, a list *gens* and an element *id*, then `Group( gens, id )` is the group generated by the elements of *gens*, with identity *id*.

Note that the value of the attribute `GeneratorsOfGroup` need not be equal to the list *gens* of generators entered as argument. Use `GroupWithGenerators` (see 37.2.2) if you want to be sure that the argument *gens* is stored as value of `GeneratorsOfGroup`.

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
```

2 ▶ `GroupWithGenerators( gens )` O
▶ `GroupWithGenerators( gens, id )` O

`GroupWithGenerators` returns the group G generated by the list *gens*. If a second argument *id* is present then this is stored as the identity element of the group. The value of the attribute `GeneratorsOfGroup` of G is equal to *gens*.

3 ▶ `GeneratorsOfGroup( G )` A

returns a list of generators of the group G. If G has been created by the command `GroupWithGenerators` (see 37.2.2), with argument *gens*, then the list returned by `GeneratorsOfGroup` will be equal to *gens*.

```
gap> g:=GroupWithGenerators([(1,2,3,4),(1,2)]);
Group([ (1,2,3,4), (1,2) ])
gap> GeneratorsOfGroup(g);
[ (1,2,3,4), (1,2) ]
```

While in this example GAP displays the group via the generating set stored in the attribute `GeneratorsOfGroup`, the methods installed for `View` (see 6.2.1) will in general display only some information about the group which may even be just the fact that it is a group.

4 ▶ `AsGroup( D )` A

if the elements of the collection D form a group the command returns this group, otherwise it returns `fail`.

```
gap> AsGroup([(1,2)]);
fail
gap> AsGroup([(),(1,2)]);
Group([ (1,2) ])
```

5 ▶ `ConjugateGroup( G, obj )` O

returns the conjugate group of G, obtained by applying the conjugating element *obj*. To form a conjugate (group) by any object acting via `^`, one can use the infix operator `^`.

```
gap> ConjugateGroup(g,(1,5));
Group([ (2,3,4,5), (2,5) ])
```

6 ▶ `IsGroup( obj )` C

A group is a magma-with-inverses (see 33.1.4) and associative (see 33.4.7) multiplication.

`IsGroup` tests whether the object *obj* fulfills these conditions, it does **not** test whether *obj* is a set of elements that forms a group under multiplication; use `AsGroup` (see 37.2.4) if you want to perform such a test. (See 13.3 for details about categories.)

```
gap> IsGroup(g);
true
```

7 ▶ `InfoGroup` V

is the info class for the generic group theoretic functions (see 7.4).

37.3 Subgroups

For the general concept of parents and subdomains, see 30.7 and 30.8. More functions that construct certain subgroups can be found in the sections 37.11, 37.12, 37.13, and 37.14.

1 ▶ `Subgroup( G, gens )` F
▶ `SubgroupNC( G, gens )` F

creates the subgroup U of G generated by *gens*. The `Parent` of U will be G. The `NC` version does not check, whether the elements in *gens* actually lie in G.

```
gap> u:=Subgroup(g,[(1,2,3),(1,2)]);
Group([ (1,2,3), (1,2) ])
```

2 ▶ `Index( G, U )` O
▶ `IndexNC( G, U )` O

For a subgroup U of the group G, `Index` returns the index $[G : U] = \frac{|G|}{|U|}$ of U in G. The `NC` version does not test whether U is contained in G.

```
gap> Index(g,u);
4
```

3 ▶ `IndexInWholeGroup( G )` A

If the family of elements of G itself forms a group P, this attribute returns the index of G in P.

4 ▶ `AsSubgroup( G, U )` O

creates a subgroup of G which contains the same elements as U

```
gap> v:=AsSubgroup(g,Group((1,2,3),(1,4)));
Group([ (1,2,3), (1,4) ])
gap> Parent(v);
Group([ (1,2,3,4), (1,2) ])
```

5 ▶ `IsSubgroup( G, U )` F

`IsSubgroup` returns `true` if U is a group that is a subset of the domain G. This is actually checked by calling `IsGroup( U )` and `IsSubset( G, U )`; note that special methods for `IsSubset` (see 28.4.1) are available that test only generators of U if G is closed under the group operations. So in most cases, for example whenever one knows already that U is a group, it is better to call only `IsSubset`.

```
gap> IsSubgroup(g,u);
true
gap> v:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> u=v;
true
gap> IsSubgroup(g,v);
true
```

6 ▶ `IsNormal( G, U )` O

returns `true` if the group G normalizes the group U and `false` otherwise.

A group G **normalizes** a group U if and only if for every $g \in G$ and $u \in U$ the element u^g is a member of U. Note that U need not be a subgroup of G.

```
gap> IsNormal(g,u);
false
```

7 ▶ `IsCharacteristicSubgroup( G, N )` O

tests whether N is invariant under all automorphisms of G.

```
gap> IsCharacteristicSubgroup(g,u);
false
```

8 ▶ `ConjugateSubgroup( G, g )` O

9 ▶ `ConjugateSubgroups( G, U )` O

returns a list of all images of the group U under conjugation action by G.

10 ▶ `IsSubnormal( G, U )` O

A subgroup U of the group G is subnormal if it is contained in a subnormal series of G.

```
gap> IsSubnormal(g,Group((1,2,3)));
false
gap> IsSubnormal(g,Group((1,2)(3,4)));
true
```

If a group U is created as a subgroup of another group G, G becomes the parent of U. There is no `universal` parent group, parent-child chains can be arbitrary long. GAP stores the result of some operations (such as `Normalizer`) with the parent as an attribute.

11 ▶ `SubgroupByProperty( G, prop )` F

creates a subgroup of G consisting of those elements fulfilling *prop* (which is a tester function). No test is done whether the property actually defines a subgroup.

12 ▶ `SubgroupShell( G )` F

creates a subgroup of G which at this point is not yet specified further (but will be later, for example by assigning a generating set).

```
gap> u:=SubgroupByProperty(g,i->3^i=3);
<subgrp of Group([ (1,2,3,4), (1,2) ]) by property>
gap> (1,3) in u; (1,4) in u; (1,5) in u;
false
true
false
gap> GeneratorsOfGroup(u);
[ (1,2), (1,4,2) ]
gap> u:=SubgroupShell(g);
<group>
```

37.4 Closures of (Sub)groups

1 ▶ `ClosureGroup( G, obj )` O

creates the group generated by the elements of G and *obj*. *obj* can be either an element or a collection of elements, in particular another group.

```
gap> g:=SmallGroup(24,12);;u:=Subgroup(g,[g.3,g.4]);
Group([ f3, f4 ])
gap> ClosureGroup(u,g.2);
Group([ f2, f3, f4 ])
gap> ClosureGroup(u,[g.1,g.2]);
Group([ f1, f2, f3, f4 ])
gap> ClosureGroup(u,Group(g.2*g.1));
Group([ f1*f2^2, f3, f4 ])
```

2 ▶ `ClosureGroupAddElm( G, elm )` F
▶ `ClosureGroupCompare( G, elm )` F
▶ `ClosureGroupIntest( G, elm )` F

These three functions together with `ClosureGroupDefault` implement the main methods for `ClosureGroup` (see 37.4.1). In the ordering given, they just add *elm* to the generators, remove duplicates and identity elements, and test whether *elm* is already contained in G.

3 ▶ `ClosureGroupDefault( G, elm )` F

This functions returns the closure of the group G with the element *elm*. If G has the attribute `AsSSortedList` then also the result has this attribute. This is used to implement the default method for `Enumerator` (see 28.2.2) and `EnumeratorSorted` (see 28.2.3).

4 ▶ `ClosureSubgroup( G, obj )` F
▶ `ClosureSubgroupNC( G, obj )` F

For a group G that stores a parent group (see 30.7), `ClosureSubgroup` calls `ClosureGroup` (see 37.4.1) with the same arguments; if the result is a subgroup of the parent of G then the parent of G is set as parent of the result, otherwise an error is raised. The check whether the result is contained in the parent of G is omitted by the `NC` version. As a wrong parent might imply wrong properties this version should be used with care.

37.5 Expressing Group Elements as Words in Generators

Using homomorphisms (see chapter 38) is is possible to express group elements as words in given generators: Create a free group (see 35.2.1) on the correct number of generators and create a homomorphism from this free group onto the group G in whose generators you want to factorize. Then the preimage of an element of G is a word in the free generators, that will map on this element again. The following example shows how to decompose elements of S_4 in the generators `(1,2,3,4)` and `(1,2)`:

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> f:=FreeGroup("x","y");
<free group on the generators [ x, y ]>
gap> hom:=GroupHomomorphismByImagesNC(f,g,GeneratorsOfGroup(f),
> GeneratorsOfGroup(g));
[ x, y ] -> [ (1,2,3,4), (1,2) ]
gap> PreImagesRepresentative(hom,(1,4));
y^-1*x^-2*y^-1*x^-1*y^-1*x
```

The following example stems from a real request to the GAP Forum. In September 2000 a GAP user working with puzzles wanted to express the permutation `(1,2)` as a word as short as possible in particular generators of the symmetric group S_{16}.

```
gap> perms := [ (1,2,3,7,11,10,9,5), (2,3,4,8,12,11,10,6),
>   (5,6,7,11,15,14,13,9), (6,7,8,12,16,15,14,10) ];;
gap> puzzle := Group( perms );;
gap> Size( puzzle );
20922789888000
gap> F := FreeGroup( "a", "b", "c", "d" );;
gap> gens := GeneratorsOfGroup( F );;
gap> hom := GroupHomomorphismByImages( F, puzzle, gens, perms );;
gap> word := PreImagesRepresentative( hom, (1,2) );;
gap> Length( word );
13
```

1 ▶ `Factorization( `*G*`, `*elm*` )` F

returns a factorization of *elm* as word in the generators of G given in the attribute `GeneratorsOfGroup`. The component G`!.factFreeMap` will contain a map *map* from the group G to the free group in which the word is expressed. The attribute `MappingGeneratorsImages` of this map gives a list of generators and corresponding letters.

The algorithm used computes all elements of the group to ensure a short word is found. Therefore this function should **not** be used when the group G has more than a few thousand elements. Because of this, one should not call this function within algorithms, but use homomorphisms instead.

```
gap> G:=SymmetricGroup( 6 );;
gap> r:=(3,4);; s:=(1,2,3,4,5,6);;
gap> # create a subgroup to force the system to use the generators r and s.
gap> H:= Subgroup(G, [ r, s ] );
Group([ (3,4), (1,2,3,4,5,6) ])
gap> Factorization( H, (1,2,3) );
x2*x1*x2*x1*x2^-2
gap> s*r*s*r*s^-2;
(1,2,3)
gap> MappingGeneratorsImages(H!.factFreeMap);
[ [ (3,4), (1,2,3,4,5,6) ], [ x1, x2 ] ]
```

37.6 Structure Descriptions

1 ▶ `StructureDescription( `*G*` )` A

The method for `StructureDescription` exhibits the structure of the given group to some extend using the strategy outlined below. The idea is to return a possibly short string which gives some insight in the structure of the considered group and can be computed reasonably quickly.

Note that non-isomorphic groups can have the same `StructureDescription`, since the structure description might not exhibit the structure of the considered group in all detail. However, isomorphic groups in different representation will always obtain the same structure description.

The `StructureDescription` is a string of the following form:

```
StructureDescription(<G>) ::=
   1                                ; trivial group
 | C<size>                          ; cyclic group
 | A<degree>                        ; alternating group
 | S<degree>                        ; symmetric group
 | D<size>                          ; dihedral group
 | Q<size>                          ; quaternion group
 | QD<size>                         ; quasidihedral group
 | PSL(<n>,<q>)                     ; projective special linear group
 | SL(<n>,<q>)                      ; special linear group
 | GL(<n>,<q>)                      ; general linear group
 | PSU(<n>,<q>)                     ; proj. special unitary group
 | O(2<n>+1,<q>)                    ; orthogonal group, type B
 | O+(2<n>,<q>)                     ; orthogonal group, type D
 | O-(2<n>,<q>)                     ; orthogonal group, type 2D
 | PSp(2<n>,<q>)                    ; proj. special symplectic group
 | Sz(<q>)                          ; Suzuki group
 | Ree(<q>)                         ; Ree group (type 2F or 2G)
 | E(6,<q>) | E(7,<q>) | E(8,<q>)   ; Lie group of exceptional type
 | 2E(6,<q>) | F(4,<q>) | G(2,<q>)
 | 3D(4,<q>)                        ; Steinberg triality group
 | M11 | M12 | M22 | M23 | M24
 | J1 | J2 | J3 | J4 | Co1 | Co2
 | Co3 | Fi22 | Fi23 | Fi24' | Suz
 | HS | McL | He | HN | Th | B
 | M | ON | Ly | Ru                  ; sporadic simple group
 | 2F(4,2)'                         ; Tits group
 | PerfectGroup(<size>,<id>)        ; the indicated group from the
                                    ; library of perfect groups
 | A x B                            ; direct product
 | N : H                            ; semidirect product
 | C(G) . G/C(G) = G' . G/G'        ; non-split extension
                                    ; (equal alternatives and
                                    ; trivial extensions omitted)
 | Phi(G) . G/Phi(G)                ; non-split extension:
                                    ; Frattini subgroup and
                                    ; Frattini factor group
```

Note that the method chooses **one** possible way of building up the given group from smaller pieces (others are possible too).

The option "short" is recognized – if this option is set, an abbreviated output format is used (e.g. `"6x3"` instead of `"C6 x C3"`).

If the `Name` attribute is not bound, but `StructureDescription` is, `View` prints the value of the attribute `StructureDescription`. The `Print`ed representation of a group is not affected by computing a `StructureDescription`.

The strategy is

1. Lookup in precomputed list, if the order of G is not larger than 100 and not equal to 64.
2. If G is abelian: decompose it into cyclic factors in "elementary divisors style", e.g. `"C2 x C3 x C3"` is `"C6 x C3"`.

3. Recognize alternating groups, symmetric groups, dihedral groups, quasidihedral groups, quaternion groups, PSL's, SL's, GL's and simple groups not listed so far as basic building blocks.
4. Decompose into a direct product of irreducible factors.
5. Recognize semidirect products (N:H), where N is normal. Select a pair N, H with the following preferences:

1. H is abelian
2. N is abelian

2a. N has many abelian invariants

3. N is a direct product

3a. N has many direct factors

4. $\phi : H \to \mathrm{Aut}(N)$, $h \mapsto (n \mapsto n^h)$ is injective.

6. Fall back to non-splitting extensions: If the centre or the commutator factor group is non-trivial, write G as $\mathrm{Z}(G).G/\mathrm{Z}(G)$ resp. $G'.G/G'$. Otherwise if the Frattini subgroup is non-trivial, write G as $\Phi(G).G/\Phi(G)$.
7. If no decomposition is found (maybe this is not the case for any finite group) try to identify G in the perfect groups library. If also this fails return a string describing this situation.

```
gap> l := AllSmallGroups(12);;
gap> List(l,StructureDescription);; l;
[ C3 : C4, C12, A4, D12, C6 x C2 ]
gap> List(AllSmallGroups(40),G->StructureDescription(G:short));
[ "5:8", "40", "5:8", "5:Q8", "4xD10", "D40", "2x(5:4)", "(10x2):2", "20x2",
  "5xD8", "5xQ8", "2x(5:4)", "2^2xD10", "10x2^2" ]
gap> List(AllTransitiveGroups(DegreeAction,6),G->StructureDescription(G:short));
[ "6", "S3", "D12", "A4", "3xS3", "2xA4", "S4", "S4", "S3xS3", "(3^2):4",
  "2xS4", "A5", "(S3xS3):2", "S5", "A6", "S6" ]
gap> StructureDescription(PSL(4,2));
"A8"
```

37.7 Cosets

1 ▶ `RightCoset( U, g )` O

returns the right coset of U with representative g, which is the set of all elements of the form ug for all $u \in U$. g must be an element of a larger group G which contains U. For element operations such as `in` a right coset behaves like a set of group elements.

Right cosets are external orbits for the action of U which acts via `OnLeftInverse`. Of course the action of a larger group G on right cosets is via `OnRight`.

```
gap> u:=Group((1,2,3), (1,2));;
gap> c:=RightCoset(u,(2,3,4));
RightCoset(Group( [ (1,2,3), (1,2) ] ),(2,3,4))
gap> ActingDomain(c);
Group([ (1,2,3), (1,2) ])
gap> Representative(c);
(2,3,4)
gap> Size(c);
6
```

```
gap> AsList(c);
[ (2,3,4), (1,4,2), (1,3)(2,4), (2,4), (1,4,2,3), (1,3,4,2) ]
```

2 ▶ `RightCosets( G, U )` F
▶ `RightCosetsNC( G, U )` O

computes a duplicate free list of right cosets Ug for $g \in G$. A set of representatives for the elements in this list forms a right transversal of U in G. (By inverting the representatives one obtains a list of representatives of the left cosets of U.) The NC version does not check whether U is a subgroup of G.

```
gap> RightCosets(g,u);
[ RightCoset(Group( [ (1,2,3), (1,2) ] ),()),
  RightCoset(Group( [ (1,2,3), (1,2) ] ),(1,3)(2,4)),
  RightCoset(Group( [ (1,2,3), (1,2) ] ),(1,4)(2,3)),
  RightCoset(Group( [ (1,2,3), (1,2) ] ),(1,2)(3,4)) ]
```

3 ▶ `CanonicalRightCosetElement( U, g )` O

returns a "canonical" representative of the coset Ug which is independent of the given representative g. This can be used to compare cosets by comparing their canonical representatives. The representative chosen to be the "canonical" one is representation dependent and only guaranteed to remain the same within one GAP session.

```
gap> CanonicalRightCosetElement(u,(2,4,3));
(3,4)
```

4 ▶ `IsRightCoset( obj )` C

The category of right cosets.

GAP does not provide left cosets as a separate data type, but as the left coset gU consists of exactly the inverses of the elements of the right coset Ug^{-1} calculations with left cosets can be emulated using right cosets by inverting the representatives.

37.8 Transversals

1 ▶ `RightTransversal( G, U )` O

A right transversal t is a list of representatives for the set $U\backslash G$ of right cosets (consisting of cosets Ug) of U in G.

The object returned by `RightTransversal` is not a plain list, but an object that behaves like an immutable list of length $[G:U]$, except if U is the trivial subgroup of G in which case `RightTransversal` may return the sorted plain list of coset representatives.

The operation `PositionCanonical(`t`,`g`)`, called for a transversal t and an element g of G, will return the position of the representative in t that lies in the same coset of U as the element g does. (In comparison, `Position` will return `fail` if the element is not equal to the representative.) Functions that implement group actions such as `Action` or `Permutation` (see Chapter 39) use `PositionCanonical`, therefore it is possible to "act" on a right transversal to implement the action on the cosets. This is often much more efficient than acting on cosets.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> u:=Subgroup(g,[(1,2,3),(1,2)]);;
gap> rt:=RightTransversal(g,u);
RightTransversal(Group([ (1,2,3,4), (1,2) ]),Group([ (1,2,3), (1,2) ]))
gap> Length(rt);
4
gap> Position(rt,(1,2,3));
fail
```

Note that the elements of a right transversal are not necessarily "canonical" in the sense of `CanonicalRightCosetElement` (see 37.7.3), but we may compute a list of canonical coset representatives by calling that function.

```
gap> List(RightTransversal(g,u),i->CanonicalRightCosetElement(u,i));
[ (), (2,3,4), (1,2,3,4), (3,4) ]
```

The operation `PositionCanonical` is described in section 21.16.2.

```
gap> PositionCanonical(rt,(1,2,3));
1
gap> rt[1];
()
```

37.9 Double Cosets

1 ▶ `DoubleCoset( U, g, V )` O

The groups U and V must be subgroups of a common supergroup G of which g is an element. This command constructs the double coset UgV which is the set of all elements of the form ugv for any $u \in U$, $v \in V$. For element operations such as `in`, a double coset behaves like a set of group elements. The double coset stores U in the attribute `LeftActingGroup`, g as `Representative`, and V as `RightActingGroup`.

2 ▶ `RepresentativesContainedRightCosets( D )` A

A double coset UgV can be considered as an union of right cosets Uh_i. (it is the union of the orbit of Ug under right multiplication by V.) For a double coset $D{=}UgV$ this returns a set of representatives h_i such that $D = \bigcup_{h_i} Uh_i$. The representatives returned are canonical for U (see 37.7.3) and form a set.

```
gap> u:=Subgroup(g,[(1,2,3),(1,2)]);;v:=Subgroup(g,[(3,4)]);;
gap> c:=DoubleCoset(u,(2,4),v);
DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(2,4),Group( [ (3,4) ] ))
gap> (1,2,3) in c;
false
gap> (2,3,4) in c;
true
gap> LeftActingGroup(c);
Group([ (1,2,3), (1,2) ])
gap> RightActingGroup(c);
Group([ (3,4) ])
gap> RepresentativesContainedRightCosets(c);
[ (2,3,4) ]
```

3 ▶ `DoubleCosets( G, U, V )` O
▶ `DoubleCosetsNC( G, U, V )` O

computes a duplicate free list of all double cosets UgV for $g \in G$. U and V must be subgroups of the group G. The NC version does not check whether U and V are both subgroups of G.

```
gap> dc:=DoubleCosets(g,u,v);
[ DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(),Group( [ (3,4) ] )),
  DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(1,3)(2,4),Group( [ (3,4) ] )),
  DoubleCoset(Group( [ (1,2,3), (1,2) ] ),(1,4)(2,3),Group( [ (3,4) ] )) ]
gap> List(dc,Representative);
[ (), (1,3)(2,4), (1,4)(2,3) ]
```

4 ► `IsDoubleCoset( obj )` C

The category of double cosets.

5 ► `DoubleCosetRepsAndSizes( G, U, V )` O

returns a list of double coset representatives and their sizes, the entries are lists of the form [*rep*, *size*]. This operation is faster that `DoubleCosetsNC` because no double coset objects have to be created.

```
gap> dc:=DoubleCosetRepsAndSizes(g,u,v);
[ [ (), 12 ], [ (1,3)(2,4), 6 ], [ (1,4)(2,3), 6 ] ]
```

6 ► `InfoCoset` V

The information function for coset and double coset operations is `InfoCoset`.

37.10 Conjugacy Classes

1 ► `ConjugacyClass( G, g )` O

creates the conjugacy class in G with representative g. This class is an external set, so functions such as `Representative` (which returns g), `ActingDomain` (which returns G), `StabilizerOfExternalSet` (which returns the centralizer of g) and `AsList` work for it.

A conjugacy class is an external orbit (39.11.9) of group elements with the group acting by conjugation on it. Thus element tests or operation representatives can be computed. The attribute `Centralizer` gives the centralizer of the representative (which is the same result as `StabilizerOfExternalSet`). (This is a slight abuse of notation: This is **not** the centralizer of the class as a **set** which would be the standard behaviour of `Centralizer`.)

2 ► `ConjugacyClasses( G )` A

returns the conjugacy classes of elements of G as a list of `ConjugacyClass`es of G (see `ConjugacyClass` (37.10.1) for details). It is guaranteed that the class of the identity is in the first position, the further arrangement depends on the method chosen (and might be different for equal but not identical groups).

For very small groups (of size up to 500) the classes will be computed by the conjugation action of G on itself (see 37.10.4). This can be deliberately switched off using the "`noaction`" option shown below.

For solvable groups, the default method to compute the classes is by homomorphic lift (see section 43.17).

For other groups the method of [Hul00] is employed.

`ConjugacyClasses` supports the following options that can be used to modify this strategy:

`random`
: The classes are computed by random search. See `ConjugacyClassesByRandomSearch` (37.10.3) below.

`action`
: The classes are computed by action of G on itself See `ConjugacyClassesByOrbits` (37.10.4) below.

`noaction`
: Even for small groups `ConjugacyClassesByOrbits` (37.10.4) is not used as a default. This can be useful if the elements of the group use a lot of memory.

```
gap> g:=SymmetricGroup(4);;
gap> cl:=ConjugacyClasses(g);
[ ()^G, (1,2)^G, (1,2)(3,4)^G, (1,2,3)^G, (1,2,3,4)^G ]
gap> Representative(cl[3]);Centralizer(cl[3]);
(1,2)(3,4)
Group([ (1,2), (1,3)(2,4), (3,4) ])
gap> Size(Centralizer(cl[5]));
4
gap> Size(cl[2]);
6
```

In general, you will not need to have to influence the method, but simply call `ConjugacyClasses` – GAP will try to select a suitable method on its own. The method specifications are provided here mainly for expert use.

3 ▶ `ConjugacyClassesByRandomSearch( G )` F

computes the classes of the group G by random search. This works very efficiently for almost simple groups.

This function is also accessible via the option `random` to `ConjugacyClass`.

4 ▶ `ConjugacyClassesByOrbits( G )` F

computes the classes of the group G as orbits of G on its elements. This can be quick but unsurprisingly may also take a lot of memory if G becomes larger. All the classes will store their element list and thus a membership test will be quick as well.

This function is also accessible via the option `action` to `ConjugacyClass`.

Typically, for small groups (roughly of order up to 10^3) the computation of classes as orbits under the action is fastest; memory restrictions (and the increasing cost of eliminating duplicates) make this less efficient for larger groups.

Calculation by random search has the smallest memory requirement, but in generally performs worse, the more classes are there.

The folowing example shows the effect of this for a small group with many classes:

```
gap> h:=Group((4,5)(6,7,8),(1,2,3)(5,6,9));;ConjugacyClasses(h:noaction);;time;
110
gap> h:=Group((4,5)(6,7,8),(1,2,3)(5,6,9));;ConjugacyClasses(h:random);;time;
300
gap> h:=Group((4,5)(6,7,8),(1,2,3)(5,6,9));;ConjugacyClasses(h:action);;time;
30
```

5 ▶ `NrConjugacyClasses( G )` A

returns the number of conjugacy classes of G.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> NrConjugacyClasses(g);
5
```

6 ▶ `RationalClass( G, g )` O

creates the rational class in G with representative g. A rational class consists of all elements that are conjugate to g or to a power g^i where i is coprime to the order of g. Thus a rational class can be interpreted as a conjugacy class of cyclic subgroups. A rational class is an external set (39.11.1) of group elements with the group acting by conjugation on it, but not an external orbit.

7 ► `RationalClasses( G )` A

returns a list of the rational classes of the group G. (See 37.10.6.)

```
gap> RationalClasses(DerivedSubgroup(g));
[ RationalClass( AlternatingGroup( [ 1 .. 4 ] ), () ),
  RationalClass( AlternatingGroup( [ 1 .. 4 ] ), (1,2)(3,4) ),
  RationalClass( AlternatingGroup( [ 1 .. 4 ] ), (1,2,3) ) ]
```

8 ► `GaloisGroup( ratcl )` A

Suppose that *ratcl* is a rational class of a group G with representative g. The exponents i for which g^i lies already in the ordinary conjugacy class of g, form a subgroup of the **prime residue class group** P_n (see 15.2.3), the so-called **Galois group** of the rational class. The prime residue class group P_n is obtained in GAP as `Units( Integers mod n )`, the unit group of a residue class ring. The Galois group of a rational class *rcl* is stored in the attribute `GaloisGroup(`*rcl*`)` as a subgroup of this group.

9 ► `IsConjugate( G, x, y )` O
► `IsConjugate( G, U, V )` O

tests whether the elements x and y or the subgroups U and V are conjugate under the action of G. (They do not need to be contained in G.) This command is only a shortcut to `RepresentativeOperation`.

```
gap> IsConjugate(g,Group((1,2,3,4),(1,3)),Group((1,3,2,4),(1,2)));
true
```

`RepresentativeAction` (see 39.5.1) can be used to obtain conjugating elements.

```
gap> RepresentativeAction(g,(1,2),(3,4));
(1,3)(2,4)
```

37.11 Normal Structure

For the operations `Centralizer` and `Centre`, see Chapter 33.

1 ► `Normalizer( G, U )` O
► `Normalizer( G, g )` O

Computes the normalizer $N_G(U)$, that is the stabilizer of U under the conjugation action of G. The second form computes $N_G(\langle g \rangle)$.

```
gap> Normalizer(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3), (2,3) ])
```

2 ► `Core( S, U )` O

If S and U are groups of elements in the same family, this operation returns the core of U in S, that is the intersection of all S-conjugates of U.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> Core(g,Subgroup(g,[(1,2,3,4)]));
Group(())
```

3 ► `PCore( G, p )` F

The p-**core** of G is the largest normal p-subgroup of G. It is the core of a p-Sylow subgroup of G.

```
gap> PCore(g,2);
Group([ (1,4)(2,3), (1,3)(2,4) ])
```

4 ▶ `NormalClosure( G, U )` O

The normal closure of U in G is the smallest normal subgroup of G which contains U.

```
gap> NormalClosure(g,Subgroup(g,[(1,2,3)]));
Group([ (1,2,3), (2,3,4) ])
```

5 ▶ `NormalIntersection( G, U )` O

computes the intersection of G and U, assuming that G is normalized by U. This works faster than `Intersection`, but will not produce the intersection if G is not normalized by U.

```
gap> NormalIntersection(Group((1,2)(3,4),(1,3)(2,4)),Group((1,2,3,4)));
Group([ (1,3)(2,4) ])
```

6 ▶ `Complementclasses( G, N )` O

Let N be a normal subgroup of G. This command returns a set of representatives for the conjugacy classes of complements of N in G. Complements are subgroups U of G which intersect trivially with N and together with N generate G.

At the moment only methods for a solvable N are available.

```
gap> Complementclasses(g,Group((1,2)(3,4),(1,3)(2,4)));
[ Group([ (3,4), (2,4,3) ]) ]
```

7 ▶ `InfoComplement` V

Info class for the complement routines.

37.12 Specific and Parametrized Subgroups

The Centre of a group (the subgroup of those elements that commute with all other elements of the group) can be computed by the operation `Centre` (see 33.4.5).

1 ▶ `TrivialSubgroup( G )` A

```
gap> TrivialSubgroup(g);
Group(())
```

2 ▶ `CommutatorSubgroup( G, H )` O

If G and H are two groups of elements in the same family, this operation returns the group generated by all commutators $[g,h] = g^{-1}h^{-1}gh$ (see 30.12.3) of elements $g \in G$ and $h \in H$, that is the group $\langle [g,h] \mid g \in G, h \in H \rangle$.

```
gap> CommutatorSubgroup(Group((1,2,3),(1,2)),Group((2,3,4),(3,4)));
Group([ (1,4)(2,3), (1,3,4) ])
gap> Size(last);
12
```

3 ▶ `DerivedSubgroup( G )` A

The derived subgroup G' of G is the subgroup generated by all commutators of pairs of elements of G. It is normal in G and the factor group G/G' is the largest abelian factor group of G.

```
gap> DerivedSubgroup(g);
Group([ (1,3,2), (1,4,3) ])
```

4 ▶ `CommutatorLength( G )` A

returns the minimal number n such that each element in the derived subgroup (see 37.12.3) of the group G can be written as a product of (at most) n commutators of elements in G.

```
gap> CommutatorLength( g );
1
```

5 ▶ `FittingSubgroup( G )` A

The Fitting subgroup of a group G is its largest nilpotent normal subgroup.

```
gap> FittingSubgroup(g);
Group([ (1,3)(2,4), (1,4)(2,3) ])
```

6 ▶ `FrattiniSubgroup( G )` A

The Frattini subgroup of a group G is the intersection of all maximal subgroups of G.

```
gap> FrattiniSubgroup(g);
Group(())
```

7 ▶ `PrefrattiniSubgroup( G )` A

returns a Prefrattini subgroup of the finite solvable group G. A factor M/N of G is called a Frattini factor if $M/N \leq \phi(G/N)$ holds. The group P is a Prefrattini subgroup of G if P covers each Frattini chief factor of G, and if for each maximal subgroup of G there exists a conjugate maximal subgroup, which contains P. In a finite solvable group G the Prefrattini subgroups form a characteristic conjugacy class of subgroups and the intersection of all these subgroups is the Frattini subgroup of G.

```
gap> G := SmallGroup( 60, 7 );
<pc group of size 60 with 4 generators>
gap> P := PrefrattiniSubgroup(G);
Group([ f2 ])
gap> Size(P);
2
gap> IsNilpotent(P);
true
gap> Core(G,P);
Group([  ])
gap> FrattiniSubgroup(G);
Group([  ])
```

8 ▶ `PerfectResiduum( G )` A

is the smallest normal subgroup of G that has a solvable factor group.

```
gap> PerfectResiduum(Group((1,2,3,4,5),(1,2)));
Group([ (1,3,2), (2,4,3), (2,3)(4,5) ])
```

9 ▶ `RadicalGroup( G )` A

is the radical of G, i.e., the largest solvable normal subgroup of G.

```
gap> RadicalGroup(SL(2,5));
<group of 2x2 matrices of size 2 in characteristic 5>
gap> Size(last);
2
```

10 ▶ `Socle( G )` A

The socle of the group G is the subgroup generated by all minimal normal subgroups.

```
gap> Socle(g);
Group([ (1,4)(2,3), (1,2)(3,4) ])
```

11 ▶ `SupersolvableResiduum( G )` A

is the supersolvable residuum of the group G, that is, its smallest normal subgroup N such that the factor group G/N is supersolvable.

```
gap> SupersolvableResiduum(g);
Group([ (1,2)(3,4), (1,4)(2,3) ])
```

12 ▶ `PRump( G, p )` F

The p-**rump** of a group G is the subgroup $G'G^p$ for a prime p.

@example missing!@

37.13 Sylow Subgroups and Hall Subgroups

1 ▶ `SylowSubgroup( G, p )` F

returns a Sylow p subgroup of the finite group G. This is a p-subgroup of G whose index in G is coprime to p. `SylowSubgroup` computes Sylow subgroups via the operation `SylowSubgroupOp`.

```
gap> g:=SymmetricGroup(4);;
gap> SylowSubgroup(g,2);
Group([ (1,2), (3,4), (1,3)(2,4) ])
```

With respect to the following GAP functions, please note that by theorems of P. Hall, a group G is solvable if and only if one of the following conditions holds.

1. For each prime p dividing the order of G, there exists a p-complement (see 37.13.2).
2. For each set P of primes dividing the order of G, there exists a P-Hall subgroup (see 37.13.3).
3. G has a Sylow system (see 37.13.4).
4. G has a complement system (see 37.13.5).

2 ▶ `SylowComplement( G, p )` F

returns a p-Sylow complement of the finite group G. This is a subgroup U of order coprime to p such that the index $[G : U]$ is a p-power. At the moment methods exist only if G is solvable and GAP will issue an error if G is not solvable.

```
gap> SylowComplement(g,3);
Group([ (3,4), (1,4)(2,3), (1,3)(2,4) ])
```

3 ▶ `HallSubgroup( G, P )` F

computes a P-Hall subgroup for a set P of primes. This is a subgroup the order of which is only divisible by primes in P and whose index is coprime to all primes in P. The function computes Hall subgroups via

the operation `HallSubgroupOp`. At the moment methods exist only if G is solvable and GAP will issue an error if G is not solvable.

```
gap> h:=SmallGroup(60,10);;
gap> u:=HallSubgroup(h,[2,3]);
Group([ f1, f2, f3 ])
gap> Size(u);
12
```

4 ▶ `SylowSystem( G )` A

A Sylow system of a group G is a set of Sylow subgroups of G such that every pair of Sylow subgroups from this set commutes as subgroups. Sylow systems exist only for solvable groups. The operation returns `fail` if the group G is not solvable.

```
gap> h:=SmallGroup(60,10);;
gap> SylowSystem(h);
[ Group([ f1, f2 ]), Group([ f3 ]), Group([ f4 ]) ]
gap> List(last,Size);
[ 4, 3, 5 ]
```

5 ▶ `ComplementSystem( G )` A

A complement system of a group G is a set of Hall-p'-subgroups of G, where p' runs through the subsets of prime factors of $|G|$ that omit exactly one prime. Every pair of subgroups from this set commutes as subgroups. Complement systems exist only for solvable groups, therefore `ComplementSystem` returns `fail` if the group G is not solvable.

```
gap> ComplementSystem(h);
[ Group([ f3, f4 ]), Group([ f1, f2, f4 ]), Group([ f1, f2, f3 ]) ]
gap> List(last,Size);
[ 15, 20, 12 ]
```

6 ▶ `HallSystem( G )` A

returns a list containing one Hall-P subgroup for each set P of primes which occur in the order of G. Hall systems exist only for solvable groups. The operation returns `fail` if the group G is not solvable.

```
gap> HallSystem(h);
[ Group([  ]), Group([ f1, f2 ]), Group([ f1, f2, f3 ]),
  Group([ f1, f2, f3, f4 ]), Group([ f1, f2, f4 ]), Group([ f3 ]),
  Group([ f3, f4 ]), Group([ f4 ]) ]
gap> List(last,Size);
[ 1, 4, 12, 60, 20, 3, 15, 5 ]
```

37.14 Subgroups characterized by prime powers

1 ▶ `Omega( G, p[, n] )` F

For a p-group G, one defines $\Omega_n(G) = \{g \in G \mid g^{p^n} = 1\}$. The default value for n is `1`.

@At the moment methods exist only for abelian G and n=1.@

```
gap> h:=SmallGroup(16,10);
<pc group of size 16 with 4 generators>
gap> Omega(h,2);
Group([ f4, f2, f3 ])
```

2 ▶ `Agemo( G, p[, n] )` F

For a p-group G, one defines $\mho_n(G) = \langle g^{p^n} \mid g \in G \rangle$. The default value for n is `1`.

```
gap> Agemo(h,2);Agemo(h,2,2);
Group([ f4 ])
Group([  ])
```

37.15 Group Properties

Some properties of groups can be defined not only for groups but also for other structures. For example, nilpotency and solvability make sense also for algebras. Note that these names refer to different definitions for groups and algebras, contrary to the situation with finiteness or commutativity. In such cases, the name of the function for groups got a suffix `Group` to distinguish different meanings for different structures.

1 ▶ `IsCyclic(` *G* `)` P

A group is **cyclic** if it can be generated by one element. For a cyclic group, one can compute a generating set consisting of only one element using `MinimalGeneratingSet` (see 37.22.3).

2 ▶ `IsElementaryAbelian(` *G* `)` P

A group G is elementary abelian if it is commutative and if there is a prime p such that the order of each element in G divides p.

3 ▶ `IsNilpotentGroup(` *G* `)` P

A group is **nilpotent** if the lower central series (see 37.17.11 for a definition) reaches the trivial subgroup in a finite number of steps.

4 ▶ `NilpotencyClassOfGroup(` *G* `)` A

The nilpotency class of a nilpotent group G is the number of steps in the lower central series of G (see 37.17.11);

If G is not nilpotent an error is issued.

5 ▶ `IsPerfectGroup(` *G* `)` P

A group is **perfect** if it equals its derived subgroup (see 37.12.3).

6 ▶ `IsSolvableGroup(` *G* `)` P

A group is **solvable** if the derived series (see 37.17.7 for a definition) reaches the trivial subgroup in a finite number of steps.

For finite groups this is the same as being polycyclic (see 37.15.7), and each polycyclic group is solvable, but there are infinite solvable groups that are not polycyclic.

7 ▶ `IsPolycyclicGroup(` *G* `)` P

A group is polycyclic if it has a subnormal series with cyclic factors. For finite groups this is the same as if the group is solvable (see 37.15.6).

8 ▶ `IsSupersolvableGroup(` *G* `)` P

A finite group is **supersolvable** if it has a normal series with cyclic factors.

9 ▶ `IsMonomialGroup(` *G* `)` P

A finite group is **monomial** if every irreducible complex character is induced from a linear character of a subgroup.

10 ▶ `IsSimpleGroup(` *G* `)` P

A group is **simple** if it is nontrivial and has no nontrivial normal subgroups.

11 ▶ `IsomorphismTypeInfoFiniteSimpleGroup( G )` F

For a finite simple group G, `IsomorphismTypeInfoFiniteSimpleGroup` returns a record with components `series`, `name` and possibly `parameter`, describing the isomorphism type of G. The component `name` is a string that gives name(s) for G, and `series` is a string that describes the following series.

(If different characterizations of G are possible only one is given by `series` and `parameter`, while `name` may give several names.)

`"A"` Alternating groups, `parameter` gives the natural degree.

`"L"` Linear groups (Chevalley type A), `parameter` is a list $[n,q]$ that indicates $L(n, q)$.

`"2A"` Twisted Chevalley type ${}^2A$, `parameter` is a list $[n,q]$ that indicates ${}^2A(n, q)$.

`"B"` Chevalley type B, `parameter` is a list $[n,q]$ that indicates $B(n, q)$.

`"2B"` Twisted Chevalley type ${}^2B$, `parameter` is a value q that indicates ${}^2B(2, q)$.

`"C"` Chevalley type C, `parameter` is a list $[n,q]$ that indicates $C(n, q)$.

`"D"` Chevalley type D, `parameter` is a list $[n,q]$ that indicates $D(n, q)$.

`"2D"` Twisted Chevalley type ${}^2D$, `parameter` is a list $[n,q]$ that indicates ${}^2D(n, q)$.

`"3D"` Twisted Chevalley type ${}^3D$, `parameter` is a value q that indicates ${}^3D(4, q)$.

`"E"` Exceptional Chevalley type E, `parameter` is a list $[n,q]$ that indicates $E_n(q)$. The value of n is 6,7 or 8.

`"2E"` Twisted exceptional Chevalley type E_6, `parameter` is a value q that indicates ${}^2E_6(q)$.

`"F"` Exceptional Chevalley type F, `parameter` is a value q that indicates $F(4, q)$.

`"2F"` Twisted exceptional Chevalley type ${}^2F$ (Ree groups), `parameter` is a value q that indicates ${}^2F(4, q)$.

`"G"` Exceptional Chevalley type G, `parameter` is a value q that indicates $G(2, q)$.

`"2G"` Twisted exceptional Chevalley type ${}^2G$ (Ree groups), `parameter` is a value q that indicates ${}^2G(2, q)$.

`"Spor"` Sporadic groups, `name` gives the name.

`"Z"` Cyclic groups of prime size, `parameter` gives the size.

An equal sign in the name denotes different naming schemes for the same group, a tilde sign abstract isomorphisms between groups constructed in a different way.

```
gap> IsomorphismTypeInfoFiniteSimpleGroup(Group((4,5)(6,7),(1,2,4)(3,5,6)));
rec( series := "L", parameter := [ 2, 7 ],
  name := "A(1,7) = L(2,7) ~ B(1,7) = O(3,7) ~ C(1,7) = S(2,7) ~ 2A(1,7) = U(2\
,7) ~ A(2,2) = L(3,2)" )
```

12 ▶ `IsFinitelyGeneratedGroup( G )` P

tests whether the group G can be generated by a finite number of generators. (This property is mainly used to obtain finiteness conditions.)

13 ▶ `IsSubsetLocallyFiniteGroup( U )` P

A group is called locally finite if every finitely generated subgroup is finite. This property checks whether the group U is a subset of a locally finite group. This is used to check whether finite generation will imply finiteness, as it does for example for permutation groups.

14 ▶ `IsPGroup( G )` P

A **p-group** is a finite group whose order (see 28.3.6) is of the form p^n for a prime integer p and a nonnegative integer n. `IsPGroup` returns `true` if G is a p-group, and `false` otherwise.

15 ▶ `PrimePGroup( G )` A

If G is a nontrivial p-group (see 37.15.14), `PrimePGroup` returns the prime integer p; if G is trivial then `PrimePGroup` returns `fail`. Otherwise an error is issued.

16 ▶ `PClassPGroup( G )` A

The p-class of a p-group G (see 37.15.14) is the length of the lower p-central series (see 37.17.13) of G. If G is not a p-group then an error is issued.

17 ▶ `RankPGroup( G )` A

For a p-group G (see 37.15.14), `RankPGroup` returns the **rank** of G, which is defined as the minimal size of a generating system of G. If G is not a p-group then an error is issued.

```
gap> h:=Group((1,2,3,4),(1,3));;
gap> PClassPGroup(h);
2
gap> RankPGroup(h);
2
```

Note that the following functions, although they are mathematical properties, are not properties in the sense of GAP (see 13.5 and 13.7), as they depend on a parameter.

18 ▶ `IsPSolvable( G, p )` F

A group is p-solvable if every chief factor is either not divisible by p or solvable.

@Currently no method is installed!@

19 ▶ `IsPNilpotent( G, p )` F

A group is p-nilpotent if it possesses a normal p-complement.

37.16 Numerical Group Attributes

1 ▶ `AbelianInvariants( G )` A

returns the abelian invariants (also sometimes called primary decomposition) of the commutator factor group of the group G. These are given as a list of prime-powers or zeroes and describe the structure of G/G' as a direct product of cyclic groups of prime power (or infinite) order.

(See 37.22.5 to obtain actual generators).

```
gap> g:=Group((1,2,3,4),(1,2),(5,6));;
gap> AbelianInvariants(g);
[ 2, 2 ]
```

2 ▶ `Exponent( G )` A

The exponent e of a group G is the lcm of the orders of its elements, that is, e is the smallest integer such that $g^e = 1$ for all $g \in G$

```
gap> Exponent(g);
12
```

Again the following are mathematical attributes, but not GAP `Attributes` as they are depending on a parameter:

3 ▶ `EulerianFunction( G, n )` O

returns the number of n-tuples $(g_1, g_2, \ldots g_n)$ of elements of the group G that generate the whole group G. The elements of an n-tuple need not be different. If the Library of Tables of Marks (see Chapter 68) covers the group G, you may also use `EulerianFunctionByTom` (see 68.9.9).

```
gap> EulerianFunction(g,2);
432
```

37.17 Subgroup Series

In group theory many subgroup series are considered, and GAP provides commands to compute them. In the following sections, there is always a series $G = U_1 > U_2 > \cdots > U_m = \langle 1 \rangle$ of subgroups considered. A series also may stop without reaching G or $\langle 1 \rangle$.

A series is called **subnormal** if every U_{i+1} is normal in U_i.

A series is called **normal** if every U_i is normal in G.

A series of normal subgroups is called **central** if U_i/U_{i+1} is central in G/U_{i+1}.

We call a series **refinable** if intermediate subgroups can be added to the series without destroying the properties of the series.

Unless explicitly declared otherwise, all subgroup series are descending. That is they are stored in decreasing order.

1 ▶ `ChiefSeries( G )` A

is a series of normal subgroups of G which cannot be refined further. That is there is no normal subgroup N of G with $U_i > N > U_{i+1}$. This attribute returns **one** chief series (of potentially many possibilities).

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> ChiefSeries(g);
[ Group([ (1,2,3,4), (1,2) ]), Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group(()) ]
```

2 ▶ `ChiefSeriesThrough( G, l )` O

is a chief series of the group G going through the normal subgroups in the list l. l must be a list of normal subgroups of G contained in each other, sorted by descending size. This attribute returns **one** chief series (of potentially many possibilities).

3 ▶ `ChiefSeriesUnderAction( H, G )` O

returns a series of normal subgroups of G which are invariant under H such that the series cannot be refined any further. G must be a subgroup of H. This attribute returns **one** such series (of potentially many possibilities).

4 ▶ `SubnormalSeries( G, U )` O

If U is a subgroup of G this operation returns a subnormal series that descends from G to a subnormal subgroup $V \geq U$. If U is subnormal, $V = U$.

```
gap> s:=SubnormalSeries(g,Group((1,2)(3,4)));
[ Group([ (1,2,3,4), (1,2) ]), Group([ (1,2)(3,4), (1,3)(2,4) ]),
  Group([ (1,2)(3,4) ]) ]
```

5 ▶ `CompositionSeries( G )` A

A composition series is a subnormal series which cannot be refined. This attribute returns **one** composition series (of potentially many possibilities).

6 ▶ `DisplayCompositionSeries( G )` F

Displays a composition series of G in a nice way, identifying the simple factors.

```
gap> CompositionSeries(g);
[ Group([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group([ (1,3)(2,4) ]), Group(()) ]
gap> DisplayCompositionSeries(Group((1,2,3,4,5,6,7),(1,2)));
G (2 gens, size 5040)
 | Z(2)
S (5 gens, size 2520)
 | A(7)
1 (0 gens, size 1)
```

7 ► `DerivedSeriesOfGroup(` *G* `)` A

The derived series of a group is obtained by $U_{i+1} = U_i'$. It stops if U_i is perfect.

8 ► `DerivedLength(` *G* `)` A

The derived length of a group is the number of steps in the derived series. (As there is always the group, it is the series length minus 1.)

```
gap> List(DerivedSeriesOfGroup(g),Size);
[ 24, 12, 4, 1 ]
gap> DerivedLength(g);
3
```

9 ► `ElementaryAbelianSeries(` *G* `)` A
► `ElementaryAbelianSeriesLargeSteps(` *G* `)` A
► `ElementaryAbelianSeries( [`*G*`,` *NT1*`,` *NT2*`, ...] )` A

returns a series of normal subgroups of G such that all factors are elementary abelian. If the group is not solvable (and thus no such series exists) it returns `fail`.

The variant `ElementaryAbelianSeriesLargeSteps` tries to make the steps in this series large (by eliminating intermediate subgroups if possible) at a small additional cost.

In the third variant, an elementary abelian series through the given series of normal subgroups is constructed.

```
gap> List(ElementaryAbelianSeries(g),Size);
[ 24, 12, 4, 1 ]
```

10 ► `InvariantElementaryAbelianSeries(` *G*`,` *morph*`[,` *N* `[,` *fine*`]] )` O

For a (solvable) group G and a list of automorphisms *morph* of G, this command finds a normal series of G with elementary abelian factors such that every group in this series is invariant under every automorphism in *morph*.

If a normal subgroup N of G which is invariant under *morph* is given, this series is chosen to contain N. No tests are performed to check the validity of the arguments.

The series obtained will be constructed to prefer large steps unless *fine* is given as `true`.

```
gap> g:=Group((1,2,3,4),(1,3));
Group([ (1,2,3,4), (1,3) ])
gap> hom:=GroupHomomorphismByImages(g,g,GeneratorsOfGroup(g),
> [(1,4,3,2),(1,4)(2,3)]);
[ (1,2,3,4), (1,3) ] -> [ (1,4,3,2), (1,4)(2,3) ]
gap> InvariantElementaryAbelianSeries(g,[hom]);
[ Group([ (1,2,3,4), (1,3) ]), Group([ (1,3)(2,4) ]), Group(()) ]
```

11 ► `LowerCentralSeriesOfGroup(` *G* `)` A

The lower central series of a group G is defined as $U_{i+1} := [G, U_i]$. It is a central series of normal subgroups. The name derives from the fact that U_i is contained in the i-th step subgroup of any central series.

12 ▶ `UpperCentralSeriesOfGroup( G )` A

The upper central series of a group G is defined as an ending series $U_i/U_{i+1} := Z(G/U_{i+1})$. It is a central series of normal subgroups. The name derives from the fact that U_i contains every i-th step subgroup of a central series.

13 ▶ `PCentralSeries( G, p )` F

The p-central series of G is defined by $U_1 := G$, $U_i := [G, U_{i-1}]U_{i-1}^p$.

14 ▶ `JenningsSeries( G )` A

For a p-group G, this function returns its Jennings series. This series is defined by setting $G_1 = G$ and for $i \geq 0$, $G_{i+1} = [G_i, G]G_j^p$, where j is the smallest integer $\geq i/p$.

15 ▶ `DimensionsLoewyFactors( G )` A

This operation computes the dimensions of the factors of the Loewy series of G. (See [HB82], p. 157 for the slightly complicated definition of the Loewy Series.)

The dimensions are computed via the `JenningsSeries` without computing the Loewy series itself.

```
gap> G:= SmallGroup( 3^6, 100 );
<pc group of size 729 with 6 generators>
gap> JenningsSeries( G );
[ <pc group of size 729 with 6 generators>, Group([ f3, f4, f5, f6 ]),
  Group([ f4, f5, f6 ]), Group([ f5, f6 ]), Group([ f5, f6 ]),
  Group([ f5, f6 ]), Group([ f6 ]), Group([ f6 ]), Group([ f6 ]),
  Group([ <identity> of ... ]) ]
gap> DimensionsLoewyFactors(G);
[ 1, 2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 27, 27,
  27, 27, 27, 27, 27, 27, 27, 26, 25, 23, 22, 20, 19, 17, 16, 14, 13, 11, 10,
  8, 7, 5, 4, 2, 1 ]
```

16 ▶ `AscendingChain( G, U )` F

This function computes an ascending chain of subgroups from U to G. This chain is given as a list whose first entry is U and the last entry is G. The function tries to make the links in this chain small.

The option `refineIndex` can be used to give a bound for refinements of steps to avoid GAP trying to enforce too small steps.

17 ▶ `IntermediateGroup( G, U )` F

This routine tries to find a subgroup E of G, such that $G > E > U$. If U is maximal, it returns `fail`. This is done by finding minimal blocks for the operation of G on the right cosets of U.

18 ▶ `IntermediateSubgroups( G, U )` O

returns a list of all subgroups of G that properly contain U; that is all subgroups between G and U. It returns a record with components `subgroups` which is a list of these subgroups as well as a component `inclusions` which lists all maximality inclusions among these subgroups. A maximality inclusion is given as a list `[i,j]` indicating that subgroup number i is a maximal subgroup of subgroup number j, the numbers 0 and 1+length(`subgroups`) are used to denote U and G respectively.

37.18 Factor Groups

1 ▶ `NaturalHomomorphismByNormalSubgroup( G, N )` F
▶ `NaturalHomomorphismByNormalSubgroupNC( G, N )` F

returns a homomorphism from G to another group whose kernel is N. GAP will try to select the image group as to make computations in it as efficient as possible. As the factor group G/N can be identified with the image of G this permits efficient computations in the factor group. The homomorphism returned is not necessarily surjective, so `ImagesSource` should be used instead of `Range` to get a group isomorphic to the factor group. The `NC` variant does not check whether N is normal in G.

2 ▶ `FactorGroup( G, N )` F
▶ `FactorGroupNC( G, N )` O

returns the image of the `NaturalHomomorphismByNormalSubgroup(`G`,`N`)`. The `NC` version does not test whether N is normal in G.

```
gap> g:=Group((1,2,3,4),(1,2));;n:=Subgroup(g,[(1,2)(3,4),(1,3)(2,4)]);;
gap> hom:=NaturalHomomorphismByNormalSubgroup(g,n);
[ (1,2,3,4), (1,2) ] -> [ f1*f2, f1 ]
gap> Size(ImagesSource(hom));
6
gap> FactorGroup(g,n);
Group([ f1, f2 ])
```

3 ▶ `CommutatorFactorGroup( G )` A

computes the commutator factor group G/G' of the group G.

```
gap> CommutatorFactorGroup(g);
Group([ f1 ])
```

4 ▶ `MaximalAbelianQuotient( grp )` A

returns an epimorphism from *grp* onto the maximal abelian quotient of *grp*. the kernel of this epimorphism is the derived subgroup.

5 ▶ `HasAbelianFactorGroup( G, N )` O

tests whether G/N is abelian (without explicitly constructing the factor group).

6 ▶ `HasElementaryAbelianFactorGroup( G, N )` O

tests whether G/N is elementary abelian (without explicitly constructing the factor group).

```
gap> HasAbelianFactorGroup(g,n);
false
gap> HasAbelianFactorGroup(DerivedSubgroup(g),n);
true
```

7 ▶ `CentralizerModulo( G, N, elm )` O

Computes the full preimage of the centralizer $C_{G/N}(elm \cdot N)$ in G (without necessarily constructing the factor group).

```
gap> CentralizerModulo(g,n,(1,2));
Group([ (3,4), (1,3)(2,4), (1,4)(2,3) ])
```

37.19 Sets of Subgroups

1 ▶ `ConjugacyClassSubgroups( G, U )` O

generates the conjugacy class of subgroups of G with representative U. This class is an external set, so functions such as `Representative`, (which returns U), `ActingDomain` (which returns G), `StabilizerOfExternalSet` (which returns the normalizer of U), and `AsList` work for it.

(The use the `[]` list access to select elements of the class is considered obsolescent and will be removed in future versions. Use `ClassElementLattice` instead.)

```
gap> g:=Group((1,2,3,4),(1,2));;IsNaturalSymmetricGroup(g);;
gap> cl:=ConjugacyClassSubgroups(g,Subgroup(g,[(1,2)]));
Group( [ (1,2) ] )^G
gap> Size(cl);
6
gap> ClassElementLattice(cl,4);
Group([ (2,3) ])
```

2 ▶ `IsConjugacyClassSubgroupsRep( obj )` R
▶ `IsConjugacyClassSubgroupsByStabilizerRep( obj )` R

Is the representation GAP uses for conjugacy classes of subgroups. It can be used to check whether an object is a class of subgroups. The second representation `IsConjugacyClassSubgroupsByStabilizerRep` in addition is an external orbit by stabilizer and will compute its elements via a transversal of the stabilizer.

3 ▶ `ConjugacyClassesSubgroups( G )` A

This attribute returns a list of all conjugacy classes of subgroups of the group G. It also is applicable for lattices of subgroups (see 37.20.1). The order in which the classes are listed depends on the method chosen by GAP. For each class of subgroups, a representative can be accessed using `Representative` (see 28.3.7).

```
gap> ConjugacyClassesSubgroups(g);
[ Group( () )^G, Group( [ (1,3)(2,4) ] )^G, Group( [ (1,2) ] )^G,
  Group( [ (2,4,3) ] )^G, Group( [ (1,4)(2,3), (1,3)(2,4) ] )^G,
  Group( [ (1,2)(3,4), (1,2) ] )^G, Group( [ (1,2)(3,4), (1,4,2,3) ] )^G,
  Group( [ (3,4), (2,4,3) ] )^G, Group( [ (1,3)(2,4), (1,4)(2,3), (1,2) ] )^G,
  Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3) ] )^G,
  Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3), (1,2) ] )^G ]
```

4 ▶ `ConjugacyClassesMaximalSubgroups( G )` A

returns the conjugacy classes of maximal subgroups of G. Representatives of the classes can be computed directly by `MaximalSubgroupClassReps` (see 37.19.5).

```
gap> ConjugacyClassesMaximalSubgroups(g);
[ AlternatingGroup( [ 1 .. 4 ] )^G, Group( [ (1,2,3), (1,2) ] )^G,
  Group( [ (1,2), (3,4), (1,3)(2,4) ] )^G ]
```

5 ▶ `MaximalSubgroupClassReps( G )` A

returns a list of conjugacy representatives of the maximal subgroups of G.

```
gap> MaximalSubgroupClassReps(g);
[ Alt( [ 1 .. 4 ] ), Group([ (1,2,3), (1,2) ]),
  Group([ (1,2), (3,4), (1,3)(2,4) ]) ]
```

6 ▶ `MaximalSubgroups( G )` A

returns a list of all maximal subgroups of G. This may take up much space, therefore the command should be avoided if possible. See 37.19.4.

```
gap> MaximalSubgroups(Group((1,2,3),(1,2)));
[ Group([ (1,2,3) ]), Group([ (2,3) ]), Group([ (1,2) ]), Group([ (1,3) ]) ]
```

7 ► `NormalSubgroups( G )` A

returns a list of all normal subgroups of G.

```
gap> g:=SymmetricGroup(4);;NormalSubgroups(g);
[ Group(()), Group([ (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]), Sym( [ 1 .. 4 ] ) ]
```

The algorithm used for the computation of normal subgroups of permutation groups and pc groups is described in [Hul98].

8 ► `MaximalNormalSubgroups( G )` A

is a list containing those proper normal subgroups of the group G that are maximal among the proper normal subgroups.

```
gap> MaximalNormalSubgroups( g );
[ Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]) ]
```

9 ► `MinimalNormalSubgroups( G )` A

is a list containing those nontrivial normal subgroups of the group G that are minimal among the nontrivial normal subgroups.

```
gap> MinimalNormalSubgroups( g );
[ Group([ (1,4)(2,3), (1,3)(2,4) ]) ]
```

37.20 Subgroup Lattice

The GAP package XGAP permits a graphical display of the lattice of subgroups in a nice way.

1 ► `LatticeSubgroups( G )` A

computes the lattice of subgroups of the group G. This lattice has the conjugacy classes of subgroups as attribute `ConjugacyClassesSubgroups` (see 37.19.3) and permits one to test maximality/minimality relations.

```
gap> g:=SymmetricGroup(4);;
gap> l:=LatticeSubgroups(g);
<subgroup lattice of Sym( [ 1 .. 4 ] ), 11 classes, 30 subgroups>
gap> ConjugacyClassesSubgroups(l);
[ Group( () )^G, Group( [ (1,3)(2,4) ] )^G, Group( [ (1,2) ] )^G,
  Group( [ (2,4,3) ] )^G, Group( [ (1,4)(2,3), (1,3)(2,4) ] )^G,
  Group( [ (1,2)(3,4), (1,2) ] )^G, Group( [ (1,2)(3,4), (1,4,2,3) ] )^G,
  Group( [ (3,4), (2,4,3) ] )^G, Group( [ (1,3)(2,4), (1,4)(2,3), (1,2) ] )^G,
  Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3) ] )^G,
  Group( [ (1,3)(2,4), (1,4)(2,3), (2,4,3), (1,2) ] )^G ]
```

2 ► `ClassElementLattice( C, n )` O

For a class C of subgroups, obtained by a lattice computation, this operation returns the n-th conjugate subgroup in the class.

Because of other methods installed, `AsList(C)` can give a different arrangement of the class elements!

3 ▶ `MaximalSubgroupsLattice( lat )` A

For a lattice *lat* of subgroups this attribute contains the maximal subgroup relations among the subgroups of the lattice. It is a list, corresponding to the `ConjugacyClassesSubgroups` of the lattice, each entry giving a list of the maximal subgroups of the representative of this class. Every maximal subgroup is indicated by a list of the form [*cls*,*nr*] which means that the *nr*st subgroup in class number *cls* is a maximal subgroup of the representative.

The number *nr* corresponds to access via `ClassElementLattice` and **not** necessarily the `AsList` arrangement! See also 37.20.4.

```
gap> MaximalSubgroupsLattice(l);
[ [  ], [ [ 1, 1 ] ], [ [ 1, 1 ] ], [ [ 1, 1 ] ],
  [ [ 2, 1 ], [ 2, 2 ], [ 2, 3 ] ], [ [ 3, 1 ], [ 3, 6 ], [ 2, 3 ] ],
  [ [ 2, 3 ] ], [ [ 4, 1 ], [ 3, 4 ], [ 3, 5 ], [ 3, 6 ] ],
  [ [ 7, 1 ], [ 6, 1 ], [ 5, 1 ] ],
  [ [ 5, 1 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ] ],
  [ [ 10, 1 ], [ 9, 1 ], [ 9, 2 ], [ 9, 3 ], [ 8, 1 ], [ 8, 2 ], [ 8, 3 ],
      [ 8, 4 ] ] ]
gap> last[6];
[ [ 3, 1 ], [ 3, 6 ], [ 2, 3 ] ]
gap> u1:=Representative(ConjugacyClassesSubgroups(l)[6]);
Group([ (1,2)(3,4), (1,2) ])
gap> u2:=ClassElementLattice(ConjugacyClassesSubgroups(l)[3],1);;
gap> u3:=ClassElementLattice(ConjugacyClassesSubgroups(l)[3],6);;
gap> u4:=ClassElementLattice(ConjugacyClassesSubgroups(l)[2],3);;
gap> IsSubgroup(u1,u2);IsSubgroup(u1,u3);IsSubgroup(u1,u4);
true
true
true
```

4 ▶ `MinimalSupergroupsLattice( lat )` A

For a lattice *lat* of subgroups this attribute contains the minimal supergroup relations among the subgroups of the lattice. It is a list, corresponding to the `ConjugacyClassesSubgroups` of the lattice, each entry giving a list of the minimal supergroups of the representative of this class. Every minimal supergroup is indicated by a list of the form [*cls*,*nr*] which means that the *nr*st subgroup in class number *cls* is a minimal supergroup of the representative.

The number *nr* corresponds to access via `ClassElementLattice` and **not** necessarily the `AsList` arrangement! See also 37.20.3.

```
gap> MinimalSupergroupsLattice(l);
[ [ [ 2, 1 ], [ 2, 2 ], [ 2, 3 ], [ 3, 1 ], [ 3, 2 ], [ 3, 3 ], [ 3, 4 ],
      [ 3, 5 ], [ 3, 6 ], [ 4, 1 ], [ 4, 2 ], [ 4, 3 ], [ 4, 4 ] ],
  [ [ 5, 1 ], [ 6, 2 ], [ 7, 2 ] ], [ [ 6, 1 ], [ 8, 2 ], [ 8, 4 ] ],
  [ [ 8, 1 ], [ 10, 1 ] ], [ [ 9, 1 ], [ 9, 2 ], [ 9, 3 ], [ 10, 1 ] ],
  [ [ 9, 1 ] ], [ [ 9, 1 ] ], [ [ 11, 1 ] ], [ [ 11, 1 ] ], [ [ 11, 1 ] ],
  [  ] ]
gap> last[3];
[ [ 6, 1 ], [ 8, 2 ], [ 8, 4 ] ]
gap> u5:=ClassElementLattice(ConjugacyClassesSubgroups(l)[8],2);
Group([ (1,3), (1,3,2) ])
gap> u6:=ClassElementLattice(ConjugacyClassesSubgroups(l)[8],4);
Group([ (1,4), (1,4,2) ])
```

```
gap> IsSubgroup(u5,u2);
true
gap> IsSubgroup(u6,u2);
true
```

5 ▶ `RepresentativesPerfectSubgroups( G )` A
▶ `RepresentativesSimpleSubgroups( G )` A

returns a list of conjugacy representatives of perfect (respectively simple) subgroups of G. This uses the library of perfect groups (see 48.8.2), thus it will issue an error if the library is insufficient to determine all perfect subgroups.

```
gap> m11:=TransitiveGroup(11,6);
M(11)
gap> r:=RepresentativesPerfectSubgroups(m11);
[ Group([ (3,4,10)(5,11,6)(7,9,8), (1,9)(2,7)(3,10)(4,6) ]),
  Group([ (2,3,5)(4,10,9)(6,7,11), (1,9)(2,7)(3,10)(4,6) ]),
  Group([ (3,6)(4,11)(5,10)(8,9), (1,6,5,9)(2,11,3,10) ]),
  Group([ (3,5,8)(4,11,7)(6,9,10), (1,9)(2,7)(3,10)(4,6) ]), M(11), Group(())
 ]
gap> List(r,Size);
[ 60, 60, 360, 660, 7920, 1 ]
```

6 ▶ `ConjugacyClassesPerfectSubgroups( G )` A

returns a list of the conjugacy classes of perfect subgroups of G. (see 37.20.5.)

```
gap> ConjugacyClassesPerfectSubgroups(m11);
[ Group( [ ( 3, 4,10)( 5,11, 6)( 7, 9, 8), ( 1, 9)( 2, 7)( 3,10)( 4, 6) ] )^G,
  Group( [ ( 2, 3, 5)( 4,10, 9)( 6, 7,11), ( 1, 9)( 2, 7)( 3,10)( 4, 6) ] )^G,
  Group( [ ( 3, 6)( 4,11)( 5,10)( 8, 9), ( 1, 6, 5, 9)( 2,11, 3,10) ] )^G,
  Group( [ ( 3, 5, 8)( 4,11, 7)( 6, 9,10), ( 1, 9)( 2, 7)( 3,10)( 4, 6) ] )^G,
  M(11)^G, Group( () )^G ]
```

7 ▶ `Zuppos( G )` A

The **Zuppos** of a group are the cyclic subgroups of prime power order. (The name "Zuppo" derives from the German abbreviation for "zyklische Untergruppen von Primzahlpotenzordnung".) This attribute gives generators of all such subgroups of a group G. That is all elements of G of prime power order up to the equivalence that they generate the same cyclic subgroup.

8 ▶ `InfoLattice` V

is the information class used by the cyclic extension methods for subgroup lattice calculations.

37.21 Specific Methods for Subgroup Lattice Computations

1 ▶ `LatticeByCyclicExtension( G[, func[, noperf]] )` F

computes the lattice of G using the cyclic extension algorithm. If the function *func* is given, the algorithm will discard all subgroups not fulfilling *func* (and will also not extend them), returning a partial lattice. This can be useful to compute only subgroups with certain properties. Note however that this will **not** necessarily yield all subgroups that fulfill *func*, but the subgroups whose subgroups are used for the construction must also fulfill *func* as well. (In fact the filter *func* will simply discard subgroups in the cyclic extension algorithm. Therefore the trivial subgroup will always be included.) Also note, that for such a partial lattice maximality/minimality inclusion relations cannot be computed.

The cyclic extension algorithm requires the perfect subgroups of G. However GAP cannot analyze the function *func* for its implication but can only apply it. If it is known that *func* implies solvability, the computation of the perfect subgroups can be avoided by giving a third parameter *noperf* set to `true`.

```
gap> g:=WreathProduct(Group((1,2,3),(1,2)),Group((1,2,3,4)));;
gap> l:=LatticeByCyclicExtension(g,function(G)
> return Size(G) in [1,2,3,6];end);
<subgroup lattice of <permutation group of size 5184 with 9 generators>,
47 classes, 2628 subgroups, restricted under further condition l!.func>
```

The total number of classes in this example is much bigger, as the following example shows:

```
gap> LatticeSubgroups(g);
<subgroup lattice of <permutation group of size 5184 with 9 generators>,
566 classes, 27134 subgroups>
```

2 ▶ `InvariantSubgroupsElementaryAbelianGroup(` G`,` *homs*`[,` *dims*`] )` F

Let G be an elementary abelian group (that is a vector space) and *homs* a set of automorphisms of G. Then this function computes all subspaces of G which are invariant under all automorphisms in *homs*. When considering G as a module for the algebra generated by *homs*, these are all submodules. If *homs* is empty, it computes all subspaces. If the optional parameter *dims* is given, only subspaces of this dimension are computed.

```
gap> g:=Group((1,2,3),(4,5,6),(7,8,9));
Group([ (1,2,3), (4,5,6), (7,8,9) ])
gap> hom:=GroupHomomorphismByImages(g,g,[(1,2,3),(4,5,6),(7,8,9)],
> [(7,8,9),(1,2,3),(4,5,6)]);
[ (1,2,3), (4,5,6), (7,8,9) ] -> [ (7,8,9), (1,2,3), (4,5,6) ]
gap> u:=InvariantSubgroupsElementaryAbelianGroup(g,[hom]);
[ Group(()), Group([ (1,2,3)(4,5,6)(7,8,9) ]),
  Group([ (1,3,2)(7,8,9), (1,3,2)(4,5,6) ]),
  Group([ (7,8,9), (4,5,6), (1,2,3) ]) ]
```

3 ▶ `SubgroupsSolvableGroup(` G`[,` *opt*`] )` F

This function (implementing the algorithm published in [Hul99]) computes subgroups of a solvable group G, using the homomorphism principle. It returns a list of representatives up to G-conjugacy.

The optional argument *opt* is a record, which may be used to put restrictions on the subgroups computed. The following record components of *opt* are recognized and have the following effects:

`actions`
: must be a list of automorphisms of G. If given, only groups which are invariant under all these automorphisms are computed. The algorithm must know the normalizer in G of the group generated by `actions` (defined formally by embedding in the semidirect product of G with *actions*). This can be given in the component `funcnorm` and will be computed if this component is not given.

`normal`
: if set to `true` only normal subgroups are guaranteed to be returned (though some of the returned subgroups might still be not normal).

`consider`
: a function to restrict the groups computed. This must be a function of five parameters, C,A,N,B,M, that are interpreted as follows: The arguments are subgroups of a factor F of G in the relation $F \geq C > A > N > B > M$. N and M are normal subgroups. C is the full preimage of the normalizer of A/N in F/N. When computing modulo M and looking for subgroups U such that $U \cap N = B$ and $\langle U, N\rangle = A$, this function is called. If it returns `false` all potential groups U

(and therefore all groups later arising from them) are disregarded. This can be used for example to compute only subgroups of certain sizes.

(**This is just a restriction to speed up computations. The function may still return (invariant) subgroups which don't fulfill this condition!**) This parameter is used to permit calculations of some subgroups if the set of all subgroups would be too large to handle.

The actual groups C, A, N and B which are passed to this function are not necessarily subgroups of G but might be subgroups of a proper factor group $F=G/H$. Therefore the `consider` function may not relate the parameter groups to G.

`retnorm`
: if set to `true` the function not only returns a list *subs* of subgroups but also a corresponding list *norms* of normalizers in the form [*subs*,*norms*].

`series`
: is an elementary abelian series of G which will be used for the computation.

`groups`
: is a list of groups to seed the calculation. Only subgroups of these groups are constructed.

```
gap> g:=Group((1,2,3),(1,2),(4,5,6),(4,5),(7,8,9),(7,8));
Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8) ])
gap> hom:=GroupHomomorphismByImages(g,g,
> [(1,2,3),(1,2),(4,5,6),(4,5),(7,8,9),(7,8)],
> [(4,5,6),(4,5),(7,8,9),(7,8),(1,2,3),(1,2)]);
[ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8) ] ->
[ (4,5,6), (4,5), (7,8,9), (7,8), (1,2,3), (1,2) ]
gap> l:=SubgroupsSolvableGroup(g,rec(actions:=[hom]));;
gap> List(l,Size);
[ 1, 3, 9, 27, 54, 2, 6, 18, 108, 4, 216, 8 ]
gap> Length(ConjugacyClassesSubgroups(g)); # to compare
162
```

4 ▶ `SizeConsiderFunction(` *size* `)` F

This function returns a function *consider* of four arguments that can be used in `SubgroupsSolvableGroup` (see 37.21.3) for the option `consider` to compute subgroups whose sizes are divisible by *size*.

```
gap> l:=SubgroupsSolvableGroup(g,rec(actions:=[hom],
> consider:=SizeConsiderFunction(6)));;
gap> List(l,Size);
[ 1, 3, 9, 27, 54, 6, 18, 108, 216 ]
```

This example shows that in general the `consider` function does not provide a perfect filter. It is guaranteed that all subgroups fulfilling the condition are returned, but not all subgroups returned necessarily fulfill the condition.

5 ▶ `ExactSizeConsiderFunction(` *size* `)` F

This function returns a function *consider* of four arguments that can be used in `SubgroupsSolvableGroup` (see 37.21.3) for the option `consider` to compute subgroups whose sizes are exactly *size*.

```
gap> l:=SubgroupsSolvableGroup(g,rec(actions:=[hom],
> consider:=ExactSizeConsiderFunction(6)));;
gap> List(l,Size);
[ 1, 3, 9, 27, 54, 6, 108, 216 ]
```

Again, the `consider` function does not provide a perfect filter. It is guaranteed that all subgroups fulfilling the condition are returned, but not all subgroups returned necessarily fulfill the condition.

6 ▶ `InfoPcSubgroup` V

Information function for the subgroup lattice functions using pcgs.

37.22 Special Generating Sets

1 ▶ `GeneratorsSmallest( G )` A

returns a "smallest" generating set for the group G. This is the lexicographically (using GAPs order of group elements) smallest list l of elements of G such that $G = \langle l \rangle$ and $l_i \notin \langle l_1, \ldots, l_{i-1} \rangle$ (in particular l_1 is not the one of the group). The comparison of two groups via lexicographic comparison of their sorted element lists yields the same relation as lexicographic comparison of their smallest generating sets.

```
gap> g:=SymmetricGroup(4);;
gap> GeneratorsSmallest(g);
[ (3,4), (2,3), (1,2) ]
```

2 ▶ `LargestElementGroup( G )` A

returns the largest element of G with respect to the ordering `<` of the elements family.

3 ▶ `MinimalGeneratingSet( G )` A

returns a generating set of G of minimal possible length.

```
gap> MinimalGeneratingSet(g);
[ (2,4,3), (1,4,2,3) ]
```

4 ▶ `SmallGeneratingSet( G )` A

returns a generating set of G which has few elements. As neither irredundancy, nor minimal length is proven it runs much faster than `MinimalGeneratingSet`. It can be used whenever a short generating set is desired which not necessarily needs to be optimal.

```
gap> SmallGeneratingSet(g);
[ (1,2), (1,2,3,4) ]
```

5 ▶ `IndependentGeneratorsOfAbelianGroup( A )` A

returns a set of generators g of prime-power order of the abelian group A such that A is the direct product of the cyclic groups generated by the g_i.

```
gap> g:=AbelianGroup(IsPermGroup,[15,14,22,78]);;
gap> List(IndependentGeneratorsOfAbelianGroup(g),Order);
[ 2, 2, 2, 3, 3, 5, 7, 11, 13 ]
```

37.23 1-Cohomology

Let G be a finite group and M an elementary abelian normal p-subgroup of G. Then the group of 1-cocycles $Z^1(G/M, M)$ is defined as

$$Z^1(G/M, M) = \{\gamma : G/M \to M \mid \forall g_1, g_2 \in G : \gamma(g_1 M \cdot g_2 M) = \gamma(g_1 M)^{g_2} \cdot \gamma(g_2 M)\}$$

and is a $GF(p)$-vector space.

The group of 1-coboundaries $B^1(G/M, M)$ is defined as

$$B^1(G/M, M) = \{\gamma : G/M \to M \mid \exists m \in M \forall g \in G : \gamma(gM) = (m^{-1})^g \cdot m\}$$

It also is a $GF(p)$-vector space.

Let α be the isomorphism of M into a row vector space $\mathcal{W}$ and $(g_1, \ldots, g_l)$ representatives for a generating set of G/M. Then there exists a monomorphism β of $Z^1(G/M, M)$ in the l-fold direct sum of $\mathcal{W}$, such that $\beta(\gamma) = (\alpha(\gamma(g_1 M)), \ldots, \alpha(\gamma(g_l M)))$ for every $\gamma \in Z^1(G/M, M)$.

1 ▶ `OneCocycles( G, M )` O
▶ `OneCocycles( gens, M )` O
▶ `OneCocycles( G, mpcgs )` O
▶ `OneCocycles( gens, mpcgs )` O

Computes the group of 1-Cocycles $Z^1(G/M, M)$. The normal subgroup M may be given by a (Modulo)Pcgs *mpcgs*. In this case the whole calculation is performed modulo the normal subgroup defined by the `DenominatorOfModuloPcgs(`*mpcgs*`)` (see 43.1). Similarly the group G may instead be specified by a set of elements *gens* that are representatives for a generating system for the factor group G/M. If this is done the 1-Cocycles are computed with respect to these generators (otherwise the routines try to select suitable generators themselves).

2 ▶ `OneCoboundaries( G, M )` O

computes the group of 1-coboundaries. Syntax of input and output otherwise is the same as with `OneCocycles` except that entries that refer to cocycles are not computed.

The operations `OneCocycles` and `OneCoboundaries` return a record with (at least) the components:

`generators`
: Is a list of representatives for a generating set of G/M. Cocycles are represented with respect to these generators.

`oneCocycles`
: A space of row vectors over GF(p), representing Z^1. The vectors are represented in dimension $a \cdot b$ where a is the length of `generators` and p^b the size of M.

`oneCoboundaries`
: A space of row vectors that represents B^1.

`cocycleToList`
: is a function to convert a cocycle (a row vector in `oneCocycles`) to a corresponding list of elements of M.

`listToCocycle`
: is a function to convert a list of elements of M to a cocycle.

`isSplitExtension`
: indicates whether G splits over M. The following components are only bound if the extension splits. Note that if M is given by a modulo pcgs all subgroups are given as subgroups of G by generators corresponding to `generators` and thus may not contain the denominator of the modulo pcgs. In this case taking the closure with this denominator will give the full preimage of the complement in the factor group.

`complement`
: One complement to M in G.

`cocycleToComplement(`*cyc*`)`
: is a function that takes a cocycle from `oneCocycles` and returns the corresponding complement to M in G (with respect to the fixed complement `complement`).

`complementToCocycle(`*U*`)`
: is a function that takes a complement and returns the corresponding cocycle.

If the factor G/M is given by a (modulo) pcgs *gens* then special methods are used that compute a presentation for the factor implicitly from the pcgs.

Note that the groups of 1-cocycles and 1-coboundaries are not `Groups` in the sense of GAP but vector spaces.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> n:=Group((1,2)(3,4),(1,3)(2,4));;
gap> oc:=OneCocycles(g,n);
rec( oneCoboundaries := <vector space over GF(2), with 2 generators>,
  oneCocycles := <vector space over GF(2), with 2 generators>,
  generators := [ (3,4), (2,4,3) ], isSplitExtension := true,
  complement := Group([ (3,4), (2,4,3) ]),
  cocycleToList := function( c ) ... end,
  listToCocycle := function( L ) ... end,
  cocycleToComplement := function( c ) ... end,
  factorGens := [ (3,4), (2,4,3) ],
  complementToCocycle := function( K ) ... end )
gap> oc.cocycleToList([ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ]);
[ (1,2)(3,4), (1,2)(3,4) ]
gap> oc.listToCocycle([(),(1,3)(2,4)]);
[ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ]
gap> oc.cocycleToComplement([ 0*Z(2), Z(2)^0, 0*Z(2), Z(2)^0 ]);
Group([ (1,2), (1,2,3) ])
gap> oc.cocycleToComplement([ 0*Z(2), 0*Z(2), Z(2)^0, 0*Z(2) ]);
Group([ (3,4), (1,3,4) ])
gap> oc.complementToCocycle(Group((1,2,4),(1,4)));
[ 0*Z(2), Z(2)^0, Z(2)^0, Z(2)^0 ]
```

The factor group $H^1(G/M, M) = Z^1(G/M, M)/B^1(G/M, M)$ is called the first cohomology group. Currently there is no function which explicitly computes this group. The easiest way to represent it is as a vector space complement to B^1 in Z^1.

If the only purpose of the calculation of H^1 is the determination of complements it might be desirable to stop calculations once it is known that the extension cannot split. This can be achieved via the more technical function `OCOneCocycles`.

3 ► `OCOneCocycles(` *ocr*, *onlySplit* `)` O

is the more technical function to compute 1-cocycles. It takes an record *ocr* as first argument which must contain at least the components `group` for G and `modulePcgs` for a (modulo) pcgs of M. This record will also be returned with components as described under `OneCocycles` (with the exception of `isSplitExtension` which is indicated by the existence of a `complement`) but components such as `oneCoboundaries` will only be computed if not already present.

If *onlySplit* is `true`, `OneCocyclesOC` returns `false` as soon as possible if the extension does not split.

4 ► `ComplementclassesEA(` G, N `)` O

computes `Complementclasses` to an elementary abelian normal subgroup N via 1-Cohomology. Normally, a user program should call `Complementclasses` (see 37.11.6) instead, which also works for a solvable (not necessarily elementary abelian) N.

5 ► `InfoCoh` V

The info class for the cohomology calculations is `InfoCoh`.

37.24 Schur Covers and Multipliers

1 ▶ `EpimorphismSchurCover( G[, pl] )` O

returns an epimorphism *epi* from a group D onto G. The group D is one (of possibly several) Schur covers of G. The group D can be obtained as the `Source` of *epi*. the kernel of *epi* is the schur multiplier of G. If *pl* is given as a list of primes, only the multiplier part for these primes is realized. At the moment, D is represented as a finitely presented group.

2 ▶ `SchurCover( G )` O

returns one (of possibly several) Schur covers of G.

At the moment this cover is represented as a finitely presented group and `IsomorphismPermGroup` would be needed to convert it to a permutation group.

If also the relation to G is needed, `EpimorphismSchurCover` should be used.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> epi:=EpimorphismSchurCover(g);
[ f1, f2, f3 ] -> [ (3,4), (2,4,3), (1,4)(2,3) ]
gap> Size(Source(epi));
48
```

If the group becomes bigger, Schur Cover calculations might become unfeasible.

There is another operation which only returns the structure of the Multiplier, and which should work for larger groups as well.

3 ▶ `AbelianInvariantsMultiplier( G )` O

returns a list of the abelian invariants of the Schur multiplier of G.

```
gap> AbelianInvariantsMultiplier(g);
[ 2 ]
gap> AbelianInvariantsMultiplier(MathieuGroup(22));
[ 4, 3 ]
```

Note that the following example will take some time.

```
gap> AbelianInvariantsMultiplier(PSU(6,2));
[ 2, 2, 3 ]
```

At the moment, this operation will not give any information about how to extend the multiplier to a Schur Cover.

37.25 Tests for the Availability of Methods

The following filters and operations indicate capabilities of GAP. They can be used in the method selection or algorithms to check whether it is feasible to compute certain operations for a given group. In general, they return `true` if good algorithms for the given arguments are available in GAP. An answer `false` indicates that no method for this group may exist, or that the existing methods might run into problems.

Typical examples when this might happen is with finitely presented groups, for which many of the methods cannot be guaranteed to succeed in all situations.

The willingness of GAP to perform certain operations may change, depending on which further information is known about the arguments. Therefore the filters used are not implemented as properties but as "other filters" (see 13.7 and 13.8).

1 ▶ `CanEasilyTestMembership( grp )` F

This filter indicates whether a group can test membership of elements in *grp* (via the operation `in`) in reasonable time. It is used by the method selection to decide whether an algorithm that relies on membership tests may be used.

2 ▶ `CanComputeSize( dom )` F

This filter indicates whether the size of the domain *dom* (which might be `infinity`) can be computed.

3 ▶ `CanComputeSizeAnySubgroup( grp )` F

This filter indicates whether *grp* can easily compute the size of any subgroup. (This is for example advantageous if one can test that a stabilizer index equals the length of the orbit computed so far to stop early.)

4 ▶ `CanComputeIndex( G, H )` F

This filter indicates whether the index $[G : H]$ (which might be `infinity`) can be computed. It assumes that $H \leq G$. (see 37.25.5)

5 ▶ `CanComputeIsSubset( A, B )` O

This filter indicates that GAP can test (via `IsSubset`) whether B is a subset of A.

6 ▶ `KnowsHowToDecompose( G )` P
▶ `KnowsHowToDecompose( G, gens )` O

Tests whether the group G can decompose elements in the generators *gens*. If *gens* is not given it tests, whether it can decompose in the generators given in `GeneratorsOfGroup`.

This property can be used for example to check whether a `GroupHomomorphismByImages` can be reasonably defined from this group.

38 Group Homomorphisms

A group homomorphism is a mapping from one group to another that respects multiplication and inverses. They are implemented as a special class of mappings, so in particular all operations for mappings, such as `Image`, `PreImage`, `PreImagesRepresentative`, `KernelOfMultiplicativeGeneralMapping`, `Source`, `Range`, `IsInjective` and `IsSurjective` (see chapter 31, in particular section 31.8) are applicable to them.

Homomorphisms can be used to transfer calculations into isomorphic groups in another representation, for which better algorithms are available. Section 38.5 explains a technique how to enforce this automatically.

Homomorphisms are also used to represent group automorphisms, and section 38.6 explains explains GAP's facilities to work with automorphism groups.

The penultimate section of this chapter, 38.9, explains how to make GAP to search for all homomorphisms between two groups which fulfill certain specifications.

38.1 Creating Group Homomorphisms

The most important way of creating group homomorphisms is to give images for a set of group generators and to extend it to the group generated by them by the homomorphism property.

1 ► `GroupHomomorphismByImages(` *G*, *H*, *gens*, *imgs* `)` F

`GroupHomomorphismByImages` returns the group homomorphism with source G and range H that is defined by mapping the list *gens* of generators of G to the list *imgs* of images in H.

If *gens* does not generate G or if the mapping of the generators does not extend to a homomorphism (i.e., if mapping the generators describes only a multi-valued mapping) then `fail` is returned.

This test can be quite expensive. If one is certain that the mapping of the generators extends to a homomorphism, one can avoid the checks by calling `GroupHomomorphismByImagesNC`. (There also is the possibility to construct potentially multi-valued mappings with `GroupGeneralMappingByImages` and to test with `IsMapping` that they are indeed homomorphisms.)

2 ► `GroupHomomorphismByImagesNC(` *G*, *H*, *gensG*, *gensH* `)` O

`GroupHomomorphismByImagesNC` creates a homomorphism as `GroupHomomorphismByImages` does, however it does not test whether *gens* generates G and that the mapping of *gens* to *imgs* indeed defines a group homomorphism. Because these tests can be expensive it can be substantially faster than `GroupHomomorphismByImages`. Results are unpredictable if the conditions do not hold.

(For creating a possibly multi-valued mapping from G to H that respects multiplication and inverses, `GroupGeneralMappingByImages` can be used.)

```
gap> gens:=[(1,2,3,4),(1,2)];
[ (1,2,3,4), (1,2) ]
gap> g:=Group(gens);
Group([ (1,2,3,4), (1,2) ])
gap> h:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> hom:=GroupHomomorphismByImages(g,h,gens,[(1,2),(1,3)]);
[ (1,2,3,4), (1,2) ] -> [ (1,2), (1,3) ]
gap> Image(hom,(1,4));
(2,3)
gap> map:=GroupHomomorphismByImages(g,h,gens,[(1,2,3),(1,2)]);
fail
```

3 ▶ `GroupGeneralMappingByImages( `*G*`, `*H*`, `*gensG*`, `*gensH*` )` O

returns a generalized mapping defined by extending the mapping from *gensG* to *gensH* homomorphically. (`GroupHomomorphismByImages` creates a `GroupGeneralMappingByImages` and tests whether it `IsMapping`.)

```
gap> map:=GroupGeneralMappingByImages(g,h,gens,[(1,2,3),(1,2)]);
[ (1,2,3,4), (1,2) ] -> [ (1,2,3), (1,2) ]
gap> IsMapping(map);
false
```

A second way to create homomorphisms is to give functions that compute image and preimage. (A similar case are homomorphisms that are induced by conjugation. Special constructors for such mappings are described in section 38.6).

4 ▶ `GroupHomomorphismByFunction( `*S*`, `*R*`, `*fun*` )` F
▶ `GroupHomomorphismByFunction( `*S*`, `*R*`, `*fun*`, `*invfun*` )` F
▶ `GroupHomomorphismByFunction( `*S*`, `*R*`, `*fun*`, 'false`, *prefun*)' F

`GroupHomomorphismByFunction` returns a group homomorphism *hom* with source S and range R, such that each element s of S is mapped to the element *fun*`( `*s*` )`, where *fun* is a GAP function.

If the argument *invfun* is bound then *hom* is a bijection between S and R, and the preimage of each element r of R is given by *invfun*`( `*r*` )`, where *invfun* is a GAP function.

In the third variant, a function *prefun* is given that can be used to compute a single preimage. In this case, the third entry must be `false`.

No test is performed on whether the functions actually give an homomorphism between both groups because this would require testing the full multiplication table.

`GroupHomomorphismByFunction` creates a mapping which `IsSPGeneralMapping`.

```
gap> hom:=GroupHomomorphismByFunction(g,h,
> function(x) if SignPerm(x)=-1 then return (1,2); else return ();fi;end);
MappingByFunction( Group([ (1,2,3,4), (1,2) ]), Group([ (1,2,3), (1,2)
 ]), function( x ) ... end )
gap> ImagesSource(hom);
Group([ (1,2), (1,2) ])
gap> Image(hom,(1,2,3,4));
(1,2)
```

The third class are epimorphisms from a group onto its factor group. Such homomorphisms can be constructed by `NaturalHomomorphismByNormalSubgroup` (see 37.18.1).

The fourth class is homomorphisms in a permutation group that are induced by an action on a set. Such homomorphisms are described in the context of group actions, see chapter 39 and in particular section 39.6.1.

5 ► `AsGroupGeneralMappingByImages( `*map*` )` A

If *map* is a mapping from one group to another this attribute returns a group general mapping that which implements the same abstract mapping. (Some operations can be performed more effective in this representation, see also 38.10.2.)

```
gap> AsGroupGeneralMappingByImages(hom);
[ (1,2,3,4), (1,2) ] -> [ (1,2), (1,2) ]
```

38.2 Operations for Group Homomorphisms

Group homomorphisms are mappings, so all the operations and properties for mappings described in chapter 31 are applicable to them. (However often much better methods, than for general mappings are available.)

Group homomorphisms will map groups to groups by just mapping the set of generators.

`KernelOfMultiplicativeGeneralMapping` can be used to compute the kernel of a group homomorphism.

```
gap> hom:=GroupHomomorphismByImages(g,h,gens,[(1,2),(1,3)]);;
gap> Kernel(hom);
Group([ (1,4)(2,3), (1,2)(3,4) ])
```

Homomorphisms can map between groups in different representations and are also used to get isomorphic groups in a different representation.

```
gap> m1:=[[0,-1],[1,0]];;m2:=[[0,-1],[1,1]];;
gap> sl2z:=Group(m1,m2);; # SL(2,Integers) as matrix group
gap> F:=FreeGroup(2);;
gap> psl2z:=F/[F.1^2,F.2^3]; #PSL(2,Z) as FP group
<fp group on the generators [ f1, f2 ]>
gap> phom:=GroupHomomorphismByImagesNC(sl2z,psl2z,[m1,m2],
> GeneratorsOfGroup(psl2z)); # the non NC-version would be expensive
[ [ [ 0, -1 ], [ 1, 0 ] ], [ [ 0, -1 ], [ 1, 1 ] ] ] -> [ f1, f2 ]
gap> Kernel(phom); # the diagonal matrices
Group([ [ [ -1, 0 ], [ 0, -1 ] ], [ [ -1, 0 ], [ 0, -1 ] ] ])
gap> p1:=(1,2)(3,4);;p2:=(2,4,5);;a5:=Group(p1,p2);;
gap> ahom:=GroupHomomorphismByImages(psl2z,a5,
> GeneratorsOfGroup(psl2z),[p1,p2]); # here homomorphism test is cheap.
[ f1, f2 ] -> [ (1,2)(3,4), (2,4,5) ]
gap> u:=PreImage(ahom,Group((1,2,3),(1,2)(4,5)));
Group(<fp, no generators known>)
gap> Index(psl2z,u);
10
gap> isofp:=IsomorphismFpGroup(u);; Image(isofp);
<fp group of size infinity on the generators [ F1, F2, F3, F4 ]>
gap> RelatorsOfFpGroup(Image(isofp));
[ F1^2, F4^2, F3^3 ]
gap> up:=PreImage(phom,u);;
gap> List(GeneratorsOfGroup(up),TraceMat);
[ -2, -2, 0, -4, 1, 0 ]
```

For an automorphism *aut*, `Inverse` returns the inverse automorphism aut^{-1}. However if *hom* is a bijective homomorphism between different groups, or if *hom* is injective and considered to be a bijection to its image,

the operation InverseGeneralMapping should be used instead. (See 30.10.8 for a further discussion of this problem.)

```
gap> iso:=IsomorphismPcGroup(g);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]) -> [ f1, f2, f3, f4 ]
gap> Inverse(iso);
#I  The mapping must be bijective and have source=range
#I  You might want to use 'InverseGeneralMapping'
fail
gap> InverseGeneralMapping(iso);
[ f1, f2, f3, f4 ] -> Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
```

38.3 Efficiency of Homomorphisms

GAP permits to create homomorphisms between arbitrary groups. This section considers the efficiency of the implementation and shows ways how to choose suitable representations. For permutation groups (see 41) or Pc groups (see 44) this is normally nothing to worry about, unless the groups get extremely large. For other groups however certain calculations might be expensive and some precaution might be needed to avoid unnecessarily expensive calculations.

In short, it is always worth to tell a mapping that it is a homomorphism (this can be done by SetIsMapping) (or to create it directly with GroupHomomorphismByImagesNC).

The basic operations required are to compute image and preimage of elements and to test whether a mapping is a homomorphism. Their cost will differ depending on the type of the mapping.

Mappings given on generators (GroupHomomorphismByImages, GroupGeneralMappingByImages)

Computing images requires to express an element of the source as word in the generators. If it cannot be done effectively (this is determined by KnowsHowToDecompose, see 37.25.6 which returns true for example for arbitrary permutation groups, for Pc groups or for finitely presented groups with the images of the free generators) the span of the generators has to be computed elementwise which can be very expensive and memory consuming.

Computing preimages adheres to the same rules with swapped rôles of generators and their images.

The test whether a mapping is a homomorphism requires the computation of a presentation for the source and evaluation of its relators in the images of its generators. For larger groups this can be expensive and GroupHomomorphismByImagesNC should be used if the mapping is known to be a homomorphism.

Action homomorphisms (ActionHomomorphism)

The calculation of images is determined by the acting function used and – for large domains – is often dominated by the search for the position of an image in a list of the domain elements. This can be improved by sorting this list if an efficient method for < to compare elements of the domain is available.

Once the images of a generating set are computed, computing preimages (which is done via the AsGroupGeneralMappingByImages) and computing the kernel bahaves the same as for a GroupHomomorphismByImages in a permutation group.

GAP will always assume that the acting function provided implements a proper group action and thus that the mapping is indeed a homomorphism.

Mappings given by functions (GroupHomomorphismByFunction, GroupGeneralMappingByFunctions)

Computing images is wholly determined by the function that performs the image calculation. If no function to compute preimages is given, computing preimages requires mapping every element of the source to find an element that maps to the requested image. This is time and memory consuming.

Testing whether a `GroupGeneralMappingByFunctions` is a homomorphism would require mapping all products of elements and thus should be avoided.

Other operations

To compute the kernel of a homomorphism (unless the mapping is known to be injective) requires the capability to compute a presentation of the image and to evaluate the relators of this presentation in preimages of the presentations generators.

The calculation of the `Image` (respectively `ImagesSource`) requires to map a generating set of the source, testing surjectivity is a comparison for equality with the range.

Testing injectivity is a test for triviality of the kernel.

The comparison of mappings is based on a lexicographic comparison of a sorted element list of the source. For groups this can be simplified:

1 ▶ `ImagesSmallestGenerators(` *map* `)` A

returns the list of images of `GeneratorsSmallest(Source(`*map*`))`. This list can be used to compare group homomorphisms. (The standard comparison is to compare the image lists on the set of elements of the source. If however x and y have the same images under a and b, certainly all their products have. Therefore it is sufficient to test this on the images of the smallest generators.)

38.4 Homomorphism for very large groups

Some homomorphisms (notably particular actions) transfer known information about the source group (such as a stabilizer chain) to the image group if this is substantially cheaper than to compute the information in the image group anew. In most cases this is no problem and in fact speeds up further calculations notably.

For a huge source group, however this can be time consuming or take a large amount of extra memory for storage. In this case it can be helpful to avoid as much automatism as possible.

The following list of tricks might be useful in such a case. (However you will lose much automatic deduction. So please restrict the use of these to cases where the standard approach does not work.)

- Compute only images (or the `PreImageRepresentative`) of group elements. Do not compute the images of (sub)groups or the full preimage of a subgroup.
- Create action homomorphisms as "surjective" (see `ActionHomomorphism`) (otherwise the range is set to be the full symmetric group) However do not compute Range or Image, but only the images of a generator set.
- If you suspect an action homomorphism to do too much internally, replace the action function with a function that does the same; i.e. replace `OnPoints` by

```
function(p,g) return p^g;end;
```

The action will be the same, but as the action function is not `OnPoints`, the extra processing for special cases is not triggered.

38.5 Nice Monomorphisms

GAP contains very efficient algorithms for some special representations of groups (for example pc groups or permutation groups) while for other representations only slow generic methods are available. In this case it can be worthwhile to do all calculations rather in an isomorphic image of the group, which is in a "better" representation. The way to achieve this in GAP is via **nice monomorphisms**.

For this mechanism to work, of course there must be effective methods to evaluate the `NiceMonomorphism` on elements and to take preimages under it. As by definition no good algorithms exist for the source group, normally this can only be achieved by using an `ActionHomomorphism` or a `GroupHomomorphismByFunction` (see also section 38.3).

1 ▶ `IsHandledByNiceMonomorphism( obj )` P

If this property is `true`, high-valued methods that translate all calculations in *obj* in the image under the `NiceMonomorphism` become available for *obj*.

2 ▶ `NiceMonomorphism( obj )` A

is a homomorphism that is defined (at least) on the whole of *obj* and whose restriction to *obj* is injective. The concrete morphism (and also the image group) will depend on the representation of *obj*.

3 ▶ `NiceObject( obj )` A

The `NiceObject` of *obj* is the image of *obj* under its `NiceMonomorphism`.

A typical example are finite matrix groups, which use a faithful action on vectors to translate all calculations in a permutation group.

```
gap> gl:=GL(3,2);
SL(3,2)
gap> IsHandledByNiceMonomorphism(gl);
true
gap> NiceObject(gl);
Group([ (5,7)(6,8), (2,3,5)(4,7,6) ])
gap> Image(NiceMonomorphism(gl),Z(2)*[[1,0,0],[0,1,1],[1,0,1]]);
(2,6)(3,4,7,8)
```

4 ▶ `IsCanonicalNiceMonomorphism( nhom )` P

A `NiceMonomorphism` *nhom* is canonical if the image set will only depend on the set of group elements but not on the generating set and `<` comparison of group elements translates through the nice monomorphism. This implies that equal objects will always have equal `NiceObject`s. In some situations however this condition would be expensive to achieve, therefore it is not guaranteed for every nice monomorphism.

38.6 Group Automorphisms

Group automorphisms are bijective homomorphism from a group onto itself. An important subclass are automorphisms which are induced by conjugation of the group itself or a supergroup.

1 ▶ `ConjugatorIsomorphism( G, g )` O

Let G be a group, and g an element in the same family as the elements of G. `ConjugatorIsomorphism` returns the isomorphism from G to G^g defined by $h \mapsto h^g$ for all $h \in G$.

If g normalizes G then `ConjugatorIsomorphism` does the same as `ConjugatorAutomorphismNC` (see 38.6.2).

2 ▶ `ConjugatorAutomorphism( G, g )` F
▶ `ConjugatorAutomorphismNC( G, g )` O

Let G be a group, and g an element in the same family as the elements of G such that g normalizes G. `ConjugatorAutomorphism` returns the automorphism of G defined by $h \mapsto h^g$ for all $h \in G$.

If conjugation by g does **not** leave G invariant, `ConjugatorAutomorphism` returns `fail`; in this case, the isomorphism from G to G^g induced by conjugation with g can be constructed with `ConjugatorIsomorphism` (see 38.6.1).

`ConjugatorAutomorphismNC` does the same as `ConjugatorAutomorphism`, except that the check is omitted whether g normalizes G.

3 ▶ `InnerAutomorphism( G, g )` F
▶ `InnerAutomorphismNC( G, g )` O

Let G be a group, and $g \in G$. `InnerAutomorphism` returns the automorphism of G defined by $h \mapsto h^g$ for all $h \in G$.

If g is **not** an element of G, `InnerAutomorphism` returns `fail`; in this case, the isomorphism from G to G^g induced by conjugation with g can be constructed with `ConjugatorIsomorphism` (see 38.6.1) or with `ConjugatorAutomorphism` (see 38.6.2).

`InnerAutomorphismNC` does the same as `InnerAutomorphism`, except that the check is omitted whether $g \in G$.

4 ▶ `IsConjugatorIsomorphism( hom )` P
▶ `IsConjugatorAutomorphism( hom )` P
▶ `IsInnerAutomorphism( hom )` P

Let *hom* be a group general mapping (see 31.8.4) with source G, say. `IsConjugatorIsomorphism` returns `true` if *hom* is induced by conjugation of G by an element g that lies in G or in a group into which G is naturally embedded in the sense described below, and `false` otherwise. Natural embeddings are dealt with in the case that G is a permutation group (see Chapter 41), a matrix group (see Chapter 42), a finitely presented group (see Chapter 45), or a group given w.r.t. a polycyclic presentation (see Chapter 44). In all other cases, `IsConjugatorIsomorphism` may return `false` if *hom* is induced by conjugation but is not an inner automorphism.

If `IsConjugatorIsomorphism` returns `true` for *hom* then an element g that induces *hom* can be accessed as value of the attribute `ConjugatorOfConjugatorIsomorphism` (see 38.6.5).

`IsConjugatorAutomorphism` returns `true` if *hom* is an automorphism (see 31.12.3) that is regarded as a conjugator isomorphism by `IsConjugatorIsomorphism`, and `false` otherwise.

`IsInnerAutomorphism` returns `true` if *hom* is a conjugator automorphism such that an element g inducing *hom* can be chosen in G, and `false` otherwise.

5 ▶ `ConjugatorOfConjugatorIsomorphism( hom )` A

For a conjugator isomorphism *hom* (see 38.6.1), `ConjugatorOfConjugatorIsomorphism` returns an element g such that mapping under *hom* is induced by conjugation with g.

To avoid problems with `IsInnerAutomorphism`, it is guaranteed that the conjugator is taken from the source of *hom* if possible.

```
gap> hgens:=[(1,2,3),(1,2,4)];;h:=Group(hgens);;
gap> hom:=GroupHomomorphismByImages(h,h,hgens,[(1,2,3),(2,3,4)]);;
gap> IsInnerAutomorphism(hom);
true
gap> ConjugatorOfConjugatorIsomorphism(hom);
(1,2,3)
gap> hom:=GroupHomomorphismByImages(h,h,hgens,[(1,3,2),(1,4,2)]);
[ (1,2,3), (1,2,4) ] -> [ (1,3,2), (1,4,2) ]
gap> IsInnerAutomorphism(hom);
false
gap> IsConjugatorAutomorphism(hom);
true
gap> ConjugatorOfConjugatorIsomorphism(hom);
(1,2)
```

38.7 Groups of Automorphisms

Group automorphism can be multiplied and inverted and thus it is possible to form groups of automorphisms.

1 ▶ `IsGroupOfAutomorphisms( G )` P

indicates whether G consists of automorphisms of another group H. The group H can be obtained from G via the attribute `AutomorphismDomain`.

2 ▶ `AutomorphismDomain( G )` A

If G consists of automorphisms of H, this attribute returns H.

3 ▶ `AutomorphismGroup( obj )` A

returns the full automorphism group of the object *obj*. The automorphisms act on the domain by the caret operator `^`. The automorphism group often stores a "NiceMonomorphism" (see 38.5.2) to a permutation group, obtained by the action on a subset of *obj*.

4 ▶ `IsAutomorphismGroup( G )` P

indicates whether G is the full automorphism group of another group H, this group is given as `AutomorphismDomain` of G.

```
gap> g:=Group((1,2,3,4),(1,3));
Group([ (1,2,3,4), (1,3) ])
gap> au:=AutomorphismGroup(g);
<group of size 8 with 3 generators>
gap> GeneratorsOfGroup(au);
[ ^(1,2,3,4), ^(1,3), [ (1,2,3,4), (2,4) ] -> [ (1,2,3,4), (1,2)(3,4) ] ]
gap> NiceObject(au);
Group([ (1,4)(2,6), (2,6)(3,5), (1,2,4,6) ])
```

5 ▶ `InnerAutomorphismsAutomorphismGroup( autgroup )` A

For an automorphism group *autgroup* of a group this attribute stores the subgroup of inner automorphisms (automorphisms induced by conjugation) of the original group.

```
gap> InnerAutomorphismsAutomorphismGroup(au);
<group with 2 generators>
```

6 ▶ `InducedAutomorphism( epi, aut )` O

Let *aut* be an automorphism of a group G and *epi*: `G -> H` an homomorphism such that `ker` *epi* is fixed under *aut*. Let U be the image of *epi*. This command returns the automorphism of U induced by *aut* via *epi*, that is the automorphism of U which maps *g^epi* to *(g^aut)^epi*, for $g \in G$.

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> n:=Subgroup(g,[(1,2)(3,4),(1,3)(2,4)]);
Group([ (1,2)(3,4), (1,3)(2,4) ])
gap> epi:=NaturalHomomorphismByNormalSubgroup(g,n);
[ (1,2,3,4), (1,2) ] -> [ f1*f2, f1 ]
gap> aut:=InnerAutomorphism(g,(1,2,3));
^(1,2,3)
gap> InducedAutomorphism(epi,aut);
^f2
```

38.8 Calculating with Group Automorphisms

Usually the best way to calculate in a group of automorphisms is to go translate all calculations to an isomorphic group in a representation, for which better algorithms are available, say a permutation group. This translation can be done automatically using a `NiceMonomorphism` (see 38.5.2.)

Once a group knows to be a group of automorphisms (this can be achieved by testing or setting the property `IsGroupOfAutomorphisms` (see 38.7.1), GAP will try itself to find such a nice monomorphism once calculations in the automorphism group are done.

Note that nice homomorphisms inherit down to subgroups, but cannot necessarily be extended from a subgroup to the whole group. Thus when working with a group of automorphisms, it can be beneficial to enforce calculation of the nice monomorphism for the whole group (for example by explicitly calling `Random(G)` and ignoring the result – it will be stored internally) at the start of the calculation. Otherwise GAP might first calculate a nice monomorphism for the subgroup, only to be forced to calculate a new nice monomorphism for the whole group later on.

1 ▶ `AssignNiceMonomorphismAutomorphismGroup( autgrp, group )` F

computes a nice monomorphism for *autgroup* acting on *group* and stores it as `NiceMonomorphism` in *autgrp*.

If the centre of `AutomorphismDomain` of *autgrp* is trivial, the operation will first try to represent all automorphisms by conjugation (in *group* or a natural parent of *group*).

If this fails the operation tries to find a small subset of *group* on which the action will be faithful.

The operation sets the attribute `NiceMonomorphism` and does not return a value.

If a good domain for a faithful permutation action is known already, a homomorphism for the action on it can be created using `NiceMonomorphismAutomGroup`. It might be stored by `SetNiceMonomorphism` (see 38.5.2).

2 ▶ `NiceMonomorphismAutomGroup( autgrp, elms, elmsgens )` F

This function creates a monomorphism for an automorphism group *autgrp* of a group by permuting the group elements in the list *elms*. This list must be chosen to yield a faithful representation. *elmsgens* is a list of generators which are a subset of *elms*. (They can differ from the groups original generators.) It does not yet assign it as `NiceMonomorphism`.

Another nice way of representing automorphisms as permutations has been described in [Sim97]. It it not yet available in GAP, a description however can be found in section 8.3 of "Extending GAP".

38.9 Searching for Homomorphisms

1 ▶ `IsomorphismGroups( G, H )` F

computes an isomorphism between the groups G and H if they are isomorphic and returns `fail` otherwise.

With the existing methods the amount of time needed grows with the size of a generating system of G. (Thus in particular for p-groups calculations can be slow.) If you do only need to know whether groups are isomorphic, you might want to consider `IdSmallGroup` (see 48.7.5) or the random isomorphism test (see 44.10.1).

```
gap> g:=Group((1,2,3,4),(1,3));;
gap> h:=Group((1,4,6,7)(2,3,5,8), (1,5)(2,6)(3,4)(7,8));;
gap> IsomorphismGroups(g,h);
[ (1,2,3,4), (1,3) ] -> [ (1,4,6,7)(2,3,5,8), (1,2)(3,7)(4,8)(5,6) ]
gap> IsomorphismGroups(g,Group((1,2,3,4),(1,2)));
fail
```

2 ▶ `GQuotients( F, G )` O

computes all epimorphisms from F onto G up to automorphisms of G. This classifies all factor groups of F which are isomorphic to G.

With the existing methods the amount of time needed grows with the size of a generating system of G. (Thus in particular for p-groups calculations can be slow.)

If the `findall` option is set to `false`, the algorithm will stop once one homomorphism has been found (this can be faster and might be sufficient if not all homomorphisms are needed).

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> h:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> quo:=GQuotients(g,h);
[ [ (1,4,2,3), (1,2,4) ] -> [ (2,3), (1,2,3) ] ]
```

3 ▶ `IsomorphicSubgroups( G, H )` O

computes all monomorphisms from H into G up to G-conjugacy of the image groups. This classifies all G-classes of subgroups of G which are isomorphic to H.

With the existing methods, the amount of time needed grows with the size of a generating system of G. (Thus in particular for p-groups calculations can be slow.) A main use of `IsomorphicSubgroups` therefore is to find nonsolvable subgroups (which often can be generated by 2 elements).

(To find p-subgroups it is often faster to compute the subgroup lattice of the sylow subgroup and to use `IdGroup` to identify the type of the subgroups.)

If the `findall` option is set to `false`, the algorithm will stop once one homomorphism has been found (this can be faster and might be sufficient if not all homomorphisms are needed).

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
gap> h:=Group((3,4),(1,2));;
gap> emb:=IsomorphicSubgroups(g,h);
[ [ (3,4), (1,2) ] -> [ (3,4), (1,2) ],
  [ (3,4), (1,2) ] -> [ (1,3)(2,4), (1,2)(3,4) ] ]
```

4 ▶ `MorClassLoop( range, classes, params, action )` F

This function loops over element tuples taken from *classes* and checks these for properties such as generating a given group, or fulfilling relations. This can be used to find small generating sets or all types of Morphisms.

The element tuples are used only up to up to inner automorphisms as all images can be obtained easily from them by conjugation while running through all of them usually would take too long.

range is a group from which these elements are taken. The classes are given in a list *classes* which is a list of records with components

`classes`
: A list of conjugacy classes `representative`
One element in the union of these classes `size`
The sum of the sizes of these classes

params is a record containing optional components:

`gens`
: generators that are to be mapped (for testing morphisms). The length of this list determines the length of element tuples considered.

`from`
: a preimage group (that contains *gens*)

`to`
: image group (which might be smaller than `range`)

`free`
: free generators, a list of the same length than the generators `gens`.

`rels`
: some relations that hold among the generators `gens`. They are given as a list [*word*,*order*] where *word* is a word in the free generators `free`.

`dom`
: a set of elements on which automorphisms act faithfully (used to do element tests in partial automorphism groups).

`aut`
: Subgroup of already known automorphisms.

action is a number whose bit-representation indicates the requirements which are enforced on the element tuples found:

1 homomorphism

2 injective

4 surjective

8 find all (otherwise stops after the first find)

If the search is for homomorphisms, the function returns homomorphisms obtained by mapping the given generators `gens` instead of element tuples.

The "Morpheus" algorithm used to find homomorphisms is described in section V.5 of [Hul96].

38.10 Representations for Group Homomorphisms

The different representations of group homomorphisms are used to indicate from what type of group to what type of group they map and thus determine which methods are used to compute images and preimages.

The information in this section is mainly relevant for implementing new methods and not for using homomorphisms.

1 ▶ `IsGroupGeneralMappingByImages( map )` R

Representation for mappings from one group to another that are defined by extending a mapping of group generators homomorphically. Instead of record components, the attribute `MappingGeneratorImages` is used to store generators and their images.

2 ▶ `IsGroupGeneralMappingByAsGroupGeneralMappingByImages( map )` R

Representation for mappings that delegate work on a `GroupHomomorphismByImages`.

3 ▶ `IsPreimagesByAsGroupGeneralMappingByImages( map )` R

Representation for mappings that delegate work for preimages to a GroupHomomorphismByImages.

4 ▶ `IsPermGroupGeneralMappingByImages( map )` R
▶ `IsPermGroupHomomorphismByImages( map )` R

is the representation for mappings that map from a perm group

5 ▶ `IsToPermGroupGeneralMappingByImages( map )` R
▶ `IsToPermGroupHomomorphismByImages( map )` R

is the representation for mappings that map to a perm group

6 ▶ `IsGroupGeneralMappingByPcgs( map )` R

is the representations for mappings that map a pcgs to images and thus may use exponents to decompose generators.

7 ▶ `IsPcGroupGeneralMappingByImages( map )` R
▶ `IsPcGroupHomomorphismByImages( map )` R

is the representation for mappings from a pc group

8 ▶ `IsToPcGroupGeneralMappingByImages( map )` R
▶ `IsToPcGroupHomomorphismByImages( map )` R

is the representation for mappings to a pc group

9 ▶ `IsFromFpGroupGeneralMappingByImages( map )` R
▶ `IsFromFpGroupHomomorphismByImages( map )` R

is the representation of mappings from an fp group.

10 ▶ `IsFromFpGroupStdGensGeneralMappingByImages( map )` R
▶ `IsFromFpGroupStdGensHomomorphismByImages( map )` R

is the representation of mappings from an fp group that give images of the standard generators.

39 Group Actions

A **group action** is a triple $(G, Omega, \mu)$, where G is a group, *Omega* a set and μ: $Omega \times G \to Omega$ a function (whose action is compatible with the group arithmetic). We call *Omega* the **domain** of the action.

In GAP, *Omega* can be a duplicate-free collection (an object that permits access to its elements via the $Omega[n]$ operation, for example a list), it does not need to be sorted (see 21.17.4).

The acting function μ is a GAP function of the form

`actfun(`*pnt*`,`*g*`)`

that returns the image $\mu(pnt, g)$ for a point $pnt \in Omega$ and a group element $g \in G$.

Groups always acts from the right, that is $\mu(\mu(pnt, g), h) = \mu(pnt, gh)$.

GAP does not test whether an acting function `actfun` satisfies the conditions for a group operation but silently assumes that is does. (If it does not, results are unpredictable.)

The first section of this chapter, 39.1, describes the various ways how operations for group actions can be called.

Functions for several commonly used action are already built into GAP. These are listed in section 39.2.

The sections 39.6 and 39.7 describe homomorphisms and mappings associated to group actions as well as the permutation group image of an action.

The other sections then describe operations to compute orbits, stabilizers, as well as properties of actions.

Finally section 39.11 describes the concept of "external sets" which represent the concept of a G-**set** and underly the actions mechanism.

39.1 About Group Actions

The syntax which is used by the operations for group actions is quite flexible. For example we can call the operation `OrbitsDomain` for the orbits of the group G on the domain *Omega* in the following ways:

`OrbitsDomain(`G`,`*Omega*`[,`*actfun*`])`

The acting function *actfun* is optional. If it is not given, the built-in action `OnPoints` (which defines an action via the caret operator `^`) is used as a default.

`OrbitsDomain(`G`,`*Omega*`,`*gens*`,`*acts*`[,`*actfun*`])`

This second version (of `OrbitsDomain`) permits one to implement an action induced by a homomorphism: If H acts on *Omega* via μ and φ: $G \to H$ is a homomorphism, G acts on *Omega* via $\mu'(\omega, g) = \mu(\omega, g^{\varphi})$:

Here *gens* must be a set of generators of G and *acts* the images of *gens* under a homomorphism φ: $G \to H$. *actfun* is the acting function for H, the call to `ExampleActionFunction` implements the induced action of G. Again, the acting function *actfun* is optional and `OnPoints` is used as a default.

The advantage of this notation is that GAP does not need to construct this homomorphism φ and the range group H as GAP objects. (If a small group G acts via complicated objects *acts* this otherwise could lead to performance problems.)

GAP does not test whether the mapping *gens* ↦ *acts* actually induces a homomorphism and the results are unpredictable if this is not the case.

`OrbitsDomain(`*extset*`) A`

A third variant is to call the operation with an external set (which then provides *G*, *Omega* and *actfun*. You will find more about external sets in section 39.11.

For operations like `Stabilizer` of course the domain must be replaced by an element of *Omega* which is to be acted on.

39.2 Basic Actions

GAP already provides acting functions for the more common actions of a group. For built-in operations such as `Stabilizer` special methods are available for many of these actions.

This section also shows how to implement different actions. (Note that every action must be from the right.)

1 ▶ `OnPoints(` *pnt*`,` *g* `)` F

returns *pnt* `^` *g*. This is for example the action of a permutation group on points, or the action of a group on its elements via conjugation. The action of a matrix group on vectors from the right is described by both `OnPoints` and `OnRight` (see 39.2.2).

2 ▶ `OnRight(` *pnt*`,` *g* `)` F

returns *pnt* `*` *g*. This is for example the action of a group on its elements via right multiplication, or the action of a group on the cosets of a subgroup. The action of a matrix group on vectors from the right is described by both `OnPoints` (see 39.2.1) and `OnRight`.

3 ▶ `OnLeftInverse(` *pnt*`,` *g* `)` F

returns g^{-1} `*` *pnt*. Forming the inverse is necessary to make this a proper action, as in GAP groups always act from the right.

(`OnLeftInverse` is used for example in the representation of a right coset as an external set (see 39.11), that is a right coset *Ug* is an external set for the group *U* acting on it via `OnLeftInverse`.)

4 ▶ `OnSets(` *set*`,` *g* `)` F

Let *set* be a proper set (see 21.19). `OnSets` returns the proper set formed by the images `OnPoints(` *pnt*`,` *g* `)` of all points *pnt* of *set*.

`OnSets` is for example used to compute the action of a permutation group on blocks.

(`OnTuples` is an action on lists that preserves the ordering of entries, see 39.2.5.)

5 ▶ `OnTuples(` *tup*`,` *g* `)` F

Let *tup* be a list. `OnTuples` returns the list formed by the images `OnPoints(` *pnt*`,` *g* `)` for all points *pnt* of *tup*.

(`OnSets` is an action on lists that additionally sorts the entries of the result, see 39.2.4.)

6 ▶ `OnPairs(` *tup*`,` *g* `)` F

is a special case of `OnTuples` (see 39.2.5) for lists *tup* of length 2.

7 ▶ `OnSetsSets(` *set*`,` *g* `)` F

Action on sets of sets; for the special case that the sets are pairwise disjoint, it is possible to use `OnSetsDisjointSets` (see 39.2.8).

8 ▶ `OnSetsDisjointSets( set, g )` F

Action on sets of pairwise disjoint sets (see also 39.2.7).

9 ▶ `OnSetsTuples( set, g )` F

Action on sets of tuples.

10 ▶ `OnTuplesSets( set, g )` F

Action on tuples of sets.

11 ▶ `OnTuplesTuples( set, g )` F

Action on tuples of tuples

```
gap> g:=Group((1,2,3),(2,3,4));;
gap> Orbit(g,1,OnPoints);
[ 1, 2, 3, 4 ]
gap> Orbit(g,(),OnRight);
[ (), (1,2,3), (2,3,4), (1,3,2), (1,3)(2,4), (1,2)(3,4), (2,4,3), (1,4,2),
  (1,4,3), (1,3,4), (1,2,4), (1,4)(2,3) ]
gap> Orbit(g,[1,2],OnPairs);
[ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ], [ 3, 1 ], [ 3, 4 ], [ 2, 1 ], [ 1, 4 ],
  [ 4, 1 ], [ 4, 2 ], [ 3, 2 ], [ 2, 4 ], [ 4, 3 ] ]
gap> Orbit(g,[1,2],OnSets);
[ [ 1, 2 ], [ 2, 3 ], [ 1, 3 ], [ 3, 4 ], [ 1, 4 ], [ 2, 4 ] ]

gap> Orbit(g,[[1,2],[3,4]],OnSetsSets);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 4 ], [ 2, 3 ] ], [ [ 1, 3 ], [ 2, 4 ] ] ]
gap> Orbit(g,[[1,2],[3,4]],OnTuplesSets);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 2, 3 ], [ 1, 4 ] ], [ [ 1, 3 ], [ 2, 4 ] ],
  [ [ 3, 4 ], [ 1, 2 ] ], [ [ 1, 4 ], [ 2, 3 ] ], [ [ 2, 4 ], [ 1, 3 ] ] ]
gap> Orbit(g,[[1,2],[3,4]],OnSetsTuples);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 1, 4 ], [ 2, 3 ] ], [ [ 1, 3 ], [ 4, 2 ] ],
  [ [ 2, 4 ], [ 3, 1 ] ], [ [ 2, 1 ], [ 4, 3 ] ], [ [ 3, 2 ], [ 4, 1 ] ] ]
gap> Orbit(g,[[1,2],[3,4]],OnTuplesTuples);
[ [ [ 1, 2 ], [ 3, 4 ] ], [ [ 2, 3 ], [ 1, 4 ] ], [ [ 1, 3 ], [ 4, 2 ] ],
  [ [ 3, 1 ], [ 2, 4 ] ], [ [ 3, 4 ], [ 1, 2 ] ], [ [ 2, 1 ], [ 4, 3 ] ],
  [ [ 1, 4 ], [ 2, 3 ] ], [ [ 4, 1 ], [ 3, 2 ] ], [ [ 4, 2 ], [ 1, 3 ] ],
  [ [ 3, 2 ], [ 4, 1 ] ], [ [ 2, 4 ], [ 3, 1 ] ], [ [ 4, 3 ], [ 2, 1 ] ] ]
```

12 ▶ `OnLines( vec, g )` F

Let *vec* be a **normed** row vector, that is, its first nonzero entry is normed to the identity of the relevant field, `OnLines` returns the row vector obtained from normalizing `OnRight( vec, g )` by scalar multiplication from the left. This action corresponds to the projective action of a matrix group on 1-dimensional subspaces.

```
gap> gl:=GL(2,5);;v:=[1,0]*Z(5)^0;
[ Z(5)^0, 0*Z(5) ]
gap> h:=Action(gl,Orbit(gl,v,OnLines),OnLines);
Group([ (2,3,5,6), (1,2,4)(3,6,5) ])
```

13 ▶ `OnIndeterminates( poly, perm )` F

A permutation *perm* acts on the multivariate polynomial *poly* by permuting the indeterminates as it permutes points.

14 ▶ `Permuted( list, perm )`

The following example demonstrates `Permuted` being used to implement a permutation action on a domain:

```
gap> g:=Group((1,2,3),(1,2));;
gap> dom:=[ "a", "b", "c" ];;
gap> Orbit(g,dom,Permuted);
[ [ "a", "b", "c" ], [ "c", "a", "b" ], [ "b", "a", "c" ], [ "b", "c", "a" ],
  [ "a", "c", "b" ], [ "c", "b", "a" ] ]
```

15 ▶ `OnSubspacesByCanonicalBasis( bas, mat )` F

implements the operation of a matrix group on subspaces of a vector space. *bas* must be a list of (linearly independent) vectors which forms a basis of the subspace in Hermite normal form. *mat* is an element of the acting matrix group. The function returns a mutable matrix which gives the basis of the image of the subspace in Hermite normal form. (In other words: it triangulizes the product of *bas* with *mat*.)

If one needs an action for which no acting function is provided by the library it can be implemented via a GAP function that conforms to the syntax

`actfun(`*omega*`,`*g*`)`

For example one could define the following function that acts on pairs of polynomials via `OnIndeterminates`:

```
OnIndeterminatesPairs:=function(polypair,g)
  return [OnIndeterminates(polypair[1],g),
          OnIndeterminates(polypair[2],g)];
end;
```

Note that this function **must** implement an action from the **right**. This is not verified by GAP and results are unpredicatble otherwise.

39.3 Orbits

If G acts on *Omega* the set of all images of $\omega \in$ *Omega* under elements of G is called the **orbit** of ω. The set of orbits of G is a partition of *Omega*.

Note that currently GAP does **not** check whether a given point really belongs to Ω. For example, consider the following example where the projective action of a matrix group on a finite vector space shall be computed.

```
gap> Orbit( GL(2,3), [ -1, 0 ] * Z(3)^0, OnLines );
[ [ Z(3), 0*Z(3) ], [ Z(3)^0, 0*Z(3) ], [ Z(3)^0, Z(3) ], [ Z(3)^0, Z(3)^0 ],
  [ 0*Z(3), Z(3)^0 ] ]
gap> Size( GL(2,3) ) / Length( last );
48/5
```

The error is that `OnLines` (see 39.2.12) acts on the set of normed row vectors (see 59.8.11) of the vector space in question, but that the seed vector is itself not such a vector.

1 ▶ `Orbit( G[, Omega], pnt, [gens, acts, ] act )` O

The orbit of the point *pnt* is the list of all images of *pnt* under the action.

(Note that the arrangement of points in this list is not defined by the operation.)

The orbit of *pnt* will always contain one element that is **equal** to *pnt*, however for performance reasons this element is not necessarily **identical** to *pnt*, in particular if *pnt* is mutable.

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> Orbit(g,1);
[ 1, 3, 2, 4 ]
gap> Orbit(g,[1,2],OnSets);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ], [ 2, 4 ] ]
```

(See Section 39.2 for information about specific actions.)

2 ► `Orbits(` *G*, *seeds*`[,` *gens*, *acts*`][,` *act*`] )` O
► `Orbits(` *xset* `)` A

returns a duplicate-free list of the orbits of the elements in *seeds* under the action *act* of *G*

(Note that the arrangement of orbits or of points within one orbit is not defined by the operation.)

3 ► `OrbitsDomain(` *G*, *Omega*`[,` *gens*, *acts*`][,` *act*`] )` O
► `OrbitsDomain(` *xset* `)` A

returns a list of the orbits of *G* on the domain *Omega* (given as lists) under the action *act*.

This operation is often faster than `Orbits`. The domain *Omega* must be closed under the action of *G*, otherwise an error can occur.

(Note that the arrangement of orbits or of points within one orbit is not defined by the operation.)

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> Orbits(g,[1..5]);
[ [ 1, 3, 2, 4 ], [ 5 ] ]
gap> OrbitsDomain(g,Arrangements([1..4],3),OnTuples);
[ [ [ 1, 2, 3 ], [ 3, 1, 2 ], [ 1, 4, 2 ], [ 2, 3, 1 ], [ 2, 1, 4 ],
      [ 3, 4, 1 ], [ 1, 3, 4 ], [ 4, 2, 1 ], [ 4, 1, 3 ], [ 2, 4, 3 ],
      [ 3, 2, 4 ], [ 4, 3, 2 ] ],
  [ [ 1, 2, 4 ], [ 3, 1, 4 ], [ 1, 4, 3 ], [ 2, 3, 4 ], [ 2, 1, 3 ],
      [ 3, 4, 2 ], [ 1, 3, 2 ], [ 4, 2, 3 ], [ 4, 1, 2 ], [ 2, 4, 1 ],
      [ 3, 2, 1 ], [ 4, 3, 1 ] ] ]
gap> OrbitsDomain(g,GF(2)^2,[(1,2,3),(1,4)(2,3)],
> [[[Z(2)^0,Z(2)^0],[Z(2)^0,0*Z(2)]],[[Z(2)^0,0*Z(2)],[0*Z(2),Z(2)^0]]]);
[ [ <an immutable GF2 vector of length 2> ],
  [ <an immutable GF2 vector of length 2>, <an immutable GF2 vector of length
      2>, <an immutable GF2 vector of length 2> ] ]
```

(See Section 39.2 for information about specific actions.)

4 ► `OrbitLength(` *G*, *Omega*, *pnt*, `[`*gens*, *acts*, `]` *act* `)` O

computes the length of the orbit of *pnt*.

5 ► `OrbitLengths(` *G*, *seeds*`[,` *gens*, *acts*`][,` *act*`] )` O
► `OrbitLengths(` *xset* `)` A

computes the lengths of all the orbits of the elements in *seegs* under the action *act* of *G*.

6 ► `OrbitLengthsDomain(` *G*, *Omega*`[,` *gens*, *acts*`][,` *act*`] )` O
► `OrbitLengthsDomain(` *xset* `)` A

computes the lengths of all the orbits of *G* on *Omega*.

This operation is often faster than `OrbitLengths`. The domain *Omega* must be closed under the action of *G*, otherwise an error can occur.

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> OrbitLength(g,[1,2,3,4],OnTuples);
12
gap> OrbitLengths(g,Arrangements([1..4],4),OnTuples);
[ 12, 12 ]
```

39.4 Stabilizers

The **Stabilizer** of an element ω is the set of all those $g \in G$ which fix ω.

1 ▶ `OrbitStabilizer( G, [Omega, ] pnt, [gens, acts, ] act )` O

computes the orbit and the stabilizer of *pnt* simultaneously in a single Orbit-Stabilizer algorithm.

The stabilizer must have G as its parent.

2 ▶ `Stabilizer( G [, Omega], pnt [, gens, acts] [, act] )` F

computes the stabilizer in G of the point *pnt*, that is the subgroup of those elements of G that fix *pnt*. The stabilizer will have G as its parent.

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> Stabilizer(g,4);
Group([ (1,3,2) ])
```

The stabilizer of a set or tuple of points can be computed by specifying an action of sets or tuples of points.

```
gap> Stabilizer(g,[1,2],OnSets);
Group([ (1,2)(3,4) ])
gap> Stabilizer(g,[1,2],OnTuples);
Group(())
gap> OrbitStabilizer(g,[1,2],OnSets);
rec( orbit := [ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 2, 3 ], [ 3, 4 ], [ 2, 4 ] ],
  stabilizer := Group([ (1,2)(3,4) ]) )
```

(See Section 39.2 for information about specific actions.)

The standard methods for all these actions are an Orbit-Stabilizer algorithm. For permutation groups backtrack algorithms are used. For solvable groups an orbit-stabilizer algorithm for solvable groups, which uses the fact that the orbits of a normal subgroup form a block system (see [LNS84]) is used.

3 ▶ `OrbitStabilizerAlgorithm( G, Omega, blist, gens, acts, pntact )` F

This operation should not be called by a user. It is documented however for purposes to extend or maintain the group actions package.

`OrbitStabilizerAlgorithm` performs an orbit stabilizer algorithm for the group G acting with the generators *gens* via the generator images *gens* and the group action *act* on the element *pnt*. (For technical reasons *pnt* and *act* are put in one record with components `pnt` and `act` respectively.)

The *pntact* record may carry a component *stabsub*. If given, this must be a subgroup stabilizing **all** points in the domain and can be used to abbreviate stabilizer calculations.

The argument *Omega* (which may be replaced by `false` to be ignored) is the set within which the orbit is computed (once the orbit is the full domain, the orbit calculation may stop). If *blist* is given it must be a bit list corresponding to *Omega* in which elements which have been found already will be "ticked off" with `true`. (In particular, the entries for the orbit of *pnt* still must be all set to `false`). Again the remaining action domain (the bits set initially to `false`) can be used to stop if the orbit cannot grow any longer. Another use of the bit list is if *Omega* is an enumerator which can determine `PositionCanonical`s very quickly. In this situation it can be worth to search images not in the orbit found so far, but via their position in *Omega* and use a the bit list to keep track whether the element is in the orbit found so far.

39.5 Elements with Prescribed Images

1 ▶ `RepresentativeAction(` *G* `[,` *Omega*`],` *d*`,` *e* `[,` *gens*`,` *acts*`] [,` *act*`] )` O

computes an element of G that maps d to e under the given action and returns `fail` if no such element exists.

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> RepresentativeAction(g,1,3);
(1,3)(2,4)
gap> RepresentativeAction(g,1,3,OnPoints);
(1,3)(2,4)
gap> RepresentativeAction(g,(1,2,3),(2,4,3));
(1,2,4)
gap> RepresentativeAction(g,(1,2,3),(2,3,4));
fail
gap> RepresentativeAction(g,Group((1,2,3)),Group((2,3,4)));
(1,2,4)
gap>  RepresentativeAction(g,[1,2,3],[1,2,4],OnSets);
(2,4,3)
gap>  RepresentativeAction(g,[1,2,3],[1,2,4],OnTuples);
fail
```

(See Section 39.2 for information about specific actions.)

Again the standard method for `RepresentativeAction` is an orbit-stabilizer algorithm, for permutation groups and standard actions a backtrack algorithm is used.

39.6 The Permutation Image of an Action

If G acts on a domain *Omega*, an enumeration of *Omega* yields a homomorphism of G into the symmetric group on $\{1, \ldots, |Omega|\}$. In GAP, the enumeration of the domain *Omega* is provided by the `Enumerator` of *Omega* (see 28.2.2) which of course is *Omega* itself if it is a list.

1 ▶ `ActionHomomorphism(` *G*`,` *Omega* `[,` *gens*`,` *acts*`] [,` *act*`] [, "surjective"] )` O
▶ `ActionHomomorphism(` *xset* `[, "surjective"] )` A
▶ `ActionHomomorphism(` *action* `)` A

computes a homomorphism from G into the symmetric group on $|Omega|$ points that gives the permutation action of G on *Omega*.

By default the homomorphism returned by `ActionHomomorphism` is not necessarily surjective (its `Range` is the full symmetric group) to avoid unnecessary computation of the image. If the optional string argument `"surjective"` is given, a surjective homomorphism is created.

The third version (which is supported only for GAP3 compatibility) returns the action homomorphism that belongs to the image obtained via `Action` (see 39.6.2).

(See Section 39.2 for information about specific actions.)

```
gap> g:=Group((1,2,3),(1,2));;
gap> hom:=ActionHomomorphism(g,Arrangements([1..4],3),OnTuples);
<action homomorphism>
gap> Image(hom);
Group([ (1,9,13)(2,10,14)(3,7,15)(4,8,16)(5,12,17)(6,11,18)(19,22,23)(20,21,
    24), (1,7)(2,8)(3,9)(4,10)(5,11)(6,12)(13,15)(14,16)(17,18)(19,21)(20,
    22)(23,24) ])
gap> Size(Range(hom));Size(Image(hom));
620448401733239439360000
6
gap> hom:=ActionHomomorphism(g,Arrangements([1..4],3),OnTuples,
> "surjective");;
gap> Size(Range(hom));
6
```

When acting on a domain, the operation `PositionCanonical` is used to determine the position of elements in the domain. This can be used to act on a domain given by a list of representatives for which `PositionCanonical` is implemented, for example a `RightTransversal` (see 37.8.1).

2 ► `Action(` *G*, *Omega* [*gens*, *acts*] [, *act*] `)` O
► `Action(` *xset* `)` A

returns the `Image` group of `ActionHomomorphism` called with the same parameters.

Note that (for compatibility reasons to be able to get the action homomorphism) this image group internally stores the action homomorphism. If *G* or *Omega* are exteremly big, this can cause memory problems. In this case compute only generator images and form the image group yourself.

(See Section 39.2 for information about specific actions.) The following code shows for example how to create the regular action of a group:

```
gap> g:=Group((1,2,3),(1,2));;
gap> Action(g,AsList(g),OnRight);
Group([ (1,4,5)(2,3,6), (1,3)(2,4)(5,6) ])
```

3 ► `SparseActionHomomorphism(` *G*, *Omega*, *start* [, *gens*, *acts*] [, *act*] `)` O
► `SortedSparseActionHomomorphism(` *G*, *Omega*, *start*[, *gens*, *acts*] [, *act*] `)` O

`SparseActionHomomorphism` computes the `ActionHomomorphism(`*G*,*dom*`[,`*gens*,*acts*`][,`*act*`])`, where *dom* is the union of the orbits `Orbit(`*G*,*pnt*`[,`*gens*,*acts*`][,`*act*`])` for all points *pnt* from *start*. If *G* acts on a very large domain *Omega* not surjectively this may yield a permutation image of substantially smaller degree than by action on *Omega*.

The operation `SparseActionHomomorphism` will only use `=` comparisons of points in the orbit. Therefore it can be used even if no good `<` comparison method exists. However the image group will depend on the generators *gens* of *G*.

The operation `SortedSparseActionHomomorphism` in contrast will sort the orbit and thus produce an image group which is not dependent on these generators.

```
gap> h:=Group(Z(3)*[[[1,1],[0,1]]]);
Group([ [ [ Z(3), Z(3) ], [ 0*Z(3), Z(3) ] ] ])
gap> hom:=ActionHomomorphism(h,GF(3)^2,OnRight);;
gap> Image(hom);
Group([ (2,3)(4,9,6,7,5,8) ])
gap> hom:=SparseActionHomomorphism(h,[Z(3)*[1,0]],OnRight);;
gap> Image(hom);
```

```
Group([ (1,2,3,4,5,6) ])
```

For an action homomorphism, the operation `UnderlyingExternalSet` (see 39.11.16) will return the external set on *Omega* which affords the action.

39.7 Action of a group on itself

Of particular importance is the action of a group on its elements or cosets of a subgroup. These actions can be obtained by using `ActionHomomorphism` for a suitable domain (for example a list of subgroups). For the following (frequently used) types of actions however special (often particularly efficient) functions are provided:

1 ▶ `FactorCosetAction( G, U, [N] )` O

This command computes the action of G on the right cosets of the subgroup U. If the normal subgroup N is given, it is stored as kernel of this action.

```
gap> g:=Group((1,2,3,4,5),(1,2));;u:=SylowSubgroup(g,2);;Index(g,u);
15
gap> FactorCosetAction(g,u);
<action epimorphism>
gap> Range(last);
Group([ (1,9,13,10,4)(2,8,14,11,5)(3,7,15,12,6),
  (1,7)(2,8)(3,9)(5,6)(10,11)(14,15) ])
```

A special case is the regular action on all elements:

2 ▶ `RegularActionHomomorphism( G )` A

returns an isomorphism from G onto the regular permutation representation of G.

3 ▶ `AbelianSubfactorAction( G, M, N )` O

Let G be a group and $M \geq N$ be subgroups of a common parent that are normal under G, such that the subfactor M/N is elementary abelian. The operation `AbelianSubfactorAction` returns a list [*phi*,*alpha*,*bas*] where *bas* is a list of elements of M which are representatives for a basis of M/N, *alpha* is a map from M into a n-dimensional row space over $GF(p)$ where $[M : N] = p^n$ that is the natural homomorphism of M by N with the quotient represented as an additive group. Finally *phi* is a homomorphism from G into $GL_n(p)$ that represents the action of G on the factor M/N.

Note: If only matrices for the action are needed, `LinearActionLayer` might be faster.

```
gap> g:=Group((1,8,10,7,3,5)(2,4,12,9,11,6),(1,9,5,6,3,10)(2,11,12,8,4,7));;
gap> c:=ChiefSeries(g);;List(c,Size);
[ 96, 48, 16, 4, 1 ]
gap> HasElementaryAbelianFactorGroup(c[3],c[4]);
true
gap> SetName(c[3],"my_group");;
gap> a:=AbelianSubfactorAction(g,c[3],c[4]);
[ [ (1,8,10,7,3,5)(2,4,12,9,11,6), (1,9,5,6,3,10)(2,11,12,8,4,7) ] ->
    [ <an immutable 2x2 matrix over GF2>, <an immutable 2x2 matrix over GF2> ]
    , MappingByFunction( my_group, ( GF(2)^
    2 ), function( e ) ... end, function( r ) ... end ),
  Pcgs([ (2,8,3,9)(4,10,5,11), (1,6,12,7)(4,10,5,11) ]) ]
gap> mat:=Image(a[1],g);
Group([ <an immutable 2x2 matrix over GF2>,
  <an immutable 2x2 matrix over GF2> ])
```

```
gap> Size(mat);
3
gap> e:=PreImagesRepresentative(a[2],[Z(2),0*Z(2)]);
(2,8,3,9)(4,10,5,11)
gap> e in c[3];e in c[4];
true
false
```

39.8 Permutations Induced by Elements and Cycles

If only the permutation image of a single element is needed, it might not be worth to create the action homomorphism, the following operations yield the permutation image and cycles of a single element.

1 ▶ `Permutation(` *g*, *Omega*`[,` *gens*, *acts*`][,` *act*`] )` F
▶ `Permutation(` *g*, *xset* `)` F

computes the permutation that corresponds to the action of g on the permutation domain *Omega* (a list of objects that are permuted). If an external set *xset* is given, the permutation domain is the `HomeEnumerator` of this external set (see Section 39.11). Note that the points of the returned permutation refer to the positions in *Omega*, even if *Omega* itself consists of integers.

If g does not leave the domain invariant, or does not map the domain injectively `fail` is returned.

2 ▶ `PermutationCycle(` *g*, *Omega*, *pnt* `[,` *act*`] )` O

computes the permutation that represents the cycle of *pnt* under the action of the element g.

```
gap> Permutation([[Z(3),-Z(3)],[Z(3),0*Z(3)]],AsList(GF(3)^2));
(2,7,6)(3,4,8)
gap> Permutation((1,2,3)(4,5)(6,7),[4..7]);
(1,2)(3,4)
gap> PermutationCycle((1,2,3)(4,5)(6,7),[4..7],4);
(1,2)
```

3 ▶ `Cycle(` *g*, *Omega*, *pnt* `[,` *act*`] )` O

returns a list of the points in the cycle of *pnt* under the action of the element g.

4 ▶ `CycleLength(` *g*, *Omega*, *pnt* `[,` *act*`] )` O

returns the length of the cycle of *pnt* under the action of the element g.

5 ▶ `Cycles(` *g*, *Omega* `[,` *act*`] )` O

returns a list of the cycles (as lists of points) of the action of the element g.

6 ▶ `CycleLengths(` *g*, *Omega*, `[,` *act*`] )` O

returns the lengths of all the cycles under the action of the element g on *Omega*.

```
gap> Cycle((1,2,3)(4,5)(6,7),[4..7],4);
[ 4, 5 ]
gap> CycleLength((1,2,3)(4,5)(6,7),[4..7],4);
2
gap> Cycles((1,2,3)(4,5)(6,7),[4..7]);
[ [ 4, 5 ], [ 6, 7 ] ]
gap> CycleLengths((1,2,3)(4,5)(6,7),[4..7]);
[ 2, 2 ]
```

39.9 Tests for Actions

1 ▶ `IsTransitive(` *G*`,` *Omega*`[,` *gens*`,` *acts*`][,` *act*`] )` O
▶ `IsTransitive(` *xset* `)` P

returns `true` if the action implied by the arguments is transitive, or `false` otherwise.

We say that a group G acts **transitively** on a domain D if and only if for every pair of points d and e there is an element g of G such that $d^g = e$.

2 ▶ `Transitivity(` *G*`,` *Omega*`[,` *gens*`,` *acts*`][,` *act*`] )` O
▶ `Transitivity(` *xset* `)` A

returns the degree k (a non-negative integer) of transitivity of the action implied by the arguments, i.e. the largest integer k such that the action is k-transitive. If the action is not transitive `0` is returned.

An action is **k-transitive** if every k-tuple of points can be mapped simultaneously to every other k-tuple.

```
gap> g:=Group((1,3,2),(2,4,3));;
gap> IsTransitive(g,[1..5]);
false
gap> Transitivity(g,[1..4]);
2
```

Note: For permutation groups, the syntax `IsTransitive(`g`)` is also permitted and tests whether the group is transitive on the points moved by it, that is the group $\langle (2,3,4),(2,3)\rangle$ is transitive (on 3 points).

3 ▶ `RankAction(` *G*`,` *Omega*`[,` *gens*`,` *acts*`][,` *act*`] )` O
▶ `RankAction(` *xset* `)` A

returns the rank of a transitive action, i.e. the number of orbits of the point stabilizer.

```
gap> RankAction(g,Combinations([1..4],2),OnSets);
4
```

4 ▶ `IsSemiRegular(` *G*`,` *Omega*`[,` *gens*`,` *acts*`][,` *act*`] )` O
▶ `IsSemiRegular(` *xset* `)` P

returns `true` if the action implied by the arguments is semiregular, or `false` otherwise.

An action is **semiregular** is the stabilizer of each point is the identity.

5 ▶ `IsRegular(` *G*`,` *Omega*`[,` *gens*`,` *acts*`][,` *act*`] )` O
▶ `IsRegular(` *xset* `)` P

returns `true` if the action implied by the arguments is regular, or `false` otherwise.

An action is **regular** if it is both semiregular (see 39.9.4) and transitive (see 39.9.1). In this case every point *pnt* of *Omega* defines a one-to-one correspondence between G and *Omega*.

```
gap> IsSemiRegular(g,Arrangements([1..4],3),OnTuples);
true
gap> IsRegular(g,Arrangements([1..4],3),OnTuples);
false
```

6 ▶ `Earns(` *G*`,` *Omega*`[,` *gens*`,` *acts*`][,` *act*`] )` O
▶ `Earns(` *xset* `)` A

returns a list of the elementary abelian regular (when acting on *Omega*) normal subgroups of G.

At the moment only methods for a primitive group G are implemented.

7 ▶ `IsPrimitive( G, Omega[, gens, acts][, act] )` O
▶ `IsPrimitive( xset )` P

returns `true` if the action implied by the arguments is primitive, or `false` otherwise.

An action is **primitive** if it is transitive and the action admits no nontrivial block systems. See 39.10.

```
gap> IsPrimitive(g,Orbit(g,(1,2)(3,4)));
true
```

39.10 Block Systems

A **block system** (system of imprimitivity) for the action of G on *Omega* is a partition of *Omega* which – as a partition – remains invariant under the action of G.

1 ▶ `Blocks( G, Omega[, seed][, gens, acts][, act] )` O
▶ `Blocks( xset[, seed] )` A

computes a block system for the action. If *seed* is not given and the action is imprimitive, a minimal nontrivial block system will be found. If *seed* is given, a block system in which *seed* is the subset of one block is computed. The action must be transitive.

```
gap> g:=TransitiveGroup(8,3);
E(8)=2[x]2[x]2
gap> Blocks(g,[1..8]);
[ [ 1, 8 ], [ 2, 3 ], [ 4, 5 ], [ 6, 7 ] ]
gap> Blocks(g,[1..8],[1,4]);
[ [ 1, 4 ], [ 2, 7 ], [ 3, 6 ], [ 5, 8 ] ]
```

(See Section 39.2 for information about specific actions.)

2 ▶ `MaximalBlocks( G, Omega [, seed] [, gens, acts] [, act] )` O
▶ `MaximalBlocks( xset [, seed] )` A

returns a block system that is maximal with respect to inclusion. maximal with respect to inclusion) for the action of G on *Omega*. If *seed* is given, a block system in which *seed* is the subset of one block is computed.

```
gap> MaximalBlocks(g,[1..8]);
[ [ 1, 2, 3, 8 ], [ 4, 5, 6, 7 ] ]
```

3 ▶ `RepresentativesMinimalBlocks( G, Omega[, gens, acts][, act] )` O
▶ `RepresentativesMinimalBlocks( xset )` A

computes a list of block representatives for all minimal (i.e blocks are minimal with respect to inclusion) nontrivial block systems for the action.

```
gap> RepresentativesMinimalBlocks(g,[1..8]);
[ [ 1, 2 ], [ 1, 3 ], [ 1, 4 ], [ 1, 5 ], [ 1, 6 ], [ 1, 7 ], [ 1, 8 ] ]
```

4 ▶ `AllBlocks( G )` A

computes a list of representatives of all block systems for a permutation group G acting transitively on the points moved by the group.

```
gap> AllBlocks(g);
[ [ 1, 8 ], [ 1, 2, 3, 8 ], [ 1, 4, 5, 8 ], [ 1, 6, 7, 8 ], [ 1, 3 ],
  [ 1, 3, 5, 7 ], [ 1, 3, 4, 6 ], [ 1, 5 ], [ 1, 2, 5, 6 ], [ 1, 2 ],
  [ 1, 2, 4, 7 ], [ 1, 4 ], [ 1, 7 ], [ 1, 6 ] ]
```

The stabilizer of a block can be computed via the action `OnSets` (see 39.2.4):

```
gap> Stabilizer(g,[1,8],OnSets);
Group([ (1,8)(2,3)(4,5)(6,7) ])
```

If *bs* is a partition of *omega*, given as a set of sets, the stabilizer under the action `OnSetsDisjointSets` (see 39.2.8) returns the largest subgroup which preserves *bs* as a block system.

```
gap> g:=Group((1,2,3,4,5,6,7,8),(1,2));;
gap> bs:=[[1,2,3,4],[5,6,7,8]];;
gap> Stabilizer(g,bs,OnSetsDisjointSets);
Group([ (6,7), (5,6), (5,8), (2,3), (3,4)(5,7), (1,4), (1,5,4,8)(2,6,3,7) ])
```

39.11 External Sets

When considering group actions, sometimes the concept of a G**-set** is used. This is the set *Omega* endowed with an action of G. The elements of the G-set are the same as those of *Omega*, however concepts like equality and equivalence of G-sets do not only consider the underlying domain *Omega* but the group action as well.

This concept is implemented in GAP via **external sets**.

1 ▶ `IsExternalSet( `*obj*` )` C

An **external set** specifies an action *act*: $Omega \times G \to Omega$ of a group G on a domain *Omega*. The external set knows the group, the domain and the actual acting function. Mathematically, an external set is the set *Omega*, which is endowed with the action of a group G via the group action *act*. For this reason GAP treats external sets as a domain whose elements are the elements of *Omega*. An external set is always a union of orbits. Currently the domain *Omega* must always be finite. If *Omega* is not a list, an enumerator for *Omega* is automatically chosen.

2 ▶ `ExternalSet( `*G*`, `*Omega*`[, `*gens*`, `*acts*`][, `*act*`] )` O

creates the external set for the action *act* of G on *Omega*. *Omega* can be either a proper set or a domain which is represented as described in 12.4 and 28.

```
gap> g:=Group((1,2,3),(2,3,4));;
gap> e:=ExternalSet(g,[1..4]);
<xset:[ 1, 2, 3, 4 ]>
gap> e:=ExternalSet(g,g,OnRight);
<xset:<enumerator of perm group>>
gap> Orbits(e);
[ [ (), (1,2)(3,4), (1,3)(2,4), (1,4)(2,3), (2,4,3), (1,4,2), (1,2,3),
      (1,3,4), (2,3,4), (1,3,2), (1,4,3), (1,2,4) ] ]
```

The following three attributes of an external set hold its constituents.

3 ▶ `ActingDomain( `*xset*` )` A

This attribute returns the group with which the external set *xset* was defined.

4 ▶ `FunctionAction( `*xset*` )` A

the acting function *act* of *xset*

5 ▶ `HomeEnumerator( `*xset*` )` A

returns an enumerator of the domain *Omega* with which *xset* was defined. For external subsets, this is different from `Enumerator( `*xset*` )`, which enumerates only the subset.

```
gap> ActingDomain(e);
Group([ (1,2,3), (2,3,4) ])
gap> FunctionAction(e)=OnRight;
true
gap> HomeEnumerator(e);
<enumerator of perm group>
```

Most operations for actions are applicable as an attribute for an external set.

6 ▶ `IsExternalSubset(` *obj* `)` R

An external subset is the restriction of an external set to a subset of the domain (which must be invariant under the action). It is again an external set.

The most prominent external subsets are orbits:

7 ▶ `ExternalSubset(` *G*, *xset*, *start*, [*gens*, *acts*,]*act* `)` O

constructs the external subset of *xset* on the union of orbits of the points in *start*.

8 ▶ `IsExternalOrbit(` *obj* `)` R

An external orbit is an external subset consisting of one orbit.

9 ▶ `ExternalOrbit(` *G*, *Omega*, *pnt*, [*gens*, *acts*,] *act* `)` O

constructs the external subset on the orbit of *pnt*. The `Representative` of this external set is *pnt*.

```
gap> e:=ExternalOrbit(g,g,(1,2,3));
(1,2,3)^G
```

Many subsets of a group, such as conjugacy classes or cosets (see 37.10.1 and 37.7.1) are implemented as external orbits.

10 ▶ `StabilizerOfExternalSet(` *xset* `)` A

computes the stabilizer of `Representative(`*xset*`)` The stabilizer must have the acting group *G* of *xset* as its parent.

```
gap> Representative(e);
(1,2,3)
gap> StabilizerOfExternalSet(e);
Group([ (1,2,3) ])
```

11 ▶ `ExternalOrbits(` *G*, *Omega*[, *gens*, *acts*][, *act*] `)` O
▶ `ExternalOrbits(` *xset* `)` A

computes a list of `ExternalOrbit`s that give the orbits of *G*.

```
gap> ExternalOrbits(g,AsList(g));
[ ()^G, (2,3,4)^G, (2,4,3)^G, (1,2)(3,4)^G ]
```

12 ▶ `ExternalOrbitsStabilizers(` *G*, *Omega*[, *gens*, *acts*][, *act*] `)` O
▶ `ExternalOrbitsStabilizers(` *xset* `)` A

In addition to `ExternalOrbits`, this operation also computes the stabilizers of the representatives of the external orbits at the same time. (This can be quicker than computing the `ExternalOrbits` first and the stabilizers afterwards.)

```
gap> e:=ExternalOrbitsStabilizers(g,AsList(g));
[ ()^G, (2,3,4)^G, (2,4,3)^G, (1,2)(3,4)^G ]
gap> HasStabilizerOfExternalSet(e[3]);
true
gap> StabilizerOfExternalSet(e[3]);
Group([ (2,4,3) ])
```

13 ▶ `CanonicalRepresentativeOfExternalSet( xset )` A

The canonical representative of an external set may only depend on *G*, *Omega*, *act* and (in the case of external subsets) `Enumerator( xset )`. It must not depend, e.g., on the representative of an external orbit. GAP does not know methods for every external set to compute a canonical representative . See 39.11.14.

14 ▶ `CanonicalRepresentativeDeterminatorOfExternalSet( xset )` A

returns a function that takes as arguments the acting group and the point. It returns a list of length 3: [*canonrep*, *stabilizercanonrep*, *conjugatingelm*]. (List components 2 and 3 are optional and do not need to be bound.) An external set is only guaranteed to be able to compute a canonical representative if it has a `CanonicalRepresentativeDeterminatorOfExternalSet`.

15 ▶ `ActorOfExternalSet( xset )` A

returns an element mapping `Representative(`*xset*`)` to `CanonicalRepresentativeOfExternalSet(`*xset*`)` under the given action.

```
gap> u:=Subgroup(g,[(1,2,3)]);;
gap> e:=RightCoset(u,(1,2)(3,4));;
gap> CanonicalRepresentativeOfExternalSet(e);
(2,4,3)
gap> ActorOfExternalSet(e);
(1,3,2)
gap> FunctionAction(e)((1,2)(3,4),last);
(2,4,3)
```

External sets also are implicitly underlying action homomorphisms:

16 ▶ `UnderlyingExternalSet( ohom )` A

The underlying set of an action homomorphism is the external set on which it was defined.

```
gap> g:=Group((1,2,3),(1,2));;
gap> hom:=ActionHomomorphism(g,Arrangements([1..4],3),OnTuples);;
gap> s:=UnderlyingExternalSet(hom);
<xset:[[ 1, 2, 3 ],[ 1, 2, 4 ],[ 1, 3, 2 ],[ 1, 3, 4 ],[ 1, 4, 2 ],
[ 1, 4, 3 ],[ 2, 1, 3 ],[ 2, 1, 4 ],[ 2, 3, 1 ],[ 2, 3, 4 ],[ 2, 4, 1 ],
[ 2, 4, 3 ],[ 3, 1, 2 ],[ 3, 1, 4 ],[ 3, 2, 1 ], ...]>
gap> Print(s,"\n");
[ [ 1, 2, 3 ], [ 1, 2, 4 ], [ 1, 3, 2 ], [ 1, 3, 4 ], [ 1, 4, 2 ],
  [ 1, 4, 3 ], [ 2, 1, 3 ], [ 2, 1, 4 ], [ 2, 3, 1 ], [ 2, 3, 4 ],
  [ 2, 4, 1 ], [ 2, 4, 3 ], [ 3, 1, 2 ], [ 3, 1, 4 ], [ 3, 2, 1 ],
  [ 3, 2, 4 ], [ 3, 4, 1 ], [ 3, 4, 2 ], [ 4, 1, 2 ], [ 4, 1, 3 ],
  [ 4, 2, 1 ], [ 4, 2, 3 ], [ 4, 3, 1 ], [ 4, 3, 2 ] ]
```

17 ▶ `SurjectiveActionHomomorphismAttr( xset )` A

returns an action homomorphism for *xset* which is surjective. (As the `Image` of this homomorphism has to be computed to obtain the range, this may take substantially longer than `ActionHomomorphism`.)

40 Permutations

GAP offers a data type **permutation** to describe the elements of permutation groups.

The points on which permutations in GAP act are the positive integers less than $2^{28}-1$, and the image of a point i under a permutation p is written i^p, which is expressed as `i^p` in GAP. (This action is also implemented by the function `OnPoints`, see 39.2.1.) If $i^p \neq i$, we say that i is **moved** by p, otherwise it is **fixed**. Permutations in GAP are entered and displayed in cycle notation, such as `(1,2,3)(4,5)`.

The preimage of the point i under the permutation p can be computed as i `/` p, without constructing the inverse of p.

For arithmetic operations for permutations and their precedence, see 30.12.

In the names of the GAP functions that deal with permutations, the word `Permutation` is usually abbreviated to `Perm`, to save typing. For example, the category test function for permutations is called `IsPerm`.

1 ► `IsPerm(` *obj* `)` C

Each **permutation** in GAP lies in the category `IsPerm`. Basic operations for permutations are `LargestMovedPoint` (see 40.2.2), multiplication of two permutations via `*`, and exponentiation `^` with first argument a positive integer i and second argument a permutation π, the result being the image of the point i under π.

2 ► `IsPermCollection(` *obj* `)` C
► `IsPermCollColl(` *obj* `)` C

are the categories for collections of permutations and collections of collections of permutations, respectively.

3 ► `PermutationsFamily` V

is the family of all permutations.

Internally, GAP stores a permutation as a list of the d images of the integers $1, \ldots, d$, where the "internal degree" d is the largest integer moved by the permutation or bigger. When a permutation is read in in cycle notation, d is always set to the largest moved integer, but a bigger d can result from a multiplication of two permutations, because the product is not shortened if it fixes d. The images are either all stored as 16-bit integers or all as 32-bit integers (actually as GAP immediate integers less than 2^{28}), depending on whether $d \leq 65536$ or not. This means that the identity permutation `()` takes $4m$ bytes if it was calculated as `(1, ..., m) * (1, ..., m)^-1`. It can take even more because the internal list has sometimes room for more than d images. For example, the maximal degree of any permutation in GAP is $m = 2^{22} - 1024 = 4{,}193{,}280$, because bigger permutations would have an internal list with room for more than 2^{22} images, requiring more than 2^{24} bytes. 2^{24}, however, is the largest possible size of an object that the GAP storage manager can deal with.

Permutations do not belong to a specific group. That means that one can work with permutations without defining a permutation group that contains them.

```
gap> (1,2,3);
(1,2,3)
gap> (1,2,3) * (2,3,4);
(1,3)(2,4)
gap> 17^(2,5,17,9,8);
9
gap> OnPoints(17,(2,5,17,9,8));
9
```

The operation `Permuted` (see 21.20.16) can be used to permute the entries of a list according to a permutation.

40.1 Comparison of Permutations

1 ▶ *p_1* `=` *p_2*
▶ *p_1* `<` *p_2*

Two permutations are equal if they move the same points and all these points have the same images under both permutations.

The permutation p_1 is smaller than p_2 if $p_1 \neq p_2$ and $i^{p_1} < i^{p_2}$ where i is the smallest point with $i^{p_1} \neq i^{p_2}$. Therefore the identity permutation is the smallest permutation. (see also 30.11)

Permutations can be compared with certain other GAP objects, see 4.11 for the details.

```
gap> (1,2,3) = (2,3,1);
true
gap> (1,2,3) * (2,3,4) = (1,3)(2,4);
true
gap> (1,2,3) < (1,3,2);      # 1^(1,2,3) = 2 < 3 = 1^(1,3,2)
true
gap> (1,3,2,4) < (1,3,4,2);  # 2^(1,3,2,4) = 4 > 1 = 2^(1,3,4,2)
false
```

2 ▶ `SmallestGeneratorPerm(` *perm* `)` F

is the smallest permutation that generates the same cyclic group as the permutation *perm*. This is very efficient, even when *perm* has large order.

```
gap> SmallestGeneratorPerm( (1,4,3,2) );
(1,2,3,4)
```

40.2 Moved Points of Permutations

1 ▶ `SmallestMovedPoint(` *perm* `)` A
▶ `SmallestMovedPoint(` *C* `)` A

is the smallest positive integer that is moved by *perm* if such an integer exists, and `infinity` if *perm* `= ()`. For *C* a collection or list of permutations, the smallest value of `SmallestMovedPoint` for the elements of *C* is returned (and `infinity` if *C* is empty).

2 ▶ `LargestMovedPoint(` *perm* `)` A
▶ `LargestMovedPoint(` *C* `)` A

For a permutation *perm*, this attribute contains the largest positive integer which is moved by *perm* if such an integer exists, and 0 if *perm* `= ()`. For *C* a collection or list of permutations, the largest value of `LargestMovedPoint` for the elements of *C* is returned (and 0 if *C* is empty).

3 ▶ `MovedPoints( perm )` A
▶ `MovedPoints( C )` A

is the proper set of the positive integers moved by at least one permutation in the collection C, respectively by the permutation *perm*.

4 ▶ `NrMovedPoints( perm )` A
▶ `NrMovedPoints( C )` A

is the number of positive integers that are moved by *perm*, respectively by at least one element in the collection C. (The actual moved points are returned by `MovedPoints`, see 40.2.3)

```
gap> SmallestMovedPointPerm((4,5,6)(7,2,8));
2
gap> LargestMovedPointPerm((4,5,6)(7,2,8));
8
gap> NrMovedPointsPerm((4,5,6)(7,2,8));
6
gap> MovedPoints([(2,3,4),(7,6,3),(5,47)]);
[ 2, 3, 4, 5, 6, 7, 47 ]
gap> NrMovedPoints([(2,3,4),(7,6,3),(5,47)]);
7
gap> SmallestMovedPoint([(2,3,4),(7,6,3),(5,47)]);
2
gap> LargestMovedPoint([(2,3,4),(7,6,3),(5,47)]);
47
gap> LargestMovedPoint([()]);
0
```

40.3 Sign and Cycle Structure

1 ▶ `SignPerm( perm )` A

The **sign** of a permutation *perm* is defined as $(-1)^k$ where k is the number of cycles of *perm* of even length.

The sign is a homomorphism from the symmetric group onto the multiplicative group $\{+1, -1\}$, the kernel of which is the alternating group.

2 ▶ `CycleStructurePerm( perm )` A

is the cycle structure (i.e. the numbers of cycles of different lengths) of *perm*. This is encoded in a list l in the following form: The i-th entry of l contains the number of cycles of *perm* of length *i+1*. If *perm* contains no cycles of length *i+1* it is not bound. Cycles of length 1 are ignored.

```
gap> SignPerm((1,2,3)(4,5));
-1
gap> CycleStructurePerm((1,2,3)(4,5,9,7,8));
[ , 1,, 1 ]
```

40.4 Creating Permutations

1 ▶ `ListPerm(` *perm* `)` F

is a list *list* that contains the images of the positive integers under the permutation *perm*. That means that *list*`[`*i*`] = `*i*`^`*perm*, where *i* lies between 1 and the largest point moved by *perm* (see 40.2.2).

2 ▶ `PermList(` *list* `)` F

is the permutation *perm* that moves points as described by the list *list*. That means that *i*`^`*perm* `=` *list*`[`*i*`]` if *i* lies between 1 and the length of *list*, and *i*`^`*perm* `=` *i* if *i* is larger than the length of the list *list*. It will return `fail` if *list* does not define a permutation, i.e., if *list* is not dense, or if *list* contains a positive integer twice, or if *list* contains an integer not in the range `[ 1 .. Length(` *list* `) ]`. If *list* contains non-integer entries an error is raised.

3 ▶ `MappingPermListList(` *src*`,` *dst* `)` F

Let *src* and *dst* be lists of positive integers of the same length, such that neither may contain an element twice. `MappingPermListList` returns a permutation *perm* such that *src*`[`*i*`]^`*perm* `=` *dst*`[`*i*`]`. *perm* fixes all points larger than the maximum of the entries in *src* and *dst*. If there are several such permutations, it is not specified which of them `MappingPermListList` returns.

4 ▶ `RestrictedPerm(` *perm*`,` *list* `)` O

`RestrictedPerm` returns the new permutation *new* that acts on the points in the list *list* in the same way as the permutation *perm*, and that fixes those points that are not in *list*. *list* must be a list of positive integers such that for each *i* in *list* the image *i*`^`*perm* is also in *list*, i.e., *list* must be the union of cycles of *perm*.

```
gap> ListPerm((3,4,5));
[ 1, 2, 4, 5, 3 ]
gap> PermList([1,2,4,5,3]);
(3,4,5)
gap> MappingPermListList([2,5,1,6],[7,12,8,2]);
(1,8,5,12,11,10,9,6,2,7,4,3)
gap> RestrictedPerm((1,2)(3,4),[3..5]);
(3,4)
```

41 Permutation Groups

1 ▶ `IsPermGroup( obj )` C

A permutation group is a group of permutations on a finite set Ω of positive integers. GAP does **not** require the user to specify the operation domain Ω when a permutation group is defined.

```
gap> g:=Group((1,2,3,4),(1,2));
Group([ (1,2,3,4), (1,2) ])
```

Permutation groups are groups and therefore all operations for groups (see Chapter 37) can be applied to them. In many cases special methods are installed for permutation groups that make computations more effective.

41.1 The Natural Action

The functions `MovedPoints`, `NrMovedPoints`, `LargestMovedPoint`, and `SmallestMovedPoint` are defined for arbitrary collections of permutations (see 40.2), in particular they can be applied to permutation groups.

```
gap> g:= Group( (2,3,5,6), (2,3) );;
gap> MovedPoints( g );  NrMovedPoints( g );
[ 2, 3, 5, 6 ]
4
gap> LargestMovedPoint( g );  SmallestMovedPoint( g );
6
2
```

The action of a permutation group on the positive integers is a group action (via the acting function `OnPoints`). Therefore all action functions can be applied (see the Chapter 39), for example `Orbit`, `Stabilizer`, `Blocks`, `IsTransitive`, `IsPrimitive`.

If one has a list of group generators and is interested in the moved points (see above) or orbits, it may be useful to avoid the explicit construction of the group for efficiency reasons. For the special case of the action of permutations on positive integers via `^`, the following functions are provided for this purpose.

1 ▶ `OrbitPerms( perms, pnt )` F

returns the orbit of the positive integer *pnt* under the group generated by the permutations in the list *perms*.

2 ▶ `OrbitsPerms( perms, D )` F

returns the list of orbits of the positive integers in the list D under the group generated by the permutations in the list *perms*.

```
gap> OrbitPerms( [ (1,2,3)(4,5), (3,6) ], 1 );
[ 1, 2, 3, 6 ]
gap> OrbitsPerms( [ (1,2,3)(4,5), (3,6) ], [ 1 .. 6 ] );
[ [ 1, 2, 3, 6 ], [ 4, 5 ] ]
```

Similarly, several functions concerning the natural action of permutation groups address stabilizer chains (see 41.5) rather than permutation groups themselves, for example `BaseStabChain` (see 41.9.1).

41.2 Computing a Permutation Representation

1 ► `IsomorphismPermGroup( G )` A

returns an isomorphism φ from the group G onto a permutation group P which is isomorphic to G. The method will select a suitable permutation representation.

```
gap> g:=SmallGroup(24,12);
<pc group of size 24 with 4 generators>
gap> iso:=IsomorphismPermGroup(g);
<action isomorphism>
gap> Image(iso,g.3*g.4);
(1,4)(2,3)(5,8)(6,7)(9,12)(10,11)(13,16)(14,15)(17,20)(18,19)(21,24)(22,23)
```

In many cases the permutation representation constructed by `IsomorphismPermGroup` is regular.

2 ► `SmallerDegreePermutationRepresentation( G )` F

Let G be a permutation group that acts transitively on its moved points. `SmallerDegreePermutationRepresentation` tries to find a faithful permutation representation of smaller degree. The result is a group homomorphism onto a permutation group, in the worst case this is the identity mapping on G.

Note that the result is not guaranteed to be a faithful permutation representation of smallest degree, or of smallest degree among the transitive permutation representations of G. Using GAP interactively, one might be able to choose subgroups of small index for which the cores intersect trivially; in this case, the actions on the cosets of these subgroups give rise to an intransitive permutation representation the degree of which may be smaller than the original degree.

The methods used might involve the use of random elements and the permutation representation (or even the degree of the representation) is not guaranteed to be the same for different calls of `SmallerDegreePermutationRepresentation`.

```
gap> image:= Image( iso );;  NrMovedPoints( image );
24
gap> small:= SmallerDegreePermutationRepresentation( image );;
gap> Image( small );
Group([ (2,3), (2,4,3), (1,3)(2,4), (1,2)(3,4) ])
```

41.3 Symmetric and Alternating Groups

The commands `SymmetricGroup` and `AlternatingGroup` (see 48.1) construct symmetric and alternating permutation groups. GAP can also detect whether a given permutation group is a symmetric or alternating group on the set of its moved points; if so then the group is called a **natural** symmetric or alternating group, respectively.

1 ► `IsNaturalSymmetricGroup( group )` P

A group is a natural symmetric group if it is a permutation group acting as symmetric group on its moved points.

2 ► `IsNaturalAlternatingGroup( group )` P

A group is a natural alternating group if it is a permutation group acting as alternating group on its moved points.

For groups that are known to be natural symmetric or natural alternating groups, very efficient methods for computing membership, conjugacy classes, Sylow subgroups etc. are used.

```
gap> g:=Group((1,5,7,8,99),(1,99,13,72));;
gap> IsNaturalSymmetricGroup(g);
true
gap> g;
Sym( [ 1, 5, 7, 8, 13, 72, 99 ] )
gap> IsNaturalSymmetricGroup( Group( (1,2)(4,5), (1,2,3)(4,5,6) ) );
false
```

The following functions can be used to check whether a given group (not necessarily a permutation group) is isomorphic to a symmetric or alternating group.

There are no methods yet for IsSymmetricGroup and IsAlternatingGroup!

3 ► `IsSymmetricGroup(` *group* `)` P

is `true` if the group *group* is isomorphic to a natural symmetric group.

4 ► `IsAlternatingGroup(` *group* `)` P

Such a group is a group isomorphic to a natural alternating group.

5 ► `SymmetricParentGroup(` *grp* `)` A

For a permutation group *grp* this function returns the symmetric group that moves the same points as *grp* does.

```
gap> SymmetricParentGroup( Group( (1,2), (4,5), (7,8,9) ) );
Sym( [ 1, 2, 4, 5, 7, 8, 9 ] )
```

41.4 Primitive Groups

1 ► `ONanScottType(` G `)` A

returns the type of G of a primitive permutation group G, according to the O'Nan-Scott classification. The labelling of the different types is not consistent in the literature, we use the following:

- 1 Affine.
- 2 Almost simple.
- 3a Diagonal, Socle consists of two normal subgroups.
- 3b Diagonal, Socle is minimal normal.
- 4a Product action with the first factor primitive of type 3a.
- 4b Product action with the first factor primitive of type 3b.
- 4c Product action with the first factor primitive of type 2.
- 5 Twisted wreath product

As it can contain letters, the type is returned as a string.

If G is not a permutation group or does not act primitively on the points moved by it, the result is undefined.

2 ► `SocleTypePrimitiveGroup(` G `)` A

returns the socle type of a primitive permutation group. The socle of a primitive group is the direct product of isomorphic simple groups, therefore the type is indicated by a record with components `series`, `parameter` (both as described under `IsomorphismTypeInfoFiniteSimpleGroup`, see 37.15.11) and `width` for the number of direct factors.

If G does not have a faithful primitive action, the result is undefined.

```
gap> g:=AlternatingGroup(5);;
gap> h:=DirectProduct(g,g);;
gap> p:=List([1,2],i->Projection(h,i));;
gap> ac:=Action(h,AsList(g),
> function(g,h) return Image(p[1],h)^-1*g*Image(p[2],h);end);;
gap> Size(ac);NrMovedPoints(ac);IsPrimitive(ac,[1..60]);
3600
60
true
gap> ONanScottType(ac);
"3a"
gap> SocleTypePrimitiveGroup(ac);
rec( series := "A", width := 2,
  name := "A(5) ~ A(1,4) = L(2,4) ~ B(1,4) = O(3,4) ~ C(1,4) = S(2,4) ~ 2A(1,4\
) = U(2,4) ~ A(1,5) = L(2,5) ~ B(1,5) = O(3,5) ~ C(1,5) = S(2,5) ~ 2A(1,5) = U\
(2,5)", parameter := 5 )
```

41.5 Stabilizer Chains

Many of the algorithms for permutation groups use a **stabilizer chain** of the group. The concepts of stabilizer chains, **bases**, and **strong generating sets** were introduced by Charles Sims in [Sim70]. A further discussion of base change is given in section 8.1 in "Extending GAP".

Let $B = [b_1, \ldots, b_n]$ be a list of points, $G^{(1)} = G$ and $G^{(i+1)} = Stab_{G^{(i)}}(b_i)$, such that $G^{(n+1)} = \{()\}$. Then the list $[b_1, \ldots, b_n]$ is called a **base** of G, the points b_i are called **base points**. A set S of generators for G satisfying the condition $< S \cap G^{(i)} > = G^{(i)}$ for each $1 \leq i \leq n$, is called a **strong generating set** (SGS) of G. (More precisely we ought to say that it is a SGS of G **relative** to B). The chain of subgroups $G^{(i)}$ of G itself is called the **stabilizer chain** of G relative to B.

Since $[b_1, \ldots, b_n]$, where n is the degree of G and b_i are the moved points of G, certainly is a base for G there exists a base for each permutation group. The number of points in a base is called the **length** of the base. A base B is called **reduced** if there exists no i such that $G^{(i)} = G^{(i+1)}$. (This however does not imply that no subset of B could also serve as a base.) Note that different reduced bases for one permutation group G may have different lengths. For example, the irreducible degree 416 permutation representation of the Chevalley Group $G_2(4)$ possesses reduced bases of length 5 and 7.

Let $R^{(i)}$ be a right transversal of $G^{(i+1)}$ in $G^{(i)}$, i.e. a set of right coset representatives of the cosets of $G^{(i+1)}$ in $G^{(i)}$. Then each element g of G has a unique representation of the form $g = r_n \ldots r_1$ with $r_i \in R^{(i)}$. The cosets of $G^{(i+1)}$ in $G^{(i)}$ are in bijective correspondence with the points in $O^{(i)} := b_i^{G^{(i)}}$. So we could represent a transversal as a list T such that $T[p]$ is a representative of the coset corresponding to the point $p \in O^{(i)}$, i.e., an element of $G^{(i)}$ that takes b_i to p. (Note that such a list has holes in all positions corresponding to points not contained in $O^{(i)}$.)

This approach however will store many different permutations as coset representatives which can be a problem if the degree n gets bigger. Our goal therefore is to store as few different permutations as possible such that we can still reconstruct each representative in $R^{(i)}$, and from them the elements in G. A **factorized inverse transversal** T is a list where $T[p]$ is a generator of $G^{(i)}$ such that $p^{T[p]}$ is a point that lies earlier in $O^{(i)}$ than p (note that we consider $O^{(i)}$ as a list, not as a set). If we assume inductively that we know an element $r \in G^{(i)}$ that takes b_i to $p^{T[p]}$, then $rT[p]^{-1}$ is an element in $G^{(i)}$ that takes b_i to p. GAP uses such factorized inverse transversals.

Another name for a factorized inverse transversal is a **Schreier tree**. The vertices of the tree are the points in $O^{(i)}$, and the root of the tree is b_i. The edges are defined as the ordered pairs $(p, p^{T[p]})$, for $p \in O^{(i)} \setminus \{b_i\}$. The edge $(p, p^{T[p]})$ is labelled with the generator $T[p]$, and the product of edge labels along the unique path from p to b_i is the inverse of the transversal element carrying b_i to p.

Before we describe the construction of stablizer chains in 41.7, we explain in 41.6 the idea of using non-deterministic algorithms; this is necessary for understanding the options available for the construction of stabilizer chains. After that, in 41.8 it is explained how a stabilizer chain is stored in GAP, 41.9 lists operations for stabilizer chains, and 41.10 lists low level routines for manipulating stabilizer chains.

41.6 Randomized Methods for Permutation Groups

For most computations with permutation groups, it is crucial to construct stabilizer chains efficiently. Sims's original construction [Sim70] is deterministic, and is called the Schreier-Sims algorithm, because it is based on Schreier's Lemma (p. 96 in [Hal59]): given $K = \langle S \rangle$ and a transversal T for K mod L, one can obtain $|S||T|$ generators for L. This lemma is applied recursively, with consecutive point stabilizers $G^{(i)}$ and $G^{(i+1)}$ playing the role of K and L.

In permutation groups of large degree, the number of Schreier generators to be processed becomes too large, and the deterministic Schreier-Sims algorithm becomes impractical. Therefore, GAP uses randomized algorithms. The method selection process, which is quite different from Version 3, works the following way.

If a group acts on not more than a hundred points, Sims's original deterministic algorithm is applied. In groups of degree greater than hundred, a heuristic algorithm based on ideas in [BCFS91] constructs a stabilizer chain. This construction is complemented by a verify-routine that either proves the correctness of the stabilizer chain or causes the extension of the chain to a correct one. The user can influence the verification process by setting the value of the record component `random` (cf. 41.7).

If `random` = 1000 then a slight extension of an unpublished method of Sims is used. The outcome of this verification process is always correct. The user also can prescribe any integer $1 \leq x \leq 999$ as the value of `random`. In this case, a randomized verification process from [BCFS91] is applied, and the result of the stabilizer chain construction is guaranteed to be correct with probability at least $x/1000$. The practical performance of the algorithm is much better than the theoretical guarantee.

If the stabilizer chain is not correct then the elements in the product of transversals $R^{(m)}R^{(m-1)} \cdots R^{(1)}$ constitute a proper subset of the group G in question. This means that a membership test with this stabilizer chain returns `false` for all elements that are not in G, but it may also return `false` for some elements of G; in other words, the result `true` of a membership test is always correct, whereas the result `false` may be incorrect.

The construction and verification phases are separated because there are situations where the verification step can be omitted; if one happens to know the order of the group in advance then the randomized construction of the stabilizer chain stops as soon as the product of the lengths of the basic orbits of the chain equals the group order, and the chain will be correct (see the `size` option of the `StabChain` command in 41.7.1).

Although the worst case running time is roughly quadratic for Sims's verification and roughly linear for the randomized one, in most examples the running time of the stabilizer chain construction with `random`= 1000 (i.e., guaranteed correct output) is about the same as the running time of randomized verification with guarantee of at least 90% correctness. Therefore, we suggest to use the default value `random`= 1000. Possible uses of `random`< 1000 are when one has to run through a large collection of subgroups, and a low value of random is used to choose quickly a candidate for more thorough examination; another use is when the user suspects that the quadratic bottleneck of the guaranteed correct verification is hit.

We will illustrate these ideas in two examples.

```
gap> h:= SL(4,7);;
gap> o:= Orbit( h, [1,0,0,0]*Z(7)^0, OnLines );;
gap> op:= Action( h, o, OnLines );;
gap> NrMovedPoints( op );
400
```

We created a permutation group on 400 points. First we compute a guaranteed correct stabilizer chain. (The `StabChain` command is described in 41.7.1.)

```
gap> h:= Group( GeneratorsOfGroup( op ) );;
gap> StabChain( h );;  time;
1120
gap> Size( h );
2317591180800
```

Now randomized verification will be used. We require that the result is guaranteed correct with probability 90%. This means that if we would do this calculation many times over, GAP would **guarantee** that in least 90% percent of all calculations the result is correct. In fact the results are much better than the guarantee, but we cannot promise that this will really happen. (For the meaning of the `random` component in the second argument of `StabChain`, see 41.7.1.)

First the group is created anew.

```
gap> h:= Group( GeneratorsOfGroup( op ) );;
gap> StabChain( h, rec( random:= 900 ) );;  time;
1410
gap> Size( h );
2317591180800
```

The result is still correct, and the running time is actually somewhat slower. If you give the algorithm additional information so that it can check its results, things become faster and the result is guaranteed to be correct.

```
gap> h:=Group( GeneratorsOfGroup( op ) );;
gap> SetSize( h, 2317591180800 );
gap> StabChain( h );;  time;
170
```

The second example gives a typical group when the verification with `random` = 1000 is slow. The problem is that the group has a stabilizer subgroup $G^{(i)}$ such that the fundamental orbit $O^{(i)}$ is split into a lot of orbits when we stabilize b_i and one additional point of $O^{(i)}$.

```
gap> p1:=PermList(Concatenation([401],[1..400]));;
gap> p2:=PermList(List([1..400],i->(i*20 mod 401)));;
gap> d:=DirectProduct(Group(p1,p2),SymmetricGroup(5));;
gap> h:=Group(GeneratorsOfGroup(d));;
gap> StabChain(h);;time;Size(h);
1030
192480
gap> h:=Group(GeneratorsOfGroup(d));;
gap> StabChain(h,rec(random:=900));;time;Size(h);
570
192480
```

When stabilizer chains of a group G are created with `random` < 1000, this is noted in the group G, by setting of the record component `random` in the value of the attribute `StabChainOptions` for G (see 41.7.2). As errors induced by the random methods might propagate, any group or homomorphism created from G inherits a `random` component in its `StabChainOptions` from the corresponding component for G.

A lot of algorithms dealing with permutation groups use randomized methods; however, if the initial stabilizer chain construction for a group is correct, these further methods will provide guaranteed correct output.

41.7 Construction of Stabilizer Chains

1 ▶ `StabChain( G[, options] )` F
▶ `StabChain( G, base )` F
▶ `StabChainOp( G, options )` O
▶ `StabChainMutable( G )` AM
▶ `StabChainMutable( permhomom )` AM
▶ `StabChainImmutable( G )` A

These commands compute a stabilizer chain for the permutation group G; additionally, `StabChainMutable` is also an attribute for the group homomorphism *permhomom* whose source is a permutation group.

`StabChainOp` is an operation with two arguments G and *options*, the latter being a record which controls some aspects of the computation of a stabilizer chain (see below); `StabChainOp` returns a **mutable** stabilizer chain. `StabChainMutable` is a **mutable** attribute for groups or homomorphisms, its default method for groups is to call `StabChainOp` with empty options record. `StabChainImmutable` is an attribute with **immutable** values; its default method dispatches to `StabChainMutable`.

`StabChain` is a function with first argument a permutation group G, and optionally a record *options* as second argument. If the value of `StabChainImmutable` for G is already known and if this stabilizer chain matches the requirements of *options*, `StabChain` simply returns this stored stabilizer chain. Otherwise `StabChain` calls `StabChainOp` and returns an immutable copy of the result; additionally, this chain is stored as `StabChainImmutable` value for G. If no *options* argument is given, its components default to the global variable `DefaultStabChainOptions` (see 41.7.3). If *base* is a list of positive integers, the version `StabChain( G, base )` defaults to `StabChain( G, rec( base:= base ) )`.

If given, *options* is a record whose components specify properties of the desired stabilizer chain or which may help the algorithm. Default values for all of them can be given in the global variable `DefaultStabChainOptions` (see 41.7.3). The following options are supported.

`base` (default an empty list)
: A list of points, through which the resulting stabilizer chain shall run. For the base B of the resulting stabilizer chain S this means the following. If the `reduced` component of *options* is `true` then those points of `base` with nontrivial basic orbits form the initial segment of B, if the `reduced` component is `false` then `base` itself is the initial segment of B. Repeated occurrences of points in `base` are ignored. If a stabilizer chain for G is already known then the stabilizer chain is computed via a base change.

`knownBase` (no default value)
: A list of points which is known to be a base for the group. Such a known base makes it easier to test whether a permutation given as a word in terms of a set of generators is the identity, since it suffices to map the known base with each factor consecutively, rather than multiplying the whole permutations (which would mean to map every point). This speeds up the Schreier-Sims algorithm which is used when a new stabilizer chain is constructed; it will not affect a base change, however. The component `knownBase` bears no relation to the `base` component, you may specify a known base `knownBase` and a desired base `base` independently.

`reduced` (default `true`)
: If this is `true` the resulting stabilizer chain S is reduced, i.e., the case $G^{(i)} = G^{(i+1)}$ does not occur. Setting `reduced` to `false` makes sense only if the component `base` (see above) is also set; in this case all points of `base` will occur in the base B of S, even if they have trivial basic orbits. Note that if `base` is just an initial segment of B, the basic orbits of the points in $B \setminus$ `base` are always nontrivial.

`tryPcgs` (default `true`)
: If this is `true` and either the degree is at most 100 or the group is known to be solvable, GAP will first try to construct a pcgs (see Chapter 43) for G which will succeed and implicitly construct a

stabilizer chain if G is solvable. If G turns out non-solvable, one of the other methods will be used. This solvability check is comparatively fast, even if it fails, and it can save a lot of time if G is solvable.

`random` (default `1000`)
: If the value is less than 1000, the resulting chain is correct with probability at least `random`/1000. The `random` option is explained in more detail in 41.6.

`size` (default `Size( G )` if this is known, i.e., if `HasSize( G )` is `true`)
: If this component is present, its value is assumed to be the order of the group G. This information can be used to prove that a non-deterministically constructed stabilizer chain is correct. In this case, GAP does a non-deterministic construction until the size is correct.

`limit` (default `Size( Parent( G ) )` or `StabChainOptions( Parent( G ) ).limit` if this is present)
: If this component is present, it must be greater than or equal to the order of G. The stabilizer chain construction stops if size `limit` is reached.

2 ▶ `StabChainOptions( G )` AM

is a record that stores the options with which the stabilizer chain stored in `StabChainImmutable` has been computed (see 41.7.1 for the options that are supported).

3 ▶ `DefaultStabChainOptions` V

are the options for `StabChain` which are set as default.

4 ▶ `StabChainBaseStrongGenerators( base, sgs, one )` F

If a base *base* for a permutation group G and a strong generating set *sgs* for G with respect to *base* are given. *one* must be the appropriate `One` (in most cases this will be `()`). This function constructs a stabilizer chain without the need to find Schreier generators; so this is much faster than the other algorithms.

5 ▶ `MinimalStabChain( G )` A

returns the reduced stabilizer chain corresponding to the base $[1, 2, 3, 4, \ldots]$.

41.8 Stabilizer Chain Records

If a permutation group has a stabilizer chain, this is stored as a recursive structure. This structure is itself a record S and it has (1) components that provide information about one level $G^{(i)}$ of the stabilizer chain (which we call the "current stabilizer") and (2) a component `stabilizer` that holds another such record, namely the stabilizer chain of the next stabilizer $G^{(i+1)}$. This gives a recursive structure where the "outermost" record representing the "topmost" stabilizer is bound to the group record component `stabChain` and has the components explained below. Note: Since the structure is recursive, **never print a stabilizer chain!** (Unless you want to exercise the scrolling capabilities of your terminal.)

`identity`
: the identity element of the current stabilizer.

`labels`
: a list of permutations which contains labels for the Schreier tree of the current stabilizer, i.e., it contains elements for the factorized inverse transversal. The first entry is this list is always the `identity`. Note that GAP tries to arrange things so that the `labels` components are identical (i.e., the same GAP object) in every stabilizer of the chain; thus the `labels` of a stabilizer do not necessarily all lie in the this stabilizer (but see `genlabels` below).

`genlabels`
: a list of integers indexing some of the permutations in the `labels` component. The `labels` addressed in this way form a generating set for the current stabilizer. If the `genlabels` component is empty, the

rest of the stabilizer chain represents the trivial subgroup, and can be ignored, e.g., when calculating the size.

`generators`
: a list of generators for the current stabilizer. Usually, it is `labels{ genlabels }`.

`orbit`
: the vertices of the Schreier tree, which form the basic orbit $b_i^{G^{(i)}}$, ordered in such a way that the base point b_i is

`transversal`
: The factorized inverse transversal found during the orbit algorithm. The element g stored at `transversal[`i`]` will map i to another point j that in the Schreier tree is closer to the base point. By iterated application (`transversal[`j`]` and so on) eventually the base point is reached and an element that maps i to the base point foiund as product.

`translabels`
: An index list such that `transversal[`j`] = labels[ translabels[`j`] ]`. This list takes up comparatively little memory and is used to speed up base changes.

`stabilizer`
: If the current stabilizer is not yet the trivial group, the stabilizer chain continues with the stabilizer of the current base point, which is again represented as a record with components `labels`, `identity`, `genlabels`, `generators`, `orbit`, `translabels`, `transversal` (and possibly `stabilizer`). This record is bound to the `stabilizer` component of the current stabilizer. The last member of a stabilizer chain is recognized by the fact that it has no `stabilizer` component bound.

It is possible that different stabilizer chains share the same record as one of their iterated `stabilizer` components.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> StabChain(g);
<stabilizer chain record, Base [ 1, 2, 3 ], Orbit length 4, Size: 24>
gap> BaseOfGroup(g);
[ 1, 2, 3 ]
gap> StabChainOptions(g);
rec( random := 1000 )
gap> DefaultStabChainOptions;
rec( reduced := true, random := 1000, tryPcgs := true )
```

41.9 Operations for Stabilizer Chains

1 ▶ `BaseStabChain( `S` )` F

returns the base belonging to the stabilizer chain S.

2 ▶ `BaseOfGroup( `G` )` A

returns a base of the permutation group G. There is **no** guarantee that a stabilizer chain stored in G corresponds to this base!

3 ▶ `SizeStabChain( `S` )` F

returns the product of the orbit lengths in the stabilizer chain S, that is, the order of the group described by S.

4 ▶ `StrongGeneratorsStabChain( `S` )` F

returns a strong generating set corresponding to the stabilizer chain S.

5 ▶ `GroupStabChain( [`*G*`, ] `*S*` )` F

constructs a permutation group with stabilizer chain S, i.e., a group with generators `Generators(` S `)` to which S is assigned as component `stabChain`. If the optional argument G is given, the result will have the parent G.

6 ▶ `OrbitStabChain(` *S*`,` *pnt* `)` F

returns the orbit of *pnt* under the group described by the stabilizer chain S.

7 ▶ `IndicesStabChain(` *S* `)` F

returns a list of the indices of the stabilizers in the stabilizer chain S.

8 ▶ `ListStabChain(` *S* `)` F

returns a list that contains at position i the stabilizer of the first $i-1$ base points in the stabilizer chain S.

9 ▶ `ElementsStabChain(` *S* `)` F

returns a list of all elements of the group described by the stabilizer chain S.

10 ▶ `InverseRepresentative(` *S*`,` *pnt* `)` F

calculates the transversal element which maps *pnt* back to the base point of S. It just runs back through the Schreier tree from *pnt* to the root and multiplies the labels along the way.

11 ▶ `SiftedPermutation(` *S*`,` *g* `)` F

sifts the permutation g through the stabilizer chain S and returns the result after the last step.

The element g is sifted as follows: g is replaced by g `* InverseRepresentative(` S`, `S`.orbit[1]^`g `)`, then S is replaced by S`.stabilizer` and this process is repeated until S is trivial or S`.orbit[1]^`g is not in the basic orbit S`.orbit`. The remainder g is returned, it is the identity permutation if and only if the original g is in the group G described by the original S.

12 ▶ `MinimalElementCosetStabChain(` *S*`,` *g* `)` F

Let G be the group described by the stabilizer chain S. This function returns a permutation h such that $Gg = Gh$ (that is, $g/h \in G$) and with the additional property that the list of images under h of the base belonging to S is minimal w.r.t. lexicographical ordering.

13 ▶ `LargestElementStabChain(` *S*`,` *id* `)` F

Let G be the group described by the stabilizer chain S. This function returns the element $h \in G$ with the property that the list of images under h of the base belonging to S is maximal w.r.t. lexicographical ordering. The second argument must be an identity element (used to start the recursion)

14 ▶ `ApproximateSuborbitsStabilizerPermGroup(` *G*`,` *pnt* `)` F

returns an approximation of the orbits of `Stabilizer(` *G*`,` *pnt* `)` on all points of the orbit `Orbit(` *G*`,` *pnt* `)`, without computing the full point stabilizer; As not all Schreier generators are used, the result may represent the orbits of only a subgroup of the point stabilizer.

41.10 Low Level Routines to Modify and Create Stabilizer Chains

These operations modify a stabilizer chain or obtain new chains with specific properties. They are rather technical and should only be used if such low-level routines are deliberately required. (For all functions in this section the parameter S is a stabilizer chain.)

1 ▶ `CopyStabChain( S )` F

This function returns a copy of the stabilizer chain S that has no mutable object (list or record) in common with S. The `labels` components of the result are possibly shared by several levels, but superfluous labels are removed. (An entry in `labels` is superfluous if it does not occur among the `genlabels` or `translabels` on any of the levels which share that `labels` component.)

This is useful for stabiliser sub-chains that have been obtained as the (iterated) `stabilizer` component of a bigger chain.

2 ▶ `CopyOptionsDefaults( G, options )` F

sets components in a stabilizer chain options record *options* according to what is known about the group G. This can be used to obtain a new stabilizer chain for G quickly.

3 ▶ `ChangeStabChain( S, base[, reduced] )` F

changes or reduces a stabilizer chain S to be adapted to the base *base*. The optional argument *reduced* is interpreted as follows.

`reduced = false` :
change the stabilizer chain, do not reduce it,

`reduced = true` :
change the stabilizer chain, reduce it.

4 ▶ `ExtendStabChain( S, base )` F

extends the stabilizer chain S so that it corresponds to base *base*. The original base of S must be a subset of *base*.

5 ▶ `ReduceStabChain( S )` F

changes the stabilizer chain S to a reduced stabilizer chain by eliminating trivial steps.

6 ▶ `RemoveStabChain( S )` F

S must be a stabilizer record in a stabilizer chain. This chain then is cut off at S by changing the entries in S. This can be used to remove trailing trivial steps.

7 ▶ `EmptyStabChain( labels, id[, pnt] )` F

constructs a stabilizer chain for the trivial group with `identity=`*id* and `labels=`$\{id\} \cup$ *labels* (but of course with `genlabels=[ ]` and `generators=[ ]`). If the optional third argument *pnt* is present, the only stabilizer of the chain is initialized with the one-point basic orbit `[ pnt ]` and with `translabels` and `transversal` components.

8 ▶ `InsertTrivialStabilizer( S, pnt )` F

`InsertTrivialStabilizer` initializes the current stabilizer with *pnt* as `EmptyStabChain` did, but assigns the original S to the new S`.stabilizer` component, such that a new level with trivial basic orbit (but identical `labels` and `ShallowCopy`ed `genlabels` and `generators`) is inserted. This function should be used only if *pnt* really is fixed by the generators of S, because then new generators can be added and the orbit and transversal at the same time extended with `AddGeneratorsExtendSchreierTree`.

9 ▶ `IsFixedStabilizer( S, pnt )` F

returns `true` if *pnt* is fixed by all generators of S and `false` otherwise.

10 ▶ `AddGeneratorsExtendSchreierTree( S, new )` F

adds the elements in *new* to the list of generators of S and at the same time extends the orbit and transversal. This is the only legal way to extend a Schreier tree (because this involves careful handling of the tree components).

41.11 Backtrack

A main use for stabilizer chains is in backtrack algorithms for permutation groups. GAP implements a partition-backtrack algorithm as described in [Leo91] and refined in [The97].

1 ▶ `SubgroupProperty( G, Pr[, L ] )` F

Pr must be a one-argument function that returns `true` or `false` for elements of G and the subset of elements of G that fulfill Pr must be a subgroup. (**If the latter is not true the result of this operation is unpredictable!**) This command computes this subgroup. The optional argument L must be a subgroup of the set of all elements fulfilling Pr and can be given if known in order to speed up the calculation.

2 ▶ `ElementProperty( G, Pr[, L[, R]] )` F

`ElementProperty` returns an element π of the permutation group G such that the one-argument function Pr returns `true` for π. It returns `fail` if no such element exists in G. The optional arguments L and R are subgroups of G such that the property Pr has the same value for all elements in the cosets Lg and gR, respectively.

A typical example of using the optional subgroups L and R is the conjugacy test for elements a and b for which one can set $L := C_G(a)$ and $R := C_G(b)$.

```
gap> propfun:= el -> (1,2,3)^el in [ (1,2,3), (1,3,2) ];;
gap> SubgroupProperty( g, propfun, Subgroup( g, [ (1,2,3) ] ) );
Group([ (1,2,3), (2,3) ])
gap> ElementProperty( g, el -> Order( el ) = 2 );
(2,4)
```

Chapter 40 describes special operations to construct permutations in the symmetric group without using backtrack constructions.

Backtrack routines are also called by the methods for permutation groups that compute centralizers, normalizers, intersections, conjugating elements as well as stabilizers for the operations of a permutation group `OnPoints`, `OnSets`, `OnTuples` and `OnSetSets`. Some of these methods use more specific refinements than `SubgroupProperty` or `ElementProperty`. For the definition of refinements, and how one can define refinements, see Section 8.2 in "Extending GAP".

3 ▶ `TwoClosure( G )` A

The **2-closure** of a transitive permutation group G on n points is the largest subgroup of S_n which has the same orbits on sets of ordered pairs of points as the group G has. It also can be interpreted as the stabilizer of the orbital graphs of G.

```
gap> TwoClosure(Group((1,2,3),(2,3,4)));
Sym( [ 1 .. 4 ] )
```

4 ▶ `InfoBckt` V

is the info class for the partition backtrack routines.

41.12 Working with large degree permutation groups

Permutation groups of large degree (usually at least a few 10000) can pose a challenge to the heuristics used in the algorithms for permutation groups. This section lists a few useful tricks that may speed up calculations with such large groups enormously.

The first aspect concerns solvable groups: A lot of calculations (including an initial stabilizer chain computation thanks to the algorithm from [Sim90]) are faster if a permutation group is known to be solvable. On the other hand, proving nonsolvability can be expensive for higher degrees. Therefore GAP will automatically test a permutation group for solvability, only if the degree is not exceeding 100. (See also the `tryPcgs` component of `StabChainOptions`.) It is therefore beneficial to tell a group of larger degree, which is known to be solvable, that it is, using `SetIsSolvableGroup(`G`,true)`.

The second aspect concerns memory usage. A permutation on more than 65536 points requires 4 byte per point for storing. So permutations on 256000 poitns require roughly 1MB of storage per permutation. Just storing the permutations required for a stabilizer chain might already go beyond the available memory, in particular if the base is not very short. In such a situation it can be useful, to replace the permutations by straight line program elements (see 35.9).

The following code gives an example of usage: We create a group of degree 231000. Using straight line program elements, one can compute a stabilizer chain in about 200 MB of memory.

```
gap> Read("largeperms"); # read generators from file
gap> gens:=StraightLineProgGens(permutationlist);;
gap> g:=Group(gens);
<permutation group with 5 generators>
gap> # use random algorithm (faster, but result is monte carlo)
gap> StabChainOptions(g).random:=1;;
gap> Size(g); # enforce computation of a stabilizer chain
352969829814506607555723283375823418805608027364917220787701179633600 0
```

Without straight line program elements, the same calculation runs into memory problems after a while even with 512MB of workspace:

```
gap> h:=Group(permutationlist);
<permutation group with 5 generators>
gap> StabChainOptions(h).random:=1;;
gap> Size(h);
exceeded the permitted memory (‘-o’ command line option) at
mlimit := 1; called from
SCRMakeStabStrong( S.stabilizer, [ g ], param, orbits, where, basesize,
 base, correct, missing, false ); called from
 SCRMakeStabStrong( S.stabilizer, [ g ], param, orbits, where, basesize,
...
```

The advantage in memory usage however is paid for in runtime: Comparisons of elements become much more expensive. One can avoid some of the related problems by registering a known base with the straight line program elements (see `StraightLineProgGens`). In this case element comparison will only compare the images of the given base points. If we are planning to do extensive calculations with the group, it can even be worth to recreate it with straight line program elements knowing a previously computed base:

```
gap> # get the base we computed already
gap> bas:=BaseStabChain(StabChainMutable(g));
[ 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55,
...
  2530, 2533, 2554, 2563, 2569 ]
gap> gens:=StraightLineProgGens(permutationlist,bas);;
gap> g:=Group(gens);;
gap> SetSize(g,
> 352969829814506607555723283375823418805608027364917220787701179633600 0);
gap> Random(g);; # enforce computation of a stabilizer chain
```

As we know already base and size, this second stabilizer chain calculation is much faster than the first one and takes less memory.

42 Matrix Groups

Matrix groups are groups generated by invertible square matrices.

In the following example we temporarily increase the line length limit from its default value 80 to 83 in order to get a nicer output format.

```
gap> m1 := [ [ Z(3)^0, Z(3)^0,   Z(3) ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ 0*Z(3),   Z(3), 0*Z(3) ] ];;
gap> m2 := [ [   Z(3),   Z(3), Z(3)^0 ],
>            [   Z(3), 0*Z(3),   Z(3) ],
>            [ Z(3)^0, 0*Z(3),   Z(3) ] ];;
gap> SizeScreen([ 83, ]);;
gap> m := Group( m1, m2 );
Group(
[ [ [ Z(3)^0, Z(3)^0, Z(3) ], [ Z(3), 0*Z(3), Z(3) ], [ 0*Z(3), Z(3), 0*Z(3) ] ],
  [ [ Z(3), Z(3), Z(3)^0 ], [ Z(3), 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3), Z(3) ] ] ])
gap> SizeScreen([ 80, ]);;
```

1 ▶ `IsMatrixGroup(` *grp* `)` C

For most operations, GAP only provides methods for finite matrix groups. Many calculations in finite matrix groups are done via a `NiceMonomorphism` (see 38.5) that represents a faithful action on vectors.

42.1 Attributes and Properties for Matrix Groups

1 ▶ `DimensionOfMatrixGroup(` *mat-grp* `)` A

The dimension of the matrix group.

2 ▶ `DefaultFieldOfMatrixGroup(` *mat-grp* `)` A

Is a field containing all the matrix entries. It is not guaranteed to be the smallest field with this property.

3 ▶ `FieldOfMatrixGroup(` *matgrp* `)` A

The smallest field containing all the matrix entries of all elements of the matrix group *matgrp*. As the calculation of this can be hard, this should only be used if one **really** needs the smallest field, use `DefaultFieldOfMatrixGroup` to get (for example) the characteristic.

```
gap> DimensionOfMatrixGroup(m);
3
gap> DefaultFieldOfMatrixGroup(m);
GF(3)
```

4 ▶ `TransposedMatrixGroup(` *matgrp* `)` A

returns the transpose of the matrix group *matgrp*. The transpose of the transpose of *matgrp* is identical to *matgrp*.

```
gap> G := Group( [[0,-1],[1,0]] );
Group([ [ [ 0, -1 ], [ 1, 0 ] ] ])
gap> T := TransposedMatrixGroup( G );
Group([ [ [ 0, 1 ], [ -1, 0 ] ] ])
gap> IsIdenticalObj( G, TransposedMatrixGroup( T ) );
true
```

42.2 Actions of Matrix Groups

The basic operations for groups are described in Chapter 39, special actions for **matrix** groups mentioned there are `OnLines`, `OnRight`, and `OnSubspacesByCanonicalBasis`.

For subtleties concerning multiplication from the left or from the right, see 42.6.

1 ▶ `ProjectiveActionOnFullSpace( `*G*`, `*F*`, `*n*` )` F

Let G be a group of n by n matrices over a field contained in the finite field F. `ProjectiveActionOnFullSpace` returns the image of the projective action of G on the full row space F^n.

2 ▶ `ProjectiveActionHomomorphismMatrixGroup( `*G*` )` F

returns an action homomorphism for a faithful projective action of G on the underlying vector space. (Note: The action is not necessarily on the full space, if a smaller subset can be found on which the action is faithful.)

3 ▶ `BlowUpIsomorphism( `*matgrp*`, `*B*` )` F

For a matrix group *matgrp* and a basis B of a field extension L/K, say, such that the entries of all matrices in *matgrp* lie in L, `BlowUpIsomorphism` returns the isomorphism with source *matgrp* that is defined by mapping the matrix A to `BlownUpMat`(A, B), see 24.12.3.

```
gap> g:= GL(2,4);;
gap> B:= CanonicalBasis( GF(4) );;  BasisVectors( B );
[ Z(2)^0, Z(2^2) ]
gap> iso:= BlowUpIsomorphism( g, B );;
gap> Display( Image( iso, [ [ Z(4), Z(2) ], [ 0*Z(2), Z(4)^2 ] ] ) );
 . 1 1 .
 1 1 . 1
 . . 1 1
 . . 1 .
gap> img:= Image( iso, g );
<matrix group with 2 generators>
gap> Index( GL(4,2), img );
112
```

42.3 GL and SL

1 ▶ `IsGeneralLinearGroup( `*grp*` )` P
▶ `IsGL( `*grp*` )` P

The General Linear group is the group of all invertible matrices over a ring. This property tests, whether a group is isomorphic to a General Linear group. (Note that currently only a few trivial methods are available for this operation. We hope to improve this in the future.)

2 ▶ `IsNaturalGL( matgrp )` P

This property tests, whether a matrix group is the General Linear group in the right dimension over the (smallest) ring which contains all entries of its elements. (Currently, only a trivial test that computes the order of the group is available.)

3 ▶ `IsSpecialLinearGroup( grp )` P
▶ `IsSL( grp )` P

The Special Linear group is the group of all invertible matrices over a ring, whose determinant is equal to 1. This property tests, wether a group is isomorphic to a Special Linear group. (Note that currently only a few trivial methods are available for this operation. We hope to improve this in the future.)

4 ▶ `IsNaturalSL( matgrp )` P

This property tests, whether a matrix group is the Special Linear group in the right dimension over the (smallest) ring which contains all entries of its elements. (Currently, only a trivial test that computes the order of the group is available.)

```
gap> IsNaturalGL(m);
false
```

5 ▶ `IsSubgroupSL( matgrp )` P

This property tests, whether a matrix group is a subgroup of the Special Linear group in the right dimension over the (smallest) ring which contains all entries of its elements.

(See also section 48.2.)

42.4 Invariant Forms

1 ▶ `InvariantBilinearForm( matgrp )` A

This attribute describes a bilinear form that is invariant under the matrix group *matgrp*. The form is given by a record with the component `matrix` which is a matrix m such that for every generator g of *matgrp* the equation $g \cdot m \cdot g^{tr} = m$ holds.

2 ▶ `IsFullSubgroupGLorSLRespectingBilinearForm( matgrp )` P

This property tests, whether a matrix group *matgrp* is the full subgroup of GL or SL (the property `IsSubgroupSL` determines which it is) respecting the `InvariantBilinearForm` of *matgrp*.

3 ▶ `InvariantSesquilinearForm( matgrp )` A

This attribute describes a sesquilinear form that is invariant under the matrix group *matgrp* over the field F with q^2 elements, say. The form is given by a record with the component `matrix` which is is a matrix m such that for every generator g of *matgrp* the equation $g \cdot m \cdot (g^{tr})^f$ holds, where f is the automorphism of F that raises each element to the q-th power. (f can be obtained as a power of `FrobeniusAutomorphism( F )`, see 57.4.1.)

4 ▶ `IsFullSubgroupGLorSLRespectingSesquilinearForm( matgrp )` P

This property tests, whether a matrix group *matgrp* is the full subgroup of GL or SL (the property `IsSubgroupSL` determines which it is) respecting the `InvariantSesquilinearForm` of *matgrp*.

5 ▶ `InvariantQuadraticForm( matgrp )` A

For a matrix group *matgrp*, `InvariantQuadraticForm` returns a record containing at least the component `matrix` whose value is a matrix Q. The quadratic form q on the natural vector space V on which *matgrp* acts is given by $q(v) = vQv^{tr}$, and the invariance under *matgrp* is given by the equation $q(v) = q(vM)$ for

all $v \in V$ and M in *matgrp*. (Note that the invariance of q does **not** imply that the matrix Q is invariant under *matgrp*.)

q is defined relative to an invariant symmetric bilinear form f (see 42.4.1), via the equation $q(\lambda x + \mu y) = \lambda^2 q(x) + \lambda\mu f(x,y) + \mu^2 q(y)$ (see Chapter 3.4 in [CCN+85]). If f is represented by the matrix F then this implies $F = Q + Q^{tr}$. In characteristic different from 2, we have $q(x) = f(x,x)/2$, so Q can be chosen as the strictly upper triangular part of F plus half of the diagonal part of F. In characteristic 2, F does not determine Q but still Q can be chosen as an upper (or lower) triangular matrix.

Whenever the `InvariantQuadraticForm` value is set in a matrix group then also the `InvariantBilinearForm` value can be accessed, and the two values are compatible in the above sense.

6 ▶ `IsFullSubgroupGLorSLRespectingQuadraticForm(` *matgrp* `)` P

This property tests, whether the matrix group *matgrp* is the full subgroup of GL or SL (the property `IsSubgroupSL` determines which it is) respecting the `InvariantQuadraticForm` value of *matgrp*.

```
gap> g:= Sp( 2, 3 );;
gap> m:= InvariantBilinearForm( g ).matrix;
[ [ 0*Z(3), Z(3)^0 ], [ Z(3), 0*Z(3) ] ]
gap> [ 0, 1 ] * m * [ 1, -1 ];          # evaluate the bilinear form
Z(3)
gap> IsFullSubgroupGLorSLRespectingBilinearForm( g );
true
gap> g:= SU( 2, 4 );;
gap> m:= InvariantSesquilinearForm( g ).matrix;
[ [ 0*Z(2), Z(2)^0 ], [ Z(2)^0, 0*Z(2) ] ]
gap> [ 0, 1 ] * m * [ 1, 1 ];           # evaluate the bilinear form
Z(2)^0
gap> IsFullSubgroupGLorSLRespectingSesquilinearForm( g );
true
gap> g:= GO( 1, 2, 3 );;
gap> m:= InvariantBilinearForm( g ).matrix;
[ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3) ] ]
gap> [ 0, 1 ] * m * [ 1, 1 ];           # evaluate the bilinear form
Z(3)^0
gap> q:= InvariantQuadraticForm( g ).matrix;
[ [ 0*Z(3), Z(3)^0 ], [ 0*Z(3), 0*Z(3) ] ]
gap> [ 0, 1 ] * q * [ 0, 1 ];           # evaluate the quadratic form
0*Z(3)
gap> IsFullSubgroupGLorSLRespectingQuadraticForm( g );
true
```

42.5 Matrix Groups in Characteristic 0

Most of the functions described in this and the following section have implementations which use functions from the GAP package Carat. If Carat is not installed or not compiled, no suitable methods are available.

1 ▶ `IsCyclotomicMatrixGroup(` G `)` P

tests whether all matrices in G have cyclotomic entries.

2 ▶ `IsRationalMatrixGroup(` G `)` P

tests whether all matrices in G have rational entries.

3 ▶ `IsIntegerMatrixGroup( G )` P

tests whether all matrices in G have integer entries.

4 ▶ `IsNaturalGLnZ( G )` P

tests whether G is $GL_n(\mathbb{Z})$ in its natural representation by $n \times n$ integer matrices. (The dimension n will be read off the generating matrices.)

```
gap> IsNaturalGLnZ( GL( 2, Integers ) );
true
```

5 ▶ `IsNaturalSLnZ( G )` P

tests whether G is $SL_n(\mathbb{Z})$ in its natural representation by $n \times n$ integer matrices. (The dimension n will be read off the generating matrices.)

```
gap> IsNaturalSLnZ( SL( 2, Integers ) );
true
```

6 ▶ `InvariantLattice( G )` A

returns a matrix B, whose rows form a basis of a $\mathbb{Z}$-lattice that is invariant under the rational matrix group G acting from the right. It returns `fail` if the group is not unimodular. The columns of the inverse of B span a $\mathbb{Z}$-lattice invariant under G acting from the left.

7 ▶ `NormalizerInGLnZ( G )` A

is an attribute used to store the normalizer of G in $GL_n(\mathbb{Z})$, where G is an integer matrix group of dimension n. This attribute is used by `Normalizer( GL( n, Integers ), G )`.

8 ▶ `CentralizerInGLnZ( G )` A

is an attribute used to store the centralizer of G in $GL_n(\mathbb{Z})$, where G is an integer matrix group of dimension n. This attribute is used by `Centralizer( GL( n, Integers ), G )`.

9 ▶ `ZClassRepsQClass( G )` A

The conjugacy class in $GL_n(\mathbb{Q})$ of the finite integer matrix group G splits into finitely many conjugacy classes in $GL_n(\mathbb{Z})$. `ZClassRepsQClass( G )` returns representative groups for these.

10 ▶ `IsBravaisGroup( G )` P

test whether G coincides with its Bravais group (see 42.5.11).

11 ▶ `BravaisGroup( G )` A

returns the Bravais group of a finite integer matrix group G. If C is the cone of positive definite quadratic forms Q invariant under $g \to g * Q * g^{tr}$ for all $g \in G$, then the Bravais group of G is the maximal subgroup of $GL_n(\mathbb{Z})$ leaving the forms in that same cone invariant. Alternatively, the Bravais group of G can also be defined with respect to the action $g \to g^{tr} * Q * g$ on positive definite quadratic forms Q. This latter definition is appropriate for groups G acting from the right on row vectors, whereas the former definition is appropriate for groups acting from the left on column vectors. Both definitions yield the same Bravais group.

12 ▶ `BravaisSubgroups( G )` A

returns the subgroups of the Bravais group of G, which are themselves Bravais groups.

13 ▶ `BravaisSupergroups( G )` A

returns the subgroups of $GL_n(\mathbb{Z})$ that contain the Bravais group of G and are Bravais groups themselves.

14 ▶ `NormalizerInGLnZBravaisGroup( G )` A

returns the normalizer of the Bravais group of G in the appropriate $GL_n(\mathbb{Z})$.

42.6 Acting OnRight and OnLeft

In GAP, matrices by convention act on row vectors from the right, whereas in crystallography the convention is to act on column vectors from the left. The definition of certain algebraic objects important in crystallography implicitly depends on which action is assumed. This holds true in particular for quadratic forms invariant under a matrix group. In a similar way, the representation of affine crystallographic groups, as they are provided by the GAP package CrystGap, depends on which action is assumed. Crystallographers are used to the action from the left, whereas the action from the right is the natural one for GAP. For this reason, a number of functions which are important in crystallography, and whose result depends on which action is assumed, are provided in two versions, one for the usual action from the right, and one for the crystallographic action from the left.

For every such function, this fact is explicitly mentioned. The naming scheme is as follows: If `SomeThing` is such a function, there will be functions `SomeThingOnRight` and `SomeThingOnLeft`, assuming action from the right and from the left, respectively. In addition, there is a generic function `SomeThing`, which returns either the result of `SomeThingOnRight` or `SomeThingOnLeft`, depending on the global variable `CrystGroupDefaultAction`.

1 ► `CrystGroupDefaultAction` V

can have either of the two values `RightAction` and `LeftAction`. The initial value is `RightAction`. For functions which have variants OnRight and OnLeft, this variable determines which variant is returned by the generic form. The value of `CrystGroupDefaultAction` can be changed with with the function `SetCrystGroupDefaultAction`.

2 ► `SetCrystGroupDefaultAction(` *action* `)` F

allows to set the value of the global variable `CrystGroupDefaultAction`. Only the arguments `RightAction` and `LeftAction` are allowed. Initially, the value of `CrystGroupDefaultAction` is `RightAction`

43 Polycyclic Groups

A group G is **polycyclic** if there exists a subnormal series $G = C_1 > C_2 > \ldots > C_n > C_{n+1} = \{1\}$ with cyclic factors. Such a series is called **pc series** of G.

Every polycyclic group is solvable and every finite solvable group is polycyclic. However, there are infinite solvable groups which are not polycyclic.

In GAP there exists a large number of methods for polycyclic groups which are based upon the polycyclic structure of these groups. These methods are usually very efficient and hence GAP tries to use them whenever possible.

In GAP 3 these methods have been available for AgGroups only; that is, for groups defined via a power-commutator presentation, see Chapter 44 for the GAP 4 analogon. This has changed in GAP 4 where these methods can be applied to many types of groups. For example, the methods can be applied to permutation groups or matrix groups which are known to be polycyclic. The only exception is the representation as finitely presented group for which the polycyclic methods cannot be used in general.

At the current state of implementations the methods for polycyclic groups can only be applied to finite groups. However, a more general implementation is planned.

43.1 Polycyclic Generating Systems

Let G be a polycyclic group with a pc series as above. A **polycyclic generating sequence** (**pcgs** for short) of G is a sequence $P := (g_1, \ldots, g_n)$ of elements of G such that $C_i = \langle C_{i+1}, g_i \rangle$ for $1 \leq i \leq n$. Note that each polycyclic group has a pcgs, but except for very small groups, a pcgs is not unique.

For each index i the subsequence of elements $(g_i, \ldots, g_n)$ forms a pcgs of the subgroup C_i. In particular, these **tails** generate the subgroups of the pc series and hence we say that the pc series is **determined** by P.

Let r_i be the index of C_{i+1} in C_i which is either a finite positive number or infinity. Then r_i is the order of $g_i C_{i+1}$ and we call the resulting list of indices the **relative orders** of the pcgs P.

Moreover, with respect to a given pcgs $(g_1, \ldots, g_n)$ each element g of G can be represented in a unique way as a product $g = g_1^{e_1} \cdot g_2^{e_2} \cdots g_n^{e_n}$ with exponents $e_i \in \{0, \ldots, r_i - 1\}$, if r_i is finite, and $e_i \in \mathbb{Z}$ otherwise. Words of this form are called **normal words** or **words in normal form**. Then the integer vector $[e_1, \ldots, e_n]$ is called the **exponent vector** of the element g. Furthermore, the smallest index k such that $e_k \neq 0$ is called the **depth** of g and e_k is the **leading exponent** of g.

For many applications we have to assume that each of the relative orders r_i is either a prime or infinity. This is equivalent to saying that there are no trivial factors in the pc series and the finite factors of the pc series are maximal refined. Then we obtain that r_i is the order of gC_{i+1} for all elements g in $C_i \setminus C_{i+1}$ and we call r_i the **relative order** of the element g.

43.2 Computing a Pcgs

Suppose a group G is given; for example, let G be a permutation or matrix group. Then we can ask GAP to compute a pcgs of this group. If G is not polycyclic, the result will be `fail`.

Note that these methods can only be applied if G is not given as finitely presented group. For finitely presented groups one can try to compute a pcgs via the polycyclic quotient methods, see 45.13.

Note also that a pcgs behaves like a list.

1 ▶ `Pcgs(` *G* `)` A

returns a pcgs for the group G. If *grp* is not polycyclic it returns `fail` **and this result is not stored as attribute value**, in particular in this case the filter `HasPcgs` is **not** set for G!

2 ▶ `IsPcgs(` *obj* `)` C

The category of pcgs.

```
gap> G := Group((1,2,3,4),(1,2));;
gap> p := Pcgs(G);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> IsPcgs( p );
true
gap> p[1];
(3,4)

gap> G := Group((1,2,3,4,5),(1,2));;
gap> Pcgs(G);
fail
```

3 ▶ `CanEasilyComputePcgs(` *grp* `)` F

This filter indicates whether it is possible to compute a pcgs for *grp* cheaply. Clearly, *grp* must be polycyclic in this case. However, not for every polycyclic group there is a method to compute a pcgs at low costs. This filter is used in the method selection mainly. Note that this filter may change its value from false to true.

```
gap> G := Group( (1,2,3,4),(1,2) );
Group([ (1,2,3,4), (1,2) ])
gap> CanEasilyComputePcgs(G);
false
gap> Pcgs(G);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> CanEasilyComputePcgs(G);
true
```

43.3 Defining a Pcgs Yourself

In a number of situations it might be useful to supply a pgcs to a group.

1 ▶ `PcgsByPcSequence(` *fam*, *pcs* `)` O
▶ `PcgsByPcSequenceNC(` *fam*, *pcs* `)` O

constructs a pcgs for the elements family *fam* from the elements in the list *pcs*. The elements must lie in the family *fam*. `PcgsByPcSequence`(`NC`) will always create a new pcgs which is not induced by any other pcgs.

```
gap> fam := FamilyObj( (1,2) );; # the family of permutations
gap> p := PcgsByPcSequence( fam, [(1,2),(1,2,3)] );
Pcgs([ (1,2), (1,2,3) ])
gap> RelativeOrders(p);
[ 2, 3 ]
gap> ExponentsOfPcElement( p, (1,3,2) );
[ 0, 2 ]
```

Note that the elementary operations for such a pcgs might be rather inefficient, since GAP has to use generic methods in this case. It might be helpful to supply the relative orders of the self-defined pcgs as well by `SetRelativeOrders( pcgs, orders )`. See also 43.4.3.

43.4 Elementary Operations for a Pcgs

1 ▶ `RelativeOrders(` *pcgs* `)` A

returns the list of relative orders of the pcgs *pcgs*.

the list of relative orders of the pcgs *pcgs*.

2 ▶ `IsFiniteOrdersPcgs(` *pcgs* `)` P

tests whether the relative orders of *pcgs* are all finite.

3 ▶ `IsPrimeOrdersPcgs(` *pcgs* `)` P

tests whether the relative orders of *pcgs* are prime numbers. Many algorithms require a pcgs to have this property. The operation `IsomorphismRefinedPcGroup` (see 44.4.8) can be of help here.

4 ▶ `PcSeries(` *pcgs* `)` A

returns the subnormal series determined by *pcgs*.

5 ▶ `GroupOfPcgs(` *pcgs* `)` A

The group generated by *pcgs*.

6 ▶ `OneOfPcgs(` *pcgs* `)` A

The identity of the group generated by *pcgs*.

```
gap> G := Group( (1,2,3,4),(1,2) );; p := Pcgs(G);;
gap> RelativeOrders(p);
[ 2, 3, 2, 2 ]
gap> IsFiniteOrdersPcgs(p);
true
gap> IsPrimeOrdersPcgs(p);
true
gap> PcSeries(p);
[ Group([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (2,4,3), (1,4)(2,3), (1,3)(2,4) ]),
  Group([ (1,4)(2,3), (1,3)(2,4) ]), Group([ (1,3)(2,4) ]), Group(()) ]
```

43.5 Elementary Operations for a Pcgs and an Element

1 ▶ `RelativeOrderOfPcElement(` *pcgs*, *elm* `)` O

The relative order of *elm* with respect to the prime order pcgs *pcgs*.

2 ▶ `ExponentOfPcElement(` *pcgs*, *elm*, *pos* `)` O

returns the *pos*-th exponent of *elm* with respect to *pcgs*.

3 ▶ `ExponentsOfPcElement(` *pcgs*, *elm* `)` O
▶ `ExponentsOfPcElement(` *pcgs*, *elm*, *posran* `)` O

returns the exponents of *elm* with respect to *pcgs*. The second form returns the exponents in the positions given in *posran*.

4 ▶ `DepthOfPcElement(` *pcgs*, *elm* `)` O

returns the depth of the element *elm* with respect to *pcgs*.

5 ▶ `LeadingExponentOfPcElement(` *pcgs*, *elm* `)` O

returns the leading exponent of *elm* with respect to *pcgs*.

6 ▶ `PcElementByExponents(` *pcgs*, *list* `)` O
▶ `PcElementByExponentsNC(` *pcgs*, *list* `)` O
▶ `PcElementByExponentsNC(` *pcgs*, *basisind*, *list* `)` O

returns the element corresponding to the exponent vector *list* with respect to *pcgs*. The exponents in *list* must be in the range of permissible exponents for *pcgs*. **It is not guaranteed that** `PcElementByExponents` **will reduce the exponents modulo the relative orders**. (You should use the operation `LinearCombinationPcgs` for this purpose.) The NC version does not check that the lengths of the lists fit together and does not check the exponent range.

The third version gives exponents only wrt. the generators in *pcgs* indexed by *basisind*.

7 ▶ `LinearCombinationPcgs(` *pcgs*, *list* `[,` *one*`] )` O

returns the product $\prod_i pcgs[i]^{list[i]}$. In contrast to `PcElementByExponents` this permits negative exponents. *pcgs* might be an list of group elements, in this case, an appropriate *one* must be given. if *list* can be empty.

```
gap> G := Group( (1,2,3,4),(1,2) );; P := Pcgs(G);;
gap> g := PcElementByExponents(P, [0,1,1,1]);
(1,2,3)
gap> ExponentsOfPcElement(P, g);
[ 0, 1, 1, 1 ]
```

8 ▶ `SiftedPcElement(` *pcgs*, *elm* `)` O

sifts *elm* through *pcgs*, reducing it if the depth is the same as the depth of one of the generators in *pcgs*. Thus the identity is returned if *elm* lies in the group generated by *pcgs*. *pcgs* must be an induced pcgs and *elm* must lie in the span of the parent of *pcgs*.

9 ▶ `CanonicalPcElement(` *ipcgs*, *elm* `)` O

reduces *elm* at the induces pcgs *ipcgs* such that the exponents of the reduced result r are zero at the depths for which there are generators in *ipcgs*. Elements, whose quotient lies in the group generated by *ipcgs* yield the same canonical element.

10 ▶ `ReducedPcElement(` *pcgs*, x, y `)` O

reduces the element x by dividing off (from the left) a power of y such that the leading coefficient of the result with respect to *pcgs* becomes zero. The elements x and y therefore have to have the same depth.

11 ▶ `CleanedTailPcElement( pcgs, elm, dep )` O

returns an element in the span of *pcgs* whose exponents for indices 1 to $dep - 1$ with respect to *pcgs* are the same as those of *elm*, the remaining exponents are undefined. This can be used to obtain more "simple" elements if only representatives in a factor are required, see 43.9.

The difference to `HeadPcElementByNumber` (see 43.5.12) is that `HeadPcElementByNumber` is guaranteed to zero out trailing coefficients while `CleantedTailPcElement` will only do this if it can be done cheaply.

12 ▶ `HeadPcElementByNumber( pcgs, elm, dep )` O

returns an element in the span of *pcgs* whose exponents for indices 1 to $dep - 1$ with respect to *pcgs* are the same as those of *elm*, the remaining exponents are zero. This can be used to obtain more "simple" elements if only representatives in a factor are required.

43.6 Exponents of Special Products

There are certain products of elements whose exponents are used often within algorithms, and which might be obtained more easily than by computing the product first and to obtain its exponents afterwards. The operations in this section provide a way to obtain such exponent vectors directly.

(The circumstances under which these operations give a speedup depend very much on the pcgs and the representation of elements that is used. So the following operations are not guaranteed to give a speedup in every case, however the default methods are not slower than to compute the exponents of a product and thus these operations should **always** be used if applicable.)

1 ▶ `ExponentsConjugateLayer( mpcgs, elm, e )` O

Computes the exponents of elm^e with respect to *mpcgs*; *elm* must be in the span of *mpcgs*, *e* a pc element in the span of the parent pcgs of *mpcgs* and *mpcgs* must be the modulo pcgs for an abelian layer. (This is the usual case when acting on a chief factor). In this case if *mpcgs* is induced by the family pcgs, the exponents can be computed directly by looking up exponents without having to compute in the group and having to collect a potential tail.

The second class are exponents of products of the generators which make up the pcgs. If the pcgs used is a `FamilyPcgs` these exponents can be looked up and do not need to be computed.

2 ▶ `ExponentsOfRelativePower( pcgs, i )` O

For $p = pcgs[i]$ this function returns the exponent vector with respect to *pcgs* of the element p^e where e is the relative order of p in *pcgs*. For the family pcgs or pcgs induced by it, this might be faster than computing the element and computing its exponent vector.

3 ▶ `ExponentsOfConjugate( pcgs, i, j )` O

returns the exponents of $pcgs[i]^{pcgs}[j]$ with respect to *pcgs*. For the family pcgs or pcgs induced by it, this might be faster than computing the element and computing its exponent vector.

4 ▶ `ExponentsOfCommutator( pcgs, i, j )` O

returns the exponents of the commutatior $\mathrm{Comm}(pcgs[i], pcgs[j])$ with respect to *pcgs*. For the family pcgs or pcgs induced by it, this might be faster than computing the element and computing its exponent vector.

43.7 Subgroups of Polycyclic Groups - Induced Pcgs

Let U be a subgroup of G and let P be a pcgs of G as above such that P determines the subnormal series $G = C_1 > \ldots > C_{n+1} = \{1\}$. Then the series of subgroups $U \cap C_i$ is a subnormal series of U with cyclic or trivial factors. Hence, if we choose an element $u_{i_j} \in (U \cap C_{i_j}) \setminus (U \cap C_{i_j+1})$ whenever this factor is non-trivial, then we obtain a pcgs $Q = (u_{i_1}, \ldots, u_{i_m})$ of U. We say that Q is an **induced pcgs** with respect to P. The pcgs P is the **parent pcgs** to the induced pcgs Q.

Note that the pcgs Q is induced with respect to P if and only if the matrix of exponent vectors of the elements u_{i_j} with respect to P is in upper triangular form. Thus Q is not unique in general.

In particular, the elements of an induced pcgs do **not necessarily** have leading coefficient 1 relative to the inducing pcgs. The attribute `LeadCoeffsIGS` (see 43.7.7) holds the leading coefficients in case they have to be renormed in an algorithm.

Each induced pcgs is a pcgs and hence allows all elementary operations for pcgs. On the other hand each pcgs could be transformed into an induced pcgs for the group defined by the pcgs, but note that an arbitrary pcgs is in general not an induced pcgs for technical reasons.

An induced pcgs is "compatible" with its parent.

1 ▶ `IsInducedPcgs(` *pcgs* `)` C

The category of induced pcgs. This a subcategory of pcgs.

2 ▶ `InducedPcgsByPcSequence(` *pcgs*, *pcs* `)` O
▶ `InducedPcgsByPcSequenceNC(` *pcgs*, *pcs* `)` O
▶ `InducedPcgsByPcSequenceNC(` *pcgs*, *pcs*, *depths* `)` O

If *pcs* is a list of elements that form an induced pcgs with respect to *pcgs* this operation returns an induced pcgs with these elements.

In the third version, the depths of *pcs* with respect to *pcgs* can be given (they are computed anew otherwise).

3 ▶ `ParentPcgs(` *pcgs* `)` A

returns the pcgs by which *pcgs* was induced. If *pcgs* was not induced, it simply returns *pcgs*.

```
gap> G := Group( (1,2,3,4),(1,2) );;
gap> P := Pcgs(G);;
gap> K := InducedPcgsByPcSequence( P, [(1,2,3,4),(1,3)(2,4)] );
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
gap> ParentPcgs( K );
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> IsInducedPcgs( K );
true
```

In [LNS84] a "non-commutative gauss" algorithm is described to compute an induced pcgs of a subgroup U from a generating set of U. This can be called in GAP via one of the following commands.

4 ▶ `InducedPcgs(` *pcgs*, *grp* `)` O

computes a pcgs for *grp* which is induced by *pcgs*. If *pcgs* has a parent pcgs, then the result is induced with respect to this parent pcgs.

5 ▶ `InducedPcgsByGenerators(` *pcgs*, *gens* `)` O
▶ `InducedPcgsByGeneratorsNC(` *pcgs*, *gens* `)` O

returns an induced pcgs with respect to *pcgs* for the subgroup generated by *gens*.

6 ► `InducedPcgsByPcSequenceAndGenerators( pcgs, ind, gens )` O

returns an induced pcgs with respect to *pcgs* of the subgroup generated by *ind* and *gens*. Here *ind* must be an induced pcgs with respect to *pcgs* (or a list of group elements that form such an igs) and it will be used as initial sequence for the computation.

```
gap> G := Group( (1,2,3,4),(1,2) );;  P := Pcgs(G);;
gap> I := InducedPcgsByGenerators( P, [(1,2,3,4)] );
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
gap> J := InducedPcgsByPcSequenceAndGenerators( P, I, [(1,2)] );
Pcgs([ (1,2,3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
```

7 ► `LeadCoeffsIGS( igs )` A

This attribute is used to store leading coefficients with respect to the parent pcgs. the i-th entry - if bound - is the leading exponent of the element of *igs* that has depth i in the parent. (It cannot be assigned to a component in `InducedPcgsByPcSequenceNC` as the permutation group methods call it from within the postprocessing, before this postprocessing however no coefficients may be computed.)

8 ► `ExtendedPcgs( N, gens )` O

extends the pcgs N (induced wrt. *home*) to a new induced pcgs by prepending *gens*. No checks are performed that this really yields an induced pcgs.

To create a subgroup generated by an induced pcgs such that the induced pcgs gets stored automatically there is the following operation.

9 ► `SubgroupByPcgs( G, pcgs )` O

43.8 Subgroups of Polycyclic Groups - Canonical Pcgs

The induced pcgs Q of U is called **canonical** if the matrix of exponent vectors contains normed vectors only and above each leading entry in the matrix there are 0's only. The canonical pcgs of U with respect to P is unique and hence such pcgs can be used to compare subgroups.

1 ► `IsCanonicalPcgs( pcgs )` P

An induced pcgs is canonical if the matrix of the exponent vectors of the elements of *pcgs* with respect to `ParentPcgs(`*pcgs*`)` is in Hermite normal form (see [LNS84]). While a subgroup can have various induced pcgs with respect to a parent pcgs a canonical pcgs is unique.

2 ► `CanonicalPcgs( pcgs )` A

returns the canonical pcgs corresponding to the induced pcgs *pcgs*.

```
gap> G := Group((1,2,3,4),(5,6,7));
Group([ (1,2,3,4), (5,6,7) ])
gap> P := Pcgs(G);
Pcgs([ (5,6,7), (1,2,3,4), (1,3)(2,4) ])
gap> I := InducedPcgsByPcSequence(P, [(5,6,7)*(1,3)(2,4),(1,3)(2,4)] );
Pcgs([ (1,3)(2,4)(5,6,7), (1,3)(2,4) ])
gap> CanonicalPcgs(I);
Pcgs([ (5,6,7), (1,3)(2,4) ])
```

43.9 Factor Groups of Polycyclic Groups - Modulo Pcgs

Let N be a normal subgroup of G such that G/N is polycyclic with pcgs $(h_1N, \ldots, h_rN)$. Then we call the sequence of preimages $(h_1, \ldots h_r)$ a **modulo pcgs** of G/N. G is called the **numerator** of the modulo pcgs and N is the **denominator** of the modulo pcgs.

Modulo pcgs are often used to facilitate efficient computations with factor groups, since they allow computations with factor groups without formally defining the factor group at all.

All elementary operations of pcgs, see Sections 43.4 and 43.5, apply to modulo pcgs as well. However, it is in general not possible to compute induced pcgs with respect to a modulo pcgs.

1 ▶ `ModuloPcgs(` *G*`,` *N* `)` O

returns a modulo pcgs for the factor G/N which must be solvable, which N may be insolvable.

`ModuloPcgs` will return **a** pcgs for the factor, there is no guarantee that it will be "compatible" with any other pcgs. If this is required, the `mod` operator must be used on induced pcgs, see below.

2 ▶ `IsModuloPcgs(` *obj* `)` C

The category of modulo pcgs. Note that each pcgs is a modulo pcgs for the trivial subgroup.

Additionally there are two more elementary operations for modulo pcgs.

3 ▶ `NumeratorOfModuloPcgs(` *pcgs* `)` A

returns a generating set for the numerator of the modulo pcgs *pcgs*.

4 ▶ `DenominatorOfModuloPcgs(` *pcgs* `)` A

returns a generating set for the denominator of the modulo pcgs *pcgs*.

```
gap> G := Group( (1,2,3,4,5),(1,2) );
Group([ (1,2,3,4,5), (1,2) ])
gap> P := ModuloPcgs(G, DerivedSubgroup(G) );
Pcgs([ (4,5) ])
gap> NumeratorOfModuloPcgs(P);
[ (1,2,3,4,5), (1,2) ]
gap> DenominatorOfModuloPcgs(P);
[ (1,3,2), (2,4,3), (2,3)(4,5) ]
gap> RelativeOrders(P);
[ 2 ]
gap> ExponentsOfPcElement( P, (1,2,3,4,5) );
[ 0 ]
gap> ExponentsOfPcElement( P, (4,5) );
[ 1 ]
```

Modulo Pcgs can also be built from compatible induced pcgs. Let G be a group with pcgs P and let I be an induced pcgs of a normal subgroup N of G. (Respectively: P and I are both induced with respect to the **same** Pcgs.) Then we can compute a modulo pcgs of G mod N by

5 ▶ *P* `mod` *I*

Note that in this case we obtain the advantage that the `NumeratorOfModuloPcgs` and the `DenominatorOfModuloPcgs` are just P and I, respectively, and hence are unique.

The resulting modulo pcgs will consist of a subset of P and will be "compatible" with P (or its parent).

```
gap> G := Group((1,2,3,4));;
gap> P := Pcgs(G);
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
gap> I := InducedPcgsByGenerators(P, [(1,3)(2,4)]);
Pcgs([ (1,3)(2,4) ])
gap> M := P mod I;
[ (1,2,3,4) ]
gap> NumeratorOfModuloPcgs(M);
Pcgs([ (1,2,3,4), (1,3)(2,4) ])
gap> DenominatorOfModuloPcgs(M);
Pcgs([ (1,3)(2,4) ])
```

6 ► `CorrespondingGeneratorsByModuloPcgs( mpcgs, imgs )` O

let *mpcgs* be a modulo pcgs for a factor of a group G and let U be a subgroup of G generated by *imgs* such that U covers the factor for the modulo pcgs. Then this function computes elements in U corresponding to the generators of the modulo pcgs.

Note that the computation of induced generating sets is not possible for some modulo pcgs.

7 ► `CanonicalPcgsByGeneratorsWithImages( pcgs, gens, imgs )` O

computes a canonical, *pcgs*-induced pcgs for the span of *gens* and simultaneously does the same transformations on *imgs*, preserving thus a correspondence between *gens* and *imgs*. This operation is used to represent homomorphisms from a pc group.

43.10 Factor Groups of Polycyclic Groups in their Own Representation

If substantial calculations are done in a factor it might be worth still to construct the factor group in its own representation (for example by calling `PcGroupWithPcgs` on a modulo pcgs, see 44.5.1).

The following functions are intended for working with factor groups obtained by factoring out the tail of a pcgs. They provide a way to map elements or induced pcgs quickly in the factor (respectively to take preimages) without the need to construct a homomorphism.

The setup is always a pcgs *pcgs* of G and a pcgs *fpcgs* of a factor group $H = G/N$ which corresponds to a head of *pcgs*.

No tests for validity of the input are performed.

1 ► `ProjectedPcElement( pcgs, fpcgs, elm )` F

returns the image in H of an element *elm* of G.

2 ► `ProjectedInducedPcgs( pcgs, fpcgs, ipcgs )` F

ipcgs must be an induced pcgs with respect to *pcgs*. This operation returns an induced pcgs with respect to *fpcgs* consisting of the nontrivial images of *ipcgs*.

3 ► `LiftedPcElement( pcgs, fpcgs, elm )` F

returns a preimage in G of an element *elm* of H.

4 ► `LiftedInducedPcgs( pcgs, fpcgs, ipcgs, ker )` F

ipcgs must be an induced pcgs with respect to *fpcgs*. This operation returns an induced pcgs with respect to *pcgs* consisting of the preimages of *ipcgs*, appended by the elements in *ker* (assuming there is a bijection of *pcgs* mod *ker* to *fpcgs*). *ker* might be a simple element list.

43.11 Pcgs and Normal Series

By definition, a pcgs determines a pc series of its underlying group. However, in many applications it will be necessary that this pc series refines a normal series with certain properties; for example, a normal series with abelian factors.

There are functions in GAP to compute a pcgs through a normal series with elementary abelian factors, a central series or the lower p-central series. See also Section 43.13 for a more explicit possibility.

1 ▶ `IsPcgsElementaryAbelianSeries( pcgs )` P

returns `true` if the pcgs *pcgs* refines an elementary abelian series. `IndicesEANormalSteps` then gives the indices in the Pcgs, at which the subgroups of this series start.

2 ▶ `PcgsElementaryAbelianSeries( G )` A
▶ `PcgsElementaryAbelianSeries( [G, N1, N2, ....] )` A

computes a pcgs for G that refines an elementary abelian series. `IndicesEANormalSteps` gives the indices in the Pcgs, at which the normal subgroups of this series start. The second variant returns a pcgs that runs through the normal subgroups *N1*, *N2*, etc.

3 ▶ `IndicesEANormalSteps( pcgs )` A

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with elementary abelian factors (for example from calling `PcgsElementaryAbelianSeries`) Then `IndicesEANormalSteps` returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroup of G. If i is an element of this list, $(g_i, \ldots, g_n)$ is a normal subgroup of G. The list always starts with 1 and ends with n+1. (These indices form **one** series with elementary abelian subfactors, not necessarily the most refined one.)

The attribute `EANormalSeriesByPcgs` returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and `IndicesEANormalSteps` is not guaranteed to work in this situation.

Typically, `IndicesEANormalSteps` is set by `PcgsElementaryAbelianSeries`.

4 ▶ `EANormalSeriesByPcgs( pcgs )` A

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with elementary abelian factors (for example from calling `PcgsElementaryAbelianSeries`). This attribute returns the actual series of normal subgroups, corresponding to `IndicesEANormalSteps`.

5 ▶ `IsPcgsCentralSeries( pcgs )` P

returns `true` if the pcgs *pcgs* refines an central elementary abelian series. `IndicesCentralNormalSteps` then gives the indices in the Pcgs, at which the subgroups of this series start.

6 ▶ `PcgsCentralSeries( G )` A

computes a pcgs for G that refines a central elementary abelian series. `IndicesCentralNormalSteps` gives the indices in the Pcgs, at which the normal subgroups of this series start.

7 ▶ `IndicesCentralNormalSteps( pcgs )` A

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with central elementary abelian factors (for example from calling `PcgsCentralSeries`) Then `IndicesCentralNormalSteps` returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroup of G. If i is an element of this list, $(g_i, \ldots, g_n)$ is a normal subgroup of G. The list always starts with 1 and ends with n+1. (These indices form **one** series with central elementary abelian subfactors, not necessarily the most refined one.)

The attribute `CentralNormalSeriesByPcgs` returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and `IndicesCentralNormalSteps` is not guaranteed to work in this situation.

Typically, `IndicesCentralNormalSteps` is set by `PcgsCentralSeries`.

8 ▶ `CentralNormalSeriesByPcgs( pcgs )` A

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with central elementary abelian factors (for example from calling `PcgsCentralSeries`). This attribute returns the actual series of normal subgroups, corresponding to `IndicesCentralNormalSteps`.

9 ▶ `IsPcgsPCentralSeriesPGroup( pcgs )` P

returns `true` if the pcgs *pcgs* refines an p-central elementary abelian series for a p-group. `IndicesPCentralNormalStepsPGroup` then gives the indices in the Pcgs, at which the subgroups of this series start.

10 ▶ `PcgsPCentralSeriesPGroup( G )` A

computes a pcgs for the p-group G that refines a p-central elementary abelian series. `IndicesPCentralNormalStepsPGroup` gives the indices in the Pcgs, at which the normal subgroups of this series start.

11 ▶ `IndicesPCentralNormalStepsPGroup( pcgs )` A

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with p-central elementary abelian factors (for example from calling `PcgsPCentralSeriesPGroup`). Then `IndicesPCentralNormalStepsPGroup` returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroup of G. If i is an element of this list, $(g_i, \ldots, g_n)$ is a normal subgroup of G. The list always starts with 1 and ends with n+1. (These indices form **one** series with central elementary abelian subfactors, not necessarily the most refined one.)

The attribute `PCentralNormalSeriesByPcgsPGroup` returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and `IndicesPCentralNormalStepsPGroup` is not guaranteed to work in this situation.

Typically, `IndicesPCentralNormalStepsPGroup` is set by `PcgsPCentralSeriesPGroup`.

12 ▶ `PCentralNormalSeriesByPcgsPGroup( pcgs )` A

Let *pcgs* be a pcgs obtained as corresponding to a series of normal subgroups with p-central elementary abelian factors (for example from calling `PcgsPCentralSeriesPGroup`). This attribute returns the actual series of normal subgroups, corresponding to `IndicesPCentralNormalStepsPGroup`.

13 ▶ `IsPcgsChiefSeries( pcgs )` P

returns `true` if the pcgs *pcgs* refines a chief series. `IndicesChiefNormalSteps` then gives the indices in the Pcgs, at which the subgroups of this series start.

14 ▶ `PcgsChiefSeries( G )` A

computes a pcgs for G that refines a chief series. `IndicesChiefNormalSteps` gives the indices in the Pcgs, at which the normal subgroups of this series start.

15 ▶ `IndicesChiefNormalSteps( pcgs )` A

Let *pcgs* be a pcgs obtained as corresponding to a chief series for example from calling `PcgsChiefSeries`). Then `IndicesChiefNormalSteps` returns a sorted list of integers, indicating the tails of *pcgs* which generate these normal subgroup of G. If i is an element of this list, $(g_i, \ldots, g_n)$ is a normal subgroup of G. The list always starts with 1 and ends with n+1. (These indices form **one** series with elementary abelian subfactors, not necessarily the most refined one.)

The attribute `ChiefNormalSeriesByPcgs` returns the actual series of subgroups.

For arbitrary pcgs not obtained as belonging to a special series such a set of indices not necessarily exists, and `IndicesChiefNormalSteps` is not guaranteed to work in this situation.

Typically, `IndicesChiefNormalSteps` is set by `PcgsChiefSeries`.

16 ► `ChiefNormalSeriesByPcgs(` *pcgs* `)` A

Let *pcgs* be a pcgs obtained as corresponding to a chief series (for example from calling `PcgsChiefSeries`). This attribute returns the actual series of normal subgroups, corresponding to `IndicesChiefNormalSteps`.

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> p:=PcgsElementaryAbelianSeries(g);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> IndicesEANormalSteps(p);
[ 1, 2, 3, 5 ]
gap> g:=Group((1,2,3,4),(1,5)(2,6)(3,7)(4,8));;
gap> p:=PcgsCentralSeries(g);
Pcgs([ (1,5)(2,6)(3,7)(4,8), (5,6,7,8), (5,7)(6,8), (1,4,3,2)(5,6,7,8),
  (1,3)(2,4)(5,7)(6,8) ])
gap> IndicesCentralNormalSteps(p);
[ 1, 2, 4, 5, 6 ]
gap> q:=PcgsPCentralSeriesPGroup(g);
Pcgs([ (1,5)(2,6)(3,7)(4,8), (5,6,7,8), (5,7)(6,8), (1,4,3,2)(5,6,7,8),
  (1,3)(2,4)(5,7)(6,8) ])
gap> IndicesPCentralNormalStepsPGroup(q);
[ 1, 3, 5, 6 ]
```

17 ► `IndicesNormalSteps(` *pcgs* `)` A

returns the indices of **all** steps in the pc series, which are normal in the group defined by the pcgs.

(In general, this function yields a slower performance than the more specialized index functions for elementary abelian series etc.)

18 ► `NormalSeriesByPcgs(` *pcgs* `)` A

returns the subgroups the pc series, which are normal in the group defined by the pcgs.

(In general, this function yields a slower performance than the more specialized index functions for elementary abelian series etc.)

43.12 Sum and Intersection of Pcgs

1 ► `SumFactorizationFunctionPcgs(` *parentpcgs*, *n*, *u*, *kerpcgs* `)` O

computes the sum and intersection of the lists n and u whose elements form modulo pcgs induced by *parentpcgs* for two subgroups modulo a kernel given by *kerpcgs*. If *kerpcgs* is a tail if the *parent-pcgs* it is sufficient to give the starting depth, this can be more efficient than to construct an explicit pcgs. The factor group modulo *kerpcgs* generated by n must be elementary abelian and normal under u.

The function returns a record with components

`sum`
: Elements that form a modulo pcgs for the span of both subgroups.

`intersection`
: Elements that form a modulo pcgs for the intersection of both subgroups.

`factorization`
: A function that returns for an element x in the span of `sum` a record with components `u` and `n` that give its decomposition.

The record components `sum` and `intersection` are **not** pcgs but only lists of pc elements (to avoid unnecessary creation of InducedPcgs).

43.13 Special Pcgs

In short, a special pcgs is a pcgs which has particularly nice properties, for example it always refines an elementary abelian series, for p-groups it even refines a central series. These nice properties permit particularly efficient algorithms.

Let G be a finite polycyclic group. A **special pcgs** of G is a pcgs which is closely related to a Hall system and the maximal subgroups of G. These pcgs have been introduced by C. R. Leedham-Green who also gave an algorithm to compute them. Improvements to this algorithm are due to Bettina Eick. For a more detailed account of their definition the reader is referred to [Eic97]

To introduce the definition of special pcgs we first need to define the **LG-series** and **head complements** of a finite polycyclic group G. Let $G = G_1 > G_2 > \ldots G_m > G_{m+1} = \{1\}$ be the lower nilpotent series of G; that is, G_i is the smallest normal subgroup of G_{i-1} with nilpotent factor. To obtain the LG-series of G we need to refine this series. Thus consider a factor $F_i := G_i/G_{i+1}$. Since F_i is finite nilpotent, it is a direct product of its Sylow subgroups, say $F_i = P_{i,1} \cdots P_{i,r_i}$. For each Sylow p_j-subgroup $P_{i,j}$ we can consider its lower p_j-central series. To obtain a characteristic central series with elementary abelian factors of F_i we loop over its Sylow subgroups. Each time we consider $P_{i,j}$ in this process we take the next step of its lower p_j-central series into the series of F_i. If there is no next step, then we just skip the consideration of $P_{i,j}$. Note that the second term of the lower p-central series of a p-group is in fact its Frattini subgroup. Thus the Frattini subgroup of F_i is contained in the computed series of this group. We denote the Frattini subgroup of $F_i = G_i/G_{i+1}$ by G_i^*/G_{i+1}.

The factors G_i/G_i^* are called the heads of G, while the (possibly trivial) factors G_i^*/G_{i+1} are the tails of G. A head complement of G is a subgroup U of G such that U/G_i^* is a complement to the head G_i/G_i^* in G/G_i^* for some i.

Now we are able to define a special pcgs of G. It is a pcgs of G with three additional properties. First, the pc series determined by the pcgs refines the LG-series of G. Second, a special pcgs **exhibits** a Hall system of the group G; that is, for each set of primes π the elements of the pcgs with relative order in π form a pcgs of a Hall π-subgroup in a Hall system of G. Third, a special pcgs exhibits a head complement for each head of G.

To record information about the LG-series with the special pcgs we define the **LGWeights** of the special pcgs. These weights are a list which contains a weight w for each elements g of the special pcgs. Such a weight w represents the smallest subgroup of the LG-series containing g.

Since the LG-series is defined in terms of the lower nilpotent series, Sylow subgroups of the factors and lower p-central series of the Sylow subgroup, the weight w is a triple. More precisely, g is contained in the $w[1]$th term U of the lower nilpotent series of G, but not in the next smaller one V. Then $w[3]$ is a prime such that gV is contained in the Sylow $w[3]$-subgroup P/V of U/V. Moreover, gV is contained in the $w[2]$th term of the lower p-central series of P/V.

There are two more attributes of a special pcgs containing information about the LG-series: the list **LGLayers** and the list **LGFirst**. The list of layers corresponds to the elements of the special pcgs and denotes the layer of the LG-series in which an element lies. The list LGFirst corresponds to the LG-series and gives the number of the first element in the special pcgs of the corresponding subgroup.

1 ▶ `IsSpecialPcgs( obj )` P

tests whether *obj* is a special pcgs.

2 ▶ `SpecialPcgs( pcgs )` A
▶ `SpecialPcgs( G )` A

computes a special pcgs for the group defined by *pcgs* or for G.

3 ▶ `LGWeights( pcgs )` A

returns the LGWeights of the special pcgs *pcgs*.

4 ► `LGLayers( pcgs )` A

returns the layers of the special pcgs *pcgs*.

5 ► `LGFirst( pcgs )` A

returns the first indices for each layer of the special pcgs *pcgs*.

6 ► `LGLength( G )` A

returns the Length of the LG-series of the group G, if G is solvable and *fail* otherwise.

```
gap> G := SmallGroup( 96, 220 );
<pc group of size 96 with 6 generators>
gap> spec := SpecialPcgs( G );
Pcgs([ f1, f2, f3, f4, f5, f6 ])
gap> LGWeights(spec);
[ [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 3 ],
  [ 1, 2, 2 ] ]
gap> LGLayers(spec);
[ 1, 1, 1, 1, 2, 3 ]
gap> LGFirst(spec);
[ 1, 5, 6, 7 ]
gap> LGLength( G );
3

gap> p := SpecialPcgs( Pcgs( SmallGroup( 96, 120 ) ) );
Pcgs([ f1, f2, f3, f4, f5, f6 ])
gap> LGWeights(p);
[ [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 2, 2 ], [ 1, 3, 2 ],
  [ 2, 1, 3 ] ]
```

Thus the first group, `SmallGroup(96, 220)`, has a lower nilpotent series of length 1; that is, the group is nilpotent. It is a direct product of its Sylow subgroups. Moreover the Sylow 2-subgroup is generated by $f1, f2, f3, f4, f6$ and the Sylow 3-subgroup is generated by $f5$. The lower 2-central series of the Sylow 2-subgroup has length 2 and the second subgroup in this series is generated by $f6$.

The second group, `SmallGroup(96, 120)`, has a lower nilpotent series of length 2 and hence is not nilpotent. The second subgroup in this series is just the Sylow 3-subgroup and it is generated by $f6$. The subgroup generated by $f1, \ldots, f5$ is a Sylow 2-subgroup of the group and also a head complement to the second head of the group. Its lower 2-central series has length 2.

In this example the `FamilyPcgs` of the groups used was a special pcgs, but this is not necessarily the case. For performance reasons it can be worth to enforce this, see 44.5.3.

7 ► `IsInducedPcgsWrtSpecialPcgs( pcgs )` P

tests whether *pcgs* is induced with respect to a special pcgs.

8 ► `InducedPcgsWrtSpecialPcgs( G )` A

computes an induced pcgs with respect to the special pcgs of the parent of G.

`InducedPcgsWrtSpecialPcgs` will return a pcgs induced by **a** special pcgs (which might differ from the one you had in mind). If you need an induced pcgs compatible with a **given** special pcgs use `InducedPcgs` for this special pcgs.

43.14 Action on Subfactors Defined by a Pcgs

When working with a polycyclic group, one often needs to compute matrix operations of the group on a factor of the group. For this purpose there are the following functions.

1 ▶ `VectorSpaceByPcgsOfElementaryAbelianGroup( mpcgs, fld )` F

returns the vector space over *fld* corresponding to the modulo pcgs *mpcgs*. Note that *mpcgs* has to define an elementary abelian p-group where p is the characteristic of *fld*.

2 ▶ `LinearOperation( gens, basisvectors, linear )` O
▶ `LinearAction( gens, basisvectors, linear )` O

returns a list of matrices, one for each element of *gens*, which corresponds to the matrix action of the elements in *gens* on the basis *basisvectors* via *linear*.

3 ▶ `LinearOperationLayer( G, gens, pcgs )` F
▶ `LinearActionLayer( G, gens, pcgs )` F

returns a list of matrices, one for each element of *gens*, which corresponds to the matrix action of G on the vector space corresponding to the modulo pcgs *pcgs*.

In certain situations, for example within the computation of conjugacy classes of finite soluble groups as described in [MN89], affine actions of groups are required. For this purpose we introduce the following functions.

4 ▶ `AffineOperation( gens, basisvectors, linear, transl )` O
▶ `AffineAction( gens, basisvectors, linear, transl )` O

return a list of matrices, one for each element of *gens*, which corresponds to the affine action of the elements in *gens* on the basis *basisvectors* via *linear* with translation *transl*.

5 ▶ `AffineOperationLayer( G, gens, pcgs, transl )` F
▶ `AffineActionLayer( G, gens, pcgs, transl )` F

returns a list of matrices, one for each element of *gens*, which corresponds to the affine action of G on the vector space corresponding to the modulo pcgs *pcgs* with translation *transl*.

```
gap> G := SmallGroup( 96, 51 );
<pc group of size 96 with 6 generators>
gap> spec := SpecialPcgs( G );
Pcgs([ f1, f2, f3, f4, f5, f6 ])
gap> LGWeights( spec );
[ [ 1, 1, 2 ], [ 1, 1, 2 ], [ 1, 1, 3 ], [ 1, 2, 2 ], [ 1, 2, 2 ],
  [ 1, 3, 2 ] ]
gap> mpcgs := InducedPcgsByPcSequence( spec, spec{[4,5,6]} );
Pcgs([ f4, f5, f6 ])
gap> npcgs := InducedPcgsByPcSequence( spec, spec{[6]} );
Pcgs([ f6 ])
gap> modu := mpcgs mod npcgs;
[ f4, f5 ]
gap> mat:=LinearActionLayer( G, spec{[1,2,3]}, modu );
[ <an immutable 2x2 matrix over GF2>, <an immutable 2x2 matrix over GF2>,
  <an immutable 2x2 matrix over GF2> ]
gap> Print( mat, "\n" );
[ [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ] ]
```

43.15 Orbit Stabilizer Methods for Polycyclic Groups

If a pcgs *pcgs* is known for a group G, then orbits and stabilizers can be computed by a special method which is particularly efficient. Note that within this function only the elements in *pcgs* and the relative orders of *pcgs* are needed. Hence this function works effectively even if the elementary operations for *pcgs* are slow.

1 ▶ `StabilizerPcgs(` *pcgs*`,` *pt* `)`

2 ▶ `Pcgs_OrbitStabilizer(` *pcgs*`,` *domain*`,` *pnt*`,` *oprs*`,` *opr* `)` F

runs a solvable group orbit-stabilizer algorithm on *pnt* with *pcgs* acting via the images *oprs* and the operation function *opr*. The domain *domain* can be used to speed up search, if it is not known, `false` can be given instead. The function returns a record with components `orbit`, `stabpcgs` and `lengths`, the latter indicating the lengths of the orbit whenever it got extended. This can be used to recompute transversal elements. This function should be used only inside algorithms when speed is essential.

43.16 Operations which have Special Methods for Groups with Pcgs

For the following methods there are special operations for groups with pcgs installed:

`IsNilpotent`, `IsSupersolvable`, `Size`, `CompositionSeries`, `ConjugacyClasses`, `Centralizer`, `FrattiniSubgroup`, `PrefrattiniSubgroup`, `MaximalSubgroups` and related operations, `HallSystem` and related operations, `MinimalGeneratingSet`, `Centre`, `Intersection`, `AutomorphismGroup`, `IrreducibleModules`.

43.17 Conjugacy Classes in Solvable Groups

There are a variety of algorithms to compute conjugacy classes and centralizers in solvable groups via epimorphic images ([FN79], [MN89], [The93]). Usually these are only invoked as methods, but it is possible to access the algorithm directly.

@The syntax of this function may change in a future rewrite!@

1 ▶ `ClassesSolvableGroup(` G`,` *mode* `[,` *opt*`] )` F

computes conjugacy classes and centralizers in solvable groups. G is the acting group. *mode* indicates the type of the calculation:

0 Conjugacy classes

4 Conjugacy test for the two elements in *opt*`.candidates`

In mode 0 the function returns a list of records containing components *representative* and *centralizer*. In mode *4* it returns a conjugating element.

The optional record *opt* may contain the following components that will affect the algorithms behaviour:

`pcgs`
: is a pcgs that will be used for the calculation. The attribute `EANormalSeriesByPcgs` must return an appropriate series of normal subgroups with elementary abelian factors among them. The algorithm will step down this series. In the case of the calculation of rational classes, it must be a pcgs refining a central series.

`candidates`
: is a list of elements for which canonical representatives are to be computed or for which a conjugacy test is performed. They must be given in mode 4. In mode 0 a list of classes corresponding to *candidates* is returned (which may contain duplicates). The *representatives* chosen are canonical with respect to *pcgs*. The records returned also contain components *operator* such that (*candidate* ^ *operator*) =*representative*.

`consider`

is a function *consider*(*fhome*,*rep*,*cenp*,K,L). Here *fhome* is a home pcgs for the factor group F in which the calculation currently takes place, *rep* is an element of the factor and *cenp* is a pcgs for the centralizer of *rep* modulo K. In mode 0, when lifting from F/K to F/L (note: for efficiency reasons, F can be different from G or L might be not trivial) this function is called before performing the actual lifting and only those representatives for which it returns `true` are passed to the next level. This permits for example the calculation of only those classes with small centralizers or classes of restricted orders.

2 ▶ `CentralizerSizeLimitConsiderFunction( sz )` F

returns a function (of the form func(*fhome*,*rep*,*cen*,K,L))that can be used in `ClassesSolvableGroup` as the *consider* component of the options record. It will restrict the lifting to those classes, for which the size of the centralizer (in the factor) is at most *sz*.

See also `SubgroupsSolvableGroup` (37.21.3).

44 Pc Groups

PcGroups are polycyclic groups that use the polycyclic presentation for element arithmetic. This presentation gives them a "natural" pcgs, the `FamilyPcgs` (see 44.1.1) with respect to which pcgs operations as described in chapter 43 are particularly efficient.

Let G be a polycyclic group with pcgs $P = (g_1, \ldots, g_n)$ and corresponding relative orders $(r_1, \ldots, r_n)$. Recall that the r_i are positive integers or infinity and let I be the set of indices i with r_i a positive integer. Then G has a finite presentation on the generators $g_1, \ldots, g_n$ with relations of the following form.

$$\begin{aligned} g_i^{r_i} &= g_{i+1}^{a(i,i,i+1)} \cdots g_n^{a(i,i,n)} \\ &\quad \text{for } 1 \leq i \leq n \text{ and } i \in I \\ g_i^{-1} g_j g_i &= g_{i+1}^{a(i,j,i+1)} \cdots g_n^{a(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \end{aligned}$$

For infinite groups we need additionally

$$\begin{aligned} g_i^{-1} g_j^{-1} g_i &= g_{i+1}^{b(i,j,i+1)} \cdots g_n^{b(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \text{ and } j \notin I \\ g_i g_j g_i^{-1} &= g_{i+1}^{c(i,j,i+1)} \cdots g_n^{c(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \text{ and } i \notin I \\ g_i g_j^{-1} g_i^{-1} &= g_{i+1}^{d(i,j,i+1)} \cdots g_n^{d(i,j,n)} \\ &\quad \text{for } 1 \leq i < j \leq n \text{ and } i, j, \notin I \end{aligned}$$

Here the right hand sides are assumed to be words in normal form; that is, for $k \in I$ we have for all exponents $0 \leq a(i,j,k), b(i,j,k), c(i,j,k), d(i,j,k) < r_k$.

A finite presentation of this type is called a **power-conjugate presentation** and a **pc group** is a polycyclic group defined by a power-conjugate presentation. Instead of conjugates we could just as well work with commutators and then the presentation would be called a **power-commutator** presentation. Both types of presentation are abbreviated as **pc presentation**. Note that a pc presentation is a rewriting system.

Clearly, whenever a group G with pcgs P is given, then we can write down the corresponding pc presentation. On the other hand, one may just write down a presentation on n abstract generators $g_1, \ldots, g_n$ with relations of the above form and define a group H by this. Then the subgroups $C_i = \langle g_i, \ldots, g_n \rangle$ of H form a subnormal series whose factors are cyclic or trivial. In the case that all factors are non-trivial, we say that the pc presentation of H is **confluent**. Note that GAP 4 can only work correctly with pc groups defined by a confluent pc presentation.

At the current level of implementation GAP can only deal with finite pc groups. This will be extended in near future.

Algorithms for pc groups use the methods for polycyclic groups described in chapter 43.

44.1 The family pcgs

Clearly, the generators of a power-conjugate presentation of a pc group G form a pcgs of the pc group. This pcgs is called the **family pcgs**.

1 ► `FamilyPcgs(` *grp* `)` A

2 ► `IsFamilyPcgs(` *pcgs* `)` P

3 ► `InducedPcgsWrtFamilyPcgs(` *grp* `)` A

4 ► `IsParentPcgsFamilyPcgs(` *pcgs* `)` P

This property indicates that the pcgs *pcgs* is induced with respect to a family pcgs.

In GAP 3 the family pcgs had been the only pcgs allowed for a pc group. Note that this has changed in GAP 4 where a pc group may have several independent polycyclic generating sequences.

However, the elementary operations for a non-family pcgs may not be as efficient as the elementary operations for the family pcgs.

This can have a significant influence on the performance of algorithms for polycyclic groups. Many algorithms require a pcgs that corresponds to an elementary abelian series (see 43.11.2) or even a special pcgs (see 43.13). If the family pcgs has the required properties, it will be used for these purposes, if not GAP has to work with respect to a new pcgs which is **not** the family pcgs and thus takes longer for elementary calculations like `ExponentsOfPcElement`.

Therefore, if the family pcgs chosen for arithmetic is not of importance it might be worth to **change** to another, nicer, pcgs to speed up calculations. This can be achieved, for example, by using the `Range` of the isomorphism obtained by `IsomorphismSpecialPcGroup` (see 44.5.3).

44.2 Elements of pc groups

The elements of a pc group G are always represented as words in normal form with respect to the family pcgs of G. Thus it is straightforward to compare elements of pc group, since this boils down to a mere comparison of exponent vectors with respect to the family pcgs. In particular, the word problem is efficiently solvable in pc groups.

1 ► *pcword* `=` *pcword*
► *pcword* `<` *pcword*

However, multiplication and inversion of elements in pc groups is not as straightforward as in arbitrary finitely presented groups where a simple concatenation or reversion of the corresponding words is sufficient (but one cannot solve the word problem).

To multiply to elements in a pc group, we first concatenate the corresponding words and then use an algorithm called `collection` to transform the new word into a word in normal form.

```
gap> g := FamilyPcgs( SmallGroup( 24, 12 ) );
Pcgs([ f1, f2, f3, f4 ])
gap> g[4] * g[1];
f1*f3
gap> (g[2] * g[3])^-1;
f2^2*f3*f4
```

44.3 Pc groups versus fp groups

In theory pc groups are finitely presented groups. In practice the arithmetic in pc groups is different from the arithmetic in fp groups. Thus for technical reasons the pc groups in GAP do not form a subcategory of the fp groups and hence the methods for fp groups cannot be applied to pc groups in general.

1 ▶ `IsPcGroup( G )` C

tests whether G is a pc group.

```
gap> G := SmallGroup( 24, 12 );
<pc group of size 24 with 4 generators>
gap> IsPcGroup( G );
true
gap> IsFpGroup( G );
false
```

Note that it is possible to convert a pc group to a fp group in GAP. The following function computes the power-commutator presentation defined by *pcgs*. The string *str* can be used to give a name to the generators of the fp group.

2 ▶ `IsomorphismFpGroupByPcgs( pcgs, str )`

```
gap> p := FamilyPcgs( SmallGroup( 24, 12 ) );
Pcgs([ f1, f2, f3, f4 ])
gap> iso := IsomorphismFpGroupByPcgs( p, "g" );
[ f1, f2, f3, f4 ] -> [ g1, g2, g3, g4 ]
gap> F := Image( iso );
<fp group of size 24 on the generators [ g1, g2, g3, g4 ]>
gap> RelatorsOfFpGroup( F );
[ g1^2, g2^-1*g1^-1*g2*g1*g2^-1, g3^-1*g1^-1*g3*g1*g4^-1*g3^-1,
  g4^-1*g1^-1*g4*g1*g4^-1*g3^-1, g2^3, g3^-1*g2^-1*g3*g2*g4^-1*g3^-1,
  g4^-1*g2^-1*g4*g2*g3^-1, g3^2, g4^-1*g3^-1*g4*g3, g4^2 ]
```

44.4 Constructing Pc Groups

If necessary, you can supply GAP with a pc presentation by hand. (Although this is the most tedious way to input a pc group.) Note that the pc presentation has to be confluent in order to work with the pc group in GAP.

(If you have already a suitable pcgs in another representation, use `PcGroupWithPcgs`, see below 44.5.1.)

One way is to define a finitely presented group with a pc presentation in GAP and then convert this presentation into a pc group. Note that this does not work for arbitrary presentations of polycyclic groups, see Chapter 45.13 for further information.

For performance reasons it is beneficial to enforce a "syllable" representation in the free group (see 35.6).

1 ▶ `PcGroupFpGroup( G )` F

creates a PcGroup P from an FpGroup (see Chapter 45) G whose presentation is polycyclic. The resulting group P has generators corresponding to the generators of G. They are printed in the same way as generators of G, but they lie in a different family. If the pc presentation of G is not confluent, an error message occurs.

```
gap> F := FreeGroup(IsSyllableWordsFamily,"a","b","c","d");;
gap> a := F.1;; b := F.2;; c := F.3;; d := F.4;;
gap> rels := [a^2, b^3, c^2, d^2, Comm(b,a)/b, Comm(c,a)/d, Comm(d,a),
>             Comm(c,b)/(c*d), Comm(d,b)/c, Comm(d,c)];
[ a^2, b^3, c^2, d^2, b^-1*a^-1*b*a*b^-1, c^-1*a^-1*c*a*d^-1, d^-1*a^-1*d*a,
  c^-1*b^-1*c*b*d^-1*c^-1, d^-1*b^-1*d*b*c^-1, d^-1*c^-1*d*c ]
gap> G := F / rels;
<fp group on the generators [ a, b, c, d ]>
gap> H := PcGroupFpGroup( G );
<pc group of size 24 with 4 generators>
```

Equivalently to the above method one can initiate a collector of a pc group by hand and use it to define a pc group. In GAP there are different collectors for different collecting strategies; at the moment, there are two collectors to choose from: the single collector for finite pc groups and the combinatorial collector for finite p-groups. See [Sim94] for further information on collecting strategies.

A collector is initiated by underlying free group to the pc presented group and the relative orders of the pc series. Then one adds the right hand sides of the power and the commutator or conjugate relations one by one. Note that omitted relators are assumed to be trivial.

2 ▶ `SingleCollector( `*fgrp*`, `*relorders*` )`
▶ `CombinatorialCollector( `*fgrp*`, `*relorders*` )`

Then the right hand sides of the pc presentation have to be declared. Let $f_1, \ldots, f_n$ be the generators of the underlying free group *fgrp*.

A combinatorial collector can only be set up for a finite p-group. Here, the relative orders *relorders* must all be equal and a prime.

3 ▶ `SetConjugate( `*coll*`, `i`, `j`, `w` )`

set the conjugate $f_j^{f_i}$ to equal w where w is a word in $f_{i+1}, \ldots, f_n$ and $i < j$.

4 ▶ `SetCommutator( `*coll*`, `j`, `i`, `w` )`

set the commutator of f_j and f_i to equal w where w is a word in $f_{i+1}, \ldots, f_n$ and $i < j$.

5 ▶ `SetPower( `*coll*`, `i`, `w` )`

set the power $f_i^{r_i}$ to equal w where w is a word in $f_{i+1}, \ldots, f_n$.

Finally, the collector has to be converted to a group.

6 ▶ `GroupByRws( `*coll* `)`
▶ `GroupByRwsNC( `*coll* `)`

creates a group from a rewriting system. In the first version it is checked whether the rewriting system is confluent, in the second version this is assumed to be true.

7 ▶ `IsConfluent( `G` )`

checks whether the pc group G has been build from a collector with a confluent power-commutator presentation.

```
gap> F := FreeGroup(IsSyllableWordsFamily, 2 );;
gap> coll1 := SingleCollector( F, [2,3] );
<<single collector, 8 Bits>>
gap> SetConjugate( coll1, 2, 1, F.2 );
gap> SetPower( coll1, 1, F.2 );
gap> G1 := GroupByRws( coll1 );
<pc group of size 6 with 2 generators>
gap> IsConfluent(G1);
true
gap> IsAbelian(G1);
true

gap> coll2 := SingleCollector( F, [2,3] );
<<single collector, 8 Bits>>
gap> SetConjugate( coll2, 2, 1, F.2^2 );
gap> G2 := GroupByRws( coll2 );
<pc group of size 6 with 2 generators>
gap> IsAbelian(G2);
false
```

With the above methods a pc group with arbitrary defining pcgs can be constructed. However, for almost all applications within GAP we need to have a pc group whose defining pcgs is a prime order pcgs. Hence the following functions are useful.

8 ▶ `IsomorphismRefinedPcGroup( G )` A

returns an isomorphism from G onto an isomorphic PC group whose family pcgs is a prime order pcgs.

9 ▶ `RefinedPcGroup( G )` A

returns the range of `IsomorphismRefinedPcGroup(`G`)`.

44.5 Computing Pc Groups

Another possibility to get a pc group in GAP is to convert a polycyclic group given by some other representation to a pc group. For finitely presented groups there are various quotient methods available. For all other types of groups one can use the following functions.

1 ▶ `PcGroupWithPcgs( `*mpcgs*` )` A

creates a new Pc group G whose family pcgs is isomorphic to the (modulo) pcgs *mpcgs*.

```
gap> G := Group( (1,2,3), (3,4,1) );;
gap> PcGroupWithPcgs( Pcgs(G) );
<pc group of size 12 with 3 generators>
```

If a pcgs is only given by a list of pc elements, `PcgsByPcSequence` (see 43.3.1) can be used:

```
gap> G:=Group((1,2,3,4),(1,2));;
gap> p:=PcgsByPcSequence(FamilyObj(One(G)),
> [ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ]);
Pcgs([ (3,4), (2,4,3), (1,4)(2,3), (1,3)(2,4) ])
gap> PcGroupWithPcgs(p);
<pc group of size 24 with 4 generators>

gap> G := SymmetricGroup( 5 );
```

```
Sym( [ 1 .. 5 ] )
gap> H := Subgroup( G, [(1,2,3,4,5), (3,4,5)] );
Group([ (1,2,3,4,5), (3,4,5) ])
gap> modu := ModuloPcgs( G, H );
Pcgs([ (4,5) ])
gap> PcGroupWithPcgs(modu);
<pc group of size 2 with 1 generators>
```

2 ▶ `IsomorphismPcGroup( G )` A

returns an isomorphism from G onto an isomorphic PC group. The series chosen for this PC representation depends on the method chosen. G must be a polycyclic group of any kind, for example a solvable permutation group.

```
gap> G := Group( (1,2,3), (3,4,1) );;
gap> iso := IsomorphismPcGroup( G );
Pcgs([ (2,4,3), (1,2)(3,4), (1,3)(2,4) ]) -> [ f1, f2, f3 ]
gap> H := Image( iso );
Group([ f1, f2, f3 ])
```

3 ▶ `IsomorphismSpecialPcGroup( G )` A

returns an isomorphism from G onto an isomorphic PC group whose family pcgs is a special pcgs. (This can be beneficial to the runtime of calculations.) G may be a polycyclic group of any kind, for example a solvable permutation group.

44.6 Saving a Pc Group

As printing a polycyclic group does not display the presentation, one cannot simply print a pc group to a file to save it. For this purpose we need the following function.

1 ▶ `GapInputPcGroup( grp, string )` F

```
gap> G := SmallGroup( 24, 12 );
<pc group of size 24 with 4 generators>
gap> PrintTo( "save", GapInputPcGroup( G, "H" ) );
gap> Read( "save" );
#I A group of order 24 has been defined.
#I It is called H
gap> H = G;
false
gap> IdSmallGroup( H ) = IdSmallGroup( G );
true
```

44.7 Operations for Pc Groups

All the operations described in Chapters 37 and 43 apply to a pc group. Nearly all methods for pc groups are methods for groups with pcgs as described in Chapter 43. The only method with is special for pc groups is a method to compute intersections of subgroups, since here a pcgs of a parent group is needed and this can only by guaranteed within pc groups.

44.8 2-Cohomology and Extensions

One of the most interesting applications of pc groups is the possibility to compute with extensions of these groups by elementary abelian groups; that is, H is an extension of G by M, if there exists a normal subgroup N in H which is isomorphic to M such that H/N is isomorphic to G.

Pc groups are particularly suited for such applications, since the 2-cohomology can be computed efficiently for such groups and, moreover, extensions of pc groups by elementary abelian groups can be represented as pc groups again.

To define the elementary abelian group M together with an action of G on M we consider M as a meataxe module for G over a finite field (section 69.13.1 describes functions that can be used to obtain certain modules). For further information on meataxe modules see Chapter 67. Note that the matrices defining the module must correspond to the pcgs of the group G.

1 ▶ `TwoCoboundaries( G, M )` O

returns the group of 2-coboundaries of a pc group G by the G-module M. The generators of M must correspond to Pcgs(G). The group of coboundaries is given as vector space over the field underlying M.

2 ▶ `TwoCocycles( G, M )` O

returns the group of 2-cocycles of a pc group G by the G-module M. The generators of M must correspond to Pcgs(G). The group of cocycles is given as vector space over the field underlying M.

3 ▶ `TwoCohomology( G, M )` O

returns a record defining the second cohomology group as factor space of the space of cocycles by the space of coboundaries. G must be a pc group and the generators of M must correspond to the pcgs of G.

```
gap> G := SmallGroup( 4, 2 );
<pc group of size 4 with 2 generators>
gap> mats := List( Pcgs( G ), x -> IdentityMat( 1, GF(2) ) );
[ [ <a GF2 vector of length 1> ], [ <a GF2 vector of length 1> ] ]
gap> M := GModuleByMats( mats, GF(2) );
rec( field := GF(2), isMTXModule := true, dimension := 1,
  generators := [ <an immutable 1x1 matrix over GF2>,
      <an immutable 1x1 matrix over GF2> ] )
gap> TwoCoboundaries( G, M );
[  ]
gap> TwoCocycles( G, M );
[ [ Z(2)^0, 0*Z(2), 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ],
  [ 0*Z(2), 0*Z(2), Z(2)^0 ] ]
gap> cc := TwoCohomology( G, M );;
gap> cc.cohom;
<linear mapping by matrix, <vector space of dimension 3 over GF(2)> -> ( GF(
2)^3 )>
```

4 ▶ `Extensions( G, M )` O

returns all extensions of G by the G-module M up to equivalence as pc groups.

5 ▶ `Extension( G, M, c )` O
▶ `ExtensionNC( G, M, c )` O

returns the extension of G by the G-module M via the cocycle c as pc groups. The `NC` version does not check the resulting group for consistence.

6 ▶ `SplitExtension( G, M )`

returns the split extension of G by the G-module M.

7 ▶ `ModuleOfExtension( E )`

returns the module of an extension E of G by M. This is the normal subgroup of E which corresponds to M.

```
gap> G := SmallGroup( 4, 2 );;
gap> mats := List( Pcgs( G ), x -> IdentityMat( 1, GF(2) ) );;
gap> M := GModuleByMats( mats, GF(2) );;
gap> co := TwoCocycles( G, M );;
gap> Extension( G, M, co[2] );
<pc group of size 8 with 3 generators>
gap> SplitExtension( G, M );
<pc group of size 8 with 3 generators>
gap> Extensions( G, M );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
gap> List(last, IdSmallGroup);
[ [ 8, 5 ], [ 8, 2 ], [ 8, 3 ], [ 8, 3 ], [ 8, 2 ], [ 8, 2 ], [ 8, 3 ],
  [ 8, 4 ] ]
```

Note that the extensions returned by `Extensions` are computed up to equivalence, but not up to isomorphism.

There exists an action of the subgroup of **compatible pairs** in $Aut(G) \times Aut(M)$ which acts on the second cohomology group. 2-cocycles which lie in the same orbit under this action define isomorphic extensions of G. However, there may be isomorphic extensions of G corresponding to cocycles in different orbits.

8 ▶ `CompatiblePairs( G, M [, D] )` F

returns the group of compatible pairs of the group G with the G-module M as subgroup of the direct product of *Aut(G)* x *Aut(M)*. Here *Aut(M)* is considered as subgroup of a general linear group. The optional argument D should be a subgroup of *Aut(G)* x *Aut(M)*. If it is given, then only the compatible pairs in D are computed.

9 ▶ `ExtensionRepresentatives( G, M, P )` O

returns all extensions of G by the G-module M up to equivalence under action of P where P has to be a subgroup of the group of compatible pairs of G with M.

```
gap> G := SmallGroup( 4, 2 );;
gap> mats := List( Pcgs( G ), x -> IdentityMat( 1, GF(2) ) );;
gap> M := GModuleByMats( mats, GF(2) );;
gap> A := AutomorphismGroup( G );;
gap> B := GL( 1, 2 );;
gap> D := DirectProduct( A, B );
<group with 6 generators>
gap> P := CompatiblePairs( G, M, D );
```

```
<group of size 6 with 2 generators>
gap> ExtensionRepresentatives( G, M, P );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
gap> Extensions( G, M );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
```

See also the forthcoming GAP package on `Group Construction Methods`.

Finally we note that for the computation of split extensions it is not necessary that M must correspond to an elementary abelian group. Here it is possible to construct split extensions of arbitrary pc groups.

10 ▶ `SplitExtensions(` *G*, *aut*, *N* `)`

returns the split extensions of the pc group G by the pc group N. *aut* should be a homomorphism from G into $Aut(N)$.

In the following example we construct the holomorph of Q_8 as split extension of Q_8 by S_4.

```
gap> N := SmallGroup( 8, 4 );
<pc group of size 8 with 3 generators>
gap> IsAbelian( N );
false
gap> A := AutomorphismGroup( N );
<group of size 24 with 4 generators>
gap> iso := IsomorphismPcGroup( A );
CompositionMapping( Pcgs([ (2,6,5,3), (1,3,5)(2,4,6), (2,5)(3,6), (1,4)(3,6)
 ]) -> [ f1, f2, f3, f4 ], <action isomorphism> )
gap> H := Image( iso );
Group([ f1, f2, f3, f4 ])
gap> G := Subgroup( H, Pcgs(H){[1,2]} );
Group([ f1, f2 ])
gap> inv := InverseGeneralMapping( iso );
[ f1*f2, f2^2*f3, f4, f3 ] -> [ Pcgs([ f1, f2, f3 ]) -> [ f1*f2, f2, f3 ],
  Pcgs([ f1, f2, f3 ]) -> [ f2, f1*f2, f3 ],
  Pcgs([ f1, f2, f3 ]) -> [ f1*f3, f2, f3 ],
  Pcgs([ f1, f2, f3 ]) -> [ f1, f2*f3, f3 ] ]
gap> K := SplitExtension( G, inv, N );
<pc group of size 192 with 7 generators>
```

44.9 Coding a Pc Presentation

If one wants to store a large number of pc groups, then it can be useful to store them in a compressed format, since pc presentations can be space consuming. Here we introduce a method to code and decode pc presentations by integers. To decode a given code the size of the underlying pc group is needed as well. For the full definition and the coding and decoding procedures see [BE99a]. This method is used with the small groups library, see Section 48.7.

1 ▶ `CodePcgs( pcgs )` F

returns the code corresponding to *pcgs*.

2 ▶ `CodePcGroup( G )` F

returns the code for a pcgs of G.

3 ▶ `PcGroupCode( code, size )` F

returns a pc group of size *size* corresponding to *code*. The argument *code* must be a valid code for a pcgs, otherwise anything may happen. Valid codes are usually obtained by one of the functions `CodePcgs` or `CodePcGroup`.

4 ▶ `PcGroupCodeRec( rec )` F

Here *rec* needs to have entries .code and .order. Then PcGroupCode returns a pc group of size .order corresponding to .code.

```
gap> G := SmallGroup( 24, 12 );;
gap> p := Pcgs( G );;
gap> code := CodePcgs( p );
5790338948
gap> H := PcGroupCode( code, 24 );
<pc group of size 24 with 4 generators>
gap> map := GroupHomomorphismByImages( G, H, p, FamilyPcgs(H) );
Pcgs([ f1, f2, f3, f4 ]) -> Pcgs([ f1, f2, f3, f4 ])
gap> IsBijective(map);
true
```

44.10 Random Isomorphism Testing

The generic isomorphism test for groups may be applied to pc groups as well. However, this test is often quite time consuming. Here we describe another method to test isomorphism by a probabilistic approach.

This method takes a list of groups and a non-negative integer as input. The output is a sublist of the input list where only isomorphic copies have been removed. The integer gives a certain amount of control over the probability to detect all isomorphisms. If it is 0, then nothing will be done at all. The larger the integer is, the larger is the probability of finding all isomorphisms. However, due to the underlying method we can not guarantee that the algorithm finds all isomorphisms, no matter how large n is.

1 ▶ `RandomIsomorphismTest( list, n )` F

list must be a list of code records of pc groups and n a non-negative integer. Returns a sublist of *list* where isomorphic copies detected by the probabilistic test have been removed.

45 Finitely Presented Groups

A **finitely presented group** (in short: FpGroup) is a group generated by a finite set of **abstract generators** subject to a finite set of **relations** that these generators satisfy. Every finite group can be represented as a finitely presented group, though in almost all cases it is computationally much more efficient to work in another representation (even the regular permutation representation).

Finitely presented groups are obtained by factoring a free group by a set of relators. Their elements know about this presentation and compare accordingly.

So to create a finitely presented group you first have to generate a free group (see 35.2.1 for details). Then a list of relators is constructed as words in the generators of the free group and is factored out to obtain the finitely presented group. Its generators **are** the images of the free generators. So for example to create the group

$$\langle a, b \mid a^2, b^3, (a \cdot b)^5 \rangle$$

you can use the following commands:

```
gap> f := FreeGroup( "a", "b" );;
gap> g := f / [ f.1^2, f.2^3, (f.1*f.2)^5 ];
<fp group on the generators [ a, b ]>
```

Note that you cannot call the generators by their names. These names are not variables, but just display figures. So, if you want to access the generators by their names, you first have to introduce the respective variables and to assign the generators to them.

```
gap> GeneratorsOfGroup( g );
[ a, b ]
gap> a;
Variable: 'a' must have a value

gap> a := g.1;; b := g.2;; # assign variables
gap> GeneratorsOfGroup( g );
[ a, b ]
gap> a in f;
false
gap> a in g;
true
```

To relieve you of the tedium of typing the above assignments, **when working interactively**, there is the function `AssignGeneratorVariables` (see 35.2.5).

Note that the generators of the free group are different from the generators of the FpGroup (even though they are displayed by the same names). That means that words in the generators of the free group are not elements of the finitely presented group. Vice versa elements of the FpGroup are not words.

```
gap> a*b = b*a;
false
gap> (b^2*a*b)^2 = a^0;
true
```

Such calculations comparing elements of an FpGroup may run into problems: There exist finitely presented groups for which no algorithm exists (it is known that no such algorithm can exist) that will tell for two arbitrary words in the generators whether the corresponding elements in the FpGroup are equal.

Therefore the methods used by GAP to compute in finitely presented groups may run into warning errors, run out of memory or run forever. If the FpGroup is (by theory) known to be finite the algorithms are guaranteed to terminate (if there is sufficient memory available), but the time needed for the calculation cannot be bounded a priori. See 45.5 and 45.15.

```
gap> (b^2*a*b)^2;
b^2*a*b^3*a*b
gap> a^0;
<identity ...>
```

A consequence of our convention is that elements of finitely presented groups are not printed in a unique way. See also `SetReducedMultiplication`.

1 ▶ `IsSubgroupFpGroup(` *H* `)` C

returns `true` if *H* is a finitely presented group or a subgroup of a finitely presented group.

2 ▶ `IsFpGroup(` *G* `)` F

is a synonym for `IsSubgroupFpGroup(`*G*`)` and `IsGroupOfFamily(`*G*`)`.

Free groups are a special case of finitely presented groups, namely finitely presented groups with no relators.

Another special case are groups given by polycyclic presentations. GAP uses a special representation for these groups which is created in a different way. See chapter 44 for details.

3 ▶ `InfoFpGroup` V

The info class for functions dealing with finitely presented groups is `InfoFpGroup`.

45.1 Creating Finitely Presented Groups

1 ▶ *F*`/`*rels*

creates a finitely presented group given by the presentation $\langle gens \mid rels \rangle$ where *gens* are the generators of the free group *F*. Note that relations are entered as **relators**, i.e., as words in the generators of the free group. To enter an equation use the quotient operator, i.e., for the relation $a^b = ab$ one has to enter `a^b/(a*b)`.

```
gap> f := FreeGroup( 3 );;
gap> f / [ f.1^4, f.2^3, f.3^5, f.1*f.2*f.3 ];
<fp group on the generators [ f1, f2, f3 ]>
```

2 ▶ `FactorGroupFpGroupByRels(` *G*`,` *elts* `)` F

returns the factor group G/N of *G* by the normal closure N of *elts* where *elts* is expected to be a list of elements of *G*.

45.2 Comparison of Elements of Finitely Presented Groups

1 ▶ a = b

Two elements of a finitely presented group are equal if they are equal in this group. Nevertheless they may be represented as different words in the generators. Because of the fundamental problems mentioned in the introduction to this chapter such a test may take very long and cannot be guaranteed to finish.

The method employed by GAP for such an equality test use the underlying finitely presented group. First (unless this group is known to be infinite) GAP tries to find a faithful permutation representation by a bounded Todd-Coxeter. If this fails, a Knuth-Bendix (see 51.5) is attempted and the words are compared via their normal form.

If only elements in a subgroup are to be tested for equality it thus can be useful to translate the problem in a new finitely presented group by rewriting (see 45.10.1);

The equality test of elements underlies many "basic" calculations, such as the order of an element, and the same type of problems can arise there. In some cases, working with rewriting systems can still help to solve the problem. The "kbmag' package provides such functionality, see the package manual for further details.

2 ▶ a < b

Problems get even worse when trying to compute a total ordering on the elements of a finitely presented group. As any ordering that is guaranteed to be reproducible in different runs of GAP or even with different groups given by syntactically equal presentations would be prohibitively expensive to implement, the ordering of elements is depending on a method chosen by GAP and not guaranteed to stay the same when repeating the construction of an FpGroup. The only guarantee given for the < ordering for such elements is that it will stay the same for one family during its lifetime. The attribute `FpElmComparisonMethod` is used to obtain a comparison function for a family of FpGroup elements.

3 ▶ `FpElmComparisonMethod(` *fam* `)` A

If *fam* is the elements family of a finitely presented group this attribute returns a function `smaller(`*left*`,` *right*`)` that will be used to compare elements in *fam*.

4 ▶ `SetReducedMultiplication(` *f* `)` F
▶ `SetReducedMultiplication(` *e* `)` F
▶ `SetReducedMultiplication(` *fam* `)` F

for an fp group f, an element e of it or the family *fam* of its elements this function will force immediate reduction when multiplying, keeping words short at extra cost per multiplication.

45.3 Preimages in the Free Group

1 ▶ `FreeGroupOfFpGroup(` *G* `)` A

returns the underlying free group for the finitely presented group G. This is the group generated by the free generators provided by `FreeGeneratorsOfFpGroup(`G`)`.

2 ▶ `FreeGeneratorsOfFpGroup(` *G* `)` A
▶ `FreeGeneratorsOfWholeGroup(` *U* `)` O

`FreeGeneratorsOfFpGroup` returns the underlying free generators corresponding to the generators of the finitely presented group G which must be a full fp group.

`FreeGeneratorsOfWholeGroup` also works for subgroups of an fp group and returns the free generators of the full group that defines the family.

3 ▶ `RelatorsOfFpGroup(` *G* `)` A

returns the relators of the finitely presented group G as words in the free generators provided by `FreeGeneratorsOfFpGroup(`G`)`.

```
gap> f := FreeGroup( "a", "b" );;
gap> g := f / [ f.1^5, f.2^2, f.1^f.2*f.1 ];
<fp group on the generators [ a, b ]>
gap> Size( g );
10
gap> FreeGroupOfFpGroup( g ) = f;
true
gap> FreeGeneratorsOfFpGroup( g );
[ a, b ]
gap> RelatorsOfFpGroup( g );
[ a^5, b^2, b^-1*a*b*a ]
```

Elements of a finitely presented group are not words, but are represented using a word from the free group as representative. The following two commands obtain this representative, respectively create an element in the finitely presented group.

4 ► `UnderlyingElement( elm )` O

Let *elm* be an element of a group whose elements are represented as words with further properties. Then `UnderlyingElement` returns the word from the free group that is used as a representative for *elm*.

```
gap> w := g.1*g.2;
a*b
gap> IsWord( w );
false
gap> ue := UnderlyingElement( w );
a*b
gap> IsWord( ue );
true
```

5 ► `ElementOfFpGroup( fam, word )` O

If *fam* is the elements family of a finitely presented group and *word* is a word in the free generators underlying this finitely presented group, this operation creates the element with the representative *word* in the free group.

```
gap> ge := ElementOfFpGroup( FamilyObj( g.1 ), f.1*f.2 );
a*b
gap> ge in f;
false
gap> ge in g;
true
```

45.4 Operations for Finitely Presented Groups

Finitely presented groups are groups and so all operations for groups should be applicable to them (though not necessarily efficient methods are available.) Most methods for finitely presented groups rely on coset enumeration. See 45.5 for details.

The command `IsomorphismPermGroup` can be used to obtain a faithful permutation representation, if such a representation of small degree exists. (Otherwise it might run very long or fail.)

```
gap> f := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> g := f / [ f.1^2, f.2^3, (f.1*f.2)^5 ];
<fp group on the generators [ a, b ]>
gap> h := IsomorphismPermGroup( g );
[ a, b ] -> [ (1,2)(4,5), (2,3,4) ]
gap> u:=Subgroup(g,[g.1*g.2]);;rt:=RightTransversal(g,u);
RightTransversal(<fp group of size 60 on the generators [ a, b ]>,Group(
[ a*b ]))
gap> Image(ActionHomomorphism(g,rt,OnRight));
Group([ (1,2)(3,4)(5,7)(6,8)(9,10)(11,12), (1,3,2)(4,5,6)(7,8,9)(10,11,12) ])
```

45.5 Coset Tables and Coset Enumeration

Coset enumeration (see [Neu82] for an explanation) is one of the fundamental tools for the examination of finitely presented groups. This section describes GAP functions that can be used to invoke a coset enumeration.

Note that in addition to the built-in coset enumerator there is the GAP package ACE. Moreover, GAP provides an interactive Todd-Coxeter in the GAP package ITC which is based on the XGAP package.

1 ▶ `CosetTable( `*G*`, `*H*` )` O

returns the coset table of the finitely presented group G on the cosets of the subgroup H.

Basically a coset table is the permutation representation of the finitely presented group on the cosets of a subgroup (which need not be faithful if the subgroup has a nontrivial core). Most of the set theoretic and group functions use the regular representation of G, i.e., the coset table of G over the trivial subgroup.

The coset table is returned as a list of lists. For each generator of G and its inverse the table contains a generator list. A generator list is simply a list of integers. If l is the generator list for the generator g and if l`[`i`] = `j then generator g takes the coset i to the coset j by multiplication from the right. Thus the permutation representation of G on the cosets of H is obtained by applying `PermList` to each generator list (see 40.4.2).

The coset table is standard (see below).

For finitely presented groups, a coset table is computed by a Todd-Coxeter coset enumeration. Note that you may influence the performance of that enumeration by changing the values of the global variables `CosetTableDefaultLimit` and `CosetTableDefaultMaxLimit` described below and that the options described under `CosetTableFromGensAndRels` are recognized.

```
gap> tab := CosetTable( g, Subgroup( g, [ g.1, g.2*g.1*g.2*g.1*g.2^-1 ] ) );
[ [ 1, 4, 5, 2, 3 ], [ 1, 4, 5, 2, 3 ], [ 2, 3, 1, 4, 5 ], [ 3, 1, 2, 4, 5 ] ]
gap> List( last, PermList );
[ (2,4)(3,5), (2,4)(3,5), (1,2,3), (1,3,2) ]
gap> PrintArray( TransposedMat( tab ) );
[ [  1,  1,  2,  3 ],
  [  4,  4,  3,  1 ],
  [  5,  5,  1,  2 ],
  [  2,  2,  4,  4 ],
  [  3,  3,  5,  5 ] ]
```

The last printout in the preceding example provides the coset table in the form in which it is usually used in hand calculations: The rows correspond to the cosets, the columns correspond to the generators and their inverses in the ordering $g_1, g_1^{-1}, g_2, g_2^{-1}$. (See section 45.6 for a description on the way the numbers are assigned.)

2 ▶ `TracedCosetFpGroup( tab, word, pt )` F

Traces the coset number *pt* under the word *word* through the coset table *tab*. (Note: *word* must be in the free group, use `UnderlyingElement` if in doubt.)

```
gap> TracedCosetFpGroup(tab,UnderlyingElement(g.1),2);
4
```

3 ▶ `FactorCosetAction( G, H )`
▶ `FactorCosetOperation( G, H )`

returns the action of G on the cosets of the subgroup H of G.

```
gap> u := Subgroup( g, [ g.1, g.1^g.2 ] );
Group([ a, b^-1*a*b ])
gap> FactorCosetAction( g, u );
[ a, b ] -> [ (2,4)(5,6), (1,2,3)(4,5,6) ]
```

4 ▶ `CosetTableBySubgroup( G, H )` O

returns a coset table for the action of G on the cosets of H. The columns of the table correspond to the `GeneratorsOfGroup(`G`)`.

5 ▶ `CosetTableFromGensAndRels( fgens, grels, fsgens )` F

is an internal function which is called by the functions `CosetTable`, `CosetTableInWholeGroup` and others. It is, in fact, the proper working horse that performs a Todd-Coxeter coset enumeration. *fgens* must be a set of free generators and *grels* a set of relators in these generators. *fsgens* are subgroup generators expressed as words in these generators. The function returns a coset table with respect to *fgens*.

`CosetTableFromGensAndRels` will call `TCENUM.CosetTableFromGensAndRels`. This makes it possible to replace the built-in coset enumerator with another one by assigning `TCENUM` to another record.

The library version which is used by default performs a standard Felsch strategy coset enumeration. You can call this function explicitly as `GAPTCENUM.CosetTableFromGensAndRels` even if other coset enumerators are installed.

The expected parameters are

fgens
: generators of the free group F

grels
: relators as words in F

fsgens
: subgroup generators as words in F.

`CosetTableFromGensAndRels` processes two options (see chapter 8):

`max`
: The limit of the number of cosets to be defined. If the enumeration does not finish with this number of cosets, an error is raised and the user is asked whether she wants to continue. The default value is the value given in the variable `CosetTableDefaultMaxLimit`. (Due to the algorithm the actual limit used can be a bit higher than the number given.)

`silent`
: if set to `true` the algorithm will not raise the error mentioned under option `max` but silently return `fail`. This can be useful if an enumeration is only wanted unless it becomes too big.

6 ► `CosetTableDefaultMaxLimit` V

is the default limit for the number of cosets allowed in a coset enumeration.

A coset enumeration will not finish if the subgroup does not have finite index, and even if it has it may take many more intermediate cosets than the actual index of the subgroup is. To avoid a coset enumeration "running away" therefore GAP has a "safety stop" built in. This is controlled by the global variable `CosetTableDefaultMaxLimit`.

If this number of cosets is reached, GAP will issue an error message and prompt the user to either continue the calculation or to stop it. The default value is 256000.

See also the description of the options to `CosetTableFromGensAndRels`.

```
gap> f := FreeGroup( "a", "b" );;
gap> u := Subgroup( f, [ f.2 ] );
Group([ b ])
gap> Index( f, u );
Error, the coset enumeration has defined more than 256000 cosets
 called from
TCENUM.CosetTableFromGensAndRels( fgens, grels, fsgens ) called from
CosetTableFromGensAndRels( fgens, grels, fsgens ) called from
TryCosetTableInWholeGroup( H ) called from
CosetTableInWholeGroup( H ) called from
IndexInWholeGroup( H ) called from
...
Entering break read-eval-print loop ...
type 'return;' if you want to continue with a new limit of 512000 cosets,
type 'quit;' if you want to quit the coset enumeration,
type 'maxlimit := 0; return;' in order to continue without a limit
brk> quit;
```

At this point, a `break`-loop (see Section 6.3) has been entered. The line beginning `Error` tells you why this occurred. The next seven lines, occur if `OnBreak` has its default value of `Where` (see 6.3.3) and explains, in this case, how GAP came to be doing a coset enumeration. Then you are give a number of options of how to escape the `break`-loop: you can either continue the calculation with a larger number of permitted cosets, stop the calculation if you don't expect the enumeration to finish (like in the example above), or continue without a limit on the number of cosets. (Choosing the first option will, of course, land you back in a `break`-loop. Try it!)

Setting `CosetTableDefaultMaxLimit` (or the `max` option value, for any function that invokes a coset enumeration) to `infinity` (or to 0) will force all coset enumerations to continue until they either get a result or exhaust the whole available space. For example, each of

```
gap> CosetTableDefaultMaxLimit := 0;;
gap> Index( f, u );
```

or

```
gap> Index( f, u : max := 0 );
```

have essentially the same effect as choosing the third option (typing: `maxlimit := 0; return;`) at the `brk>` prompt above (instead of `quit;`).

7 ► `CosetTableDefaultLimit` V

is the default number of cosets with which any coset table is initialized before doing a coset enumeration.

The function performing this coset enumeration will automatically extend the table whenever necessary (as long as the number of cosets does not exceed the value of `CosetTableDefaultMaxLimit`), but this is an expensive operation. Thus, if you change the value of `CosetTableDefaultLimit`, you should set it to a number of cosets that you expect to be sufficient for your subsequent coset enumerations. On the other hand, if you make it too large, your job will unnecessarily waste a lot of space.

The default value of `CosetTableDefaultLimit` is 1000.

8 ► `MostFrequentGeneratorFpGroup( G )` F

is an internal function which is used in some applications of coset table methods. It returns the first of those generators of the given finitely presented group G which occur most frequently in the relators.

9 ► `IndicesInvolutaryGenerators( G )` A

returns the indices of those generators of the finitely presented group G which are known to be involutions. This knowledge is used by internal functions to improve the performance of coset enumerations.

45.6 Standardization of coset tables

For any two coset numbers i and j with $i < j$ the first occurrence of i in a coset table precedes the first occurrence of j with respect to the usual row-wise ordering of the table entries. Following the notation of Charles Sims' book on computation with finitely presented groups [Sim94] we call such a table a **standard coset table**.

The table entries which contain the first occurrences of the coset numbers $i > 1$ recursively provide for each i a representative of the corresponding coset in form of a unique word w_i in the generators and inverse generators of G. The first coset (which is H itself) can be represented by the empty word w_1. A coset table is standard if and only if the words w_1, w_2, ... are length-plus-lexicographic ordered (as defined in [Sim94]), for short: **lenlex**.

We would like to warn you that this standardization of coset tables is different from the concept that we have used in earlier GAP versions. That old concept ignored the columns that correspond to inverse generators and hence only considered words in the generators of G. We will call the old standard the **semilenlex** standard as it would also work in the case of semigroups where no inverses of the generators are known.

We have changed the convention from the semilenlex standard to the lenlex standard because the definiton of a standard coset table in Sims' book tends to become a kind of international standard. However, for reasons of upward compatibility GAP still offers the possibility to switch back to the old convention by just changing the value of the global variable `CosetTableStandard` from its default value `"lenlex"` to `"semilenlex"`. Then all implicit standardizations of coset tables will follow the old convention. Setting the value of `CosetTableStandard` back to `"lenlex"` again means switching back to the new convention.

1 ► `CosetTableStandard` V

specifies the definiton of a **standard coset table**. It is used whenever coset tables or augmented coset tables are created. Its value may be `"lenlex"` or `"semilenlex"`. If it is `"lenlex"` coset tables will be standardized using all their columns as defined in Charles Sims' book (this is the new default standard of GAP). If it is `"semilenlex"` they will be standardized using only their generator columns (this was the original GAP standard). The default value of `CosetTableStandard` is `"lenlex"`.

Independent of the current value of `CosetTableStandard` there is the possibility to standardize (or re-standardize) a coset table at any time using the following function.

2 ► `StandardizeTable( table, standard )` F

standardizes the given coset table *table*. The second argument is optional. It defines the standard to be used, its values may be `"lenlex"` or `"semilenlex"` specifying the new or the old convention, respectively. If no

value for the parameter *standard* is provided the function will use the global variable `CosetTableStandard` instead. Note that the function alters the given table, it does not create a copy.

```
gap> StandardizeTable( tab, "semilenlex" );
gap> PrintArray( TransposedMat( tab ) );
[ [  1,  1,  2,  4 ],
  [  3,  3,  4,  1 ],
  [  2,  2,  3,  3 ],
  [  5,  5,  1,  2 ],
  [  4,  4,  5,  5 ] ]
```

45.7 Coset tables for subgroups in the whole group

1 ▶ `CosetTableInWholeGroup(` *H* `)` A
▶ `TryCosetTableInWholeGroup(` *H* `)` O

is equivalent to `CosetTable(`*G*`,`*H*`)` where *G* is the (unique) finitely presented group such that *H* is a subgroup of *G*. It overrides a `silent` option (see 45.5.5) with `false`.

The variant `TryCosetTableInWholeGroup` does not override the `silent` option with `false` in case a coset table is only wanted if not too expensive. It will store a result that is not `fail` in the attribute `CosetTableInWholeGroup`.

2 ▶ `SubgroupOfWholeGroupByCosetTable(` *fpfam*`,` *tab* `)` F

takes a family of an fp group and a coset table *tab* and returns the subgroup of fam!.wholeGroup defined by this coset table.

See also `CosetTableBySubgroup` (45.5.4).

45.8 Augmented Coset Tables and Rewriting

1 ▶ `AugmentedCosetTableInWholeGroup(` *H* `[,` *gens*`] )` O

For a subgroup *H* of a finitely presented group, this function returns an augmented coset table. If a generator set *gens* is given, it is guaranteed that *gens* will be a subset of the primary and secondary subgroup generators of this coset table.

It is mutable so we are permitted to add further entries. However existing entries may not be changed. Any entries added however should correspond to the subgroup only and not to an homomorphism.

2 ▶ `AugmentedCosetTableMtc(` *G*`,` *H*`,` *type*`,` *string* `)` F

is an internal function used by the subgroup presentation functions described in 46.3. It applies a Modified Todd-Coxeter coset representative enumeration to construct an augmented coset table (see 46.3) for the given subgroup *H* of *G*. The subgroup generators will be named *string*1, *string*2, ...

The function accepts the options `max` and `silent` as described for the function `CosetTableFromGensAndRels` (see 45.5.5).

3 ▶ `AugmentedCosetTableRrs(` *G*`,` *table*`,` *type*`,` *string* `)` F

is an internal function used by the subgroup presentation functions described in 46.3. It applies the Reduced Reidemeister-Schreier method to construct an augmented coset table for the subgroup of *G* which is defined by the given coset table *table*. The new subgroup generators will be named *string*1, *string*2, ...

4 ▶ `RewriteWord(` *aug*`,` *word* `)` F

RewriteWord rewrites *word* (which must be a word in the underlying free group with respect to which the augmented coset table *aug* is given) in the subgroup generators given by the augmented coset table *aug*. It returns a Tietze-type word (i.e. a list of integers), referring to the primary and secondary generators of *aug*.

If *word* is not contained in the subgroup, `fail` is returned.

45.9 Low Index Subgroups

1 ▶ `LowIndexSubgroupsFpGroup( `*G*`, [`*H*`, ]`*index*`[, `*excluded*`] )` O

returns a list of representatives of the conjugacy classes of subgroups of the finitely presented group G that contain the subgroup H of G and that have index less than or equal to *index*.

If the optional argument *excluded* has been specified, then it is expected to be a list of words in the free generators of the underlying free group of G, and `LowIndexSubgroupsFpGroup` returns only those subgroups of index at most *index* that contain H, but do not contain any conjugate of any of the group elements defined by these words.

If not given, H defaults to the trivial subgroup.

The function `LowIndexSubgroupsFpGroup` finds the requested subgroups by systematically running through a tree of all potential coset tables of G of length at most *index* (where it skips all branches of that tree for which it knows in advance that they cannot provide new classes of such subgroups). The time required to do this depends, of course, on the presentation of G, but in general it will grow exponentially with the value of *index*. So you should be careful with the choice of *index*.

```
gap> li:=LowIndexSubgroupsFpGroup( g, TrivialSubgroup( g ), 10 );
[ Group(<fp, no generators known>), Group(<fp, no generators known>),
  Group(<fp, no generators known>), Group(<fp, no generators known>) ]
```

By default, the algorithm computes no generating sets for the subgroups. This can be enforced with `GeneratorsOfGroup`:

```
gap> GeneratorsOfGroup(li[2]);
[ a, b*a*b^-1 ]
```

As an example for an application of the optional parameter *excluded*, we compute all conjugacy classes of torsion free subgroups of index at most 24 in the group $G = \langle x, y, z \mid x^2, y^4, z^3, (xy)^3, (yz)^2, (xz)^3 \rangle$. It is know from theory that each torsion element of this group is conjugate to a power of x, y, z, xy, xz, or yz. (Note that this includes conjugates of y^2.)

```
gap> F := FreeGroup( "x", "y", "z" );;
gap> x := F.1;; y := F.2;; z := F.3;;
gap> G := F / [ x^2, y^4, z^3, (x*y)^3, (y*z)^2, (x*z)^3 ];;
gap> torsion := [ x, y, y^2, z, x*y, x*z, y*z ];;
gap> SetInfoLevel( InfoFpGroup, 2 );
gap> lis := LowIndexSubgroupsFpGroup( G, TrivialSubgroup( G ), 24, torsion );;
#I  LowIndexSubgroupsFpGroup called
#I   class 1 of index 24 and length 8
#I   class 2 of index 24 and length 24
#I   class 3 of index 24 and length 24
#I   class 4 of index 24 and length 24
#I   class 5 of index 24 and length 24
#I  LowIndexSubgroupsFpGroup returns 5 classes
gap> SetInfoLevel( InfoFpGroup, 0 );
```

If a particular image group is desired, the operation `GQuotients` (see 45.13) can be useful as well.

45.10 Converting Groups to Finitely Presented Groups

1 ▶ `IsomorphismFpGroup( G )` A

returns an isomorphism from the given finite group G to a finitely presented group isomorphic to G. The function first **chooses a set of generators of** G and then computes a presentation in terms of these generators.

```
gap> g := Group( (2,3,4,5), (1,2,5) );;
gap> iso := IsomorphismFpGroup( g );
[ (2,5,4,3), (1,2,3,4,5), (1,3,2,4,5) ] -> [ F1, F2, F3 ]
gap> fp := Image( iso );
<fp group of size 120 on the generators [ F1, F2, F3 ]>
gap> RelatorsOfFpGroup( fp );
[ F1^2*F2^2*F3*F2^-1, F2^-1*F1^-1*F2*F1*F2^-2*F3, F3^-1*F1^-1*F3*F1*F3^-1,
  F2^5*F3^-5, F2^5*F3^-1*F2^-1*F3^-1*F2^-1, F2^-2*F3^2*F2^-2*F3^2 ]
```

2 ▶ `IsomorphismFpGroupByGenerators( G, gens[, string] )` A
▶ `IsomorphismFpGroupByGeneratorsNC( G, gens, string )` A

returns an isomorphism from a finite group G to a finitely presented group F isomorphic to G. The generators of F correspond to the **generators of** G **given in the list** *gens*. If *string* is given it is used to name the generators of the finitely presented group.

The NC version will avoid testing whether the elements in *gens* generate G.

```
gap> SetInfoLevel( InfoFpGroup, 1 );
gap> iso := IsomorphismFpGroupByGenerators( g, [ (1,2), (1,2,3,4,5) ] );
#I  the image group has 2 gens and 5 rels of total length 39
[ (1,2), (1,2,3,4,5) ] -> [ F1, F2 ]
gap> fp := Image( iso );
<fp group of size 120 on the generators [ F1, F2 ]>
gap> RelatorsOfFpGroup( fp );
[ F1^2, F2^5, F2^-1*F1*F2^-1*F1*F2^-1*F1*F2^-1*F1,
  F2^-1*F1*F2*F1*F2^-1*F1*F2*F1*F2^-1*F1*F2*F1,
  F2^2*F1*F2^-2*F1*F2^2*F1*F2^-2*F1 ]
```

The main task of the function `IsomorphismFpGroupByGenerators` is to find a presentation of G in the provided generators *gens*. In the case of a permutation group G it does this by first constructing a stabilizer chain of G and then it works through that chain from the bottom to the top, recursively computing a presentation for each of the involved stabilizers. The method used is essentially an implementation of John Cannon's multi-stage relations-finding algorithm as described in [Neu82] (see also [Can73] for a more graph theoretical description). Moreover, it makes heavy use of Tietze transformations in each stage to avoid an explosion of the total length of the relators.

Note that because of the random methods involved in the construction of the stabilizer chain the resulting presentations of G will in general be different for repeated calls with the same arguments.

```
gap> M12 := MathieuGroup( 12 );
Group([ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6),
  (1,12)(2,11)(3,6)(4,8)(5,9)(7,10) ])
gap> gens := GeneratorsOfGroup( M12 );;
gap> iso := IsomorphismFpGroupByGenerators( M12, gens );;
#I  the image group has 3 gens and 21 rels of total length 541
gap> iso := IsomorphismFpGroupByGenerators( M12, gens );;
#I  the image group has 3 gens and 26 rels of total length 766
```

Also in the case of a permutation group G, the function `IsomorphismFpGroupByGenerators` supports the option `method` that can be used to modify the strategy. The option `method` may take the following values.

`method := "regular"`
: This may be specified for groups of small size, up to 10^5 say. It implies that the function first constructs a regular representation R of G and then a presentation of R. In general, this presentation will be much more concise than the default one, but the price is the time needed for the construction of R.

`method := [ "regular",` *bound* `]`
: This is a refinement of the previous possibility. In this case, *bound* should be an integer, and if so the method `"regular"` as described above is applied to the largest stabilizer in the stabilizer chain of G whose size does not exceed the given bound and then the multi-stage algorithm is used to work through the chain from that subgroup to the top.

`method := "fast"`
: This chooses an alternative method which essentially is a kind of multi-stage algorithm for a stabilizer chain of G but does not make any attempt do reduce the number of relators as it is done in Cannon's algorithm or to reduce their total length. Hence it is often much faster than the default method, but the total length of the resulting presentation may be huge.

`method := "default"`
: This simply means that the default method shall be used, which is the case if the option `method` is not given a value.

```
gap> iso := IsomorphismFpGroupByGenerators( M12, gens : method := "regular" );;
#I  the image group has 3 gens and 11 rels of total length 92
gap> iso := IsomorphismFpGroupByGenerators( M12, gens : method := "fast" );;
#I  the image group has 3 gens and 181 rels of total length 4169
```

Though the option `method := "regular"` is only checked in the case of a permutation group it also affects the performance and the results of the function `IsomorphismFpGroupByGenerators` for other groups, e. g. for matrix groups. This happens because, for these groups, the function first calls the function `NiceMonomorphism` to get a bijective action homomorphism from G to a suitable permutation group, P say, and then, recursively, calls itself for the group P so that now the option becomes relevant.

```
gap> G := ImfMatrixGroup( 5, 1, 3 );
ImfMatrixGroup(5,1,3)
gap> gens := GeneratorsOfGroup( G );
[ [ [ -1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 0, 0, 1, 0 ],
      [ -1, -1, -1, -1, 2 ], [ -1, 0, 0, 0, 1 ] ],
  [ [ 0, 1, 0, 0, 0 ], [ 0, 0, 1, 0, 0 ], [ 0, 0, 0, 1, 0 ],
      [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 1 ] ] ]
gap> iso := IsomorphismFpGroupByGenerators( G, gens );;
#I  the image group has 2 gens and 7 rels of total length 68
gap> iso := IsomorphismFpGroupByGenerators( G, gens : method := "regular" );;
#I  the image group has 2 gens and 6 rels of total length 56
gap> SetInfoLevel( InfoFpGroup, 0 );
gap> iso;
<composed isomorphism:[ [ [ -1, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0 ], [ 0, 0, 0, 1,\
 0 ], [ -1, -1, -1, -1, 2 ], [ -1, 0, 0, 0, 1 ] ], [ [ 0, 1, 0, 0, 0 ], [ 0, 0\
, 1, 0, 0 ], [ 0, 0, 0, 1, 0 ], [ 1, 0, 0, 0, 0 ], [ 0, 0, 0, 0, 1 ] ] ]->[ F1\
, F2 ]>
gap> ConstituentsCompositionMapping(iso);
[ <action isomorphism>,
  [ (1,55,43,29,19,71)(2,72,18,30,44,54)(3,32,20,31,4,56)(5,75,46,33,47,74)(6,
      35,48,34,7,76)(8,36)(9,63,51,37,27,79)(10,80,26,38,52,62)(11,40,28,39,
```

```
12,64)(13,59,58)(14,15,60)(17,70,41,53,42,69)(21,67,66)(22,23,68)(25,
78,49,61,50,77)(45,73), (1,13,5,2)(3,29,21,7)(4,14,33,10)(6,30,9,
15)(8,31,37,23)(11,32,22,35)(12,16,34,38)(18,53,25,19)(20,54,61,
27)(24,36,39,40)(26,55)(28,56,62,63)(41,57,45,42)(43,69,65,47)(44,58,
73,50)(46,70,49,59)(48,71,77,67)(51,72,66,75)(52,60,74,78)(68,76,79,
80) ] -> [ F1, F2 ] ]
```

Since GAP cannot decompose elements of a matrix group into generators, the resulting isomorphism is stored as a composition of a (faithful) permutation action on vectors and a homomorphism from the permutation image to the finitely presented group. In such a situation the constituent mappings can be obtained via `ConstituentsCompositionMapping` as separate GAP objects.

45.11 New Presentations and Presentations for Subgroups

`IsomorphismFpGroup` is also used to compute a new finitely presented group that is isomorphic to the subgroup of a given finitely presented group. (This is typically the only method to compute with subgroups of a finitely presented group.)

```
gap> f:=FreeGroup(2);;
gap> g:=f/[f.1^2,f.2^3,(f.1*f.2)^5];
<fp group on the generators [ f1, f2 ]>
gap> u:=Subgroup(g,[g.1*g.2]);
Group([ f1*f2 ])
gap> hom:=IsomorphismFpGroup(u);
[ <[ [ 1, 1 ] ]|f2^-1*f1^-1> ] -> [ F1 ]
gap> new:=Range(hom);
<fp group on the generators [ F1 ]>
gap> List(GeneratorsOfGroup(new),i->PreImagesRepresentative(hom,i));
[ <[ [ 1, 1 ] ]|f2^-1*f1^-1> ]
```

When working with such homomorphisms, some subgroup elements are expressed as extremely long words in the group generators. Therefore the underlying words of subgroup generators stored in the isomorphism (as obtained by `MappingGeneratorImages` and displayed when `Viewing` the homomorphism) as well as preimages under the homomorphism are stored in the form of straight line program elements (see 35.9). These will behave like ordinary words and no extra treatment should be necessary.

```
gap> r:=Range(hom).1^10;
F1^10
gap> p:=PreImagesRepresentative(hom,r);
<[ [ 1, 10 ] ]|f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^
-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1>
```

If desired, it also is possible to convert these underlying words using `EvalStraightLineProgElm`:

```
gap> r:=EvalStraightLineProgElm(UnderlyingElement(p));
f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^
-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1
gap> p:=ElementOfFpGroup(FamilyObj(p),r);
f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^
-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1
```

(If you are only interested in a finitely presented group isomorphic to U, but not in the isomorphism, you may also use the functions `PresentationViaCosetTable` and `FpGroupPresentation` (see 46.1).)

Homomorphisms can also be used to obtain an isomorphic finitely presented group with a (hopefully) simpler presentation.

1 ▶ `IsomorphismSimplifiedFpGroup( G )` A

applies Tietze transformations to a copy of the presentation of the given finitely presented group G in order to reduce it with respect to the number of generators, the number of relators, and the relator lengths.

The operation returns an isomorphism with source G, range a group H isomorphic to G, so that the presentation of H has been simplified using Tietze transformations.

```
gap> f:=FreeGroup(3);;
gap> g:=f/[f.1^2,f.2^3,(f.1*f.2)^5,f.1/f.3];
<fp group on the generators [ f1, f2, f3 ]>
gap> hom:=IsomorphismSimplifiedFpGroup(g);
[ f1, f2, f3 ] -> [ f1, f2, f1 ]
gap> Range(hom);
<fp group on the generators [ f1, f2 ]>
gap> RelatorsOfFpGroup(Range(hom));
[ f1^2, f2^3, f1*f2*f1*f2*f1*f2*f1*f2*f1*f2 ]
gap> RelatorsOfFpGroup(g);
[ f1^2, f2^3, f1*f2*f1*f2*f1*f2*f1*f2*f1*f2, f1*f3^-1 ]
```

(`IsomorphismSimplifiedFpGroup` uses Tietze transformations to simplify the presentation, see 46.2.1.)

45.12 Preimages under Homomorphisms from an FpGroup

For some subgroups of a finitely presented group the number of subgroup generators increases with the index of the subgroup. However often these generators are not needed at all for further calculations, but what is needed is the action of the cosets of the subgroup. This gives the image of the subgroup in a finite quotient and this finite quotient can be used to calculate normalizers, closures, intersections and so forth [Hul01].

The same applies for subgroups that are obtained as preimages under homomorphisms.

1 ▶ `SubgroupOfWholeGroupByQuotientSubgroup( fpfam, Q, U )` F

takes a fp group family *fpfam*, a finitely generated group Q such that the fp generators of *fam* can be mapped by an epimorphism *phi* onto `GeneratorsOfGroup(`Q`)` and a subgroup U of Q. It returns the subgroup of *fam*`!.wholeGroup` which is the full preimage of U under *phi*.

2 ▶ `IsSubgroupOfWholeGroupByQuotientRep( G )` R

is the representation for subgroups of an fp group, given by a quotient subgroup. The components G`!.quot` and G`!.sub` hold quotient, respectively subgroup.

3 ▶ `AsSubgroupOfWholeGroupByQuotient( U )` A

returns the same subgroup in the representation `AsSubgroupOfWholeGroupByQuotient`.

See also `SubgroupOfWholeGroupByCosetTable` (45.7.2) and `CosetTableBySubgroup` (45.5.4).

This technique is used by GAP for example to represent the derived subgroup, which is obtained from the quotient G/G'.

```
gap> f:=FreeGroup(2);;g:=f/[f.1^6,f.2^6,(f.1*f.2)^6];;
gap> d:=DerivedSubgroup(g);
Group(<fp, no generators known>)
gap> Index(g,d);
36
```

4 ▶ `DefiningQuotientHomomorphism( U )` F

if U is a subgroup in quotient representation (`IsSubgroupOfWholeGroupByQuotientRep`), this function returns the defining homomorphism from the whole group to U`!.quot`.

45.13 Quotient Methods

An important class of algorithms for finitely presented groups are the **quotient algorithms** which compute quotient groups of a given finitely presented group.

▶ `MaximalAbelianQuotient(`*fpgroup*`)`

as defined for general groups, this attribute returns the largest abelian quotient of *fpgroup*.

```
gap> f:=FreeGroup(2);;fp:=f/[f.1^6,f.2^6,(f.1*f.2)^12];
<fp group on the generators [ f1, f2 ]>
gap> hom:=MaximalAbelianQuotient(fp);
[ f1, f2 ] -> [ f1, f3 ]
gap> Size(Image(hom));
36
```

1 ▶ `PQuotient(` *F*`,` *p* `[,` *c*`] [,` *logord*`] [,` *ctype*`] )` F

computes a factor p-group of a finitely presented group F in form of a quotient system. The quotient system can be converted into an epimorphism from F onto the p-group computed by the function 45.13.2.

For a group G define the exponent-p central series of G inductively by $\mathcal{P}_1(G) = G$ and $\mathcal{P}_{i+1}(G) = [\mathcal{P}_i(G), G]\mathcal{P}_{i+1}(G)^p$. The factor groups modulo the terms of the lower exponent-p central series are p-groups. The group G has p-class c if $\mathcal{P}_c(G) \neq \mathcal{P}_{c+1}(G) = 1$.

The algorithm computes successive quotients modulo the terms of the exponent-p central series of F. If the parameter c is present, then the factor group modulo the $(c+1)$-th term of the exponent-p central series of F is returned. If c is not present, then the algorithm attempts to compute the largest factor p-group of F. In case F does not have a largest factor p-group, the algorithm will not terminate.

By default the algorithm computes only with factor groups of order at most p^{256}. If the parameter *logord* is present, it will compute with factor groups of order atmost p^{logord}. If this parameter is specified, then the parameter c must also be given. The present implementation produces an error message if the order of a p-quotient exceeds p^{256} or p^{logord}, respectively. Note that the order of intermediate p-groups may be larger than the final order of a p-quotient.

The parameter *ctype* determines the type of collector that is used for computations within the factor p-group. *ctype* must either be `single` in which case a simple collector from the left is used or `combinatorial` in which case a combinatorial collector from the left is used.

2 ▶ `EpimorphismQuotientSystem(` *quotsys* `)` O

For a quotient system *quotsys* obtained from the function 45.13.1, this operation returns an epimorphism $F \to P$ where F is the finitely presented group of which *quotsys* is a quotient system and P is a `PcGroup` isomorphic to the quotient of F determined by *quotsys*.

Different calls to this operation will create different groups P, each with its own family.

```
gap> PQuotient( FreeGroup(2), 5, 10, 1024, "combinatorial" );
<5-quotient system of 5-class 10 with 520 generators>
gap> phi := EpimorphismQuotientSystem( last );
[ f1, f2 ] -> [ a1, a2 ]
gap> Collected( Factors( Size( Image( phi ) ) ) );
[ [ 5, 520 ] ]
```

3 ▶ `EpimorphismPGroup(` *fpgrp*`,` *p* `)` O
▶ `EpimorphismPGroup(` *fpgrp*`,` *p*`,` *cl* `)` O

computes an epimorphism from the finitely presented group *fpgrp* to the largest p-group of p-class *cl* which is a quotient of *fpgrp*. If *cl* is omitted, the largest finite p-group quotient (of p-class up to 1000) is determined.

```
gap> hom:=EpimorphismPGroup(fp,2);
[ f1, f2 ] -> [ a1, a2 ]
gap> Size(Image(hom));
8
gap> hom:=EpimorphismPGroup(fp,3,7);
[ f1, f2 ] -> [ a1, a2 ]
gap> Size(Image(hom));
6561
```

4 ▶ `EpimorphismNilpotentQuotient( `*fpgrp*`[, `*n*`] )` F

returns an epimorphism on the class n finite nilpotent quotient of the finitely presented group *fpgrp*. If n is omitted, the largest finite nilpotent quotient (of p-class up to 1000) is taken.

```
gap> hom:=EpimorphismNilpotentQuotient(fp,7);
[ f1, f2 ] -> [ f1*f4, f2*f5 ]
gap> Size(Image(hom));
52488
```

A related operation which is also applicable to finitely presented groups is `GQuotients`, which computes all epimorphisms from a (finitely presented) group F onto a given (finite) group G, see 38.9.2.

```
gap> GQuotients(fp,Group((1,2,3),(1,2)));
[ [ f1, f2 ] -> [ (2,3), (1,2) ], [ f1, f2 ] -> [ (2,3), (1,2,3) ],
  [ f1, f2 ] -> [ (1,2,3), (1,2) ] ]
```

45.14 Abelian Invariants for Subgroups

Using variations of coset enumeration it is possible to compute the abelian invariants of a subgroup of a finitely presented group without computing a complete presentation for the subgroup in the first place. Typically, the operation `AbelianInvariants` when called for subgroups should automatically take care of this, but in case you what to have further control about the methods used, the following operations might be of use.

1 ▶ `AbelianInvariantsSubgroupFpGroup( `G`, `H` )` F

is a synonym for `AbelianInvariantsSubgroupFpGroupRrs(`G`,`H`)`.

2 ▶ `AbelianInvariantsSubgroupFpGroupMtc( `G`, `H` )` F

uses the Modified Todd-Coxeter method to compute the abelian invariants of a subgroup H of a finitely presented group G.

3 ▶ `AbelianInvariantsSubgroupFpGroupRrs( `G`, `H` )` F
▶ `AbelianInvariantsSubgroupFpGroupRrs( `G`, `*table*` )` F

uses the Reduced Reidemeister-Schreier method to compute the abelian invariants of a subgroup H of a finitely presented group G.

Alternatively to the subgroup H, its coset table *table* in G may be given as second argument.

4 ▶ `AbelianInvariantsNormalClosureFpGroup( `G`, `H` )` F

is a synonym for `AbelianInvariantsNormalClosureFpGroupRrs(`G`,`H`)`.

5 ▶ `AbelianInvariantsNormalClosureFpGroupRrs( `G`, `H` )` F

uses the Reduced Reidemeister-Schreier method to compute the abelian invariants of the normal closure of a subgroup H of a finitely presented group G.

See 46.3 for details on the different strategies.

The following example shows a calculation for the Coxeter group B_1. This calculation and a similar one for B_0 have been used to prove that $B_1'/B_1'' \cong Z_2^9 \times Z^3$ and $B_0'/B_0'' \cong Z_2^{91} \times Z^{27}$ as stated in Proposition 5 in [FJNT95].

```
gap> # Define the Coxeter group E1.
gap> F := FreeGroup( "x1", "x2", "x3", "x4", "x5" );
<free group on the generators [ x1, x2, x3, x4, x5 ]>
gap> x1 := F.1;; x2 := F.2;; x3 := F.3;; x4 := F.4;; x5 := F.5;;
gap> rels := [ x1^2, x2^2, x3^2, x4^2, x5^2,
>  ( x1 * x3 )^2, ( x2 * x4 )^2, ( x1 * x2 )^3, ( x2 * x3 )^3, ( x3 * x4 )^3,
>  ( x4 * x1 )^3, ( x1 * x5 )^3, ( x2 * x5 )^2, ( x3 * x5 )^3, ( x4 * x5 )^2,
>  ( x1 * x2 * x3 * x4 * x3 * x2 )^2 ];;
gap> E1 := F / rels;
<fp group on the generators [ x1, x2, x3, x4, x5 ]>
gap> x1 := E1.1;; x2 := E1.2;; x3 := E1.3;; x4 := E1.4;; x5 := E1.5;;
gap> # Get normal subgroup generators for B1.
gap> H := Subgroup( E1, [ x5 * x2^-1, x5 * x4^-1 ] );;
gap> # Compute the abelian invariants of B1/B1'.
gap> A := AbelianInvariantsNormalClosureFpGroup( E1, H );
[ 2, 2, 2, 2, 2, 2, 2, 2 ]
gap> # Compute a presentation for B1.
gap> P := PresentationNormalClosure( E1, H );
<presentation with 18 gens and 46 rels of total length 132>
gap> SimplifyPresentation( P );
#I  there are 8 generators and 30 relators of total length 148
gap> B1 := FpGroupPresentation( P );
<fp group on the generators [ _x1, _x2, _x3, _x4, _x6, _x7, _x8, _x11 ]>
gap> # Compute normal subgroup generators for B1'.
gap> gens := GeneratorsOfGroup( B1 );;
gap> numgens := Length( gens );;
gap> comms := [ ];;
gap> for i in [ 1 .. numgens - 1 ] do
>     for j in [i+1 .. numgens ] do
>         Add( comms, Comm( gens[i], gens[j] ) );
>     od;
> od;
gap> # Compute the abelian invariants of B1'/B1".
gap> K := Subgroup( B1, comms );;
gap> A := AbelianInvariantsNormalClosureFpGroup( B1, K );
[ 0, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2 ]
```

45.15 Testing Finiteness of Finitely Presented Groups

As a consequence of the algorithmic insolvabilities mentioned in the introduction to this chapter, there cannot be a general method that will test whether a given finitely presented group is actually finite.

Therefore testing a finitely presented group for `IsFinite` can be problematic. What GAP actually does upon a call of `IsFinite` (or if it is – probably implicitly – asked for a faithful permutation representation) is to test whether it can find (via coset enumeration) a cyclic subgroup of finite index. If it can, it rewrites the presentation to this subgroup. Since the subgroup is cyclic, its size can be checked easily from the resulting presentation, the size of the whole group is the product of the index and the subgroup size. Since however

no bound for the index of such a subgroup (if any exist) is known, such a test might continue unsuccesfully until memory is exhausted.

On the other hand, a couple of methods exist, that might prove that a group is infinite. Again, none is guaranteed to work in every case:

The first method is to find (for example via the low index algorithm, see `LowIndexSubgroupsFpGroup`) a subgroup U such that $[U : U']$ is infinite. If U has finite index, this can be checked by the operation `AbelianInvariants` (see section 45.14 for an example).

Another method is based on p-group quotients:

1 ▶ `NewmanInfinityCriterion( G, p )` F

Let G be a finitely presented group and p a prime that divides the order of G/G'. This function applies an infinity criterion due to M.F. Newman [New90] to G. (See chapter 16 of [Joh97] for a more explicit description.) It returns `true` if the criterion succeeds in proving that G is infinite and `fail` otherwise.

Note that the criterion uses the number of generators and relations in the presentation of G. Reduction of the persentation via Tietze transformations (`IsomorphismSimplifiedFpGroup`) therefore might produce an isomorphic group, for which the criterion will work better.

```
gap> g:=FibonacciGroup(2,9);
<fp group on the generators [ f1, f2, f3, f4, f5, f6, f7, f8, f9 ]>
gap> hom:=EpimorphismNilpotentQuotient(g,2);;
gap> k:=Kernel(hom);;
gap> Index(g,k);
152
gap> AbelianInvariants(k);
[ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
gap> NewmanInfinityCriterion(Kernel(hom),5);
true
```

This proves that the subgroup k (and thus the whole group g) is infinite. (This is the original example from [New90].)

46 Presentations and Tietze Transformations

A finite presentation describes a group, but usually there is a multitude of presentations that describe isomorphic groups. Therefore a presentation in GAP is different from a finitely presented group though there are ways to translate between both.

An important feature of presentations is that they can be modified (see sections 46.6 to 46.9).

If you only want to get new presentations for subgroups of a finitely presented group (and do not want to manipulate presentations yourself), chances are that the operation `IsomorphismFpGroup` already does what you want (see 45.11).

46.1 Creating Presentations

1 ► `PresentationFpGroup(` *G* `[,` *printlevel*`] )` F

creates a presentation, i.e. a Tietze object, for the given finitely presented group *G*. This presentation will be exactly as the presentation of *G* and **no** initial Tietze transformations are applied to it.

The optional *printlevel* parameter can be used to restrict or to extend the amount of output provided by Tietze transformation commands when being applied to the created presentation. The default value 1 is designed for interactive use and implies explicit messages to be displayed by most of these commands. A *printlevel* value of 0 will suppress these messages, whereas a *printlevel* value of 2 will enforce some additional output.

```
gap> f := FreeGroup( "a", "b" );
<free group on the generators [ a, b ]>
gap> g := f / [ f.1^3, f.2^2, (f.1*f.2)^3 ];
<fp group on the generators [ a, b ]>
gap> p := PresentationFpGroup( g );
<presentation with 2 gens and 3 rels of total length 11>
```

Most of the functions creating presentations and all functions performing Tietze transformations on them sort the relators by increasing lengths. The function `PresentationFpGroup` is an exception because it is intended to reflect the relators that were used to define the involved FpGroup. You may use the following command to sort the presentation.

2 ► `TzSort(` *P* `)` F

sorts the relators of the given presentation *P* by increasing lengths. There is no particular ordering defined for the relators of equal length. Note that `TzSort` does not return a new object. It changes the given presentation.

3 ► `GeneratorsOfPresentation(` *P* `)` O

returns a list of free generators that is a `ShallowCopy` of the current generators of the presentation *P*.

4 ▶ `FpGroupPresentation( P [, nam] )` F

constructs an `FpGroup` group as defined by the given Tietze presentation P.

```
gap> h := FpGroupPresentation( p );
<fp group on the generators [ a, b ]>
gap> h = g;
false
```

5 ▶ `PresentationViaCosetTable( G )` F
▶ `PresentationViaCosetTable( G, F, words )` F

constructs a presentation for a given concrete finite group. It applies the relations finding algorithm which has been described in [Can73] and [Neu82]. It automatically applies Tietze transformations to the presentation found.

If only a group G has been specified, the single stage algorithm is applied.

The operation `IsomorphismFpGroup` in contrast uses a multiple-stage algorithm using a composition series and stabilizer chains. It usually should be used rather than `PresentationViaCosetTable`. (It does not apply Tietze transformations automatically.)

If the two stage algorithm is to be used, `PresentationViaCosetTable` expects a subgroup H of G to be provided in form of two additional arguments F and *words*, where F is a free group with the same number of generators as G, and *words* is a list of words in the generators of F which supply a list of generators of H if they are evaluated as words in the corresponding generators of G.

```
gap> G := GeneralLinearGroup( 2, 7 );
GL(2,7)
gap> GeneratorsOfGroup( G );
[ [ [ Z(7), 0*Z(7) ], [ 0*Z(7), Z(7)^0 ] ],
  [ [ Z(7)^3, Z(7)^0 ], [ Z(7)^3, 0*Z(7) ] ] ]
gap> Size( G );
2016
gap> P := PresentationViaCosetTable( G );
<presentation with 2 gens and 5 rels of total length 46>
gap> TzPrintRelators( P );
#I  1. f2^3
#I  2. f1^6
#I  3. f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1*f1^-1*f2^-1
#I  4. f1*f2*f1^-1*f2^-1*f1*f2^-1*f1^-1*f2*f1*f2^-1*f1^-1*f2^-1
#I  5. f1^-3*f2*f1*f2*f1^-1*f2^-1*f1^-1*f2^-1*f1^-2*f2
```

The two stage algorithm saves an essential amount of space by constructing two coset tables of lengths $|H|$ and $|G|/|H|$ instead of just one coset table of length $|G|$. The next example shows an application of this option in the case of a subgroup of size 7920 and index 12 in a permutation group of size 95040.

```
gap> M12 := Group( [ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6),
> (1,12)(2,11)(3,6)(4,8)(5,9)(7,10) ], () );;
gap> F := FreeGroup( "a", "b", "c" );
<free group on the generators [ a, b, c ]>
gap> words := [ F.1, F.2 ];
[ a, b ]
gap> P := PresentationViaCosetTable( M12, F, words );
<presentation with 3 gens and 10 rels of total length 97>
gap> G := FpGroupPresentation( P );
<fp group on the generators [ a, b, c ]>
```

```
gap> RelatorsOfFpGroup( G );
[ c^2, b^4, a*c*a*c*a*c, a*b^-2*a*b^-2*a*b^-2, a^11,
  a^2*b*a^-2*b^-2*a*b^-1*a^2*b^-1, a*b*a^-1*b*a^-1*b^-1*a*b*a^-1*b*a^-1*b^-1,
  a^2*b*a^2*b^-2*a^-1*b*a^-1*b^-1*a^-1*b^-1,
  a*b*a*b*a^2*b^-1*a^-1*b^-1*a*c*b*c, a^4*b*a^2*b*a^-2*c*a*b*a^-1*c ]
```

Before it is returned, the resulting presentation is being simplified by appropriate calls of the function `SimplifyPresentation` (see 46.7), but without allowing any eliminations of generators. This restriction guarantees that we get a bijection between the list of generators of G and the list of generators in the presentation. Hence, if the generators of G are redundant and if you don't care for the bijection, you may get a shorter presentation by calling the function `SimplifyPresentation`, now without this restriction, once more yourself.

```
gap> H := Group(
> [ (2,5,3), (2,7,5), (1,8,4), (1,8,6), (4,8,6), (3,5,7) ], () );;
gap> P := PresentationViaCosetTable( H );
<presentation with 6 gens and 12 rels of total length 42>
gap> SimplifyPresentation( P );
#I  there are 4 generators and 10 relators of total length 36
```

If you apply the function `FpGroupPresentation` to the resulting presentation you will get a finitely presented group isomorphic to G. Note, however, that the function `IsomorphismFpGroup` (see 45.10.1) is recommended for this purpose.

46.2 SimplifiedFpGroup

1 ▶ `SimplifiedFpGroup(` *G* `)` F

applies Tietze transformations to a copy of the presentation of the given finitely presented group G in order to reduce it with respect to the number of generators, the number of relators, and the relator lengths.

`SimplifiedFpGroup` returns a group isomorphic to the given one with a presentation which has been tried to simplify via Tietze transformations.

If the connection to the original group is important, then the operation `IsomorphismSimplifiedFpGroup` (see 45.11.1) should be used instead.

```
gap> F6 := FreeGroup( 6, "G" );;
gap> G := F6 / [ F6.1^2, F6.2^2, F6.4*F6.6^-1, F6.5^2, F6.6^2,
> F6.1*F6.2^-1*F6.3, F6.1*F6.5*F6.3^-1, F6.2*F6.4^-1*F6.3,
> F6.3*F6.4*F6.5^-1, F6.1*F6.6*F6.3^-2, F6.3^4 ];;
gap> H := SimplifiedFpGroup( G );
<fp group on the generators [ G1, G3 ]>
gap> RelatorsOfFpGroup( H );
[ G1^2, G1*G3^-1*G1*G3^-1, G3^4 ]
```

In fact, the command

```
H := SimplifiedFpGroup( G );
```

is an abbreviation of the command sequence

```
P := PresentationFpGroup( G, 0 );;
SimplifyPresentation( P );
H := FpGroupPresentation( P );
```

which applies a rather simple-minded strategy of Tietze transformations to the intermediate presentation P. If, for some concrete group, the resulting presentation is unsatisfying, then you should try a more sophisticated, interactive use of the available Tietze transformation commands (see 46.7).

46.3 Subgroup Presentations

1 ▶ `PresentationSubgroup( `*G*`, `*H*`[, `*string*`] )` F

is a synonym for `PresentationSubgroupRrs(`*G*`,`*H*`[,`*string*`])`.

2 ▶ `PresentationSubgroupRrs( `*G*`, `*H*` [, `*string*`] )` F
▶ `PresentationSubgroupRrs( `*G*`, `*table*` [, `*string*`] )` F

uses the Reduced Reidemeister-Schreier method to compute a presentation P, say, for a subgroup H of a finitely presented group G. The generators in the resulting presentation will be named *string*1, *string*2, ... , the default string is `"_x"`. You may access the i-th of these generators by P!.i.

Alternatively to the subgroup H, its coset table *table* in G may be given as second argument.

```
gap> f := FreeGroup( "a", "b" );;
gap> g := f / [ f.1^2, f.2^3, (f.1*f.2)^5 ];
<fp group on the generators [ a, b ]>
gap> g1 := Size( g );
60
gap> u := Subgroup( g, [ g.1, g.1^g.2 ] );
Group([ a, b^-1*a*b ])
gap> p := PresentationSubgroup( g, u, "g" );
<presentation with 3 gens and 4 rels of total length 12>
gap> gens := GeneratorsOfPresentation( p );
[ g1, g2, g3 ]
gap> TzPrintRelators( p );
#I  1. g1^2
#I  2. g2^2
#I  3. g3*g2*g1
#I  4. g3^5
```

Note that you cannot call the generators by their names. These names are not variables, but just display figures. So, if you want to access the generators by their names, you first will have to introduce the respective variables and to assign the generators to them.

```
gap> gens[1] = g1;
false
gap> g1;
60
gap> g1 := gens[1];; g2 := gens[2];; g3 := gens[3];;
gap> g1;
g1
```

The Reduced Reidemeister-Schreier algorithm is a modification of the Reidemeister-Schreier algorithm of George Havas [Hav74]. It was proposed by Joachim Neubüser and first implemented in 1986 by Andrea Lucchini and Volkmar Felsch in the SPAS system [SPA89]. Like the Reidemeister-Schreier algorithm of George Havas, it needs only the presentation of G and a coset table of H in G to construct a presentation of H.

Whenever you call the command `PresentationSubgroupRrs`, it first obtains a coset table of H in G if not given. Next, a set of generators of H is determined by reconstructing the coset table and introducing in that process as many Schreier generators of H in G as are needed to do a Felsch strategy coset enumeration without any coincidences. (In general, though containing redundant generators, this set will be much smaller than the set of all Schreier generators. That is why we call the method the **Reduced** Reidemeister-Schreier.)

After having constructed this set of **primary subgroup generators**, say, the coset table is extended to an **augmented coset table** which describes the action of the group generators on coset representatives, i.e., on elements instead of cosets. For this purpose, suitable words in the (primary) subgroup generators have to be associated to the coset table entries. In order to keep the lengths of these words short, additional **secondary subgroup generators** are introduced as abbreviations of subwords. Their number may be large.

Finally, a Reidemeister rewriting process is used to get defining relators for H from the relators of G. As the resulting presentation of H is a presentation on primary **and** secondary generators, in general you will have to simplify it by appropriate Tietze transformations (see 46.7) or by the command `DecodeTree` (see 46.11.1) before you can use it. Therefore it is returned in the form of a presentation, P say.

Compared with the Modified Todd-Coxeter method described below, the Reduced Reidemeister-Schreier method (as well as Havas' original Reidemeister-Schreier program) has the advantage that it does not require generators of H to be given if a coset table of H in G is known. This provides a possibility to compute a presentation of the normal closure of a given subgroup (see the `PresentationNormalClosureRrs` command below).

For certain applications you may be interested in getting not only just a presentation for H, but also a relation between the involved generators of H and the generators of G. The subgroup generators in the presentation are sorted such that the primary generators precede the secondary ones. Moreover, for each secondary subgroup generator there is a relator in the presentation which expresses this generator as a word in preceding ones. Hence, all we need in addition is a list of words in the generators of G which express the primary subgroup generators. In fact, such a list is provided in the attribute `PrimaryGeneratorWords` of the resulting presentation.

3 ► `PrimaryGeneratorWords(` P `)` A

is an attribute of the presentation P which holds a list of words in the associated group generators (of the underlying free group) which express the primary subgroup generators of P.

```
gap> PrimaryGeneratorWords( p );
[ a, b^-1*a*b ]
```

4 ► `PresentationSubgroupMtc(` G`,` H `[,` *string*`] [,` *print level*`] )` F

uses the Modified Todd-Coxeter coset representative enumeration method to compute a presentation P, say, for a subgroup H of a finitely presented group G. The presentation returned is in generators corresponding to the generators of H. The generators in the resulting presentation will be named *string*1, *string*2, ... , the default string is `"_x"`. You may access the i-th of these generators by P`!.`i.

The default print level is 1. If the print level is set to 0, then the printout of the implicitly called function `DecodeTree` will be suppressed.

```
gap> p := PresentationSubgroupMtc( g, u );
#I  there are 3 generators and 4 relators of total length 12
#I  there are 2 generators and 3 relators of total length 14
<presentation with 2 gens and 3 rels of total length 14>
```

The so called Modified Todd-Coxeter method was proposed, in slightly different forms, by Nathan S. Mendelsohn and William O. J. Moser in 1966. Moser's method was proved in [BC76]. It has been generalized to cover a broad spectrum of different versions (see the survey [Neu82]).

The `Modified Todd-Coxeter` method performs an enumeration of coset representatives. It proceeds like an ordinary coset enumeration (see 45.5), but as the product of a coset representative by a group generator or its inverse need not be a coset representative itself, the Modified Todd-Coxeter has to store a kind of correction element for each coset table entry. Hence it builds up a so called **augmented coset table** of

H in G consisting of the ordinary coset table and a second table in parallel which contains the associated subgroup elements.

Theoretically, these subgroup elements could be expressed as words in the given generators of H, but in general these words tend to become unmanageable because of their enormous lengths. Therefore, a highly redundant list of subgroup generators is built up starting from the given ("**primary**") generators of H and adding additional ("**secondary**") generators which are defined as abbreviations of suitable words of length two in the preceding generators such that each of the subgroup elements in the augmented coset table can be expressed as a word of length at most one in the resulting (primary **and** secondary) subgroup generators.

Then a rewriting process (which is essentially a kind of Reidemeister rewriting process) is used to get relators for H from the defining relators of G.

The resulting presentation involves all the primary, but not all the secondary generators of H. In fact, it contains only those secondary generators which explicitly occur in the augmented coset table. If we extended this presentation by those secondary generators which are not yet contained in it as additional generators, and by the definitions of all secondary generators as additional relators, we would get a presentation of H, but, in general, we would end up with a large number of generators and relators.

On the other hand, if we avoid this extension, the current presentation will not necessarily define H although we have used the same rewriting process which in the case of the `PresentationSubgroupRrs` command computes a defining set of relators for H from an augmented coset table and defining relators of G. The different behaviour here is caused by the fact that coincidences may have occurred in the Modified Todd-Coxeter coset enumeration.

To overcome this problem without extending the presentation by all secondary generators, the `PresentationSubgroupMtc` command applies the so called **decoding tree** algorithm which provides a more economical approach. The reader is strongly recommended to carefully read section 46.11.1 where this algorithm is described in more detail. Here we will only mention that this procedure may add a lot of intermediate generators and relators (and even change the isomorphism type) in a process which in fact eliminates all secondary generators from the presentation and hence finally provides a presentation of H on the primary, i.e., the originally given, generators of H. This is a remarkable advantage of the command `PresentationSubgroupMtc` compared to the command `PresentationSubgroupRrs`. But note that, for some particular subgroup H, the Reduced Reidemeister-Schreier method might quite well produce a more concise presentation.

The resulting presentation is returned in the form of a presentation, P say.

As the function `PresentationSubgroupRrs` described above (see there for details), the function `PresentationSubgroupMtc` returns a list of the primary subgroup generators of H in the attribute `PrimaryGeneratorWords` of P. In fact, this list is not very exciting here because it is just a shallow copy of the attribute value `GeneratorsOfPresentation(H)`, however it is needed to guarantee a certain consistency between the results of the different functions for computing subgroup presentations.

Though the decoding tree routine already involves a lot of Tietze transformations, we recommend that you try to further simplify the resulting presentation by appropriate Tietze transformations (see 46.7).

5 ▶ `PresentationNormalClosureRrs(` G`,` H `[,` *string*`] )` F

uses the Reduced Reidemeister-Schreier method to compute a presentation P, say, for the normal closure of a subgroup H of a finitely presented group G. The generators in the resulting presentation will be named *string*1, *string*2, ... , the default string is `"_x"`. You may access the i-th of these generators by $P!.i$.

6 ▶ `PresentationNormalClosure(` G`,` H`[,` *string*`] )` F

is a synonym for `PresentationNormalClosureRrs(`G`,`H`[,`*string*`])`.

46.4 Relators in a Presentation

In order to speed up the Tietze transformation routines, each relator in a presentation P is internally represented by a list of positive or negative generator numbers, i.e., each factor of the proper GAP word is represented by the position number of the corresponding generator with respect to the current list of generators, or by the respective negative number, if the factor is the inverse of a generator. Note that the numbering of the generators in Tietze words is always relative to a generator list and bears no relation to the internal numbering of generators in a family of associative words.

1 ▶ `TietzeWordAbstractWord(` *word*`,` *fgens* `)` F

assumes *fgens* to be a list of free group generators and *word* to be an abstract word in these generators. It converts *word* into a Tietze word, i. e., a list of positive or negative generator numbers.

This function simply calls `LetterRepAssocWord`.

2 ▶ `AbstractWordTietzeWord(` *word*`,` *fgens* `)` F

assumes *fgens* to be a list of free group generators and *word* to be a Tietze word in these generators, i. e., a list of positive or negative generator numbers. It converts *word* to an abstract word.

This function simply calls `AssocWordByLetterRep`.

```
gap> F := FreeGroup( "a", "b", "c" ,"d");
<free group on the generators [ a, b, c, d ]>
gap> tzword := TietzeWordAbstractWord(
> Comm(F.4,F.2) * (F.3^2 * F.2)^-1, GeneratorsOfGroup( F ){[2,3,4]} );
[ -3, -1, 3, -2, -2 ]
gap> AbstractWordTietzeWord( tzword, GeneratorsOfGroup( F ){[2,3,4]} );
d^-1*b^-1*d*c^-2
```

46.5 Printing Presentations

Whenever you create a presentation P, say, or assign it to a variable, GAP will respond by printing P. However, as P may contain a lot of generators and many relators of large length, it would be annoying if the standard print facilities displayed all this information in detail. So they restrict the printout to just one line of text containing the number of generators, the number of relators, and the total length of all relators of P. As compensation, GAP offers some special print commands which display various details of a presentation.

1 ▶ `TzPrintGenerators(` *P* `[,` *list*`] )` F

prints the generators of the given Tietze presentation P together with the number of their occurrences in the relators. The optional second argument can be used to specify the numbers of the generators to be printed. Default: all generators are printed.

```
gap> G := Group( [ (1,2,3,4,5), (2,3,5,4), (1,6)(3,4) ], () );
Group([ (1,2,3,4,5), (2,3,5,4), (1,6)(3,4) ])
gap> P := PresentationViaCosetTable( G );
<presentation with 3 gens and 6 rels of total length 28>
gap> TzPrintGenerators( P );
#I  1.  f1   11 occurrences
#I  2.  f2   10 occurrences
#I  3.  f3   7 occurrences   involution
```

2 ▶ `TzPrintRelators(` *P*`[,` *list*`] )` F

prints the relators of the given Tietze presentation P. The optional second argument *list* can be used to specify the numbers of the relators to be printed. Default: all relators are printed.

```
gap> TzPrintRelators( P );
#I  1. f3^2
#I  2. f2^4
#I  3. f2^-1*f3*f2^-1*f3
#I  4. f1^5
#I  5. f1^2*f2*f1*f2^-1
#I  6. f1^-1*f3*f1*f3*f1^-1*f2^2*f3
```

3 ▶ `TzPrintLengths( P )` F

prints just a list of all relator lengths of the given presentation P.

```
gap> TzPrintLengths( P );
[ 2, 4, 4, 5, 5, 8 ]
```

4 ▶ `TzPrintStatus( P [, norepeat ] )` F

is an internal function which is used by the Tietze transformation routines to print the number of generators, the number of relators, and the total length of all relators in the given Tietze presentation P. If *norepeat* is specified as `true`, the printing is suppressed if none of the three values has changed since the last call.

```
gap> TzPrintStatus( P );
#I  there are 3 generators and 6 relators of total length 28
```

5 ▶ `TzPrintPresentation( P )` F

prints the generators and the relators of a Tietze presentation. In fact, it is an abbreviation for the successive call of the three commands `TzPrintGenerators(`P`)`, `TzPrintRelators(`P`)`, and `TzPrintStatus(`P`)`.

6 ▶ `TzPrint( P [, list] )` F

prints the current generators of the given presentation P, and prints the relators of P as Tietze words (without converting them back to abstract words as the functions `TzPrintRelators` and `TzPrintPresentation` do). The optional second argument can be used to specify the numbers of the relators to be printed. Default: all relators are printed.

```
gap> TzPrint( P );
#I  generators: [ f1, f2, f3 ]
#I  relators:
#I  1.  2  [ 3, 3 ]
#I  2.  4  [ 2, 2, 2, 2 ]
#I  3.  4  [ -2, 3, -2, 3 ]
#I  4.  5  [ 1, 1, 1, 1, 1 ]
#I  5.  5  [ 1, 1, 2, 1, -2 ]
#I  6.  8  [ -1, 3, 1, 3, -1, 2, 2, 3 ]
```

7 ▶ `TzPrintPairs( P [, n] )` F

prints the n most often occurring relator subwords of the form ab, where a and b are different generators or inverses of generators, together with the number of their occurrences. The default value of n is 10. A value $n = 0$ is interpreted as `infinity`.

The function `TzPrintPairs` is useful in the context of Tietze transformations which introduce new generators by substituting words in the current generators (see 46.9). It gives some evidence for an appropriate choice of a word of length 2 to be substituted.

```
gap> TzPrintPairs( P, 3 );
#I  1.  3  occurrences of  f2 * f3
#I  2.  2  occurrences of  f2^-1 * f3
#I  3.  2  occurrences of  f1 * f3
```

Finally, there is a function `TzPrintOptions`. It is described in section 46.12.

46.6 Changing Presentations

The functions described in this section may be used to change a presentation. Note, however, that in general they do not perform Tietze transformations because they change or may change the isomorphism type of the group defined by the presentation.

1 ▶ `AddGenerator( P )` F

extends the presentation P by a new generator.

Let i be the smallest positive integer which has not yet been used as a generator number in the given presentation. `AddGenerator` defines a new abstract generator x_i with the name `"_xi"` and adds it to the list of generators of P.

You may access the generator x_i by typing $P!.i$. However, this is only practicable if you are running an interactive job because you have to know the value of i. Hence the proper way to access the new generator is to write `GeneratorsOfPresentation(P)[Length(GeneratorsOfPresentation(P))]`.

```
gap> G := PerfectGroup( 120 );;
gap> H := Subgroup( G, [ G.1^G.2, G.3 ] );;
gap> P := PresentationSubgroup( G, H );
<presentation with 4 gens and 7 rels of total length 21>
gap> AddGenerator( P );
#I  now the presentation has 5 generators, the new generator is _x7
gap> gens := GeneratorsOfPresentation( P );
[ _x1, _x2, _x4, _x5, _x7 ]
gap> gen := gens[Length( gens )];
_x7
gap> gen = P!.7;
true
```

2 ▶ `TzNewGenerator( P )` F

is an internal function which defines a new abstract generator and adds it to the presentation P. It is called by `AddGenerator` and by several Tietze transformation commands. As it does not know which global lists have to be kept consistent, you should not call it. Instead, you should call the function `AddGenerator`, if needed.

3 ▶ `AddRelator( P, word )` F

adds the relator *word* to the presentation P, probably changing the group defined by P. *word* must be an abstract word in the generators of P.

4 ▶ `RemoveRelator( P, n )` F

removes the n-th relator from the presentation P, probably changing the group defined by P.

46.7 Tietze Transformations

The commands in this section can be used to modify a presentation by Tietze transformations.

In general, the aim of such modifications will be to **simplify** the given presentation, i.e., to reduce the number of generators and the number of relators without increasing too much the sum of all relator lengths which we will call the **total length** of the presentation. Depending on the concrete presentation under investigation one may end up with a nice, short presentation or with a very huge one.

Unfortunately there is no algorithm which could be applied to find the shortest presentation which can be obtained by Tietze transformations from a given one. Therefore, what GAP offers are some lower-level Tietze

transformation commands and, in addition, some higher-level commands which apply the lower-level ones in a kind of default strategy which of course cannot be the optimal choice for all presentations.

The design of these commands follows closely the concept of the ANU Tietze transformation program [Hav69] and its later revisions (see [HKRR84], [Rob88]).

1 ▶ `TzGo(` *P* `[,` *silent*`] )` F

automatically performs suitable Tietze transformations of the given presentation P. It is perhaps the most convenient one among the interactive Tietze transformation commands. It offers a kind of default strategy which, in general, saves you from explicitly calling the lower-level commands it involves.

If *silent* is specified as `true`, the printing of the status line by `TzGo` is suppressed if the Tietze option `printLevel` (see 46.12) has a value less than 2.

2 ▶ `SimplifyPresentation(` *P* `)` F

is a synonym for `TzGo(`P`)`.

```
gap> F2 := FreeGroup( "a", "b" );;
gap> G := F2 / [ F2.1^9, F2.2^2, (F2.1*F2.2)^4, (F2.1^2*F2.2)^3 ];;
gap> a := G.1;; b := G.2;;
gap> H := Subgroup( G, [ (a*b)^2, (a^-1*b)^2 ] );;
gap> Index( G, H );
408
gap> P := PresentationSubgroup( G, H );
<presentation with 8 gens and 36 rels of total length 111>
gap> PrimaryGeneratorWords( P );
[ b, a*b*a ]
gap> TzOptions( P ).protected := 2;
2
gap> TzOptions( P ).printLevel := 2;
2
gap> SimplifyPresentation( P );
#I  eliminating _x7 = _x5^-1
#I  eliminating _x5 = _x4
#I  eliminating _x18 = _x3
#I  eliminating _x8 = _x3
#I  there are 4 generators and 8 relators of total length 21
#I  there are 4 generators and 7 relators of total length 18
#I  eliminating _x4 = _x3^-1*_x2^-1
#I  eliminating _x3 = _x2*_x1^-1
#I  there are 2 generators and 4 relators of total length 14
#I  there are 2 generators and 4 relators of total length 13
#I  there are 2 generators and 3 relators of total length 9
gap> TzPrintRelators( P );
#I  1. _x1^2
#I  2. _x2^3
#I  3. _x2*_x1*_x2*_x1
```

Roughly speaking, `TzGo` consists of a loop over a procedure which involves two phases: In the **search phase** it calls `TzSearch` and `TzSearchEqual` described below which try to reduce the relator lengths by substituting common subwords of relators, in the **elimination phase** it calls the command `TzEliminate` described below (or, more precisely, a subroutine of `TzEliminate` in order to save some administrative overhead) which tries to eliminate generators that can be expressed as words in the remaining generators.

If `TzGo` succeeds in reducing the number of generators, the number of relators, or the total length of all relators, it displays the new status before returning (provided that you did not set the print level to zero). However, it does not provide any output if all these three values have remained unchanged, even if the command `TzSearchEqual` involved has changed the presentation such that another call of `TzGo` might provide further progress. Hence, in such a case it makes sense to repeat the call of the command for several times (or to call the command `TzGoGo` instead).

3 ▶ `TzGoGo(` *P* `)` F

calls the command `TzGo` again and again until it does not reduce the presentation any more.

The result of the Tietze transformations can be affected substantially by the options parameters (see 46.12). To demonstrate the effect of the `eliminationsLimit` parameter, we will give an example in which we handle a subgroup of index 240 in a group of order 40320 given by a presentation due to B. H. Neumann. First we construct a presentation of the subgroup, and then we apply to it the command `TzGoGo` for different values of the parameter `eliminationsLimit` (including the default value 100). In fact, we also alter the `printLevel` parameter, but this is only done in order to suppress most of the output. In all cases the resulting presentations cannot be improved any more by applying the command `TzGoGo` again, i.e., they are the best results which we can get without substituting new generators.

```
gap> F3 := FreeGroup( "a", "b", "c" );;
gap> G := F3 / [ F3.1^3, F3.2^3, F3.3^3, (F3.1*F3.2)^5,
> (F3.1^-1*F3.2)^5, (F3.1*F3.3)^4, (F3.1*F3.3^-1)^4,
> F3.1*F3.2^-1*F3.1*F3.2*F3.3^-1*F3.1*F3.3*F3.1*F3.3^-1,
> (F3.2*F3.3)^3, (F3.2^-1*F3.3)^4 ];;
gap> a := G.1;; b := G.2;; c := G.3;;
gap> H := Subgroup( G, [ a, c ] );;
gap> for i in [ 61, 62, 63, 90, 97 ] do
> Pi := PresentationSubgroup( G, H );
> TzOptions( Pi ).eliminationsLimit := i;
> Print("#I eliminationsLimit set to ",i,"\n");
> TzOptions( Pi ).printLevel := 0;
> TzGoGo( Pi );
> TzPrintStatus( Pi );
> od;
#I eliminationsLimit set to 61
#I  there are 2 generators and 104 relators of total length 7012
#I eliminationsLimit set to 62
#I  there are 2 generators and 7 relators of total length 56
#I eliminationsLimit set to 63
#I  there are 3 generators and 97 relators of total length 5998
#I eliminationsLimit set to 90
#I  there are 3 generators and 11 relators of total length 68
#I eliminationsLimit set to 97
#I  there are 4 generators and 109 relators of total length 3813
```

Similarly, we demonstrate the influence of the `saveLimit` parameter by just continuing the preceding example for some different values of the `saveLimit` parameter (including its default value 10), but without changing the `eliminationsLimit` parameter which keeps its default value 100.

```
gap> for i in [ 7 .. 11 ] do
> Pi := PresentationSubgroup( G, H );
> TzOptions( Pi ).saveLimit := i;
> Print( "#I saveLimit set to ", i, "\n" );
> TzOptions( Pi ).printLevel := 0;
> TzGoGo( Pi );
> TzPrintStatus( Pi );
> od;
#I saveLimit set to 7
#I  there are 3 generators and 99 relators of total length 2713
#I saveLimit set to 8
#I  there are 2 generators and 103 relators of total length 11982
#I saveLimit set to 9
#I  there are 2 generators and 6 relators of total length 41
#I saveLimit set to 10
#I  there are 3 generators and 118 relators of total length 13713
#I saveLimit set to 11
#I  there are 3 generators and 11 relators of total length 58
```

46.8 Elementary Tietze Transformations

1 ▶ `TzEliminate( `*P*` )` F
▶ `TzEliminate( `*P*`, `*gen*` )` F
▶ `TzEliminate( `*P*`, `*n*` )` F

tries to eliminate a generator from a presentation P via Tietze transformations.

Any relator which contains some generator just once can be used to substitute that generator by a word in the remaining generators. If such generators and relators exist, then `TzEliminate` chooses a generator for which the product of its number of occurrences and the length of the substituting word is minimal, and then it eliminates this generator from the presentation, provided that the resulting total length of the relators does not exceed the associated Tietze option parameter `spaceLimit` (see 46.12). The default value of that parameter is `infinity`, but you may alter it appropriately.

If a generator *gen* has been specified, `TzEliminate` eliminates it if possible, i. e. if there is a relator in which *gen* occurs just once. If no second argument has been specified, `TzEliminate` eliminates some appropriate generator if possible and if the resulting total length of the relators will not exceed the parameter `lengthLimit`.

If an integer n has been specified, `TzEliminate` tries to eliminate up to n generators. Note that the calls `TzEliminate(`P`)` and `TzEliminate(`P`,1)` are equivalent.

2 ▶ `TzSearch( `*P*` )` F

searches for relator subwords which, in some relator, have a complement of shorter length and which occur in other relators, too, and uses them to reduce these other relators.

The idea is to find pairs of relators r_1 and r_2 of length l_1 and l_2, respectively, such that $l_1 \leq l_2$ and r_1 and r_2 coincide (possibly after inverting or conjugating one of them) in some maximal subword w, say, of length greater than $l_1/2$, and then to substitute each copy of w in r_2 by the inverse complement of w in r_1.

Two of the Tietze option parameters which are listed in section 46.12 may strongly influence the performance and the results of the command `TzSearch`. These are the parameters `saveLimit` and `searchSimultaneous`. The first of them has the following effect:

When `TzSearch` has finished its main loop over all relators, then, in general, there are relators which have changed and hence should be handled again in another run through the whole procedure. However,

experience shows that it really does not pay to continue this way until no more relators change. Therefore, `TzSearch` starts a new loop only if the loop just finished has reduced the total length of the relators by at least `saveLimit` per cent.

The default value of `saveLimit` is 10 per cent.

To understand the effect of the option `searchSimultaneous`, we have to look in more detail at how `TzSearch` proceeds:

First, it sorts the list of relators by increasing lengths. Then it performs a loop over this list. In each step of this loop, the current relator is treated as **short relator** r_1, and a subroutine is called which loops over the succeeding relators, treating them as **long relators** r_2 and performing the respective comparisons and substitutions.

As this subroutine performs a very expensive process, it has been implemented as a C routine in the GAP kernel. For the given relator r_1 of length l_1, say, it first determines the **minimal match length** l which is $l_1/2 + 1$, if l_1 is even, or $(l_1 + 1)/2$, otherwise. Then it builds up a hash list for all subwords of length l occurring in the conjugates of r_1 or r_1^{-1}, and finally it loops over all long relators r_2 and compares the hash values of their subwords of length l against this list. A comparison of subwords which is much more expensive is only done if a hash match has been found.

To improve the efficiency of this process we allow the subroutine to handle several short relators simultaneously provided that they have the same minimal match length. If, for example, it handles n short relators simultaneously, then you save $n - 1$ loops over the long relators r_2, but you pay for it by additional fruitless subword comparisons. In general, you will not get the best performance by always choosing the maximal possible number of short relators to be handled simultaneously. In fact, the optimal choice of the number will depend on the concrete presentation under investigation. You can use the parameter `searchSimultaneous` to prescribe an upper bound for the number of short relators to be handled simultaneously.

The default value of `searchSimultaneous` is 20.

3 ▶ `TzSearchEqual( P )` F

searches for Tietze relator subwords which, in some relator, have a complement of equal length and which occur in other relators, too, and uses them to modify these other relators.

The idea is to find pairs of relators r_1 and r_2 of length l_1 and l_2, respectively, such that l_1 is even, $l_1 \leq l_2$, and r_1 and r_2 coincide (possibly after inverting or conjugating one of them) in some maximal subword w, say, of length at least $l_1/2$. Let l be the length of w. Then, if $l > l_1/2$, the pair is handled as in `TzSearch`. Otherwise, if $l = l_1/2$, then `TzSearchEqual` substitutes each copy of w in r_2 by the inverse complement of w in r_1.

The Tietze option parameter `searchSimultaneous` is used by `TzSearchEqual` in the same way as described for `TzSearch`. However, `TzSearchEqual` does not use the parameter `saveLimit`: The loop over the relators is executed exactly once.

4 ▶ `TzFindCyclicJoins( P )` F

searches for power and commutator relators in order to find pairs of generators which generate a common cyclic subgroup. It uses these pairs to introduce new relators, but it does not introduce any new generators as is done by `TzSubstituteCyclicJoins` (see 46.9.2).

More precisely: `TzFindCyclicJoins` searches for pairs of generators a and b such that (possibly after inverting or conjugating some relators) the set of relators contains the commutator $[a, b]$, a power a^n, and a product of the form $a^s b^t$ with s prime to n. For each such pair, `TzFindCyclicJoins` uses the Euclidian algorithm to express a as a power of b, and then it eliminates a.

46.9 Tietze Transformations that introduce new Generators

Some of the Tietze transformation commands listed so far may eliminate generators and hence change the given presentation to a presentation on a subset of the given set of generators, but they all do **not** introduce new generators. However, sometimes there will be the need to substitute certain words as new generators in order to improve a presentation. Therefore GAP offers the two commands `TzSubstitute` and `TzSubstituteCyclicJoins` which introduce new generators.

1 ▶ `TzSubstitute( P, word )` F
▶ `TzSubstitute( P [, n [, eliminate ] ] )` F

In the first form `TzSubstitute` expects P to be a presentation and *word* to be either an abstract word or a Tietze word in the generators of P. It substitutes the given word as a new generator of P. This is done as follows: First, `TzSubstitute` creates a new abstract generator, g say, and adds it to the presentation, then it adds a new relator $g^{-1} \cdot word$.

In its second form, `TzSubstitute` substitutes a squarefree word of length 2 as a new generator and then eliminates a generator from the extended generator list. We will describe this process in more detail below.

The parameters n and *eliminate* are optional. If you specify arguments for them, then n is expected to be a positive integer, and *eliminate* is expected to be 0, 1, or 2. The default values are $n = 1$ and $eliminate = 0$.

`TzSubstitute` first determines the n most frequently occurring relator subwords of the form $g_1 g_2$, where g_1 and g_2 are different generators or their inverses, and sorts them by decreasing numbers of occurrences.

Let ab be the last word in that list, and let i be the smallest positive integer which has not yet been used as a generator number in the presentation P so far. `TzSubstitute` defines a new abstract generator x_i named `"_x`i`"` and adds it to P (see `AddGenerator`). Then it adds the word $x_i^{-1}ab$ as a new relator to P and replaces all occurrences of ab in the relators by x_i. Finally, it eliminates some suitable generator from P.

The choice of the generator to be eliminated depends on the actual value of the parameter *eliminate*:

If *eliminate* is zero, `TzSubstitute` just calls the function `TzEliminate`. So it may happen that it is the just introduced generator x_i which now is deleted again so that you don't get any remarkable progress in simplifying your presentation. On the first glance this does not look reasonable, but it is a consequence of the request that a call of `TzSubstitute` with $eliminate = 0$ must not increase the total length of the relators.

Otherwise, if *eliminate* is 1 or 2, `TzSubstitute` eliminates the respective factor of the substituted word ab, i. e., it eliminates a if $eliminate = 1$ or b if $eliminate = 2$. In this case, it may happen that the total length of the relators increases, but sometimes such an intermediate extension is the only way to finally reduce a given presentation.

There is still another property of the command `TzSubstitute` which should be mentioned. If, for instance, `word` is an abstract word, a call

```
TzSubstitute( P, word );
```

is more or less equivalent to

```
AddGenerator( P );
g := GeneratorsOfPresentation(P)[Length(GeneratorsOfPresentation(P))];
AddRelator( P, g^-1 * word );
```

However, there is a difference: If you are tracing generator images and preimages of P through the Tietze transformations applied to P (see 46.10), then `TzSubstitute`, as a Tietze transformation of P, will update and save the respective lists, whereas a call of the function `AddGenerator` (which does not perform a Tietze transformation) will delete these lists and hence terminate the tracing.

```
gap> G := PerfectGroup( IsSubgroupFpGroup, 960, 1 );
A5 2^4
gap> P := PresentationFpGroup( G );
<presentation with 6 gens and 21 rels of total length 84>
gap> GeneratorsOfPresentation( P );
[ a, b, s, t, u, v ]
gap> TzGoGo( P );
#I  there are 3 generators and 10 relators of total length 81
#I  there are 3 generators and 10 relators of total length 80
gap> TzPrintGenerators( P );
#I  1.  a   31 occurrences   involution
#I  2.  b   26 occurrences
#I  3.  t   23 occurrences   involution
gap> a := GeneratorsOfPresentation( P )[1];;
gap> b := GeneratorsOfPresentation( P )[2];;
gap> TzSubstitute( P, a*b );
#I  now the presentation has 4 generators, the new generator is _x7
#I  substituting new generator _x7 defined by a*b
#I  there are 4 generators and 11 relators of total length 83
gap> TzGo( P );
#I  there are 3 generators and 10 relators of total length 74
gap> TzPrintGenerators( P );
#I  1.  a   23 occurrences   involution
#I  2.  t   23 occurrences   involution
#I  3.  _x7   28 occurrences
```

As an example of an application of the command `TzSubstitute` in its second form we handle a subgroup of index 266 in the Janko group J_1.

```
gap> F2 := FreeGroup( "a", "b" );;
gap> J1 := F2 / [ F2.1^2, F2.2^3, (F2.1*F2.2)^7,
> Comm(F2.1,F2.2)^10, Comm(F2.1,F2.2^-1*(F2.1*F2.2)^2)^6 ];;
gap> a := J1.1;; b := J1.2;;
gap> H := Subgroup ( J1, [ a, b^(a*b*(a*b^-1)^2) ] );;
gap> P := PresentationSubgroup( J1, H );
<presentation with 23 gens and 82 rels of total length 530>
gap> TzGoGo( P );
#I  there are 3 generators and 47 relators of total length 1368
#I  there are 2 generators and 46 relators of total length 3773
#I  there are 2 generators and 46 relators of total length 2570
gap> TzGoGo( P );
#I  there are 2 generators and 46 relators of total length 2568
gap> TzGoGo( P );
```

Here we do not get any more progress without substituting a new generator.

```
gap> TzSubstitute( P );
#I  substituting new generator _x28 defined by _x6*_x23^-1
#I  eliminating _x28 = _x6*_x23^-1
```

GAP cannot substitute a new generator without extending the total length, so we have to explicitly ask for it by using the second form of the command `TzSubstitute`. Our problem is to chose appropriate values for

the arguments n and *eliminate*. For this purpose it may be helpful to print out a list of the most frequently occurring squarefree relator subwords of length 2.

```
gap> TzPrintPairs( P );
#I  1.  504  occurrences of  _x6 * _x23^-1
#I  2.  504  occurrences of  _x6^-1 * _x23
#I  3.  448  occurrences of  _x6 * _x23
#I  4.  448  occurrences of  _x6^-1 * _x23^-1
gap> TzSubstitute( P, 2, 1 );
#I  substituting new generator _x29 defined by _x6^-1*_x23
#I  eliminating _x6 = _x23*_x29^-1
#I  there are 2 generators and 46 relators of total length 2867
gap> TzGoGo( P );
#I  there are 2 generators and 45 relators of total length 2417
#I  there are 2 generators and 45 relators of total length 2122
gap> TzSubstitute( P, 1, 2 );
#I  substituting new generator _x30 defined by _x23*_x29^-1
#I  eliminating _x29 = _x30^-1*_x23
#I  there are 2 generators and 45 relators of total length 2192
gap> TzGoGo( P );
#I  there are 2 generators and 42 relators of total length 1637
#I  there are 2 generators and 40 relators of total length 1286
#I  there are 2 generators and 36 relators of total length 807
#I  there are 2 generators and 32 relators of total length 625
#I  there are 2 generators and 22 relators of total length 369
#I  there are 2 generators and 18 relators of total length 213
#I  there are 2 generators and 13 relators of total length 141
#I  there are 2 generators and 12 relators of total length 121
#I  there are 2 generators and 10 relators of total length 101
gap> TzPrintPairs( P );
#I  1.  19  occurrences of  _x23 * _x30^-1
#I  2.  19  occurrences of  _x23^-1 * _x30
#I  3.  14  occurrences of  _x23 * _x30
#I  4.  14  occurrences of  _x23^-1 * _x30^-1
```

If we save a copy of the current presentation, then later we will be able to restart the computation from the current state.

```
gap> P1 := ShallowCopy( P );
<presentation with 2 gens and 10 rels of total length 101>
```

Just for demonstration we make an inconvenient choice:

```
gap> TzSubstitute( P, 3, 1 );
#I  substituting new generator _x31 defined by _x23*_x30
#I  eliminating _x23 = _x31*_x30^-1
#I  there are 2 generators and 10 relators of total length 122
gap> TzGoGo( P );
#I  there are 2 generators and 9 relators of total length 105
```

This presentation is worse than the one we have saved, so we restart from that presentation again.

```
gap> P := ShallowCopy( P1 );
<presentation with 2 gens and 10 rels of total length 101>
gap> TzSubstitute( P, 2, 1);
#I  substituting new generator _x31 defined by _x23^-1*_x30
#I  eliminating _x23 = _x30*_x31^-1
#I  there are 2 generators and 10 relators of total length 107
gap> TzGoGo( P );
#I  there are 2 generators and 9 relators of total length 84
#I  there are 2 generators and 8 relators of total length 75
gap> TzSubstitute( P, 2, 1);
#I  substituting new generator _x32 defined by _x30^-1*_x31
#I  eliminating _x30 = _x31*_x32^-1
#I  there are 2 generators and 8 relators of total length 71
gap> TzGoGo( P );
#I  there are 2 generators and 7 relators of total length 56
#I  there are 2 generators and 5 relators of total length 36
gap> TzPrintRelators( P );
#I  1. _x32^5
#I  2. _x31^5
#I  3. _x31^-1*_x32^-1*_x31^-1*_x32^-1*_x31^-1*_x32^-1
#I  4. _x31*_x32*_x31^-1*_x32*_x31^-1*_x32*_x31*_x32^-2
#I  5. _x31^-1*_x32^2*_x31*_x32^-1*_x31^2*_x32^-1*_x31*_x32^2
```

2 ► `TzSubstituteCyclicJoins( P )` F

tries to find pairs of commuting generators a and b, say, such that the exponent of a (i. e. the least currently known positive integer n such that a^n is a relator in P) is prime to the exponent of b. For each such pair, their product ab is substituted as a new generator, and a and b are eliminated.

46.10 Tracing generator images through Tietze transformations

Any sequence of Tietze transformations applied to a presentation, starting from some presentation P_1 and ending up with some presentation P_2, defines an isomorphism, φ say, between the groups defined by P_1 and P_2, respectively. Sometimes it is desirable to know the images of the (old) generators of P_1 or the preimages of the (new) generators of P_2 under φ. The GAP Tietze transformation functions are able to trace these images. This is not automatically done because the involved words may grow to tremendous length, but it will be done if you explicitly request for it by calling the function `TzInitGeneratorImages`.

1 ► `TzInitGeneratorImages( P )` F

expects P to be a presentation. It defines the current generators to be the "old generators" of P and initializes the (pre)image tracing. See `TzImagesOldGens` and `TzPreImagesNewGens` for details.

You can reinitialize the tracing of the generator images at any later state by just calling the function `TzInitGeneratorImages` again.

Note: A subsequent call of the function `DecodeTree` will imply that the images and preimages are deleted and reinitialized after decoding the tree.

Moreover, if you introduce a new generator by calling the function `AddGenerator` described in section 46.6, this new generator cannot be traced in the old generators. Therefore `AddGenerator` will terminate the tracing of the generator images and preimages and delete the respective lists whenever it is called.

2 ► `OldGeneratorsOfPresentation( P )` F

assumes that P is a presentation for which the generator images and preimages are being traced under Tietze transformations. It returns the list of old generators of P.

3 ▶ `TzImagesOldGens( P )` F

assumes that P is a presentation for which the generator images and preimages are being traced under Tietze transformations. It returns a list l of words in the (current) generators `GeneratorsOfPresentation(`P`)` of P such that the i-th word l`[`i`]` represents the i-th old generator `OldGeneratorsOfPresentation(`P`)[`i`]` of P.

4 ▶ `TzPreImagesNewGens( P )` F

assumes that P is a presentation for which the generator images and preimages are being traced under Tietze transformations. It returns a list l of words in the old generators `OldGeneratorsOfPresentation(`P`)` of P such that the i-th word l`[`i`]` represents the i-th (current) generator `GeneratorsOfPresentation(`P`)[`i`]` of P.

5 ▶ `TzPrintGeneratorImages( P )` F

assumes that P is a presentation for which the generator images and preimages are being traced under Tietze transformations. It displays the preimages of the current generators as Tietze words in the old generators, and the images of the old generators as Tietze words in the current generators.

```
gap> G := PerfectGroup( IsSubgroupFpGroup, 960, 1 );
A5 2^4
gap> P := PresentationFpGroup( G );
<presentation with 6 gens and 21 rels of total length 84>
gap> TzInitGeneratorImages( P );
gap> TzGo( P );
#I  there are 3 generators and 11 relators of total length 96
#I  there are 3 generators and 10 relators of total length 81
gap> TzPrintGeneratorImages( P );
#I  preimages of current generators as Tietze words in the old ones:
#I  1. [ 1 ]
#I  2. [ 2 ]
#I  3. [ 4 ]
#I  images of old generators as Tietze words in the current ones:
#I  1. [ 1 ]
#I  2. [ 2 ]
#I  3. [ 1, -2, 1, 3, 1, 2, 1 ]
#I  4. [ 3 ]
#I  5. [ -2, 1, 3, 1, 2 ]
#I  6. [ 1, 3, 1 ]
gap> gens := GeneratorsOfPresentation( P );
[ a, b, t ]
gap> oldgens := OldGeneratorsOfPresentation( P );
[ a, b, s, t, u, v ]
gap> TzImagesOldGens( P );
[ a, b, a*b^-1*a*t*a*b*a, t, b^-1*a*t*a*b, a*t*a ]
gap> for i in [ 1 .. Length( oldgens ) ] do
> Print( oldgens[i], " = ", TzImagesOldGens( P )[i], "\n" );
> od;
a = a
b = b
s = a*b^-1*a*t*a*b*a
t = t
u = b^-1*a*t*a*b
v = a*t*a
```

46.11 DecodeTree

1 ► `DecodeTree( P )` F

assumes that P is a subgroup presentation provided by the Reduced Reidemeister-Schreier or by the Modified Todd-Coxeter method (see `PresentationSubgroupRrs`, `PresentationNormalClosureRrs`, `PresentationSubgroupMtc` in section 46.3). It eliminates the secondary generators of P (see 46.3) by applying the so called "decoding tree" procedure.

`DecodeTree` is called automatically by the command `PresentationSubgroupMtc` (see 46.3.4) where it reduces P to a presentation on the given (primary) subgroup generators.

In order to explain the effect of this command we need to insert a few remarks on the subgroup presentation commands described in section 46.3. All these commands have the common property that in the process of constructing a presentation for a given subgroup H of a finitely presented group G they first build up a highly redundant list of generators of H which consists of an (in general small) list of "primary" generators, followed by an (in general large) list of "secondary" generators, and then construct a presentation P_0, say, **on a sublist of these generators** by rewriting the defining relators of G. This sublist contains all primary, but, at least in general, by far not all secondary generators.

The role of the primary generators depends on the concrete choice of the subgroup presentation command. If the Modified Todd-Coxeter method is used, they are just the given generators of H, whereas in the case of the Reduced Reidemeister-Schreier algorithm they are constructed by the program.

Each of the secondary generators is defined by a word of length two in the preceding generators and their inverses. By historical reasons, the list of these definitions is called the **subgroup generators tree** though in fact it is not a tree but rather a kind of bush.

Now we have to distinguish two cases. If P_0 has been constructed by the Reduced Reidemeister-Schreier routines, it is a presentation of H. However, if the Modified Todd-Coxeter routines have been used instead, then the relators in P_0 are valid relators of H, but they do not necessarily define H. We handle these cases in turn, starting with the latter one.

In fact, we could easily receive a presentation of H also in this case if we extended P_0 by adding to it all the secondary generators which are not yet contained in it and all the definitions from the generators tree as additional generators and relators. Then we could recursively eliminate all secondary generators by Tietze transformations using the new relators. However, this procedure turns out to be too inefficient to be of interest.

Instead, we use the so called **decoding tree** procedure (see [AMW82], [AR84]). It proceeds as follows.

Starting from $P = P_0$, it runs through a number of steps in each of which it eliminates the current "last" generator (with respect to the list of all primary and secondary generators). If the last generator g, say, is a primary generator, then the procedure terminates. Otherwise it checks whether there is a relator in the current presentation which can be used to substitute g by a Tietze transformation. If so, this is done. Otherwise, and only then, the tree definition of g is added to P as a new relator, and the generators involved are added as new generators if they have not yet been contained in P. Subsequently, g is eliminated.

Note that the extension of P by one or two new generators is **not** a Tietze transformation. In general, it will change the isomorphism type of the group defined by P. However, it is a remarkable property of this procedure, that at the end, i.e., as soon as all secondary generators have been eliminated, it provides a presentation $P = P_1$, say, which defines a group isomorphic to H. In fact, it is this presentation which is returned by the command `DecodeTree` and hence by the command `PresentationSubgroupMtc`.

If, in the other case, the presentation P_0 has been constructed by the Reduced Reidemeister-Schreier algorithm, then P_0 itself is a presentation of H, and the corresponding subgroup presentation command (`PresentationSubgroupRrs` or `PresentationNormalClosureRrs`) just returns P_0.

As mentioned in section 46.3, we recommend to further simplify this presentation before you use it. The standard way to do this is to start from P_0 and to apply suitable Tietze transformations, e.g., by calling

the commands TzGo or TzGoGo. This is probably the most efficient approach, but you will end up with a presentation on some unpredictable set of generators. As an alternative, GAP offers you the DecodeTree command which you can use to eliminate all secondary generators (provided that there are no space or time problems). For this purpose, the subgroup presentation commands do not only return the resulting presentation, but also the tree (together with some associated lists) as a kind of side result in a component P!.tree of the resulting presentation P.

Note, however, that the decoding tree routines will not work correctly any more on a presentation from which generators have already been eliminated by Tietze transformations. Therefore, to prevent you from getting wrong results by calling the DecodeTree command in such a situation, GAP will automatically remove the subgroup generators tree from a presentation as soon as one of the generators is substituted by a Tietze transformation.

Nevertheless, a certain misuse of the command is still possible, and we want to explicitly warn you from this. The reason is that the Tietze option parameters described in section 46.7 apply to the DecodeTree command as well. Hence, in case of inadequate values of these parameters, it may happen that the DecodeTree routine stops before all the secondary generators have vanished. In this case GAP will display an appropriate warning. Then you should change the respective parameters and continue the process by calling the DecodeTree command again. Otherwise, if you would apply Tietze transformations, it might happen because of the convention described above that the tree is removed and that you end up with a wrong presentation.

After a successful run of the DecodeTree command it is convenient to further simplify the resulting presentation by suitable Tietze transformations.

As an example of an explicit call of the DecodeTree command we compute two presentations of a subgroup of order 384 in a group of order 6912. In both cases we use the Reduced Reidemeister-Schreier algorithm, but in the first run we just apply the Tietze transformations offered by the TzGoGo command with its default parameters, whereas in the second run we call the DecodeTree command before.

```
gap> F2 := FreeGroup( "a", "b" );;
gap> G := F2 / [ F2.1*F2.2^2*F2.1^-1*F2.2^-1*F2.1^3*F2.2^-1,
>                F2.2*F2.1^2*F2.2^-1*F2.1^-1*F2.2^3*F2.1^-1 ];;
gap> a := G.1;;  b := G.2;;
gap> H := Subgroup( G, [ Comm(a^-1,b^-1), Comm(a^-1,b), Comm(a,b) ] );;
```

We use the Reduced Reidemeister Schreier method and default Tietze transformations to get a presentation for H.

```
gap> P := PresentationSubgroupRrs( G, H );
<presentation with 18 gens and 35 rels of total length 169>
gap> TzGoGo( P );
#I  there are 3 generators and 20 relators of total length 488
#I  there are 3 generators and 20 relators of total length 466
```

We end up with 20 relators of total length 466. Now we repeat the procedure, but we call the decoding tree algorithm before doing the Tietze transformations.

```
gap> P := PresentationSubgroupRrs( G, H );
<presentation with 18 gens and 35 rels of total length 169>
gap> DecodeTree( P );
#I  there are 9 generators and 26 relators of total length 185
#I  there are 6 generators and 23 relators of total length 213
#I  there are 3 generators and 20 relators of total length 252
#I  there are 3 generators and 20 relators of total length 244
gap> TzGoGo( P );
#I  there are 3 generators and 19 relators of total length 168
```

```
#I  there are 3 generators and 17 relators of total length 138
#I  there are 3 generators and 15 relators of total length 114
#I  there are 3 generators and 13 relators of total length 96
#I  there are 3 generators and 12 relators of total length 84
```

This time we end up with a shorter presentation.

As an example of an implicit call of the function `DecodeTree` via the command `PresentationSubgroupMtc` we handle a subgroup of index 240 in a group of order 40320 given by a presentation due to B. H. Neumann. Note that we increase the FpGroup info level to get some additional output.

```
gap> F3 := FreeGroup( "a", "b", "c" );;
gap> a := F3.1;;  b := F3.2;;  c := F3.3;;
gap> G := F3 / [ a^3, b^3, c^3, (a*b)^5, (a^-1*b)^5, (a*c)^4,
>     (a*c^-1)^4, a*b^-1*a*b*c^-1*a*c*a*c^-1, (b*c)^3, (b^-1*c)^4 ];;
gap> a := G.1;;  b := G.2;;  c := G.3;;
gap> H := Subgroup( G, [ a, c ] );;
gap> SetInfoLevel( InfoFpGroup, 1 );
gap> P := PresentationSubgroupMtc( G, H );;
#I  index = 240  total = 4737  max = 4507
#I  MTC defined 2 primary and 4444 secondary subgroup generators
#I  there are 246 generators and 617 relators of total length 2893
#I  calling DecodeTree
#I  there are 114 generators and 385 relators of total length 1860
#I  there are 69 generators and 294 relators of total length 1855
#I  there are 43 generators and 235 relators of total length 2031
#I  there are 35 generators and 207 relators of total length 2348
#I  there are 25 generators and 181 relators of total length 3055
#I  there are 19 generators and 165 relators of total length 3290
#I  there are 20 generators and 160 relators of total length 5151
#I  there are 23 generators and 159 relators of total length 8177
#I  there are 25 generators and 159 relators of total length 12241
#I  there are 29 generators and 159 relators of total length 18242
#I  there are 34 generators and 159 relators of total length 27364
#I  there are 38 generators and 159 relators of total length 41480
#I  there are 41 generators and 159 relators of total length 62732
#I  there are 45 generators and 159 relators of total length 88872
#I  there are 46 generators and 159 relators of total length 111092
#I  there are 44 generators and 155 relators of total length 158181
#I  there are 32 generators and 155 relators of total length 180478
#I  there are 7 generators and 133 relators of total length 29897
#I  there are 4 generators and 119 relators of total length 28805
#I  there are 3 generators and 116 relators of total length 35209
#I  there are 2 generators and 111 relators of total length 25658
#I  there are 2 generators and 111 relators of total length 22634
gap> TzGoGo( P );
#I  there are 2 generators and 108 relators of total length 11760
#I  there are 2 generators and 95 relators of total length 6482
#I  there are 2 generators and 38 relators of total length 1464
#I  there are 2 generators and 8 relators of total length 116
#I  there are 2 generators and 7 relators of total length 76
#I  there are 2 generators and 6 relators of total length 66
#I  there are 2 generators and 6 relators of total length 52
```

```
gap> TzPrintGenerators( P );
#I  1.  _x1   26 occurrences
#I  2.  _x2   26 occurrences
gap> TzPrint( P );
#I  generators: [ _x1, _x2 ]
#I  relators:
#I  1.  3  [ 1, 1, 1 ]
#I  2.  3  [ 2, 2, 2 ]
#I  3.  8  [ 2, -1, 2, -1, 2, -1, 2, -1 ]
#I  4.  8  [ 2, 1, 2, 1, 2, 1, 2, 1 ]
#I  5.  14  [ -1, -2, 1, 2, 1, -2, -1, 2, 1, -2, -1, -2, 1, 2 ]
#I  6.  16  [ 1, 2, 1, -2, 1, 2, 1, -2, 1, 2, 1, -2, 1, 2, 1, -2 ]
gap> K :=  FpGroupPresentation( P );
<fp group on the generators [ _x1, _x2 ]>
gap> SetInfoLevel( InfoFpGroup, 0 );
gap> Size( K );
168
```

46.12 Tietze Options

Several of the Tietze transformation commands described above are controlled by certain parameters, the **Tietze options**, which often have a tremendous influence on their performance and results. However, in each application of the commands, an appropriate choice of these option parameters will depend on the concrete presentation under investigation. Therefore we have implemented the Tietze options in such a way that they are associated to the presentation: Each presentation keeps its own set of Tietze option parameters as an attribute.

1 ▶ `TzOptions( P )` AM

is a record whose components direct the heuristics applied by the Tietze transformation functions.

You may alter the value of any of these Tietze options by just assigning a new value to the respective record component.

The following Tietze options are recognized by GAP:

`protected`:
: The first `protected` generators in a presentation P are protected from being eliminated by the Tietze transformations functions. There are only two exceptions: The option `protected` is ignored by the functions `TzEliminate(`P`,`*gen*`)` and `TzSubstitute(`P`,`n`,`*eliminate*`)` because they explicitly specify the generator to be eliminated. The default value of `protected` is 0.

`eliminationsLimit`:
: Whenever the elimination phase of the `TzGo` command is entered for a presentation P, then it will eliminate at most `eliminationsLimit` generators (except for further ones which have turned out to be trivial). Hence you may use the `eliminationsLimit` parameter as a break criterion for the `TzGo` command. Note, however, that it is ignored by the `TzEliminate` command. The default value of `eliminationsLimit` is 100.

`expandLimit`:
: Whenever the routine for eliminating more than 1 generators is called for a presentation P by the `TzEliminate` command or the elimination phase of the `TzGo` command, then it saves the given total length of the relators, and subsequently it checks the current total length against its value before each elimination. If the total length has increased to more than `expandLimit` per cent of its original value, then the routine returns instead of eliminating another generator. Hence you may

use the `expandLimit` parameter as a break criterion for the `TzGo` command. The default value of `expandLimit` is 150.

`generatorsLimit`:
: Whenever the elimination phase of the `TzGo` command is entered for a presentation P with n generators, then it will eliminate at most $n-$`generatorsLimit` generators (except for generators which turn out to be trivial). Hence you may use the `generatorsLimit` parameter as a break criterion for the `TzGo` command. The default value of `generatorsLimit` is 0.

`lengthLimit`:
: The Tietze transformation commands will never eliminate a generator of a presentation P, if they cannot exclude the possibility that the resulting total length of the relators exceeds the maximal GAP list length of $2^{31}-1$ or the value of the option `lengthLimit`. The default value of `lengthLimit` is $2^{31}-1$.

`loopLimit`:
: Whenever the `TzGo` command is called for a presentation P, then it will loop over at most `loopLimit` of its basic steps. Hence you may use the `loopLimit` parameter as a break criterion for the `TzGo` command. The default value of `loopLimit` is `infinity`.

`printLevel`:
: Whenever Tietze transformation commands are called for a presentation P with `printLevel` $= 0$, they will not provide any output except for error messages. If `printLevel` $= 1$, they will display some reasonable amount of output which allows you to watch the progress of the computation and to decide about your next commands. In the case `printLevel` $= 2$, you will get a much more generous amount of output. Finally, if `printLevel` $= 3$, various messages on internal details will be added. The default value of `printLevel` is 1.

`saveLimit`:
: Whenever the `TzSearch` command has finished its main loop over all relators of a presentation P, then it checks whether during this loop the total length of the relators has been reduced by at least `saveLimit` per cent. If this is the case, then `TzSearch` repeats its procedure instead of returning. Hence you may use the `saveLimit` parameter as a break criterion for the `TzSearch` command and, in particular, for the search phase of the `TzGo` command. The default value of `saveLimit` is 10.

`searchSimultaneous`:
: Whenever the `TzSearch` or the `TzSearchEqual` command is called for a presentation P, then it is allowed to handle up to `searchSimultaneous` short relators simultaneously (see for the description of the `TzSearch` command for more details). The choice of this parameter may heavily influence the performance as well as the result of the `TzSearch` and the `TzSearchEqual` commands and hence also of the search phase of the `TzGo` command. The default value of `searchSimultaneous` is 20.

2 ▶ `TzPrintOptions(` P `)` F

prints the current values of the Tietze options of the presentation P.

```
gap> TzPrintOptions( P );
#I  protected          = 0
#I  eliminationsLimit  = 100
#I  expandLimit        = 150
#I  generatorsLimit    = 0
#I  lengthLimit        = 2147483647
#I  loopLimit          = infinity
#I  printLevel         = 1
#I  saveLimit          = 10
#I  searchSimultaneous = 20
```

47 Group Products

This chapter describes the various group product constructions that are possible in GAP.

At the moment for some of the products methods are available only if both factors are given in the same representation or only for certain types of groups such as permutation groups and pc groups when the product can be naturally represented as a group of the same kind.

GAP does not guarantee that a product of two groups will be in a particular representation. (Exceptions are `WreathProductImprimitiveAction` and `WreathProductProductAction` which are construction that makes sense only for permutation groups, see 47.4.1).

GAP however will try to choose an efficient representation, so products of permutation groups or pc groups often will be represented as a group of the same kind again.

Therefore the only guaranteed way to relate a product to its factors is via the embedding and projection homomorphisms (see 47.5);

47.1 Direct Products

The direct product of groups is the cartesian product of the groups (considered as element sets) with component-wise multiplication.

1 ▶ `DirectProduct(` *G*, *H* `)` F
▶ `DirectProductOp(` *list*, *expl* `)` O

These functions construct the direct product of the groups given as arguments. `DirectProduct` takes an arbitrary positive number of arguments and calls the operation `DirectProductOp`, which takes exactly two arguments, namely a nonempty list of groups and one of these groups. (This somewhat strange syntax allows the method selection to choose a reasonable method for special cases, e.g., if all groups are permutation groups or pc groups.)

GAP will try to choose an efficient representation for the direct product. For example the direct product of permutation groups will be a permutation group again and the direct product of pc groups will be a pc group.

For a product P the operation `Embedding(`P`,`nr`)` returns the homomorphism embedding the nr-th factor into P. The operation `Projection(`P`,`nr`)` gives the projection of P onto the nr-th factor (see 47.5).

```
gap> g:=Group((1,2,3),(1,2));;
gap> d:=DirectProduct(g,g,g);
Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8) ])
gap> Size(d);
216
gap> e:=Embedding(d,2);
2nd embedding into Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8) ])
gap> Image(e,(1,2));
(4,5)
gap> Image(Projection(d,2),(1,2,3)(4,5)(8,9));
(1,2)
```

47.2 Semidirect Products

The semidirect product of a group N with a group G acting on N via a homomorphism α from G into the automorphism group of N is the cartesian product $G \times N$ with the multiplication $(g, n) \cdot (h, m) = (gh, n^{(h\alpha)}m)$.

1 ▶ `SemidirectProduct( G, alpha, N )` O
▶ `SemidirectProduct( autgp, N )` O

constructs the semidirect product of N with G acting via *alpha*. *alpha* must be a homomorphism from G into a group of automorphisms of N.

If N is a group, *alpha* must be a homomorphism from G into a group of automorphisms of N.

If N is a full row space over a field F, *alpha* must be a homomorphism from G into a matrix group of the right dimension over a subfield of F, or into a permutation group (in this case permutation matrices are taken).

In the second variant, *autgp* must be a group of automorphism of N, it is a shorthand for `SemidirectProduct(`*autgp*`,IdentityMapping(`*autgp*`),`N`)`. Note that (unless *autgrp* has been obtained by the operation `AutomorphismGroup`) you have to test `IsGroupOfAutomorphisms(`*autgrp*`)` to ensure that GAP knows that *autgrp* consists of group automorphisms.

```
gap> n:=AbelianGroup(IsPcGroup,[5,5]);
<pc group of size 25 with 2 generators>
gap> au:=DerivedSubgroup(AutomorphismGroup(n));;
gap> Size(au);
120
gap> p:=SemidirectProduct(au,n);
<permutation group with 4 generators>
gap> Size(p);
3000

gap> n:=Group((1,2),(3,4));;
gap> au:=AutomorphismGroup(n);;
gap> au:=First(Elements(au),i->Order(i)=3);;
gap> au:=Group(au);
<group with 1 generators>
gap> SemidirectProduct(au,n);
Error, no method found! For debugging hints type ?Recovery from NoMethodFound
Error, no 2nd choice method found for 'IsomorphismPcGroup' on 1 arguments
gap> IsGroupOfAutomorphisms(au);
true
gap> SemidirectProduct(au,n);
<pc group with 3 generators>

gap> n:=AbelianGroup(IsPcGroup,[2,2]);
<pc group of size 4 with 2 generators>
gap> au:=AutomorphismGroup(n);
<group with 4 generators>
gap> apc:=IsomorphismPcGroup(au);
CompositionMapping( Pcgs([ (2,3), (1,3,2) ]) ->
[ f1, f2 ], <action isomorphism> )
gap> g:=Image(apc);
Group([ f1, f2 ])
```

```
gap> apci:=InverseGeneralMapping(apc);
[ f1*f2^2, f1*f2^2, f1*f2, <identity> of ... ] ->
[ [ f1, f2 ] -> [ f2, f1 ], [ f1, f2 ] -> [ f2, f1 ],
  [ f1, f2 ] -> [ f1*f2, f2 ], [ f1, f2 ] -> [ f1, f2 ] ]
gap> IsGroupHomomorphism(apci);
true
gap> p:=SemidirectProduct(g,apci,n);
<pc group of size 24 with 4 generators>
gap> IsomorphismGroups(p,Group((1,2,3,4),(1,2)));
[ f1, f2, f3, f4 ] -> [ (1,2), (1,2,4), (1,2)(3,4), (1,4)(2,3) ]

gap> SemidirectProduct(SU(3,3),GF(9)^3);
<matrix group of size 4408992 with 3 generators>
gap> SemidirectProduct(Group((1,2,3),(2,3,4)),GF(5)^4);
<matrix group of size 7500 with 3 generators>

gap> g:=Group((3,4,5),(1,2,3));;
gap> mats:=[[[Z(2^2),0*Z(2)],[0*Z(2),Z(2^2)^2]],
>          [[Z(2)^0,Z(2)^0], [Z(2)^0,0*Z(2)]]];;
gap> hom:=GroupHomomorphismByImages(g,Group(mats),[g.1,g.2],mats);;
gap> SemidirectProduct(g,hom,GF(4)^2);
<matrix group of size 960 with 3 generators>
gap> SemidirectProduct(g,hom,GF(16)^2);
<matrix group of size 15360 with 4 generators>
```

For the semidirect product P of G with N, `Embedding(`P`,1)` embeds G, `Embedding(`P`,2)` embeds N. The operation `Projection(`P`)` returns the projection of P onto G (see 47.5).

```
gap> Size(Image(Embedding(p,1)));
6
gap> Embedding(p,2);
[ f1, f2 ] -> [ f3, f4 ]
gap> Projection(p);
[ f1, f2, f3, f4 ] -> [ f1, f2, <identity> of ..., <identity> of ... ]
```

47.3 Subdirect Products

The subdirect product of the groups G and H with respect to the epimorphisms $\varphi\colon G \to A$ and $\psi\colon H \to A$ (for a common group A) is the subgroup of the direct product $G \times H$ consisting of the elements (g, h) for which $g\varphi = h\psi$. It is the pull-back of the diagram:

$$\begin{array}{ccc} & & G \\ & & \big\downarrow \varphi \\ H & \xrightarrow{\psi} & A \end{array}$$

1 ▶ `SubdirectProduct(` *G* `,` *H*`,` *Ghom*`,` *Hhom* `)` O

constructs the subdirect product of G and H with respect to the epimorphisms *Ghom* from G onto a group A and *Hhom* from H onto the same group A.

For a subdirect product P, the operation `Projection(`P`,`nr returns the projections on the nr-th factor. (In general the factors do not embed in a subdirect product.)

```
gap> g:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> hom:=GroupHomomorphismByImagesNC(g,g,[(1,2,3),(1,2)],[(),(1,2)]);
[ (1,2,3), (1,2) ] -> [ (), (1,2) ]
gap> s:=SubdirectProduct(g,g,hom,hom);
Group([ (1,2,3), (1,2)(4,5), (4,5,6) ])
gap> Size(s);
18
gap> p:=Projection(s,2);
2nd projection of Group([ (1,2,3), (1,2)(4,5), (4,5,6) ])
gap> Image(p,(1,3,2)(4,5,6));
(1,2,3)
```

2 ▶ `SubdirectProducts( G, H )` F

this function computes all subdirect products of G and H up to conjugacy in Parent(G) x Parent(H). The subdirect products are returned as subgroups of this direct product.

47.4 Wreath Products

The wreath product of a group G with a permutation group P acting on n points is the semidirect product of the normal subgroup G^n with the group P which acts on G^n by permuting the components.

1 ▶ `WreathProduct( G, P )` O
▶ `WreathProduct( G, H [, hom] )` O

constructs the wreath product of the group G with the permutation group P (acting on its `MovedPoints`).

The second usage constructs the wreath product of the group G with the image of the group H under *hom* where *hom* must be a homomorphism from H into a permutation group. (If *hom* is not given, and P is not a permutation group the result of `IsomorphismPermGroup(P)` – whose degree may be dependent on the method and thus is not well-defined! – is taken for *hom*).

For a wreath product W of G with a permutation group P of degree n and $1 \leq nr \leq n$ the operation `Embedding(`W`,`$nr`)` provides the embedding of G in the nr-th component of the direct product of the base group G^n of W. `Embedding(`W`,`n`+1)` is the embedding of P into W. The operation `Projection(`W`)` provides the projection onto the acting group P (see 47.5).

```
gap> g:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> p:=Group((1,2,3));
Group([ (1,2,3) ])
gap> w:=WreathProduct(g,p);
Group([ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8), (1,4,7)(2,5,8)(3,6,9)
 ])
gap> Size(w);
648
gap> Embedding(w,1);
1st embedding into Group( [ (1,2,3), (1,2), (4,5,6), (4,5), (7,8,9), (7,8),
  (1,4,7)(2,5,8)(3,6,9) ] )
gap> Image(Embedding(w,3));
Group([ (7,8,9), (7,8) ])
gap> Image(Embedding(w,4));
Group([ (1,4,7)(2,5,8)(3,6,9) ])
gap> Image(Projection(w),(1,4,8,2,6,7,3,5,9));
```

```
(1,2,3)
```

2 ▶ `WreathProductImprimitiveAction( G, H )` F

for two permutation groups G and H this function constructs the wreath product of G and H in the imprimitive action. If G acts on l points and H on m points this action will be on $l \cdot m$ points, it will be imprimitive with m blocks of size l each.

The operations `Embedding` and `Projection` operate on this product as described for general wreath products.

```
gap> w:=WreathProductImprimitiveAction(g,p);;
gap> LargestMovedPoint(w);
9
```

3 ▶ `WreathProductProductAction( G, H )` F

for two permutation groups G and H this function constructs the wreath product in product action. If G acts on l points and H on m points this action will be on l^m points.

The operations `Embedding` and `Projection` operate on this product as described for general wreath products.

```
gap> w:=WreathProductProductAction(g,p);
<permutation group of size 648 with 7 generators>
gap> LargestMovedPoint(w);
27
```

4 ▶ `KuKGenerators( G, beta, alpha )` F

If *beta* is a homomorphism from G in a transitive permutation group, U the full preimage of the point stabilizer and and *alpha* a homomorphism defined on (a superset) of U, this function returns images of the generators of G when mapping to the wreath product $(U\,alpha) \wr (G\,beta)$. (This is the Krasner-Kaloujnine embedding theorem.)

```
gap> g:=Group((1,2,3,4),(1,2));;
gap> hom:=GroupHomomorphismByImages(g,Group((1,2)),
> GeneratorsOfGroup(g),[(1,2),(1,2)]);;
gap> u:=PreImage(hom,Stabilizer(Image(hom),1));
Group([ (2,3,4), (1,2,4) ])
gap> hom2:=GroupHomomorphismByImages(u,Group((1,2,3)),
> GeneratorsOfGroup(u),[ (1,2,3), (1,2,3) ]);;
gap> KuKGenerators(g,hom,hom2);
[ (1,4)(2,5)(3,6), (1,6)(2,4)(3,5) ]
```

47.5 Embeddings and Projections for Group Products

The relation between a group product and its factors is provided via homomorphisms, the embeddings in the product and the projections from the product. Depending on the kind of product only some of these are defined.

1 ▶ `Embedding(P,nr)` O

returns the nr-th embedding in the group product P. The actual meaning of this embedding is described in the section for the appropriate product.

2 ▶ `Projection(P[,nr])` O

returns the (nr-th) projection of the group product P. The actual meaning of the projection returned is described in the section for the appropriate product.

48 Group Libraries

When you start GAP, it already knows several groups. Currently GAP initially knows the following groups:

- some basic groups, such as cyclic groups or symmetric groups (see 48.1),
- Classical matrix groups (see 48.2),
- the transitive permutation groups of degree at most 30 (see 48.6),
- a library of groups of small order (see 48.7),
- the finite perfect groups of size at most 10^6 (excluding 11 sizes) (see 48.8).
- a library of primitive permutation groups of degree < 1000, complete for degrees < 256 (see 48.9),
- the irreducible solvable subgroups of $GL(n, p)$ for $n > 1$ and $p^n < 256$ (see 48.11),
- the irreducible maximal finite integral matrix groups of dimension at most 31 (see 48.12),

There is usually no relation between the groups in the different libraries and a group may occur in different libraries in different incarnations.

Note that a system administrator may choose to install all, or only a few, or even none of the libraries. So some of the libraries mentioned below may not be available on your installation.

48.1 Basic Groups

There are several infinite families of groups which are parametrized by numbers. GAP provides various functions to construct these groups. The functions always permit (but do not require) one to indicate a filter (see 13.2), for example `IsPermGroup`, `IsMatrixGroup` or `IsPcGroup`, in which the group shall be constructed. There always is a default filter corresponding to a "natural" way to describe the group in question. Note that not every group can be constructed in every filter, there may be theoretical restrictions (`IsPcGroup` only works for solvable groups) or methods may be available only for a few filters.

Certain filters may admit additional hints. For example, groups constructed in `IsMatrixGroup` may be constructed over a specified field, which can be given as second argument of the function that constructs the group; The default field is `Rationals`.

1 ▶ `TrivialGroup( [`*filter*`] )` F

constructs a trivial group in the category given by the filter *filter*. If *filter* is not given it defaults to `IsPcGroup`.

```
gap> TrivialGroup();
<pc group of size 1 with 0 generators>
gap> TrivialGroup( IsPermGroup );
Group(())
```

2 ▶ `CyclicGroup( [`*filt*`, ]`*n*` )` F

constructs the cyclic group of size n in the category given by the filter *filt*. If *filt* is not given it defaults to `IsPcGroup`.

```
gap> CyclicGroup(12);
<pc group of size 12 with 3 generators>
gap> CyclicGroup(IsPermGroup,12);
Group([ (1,2,3,4,5,6,7,8,9,10,11,12) ])
gap> matgrp1:= CyclicGroup( IsMatrixGroup, 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp1 );
Rationals
gap> matgrp2:= CyclicGroup( IsMatrixGroup, GF(2), 12 );
<matrix group of size 12 with 1 generators>
gap> FieldOfMatrixGroup( matgrp2 );
GF(2)
```

3 ▶ `AbelianGroup( [`*filt*`, ]`*ints*` )` F

constructs an abelian group in the category given by the filter *filt* which is of isomorphism type $C_{ints[1]} * C_{ints[2]} * \ldots * C_{ints[n]}$. *ints* must be a list of positive integers. If *filt* is not given it defaults to `IsPcGroup`. The generators of the group returned are the elements corresponding to the integers in *ints*.

```
gap> AbelianGroup([1,2,3]);
<pc group of size 6 with 3 generators>
```

4 ▶ `ElementaryAbelianGroup( [`*filt*`, ]`*n*` )` F

constructs the elementary abelian group of size n in the category given by the filter *filt*. If *filt* is not given it defaults to `IsPcGroup`.

```
gap> ElementaryAbelianGroup(8192);
<pc group of size 8192 with 13 generators>
```

5 ▶ `DihedralGroup( [`*filt*`, ]`*n*` )` F

constructs the dihedral group of size n in the category given by the filter *filt*. If *filt* is not given it defaults to `IsPcGroup`.

```
gap> DihedralGroup(10);
<pc group of size 10 with 2 generators>
```

6 ▶ `ExtraspecialGroup( [`*filt*`, ]`*order*`,` *exp*` )` F

Let *order* be of the form p^{2n+1}, for a prime integer p and a positive integer n. `ExtraspecialGroup` returns the extraspecial group of order *order* that is determined by *exp*, in the category given by the filter *filt*.

If p is odd then admissible values of *exp* are the exponent of the group (either p or p^2) or one of `'+'`, `"+"`, `'-'`, `"-"`. For $p = 2$, only the above plus or minus signs are admissible.

If *filt* is not given it defaults to `IsPcGroup`.

```
gap> ExtraspecialGroup( 27, 3 );
<pc group of size 27 with 3 generators>
gap> ExtraspecialGroup( 27, '+' );
<pc group of size 27 with 3 generators>
gap> ExtraspecialGroup( 8, "-" );
<pc group of size 8 with 3 generators>
```

7 ▶ `AlternatingGroup( [`*filt*`, ]`*deg*` )` F
▶ `AlternatingGroup( [`*filt*`, ]`*dom*` )` F

constructs the alternating group of degree *deg* in the category given by the filter *filt*. If *filt* is not given it defaults to `IsPermGroup`. In the second version, the function constructs the alternating group on the points given in the set *dom* which must be a set of positive integers.

```
gap> AlternatingGroup(5);
Alt( [ 1 .. 5 ] )
```

8 ▶ `SymmetricGroup( [`*filt*`, ]`*deg*` )` F
▶ `SymmetricGroup( [`*filt*`, ]`*dom*` )` F

constructs the symmetric group of degree *deg* in the category given by the filter *filt*. If *filt* is not given it defaults to `IsPermGroup`. In the second version, the function constructs the symmetric group on the points given in the set *dom* which must be a set of positive integers.

```
gap> SymmetricGroup(10);
Sym( [ 1 .. 10 ] )
```

Note that permutation groups provide special treatment of symmetric and alternating groups, see 41.3.

9 ▶ `MathieuGroup( [`*filt*`, ]`*degree*` )` F

constructs the Mathieu group of degree *degree* in the category given by the filter *filt*, where *degree* must be in $\{9, 10, 11, 12, 21, 22, 23, 24\}$. If *filt* is not given it defaults to `IsPermGroup`.

```
gap> MathieuGroup( 11 );
Group([ (1,2,3,4,5,6,7,8,9,10,11), (3,7,11,8)(4,10,5,6) ])
```

10 ▶ `SuzukiGroup( [`*filt*`, ]` *q* `)` F
▶ `Sz( [`*filt*`, ]` *q* `)` F

Constructs a group isomorphic to the Suzuki group Sz(*q*) over the field with q elements, where q is a non-square power of 2.

If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the Suzuki group itself.

```
gap> SuzukiGroup( 32 );
Sz(32)
```

11 ▶ `ReeGroup( [`*filt*`, ]` *q* `)` F
▶ `Ree( [`*filt*`, ]` *q* `)` F

Constructs a group isomorphic to the Ree group $2G2(q)$ where $q = 3^{1+2m}$ for m a non-negative integer.

If *filt* is not given it defaults to `IsMatrixGroup` and the generating matrices are based on [KLM01]. (No particular choice of a generating set is guaranteed.)

```
gap> ReeGroup( 27 );
Ree(27)
```

48.2 Classical Groups

The following functions return classical groups. For the linear, symplectic, and unitary groups (the latter in dimension at least 3), the generators are taken from [Tay87]; for the unitary groups in dimension 2, the isomorphism of $SU(2, q)$ and $SL(2, q)$ is used, see for example [Hup67]. The generators of the orthogonal groups are taken from [IE94] and [KL90], except that the generators of the orthogonal groups in odd dimension in even characteristic are constructed via the isomorphism to a symplectic group, see for example [Car72].

For symplectic and orthogonal matrix groups returned by the functions described below, the invariant bilinear form is stored as the value of the attribute `InvariantBilinearForm` (see 42.4.1). Analogously, the invariant sesquilinear form defining the unitary groups is stored as the value of the attribute `InvariantSesquilinearForm` (see 42.4.3). The defining quadratic form of orthogonal groups is stored as the value of the attribute `InvariantQuadraticForm` (see 42.4.5).

1 ▶ `GeneralLinearGroup( [`*filt*`, ]`*d*`, `*R*` )` F
▶ `GL( [`*filt*`, ]`*d*`, `*R*` )` F
▶ `GeneralLinearGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `GL( [`*filt*`, ]`*d*`, `*q*` )` F

The first two forms construct a group isomorphic to the general linear group GL(d, R) of all $d \times d$ matrices that are invertible over the ring R, in the category given by the filter *filt*.

The third and the fourth form construct the general linear group over the finite field with q elements.

If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the general linear group as a matrix group in its natural action (see also 42.3.2, 42.5.4).

Currently supported rings R are finite fields, the ring `Integers`, and residue class rings `Integers mod` m.

```
gap> GL(4,3);
GL(4,3)
gap> GL(2,Integers);
GL(2,Integers)
gap> GL(3,Integers mod 12);
GL(3,Z/12Z)
```

2 ▶ `SpecialLinearGroup( [`*filt*`, ]`*d*`, `*R*` )` F
▶ `SL( [`*filt*`, ]`*d*`, `*R*` )` F
▶ `SpecialLinearGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `SL( [`*filt*`, ]`*d*`, `*q*` )` F

The first two forms construct a group isomorphic to the special linear group SL(d, R) of all those $d \times d$ matrices over the ring R whose determinant is the identity of R, in the category given by the filter *filt*.

The third and the fourth form construct the special linear group over the finite field with q elements.

If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the special linear group as a matrix group in its natural action (see also 42.3.4, 42.5.5).

Currently supported rings R are finite fields, the ring `Integers`, and residue class rings `Integers mod` m.

```
gap> SpecialLinearGroup(2,2);
SL(2,2)
gap> SL(3,Integers);
SL(3,Integers)
gap> SL(4,Integers mod 4);
SL(4,Z/4Z)
```

Using the `OnLines` operation it is possible to obtain the corresponding projective groups in a permutation action:

```
gap> g:=GL(4,3);;Size(g);
24261120
gap> pgl:=Action(g,Orbit(g,Z(3)^0*[1,0,0,0],OnLines),OnLines);;
gap> Size(pgl);
12130560
```

3 ▶ `GeneralUnitaryGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `GU( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the general unitary group GU(d, q) of those $d \times d$ matrices over the field with q^2 elements that respect a fixed nondegenerate sesquilinear form, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the general unitary group itself.

```
gap> GeneralUnitaryGroup( 3, 5 );
GU(3,5)
```

4 ▶ `SpecialUnitaryGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `SU( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the special unitary group `GU(`*d*`, `*q*`)` of those $d \times d$ matrices over the field with q^2 elements whose determinant is the identity of the field and that respect a fixed nondegenerate sesquilinear form, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the special unitary group itself.

```
gap> SpecialUnitaryGroup( 3, 5 );
SU(3,5)
```

5 ▶ `SymplecticGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `Sp( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `SP( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the symplectic group Sp(d, q) of those $d \times d$ matrices over the field with q elements that respect a fixed nondegenerate symplectic form, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the symplectic group itself.

```
gap> SymplecticGroup( 4, 2 );
Sp(4,2)
```

6 ▶ `GeneralOrthogonalGroup( [`*filt*`, ][`*e*`, ]`*d*`, `*q*` )` F
▶ `GO( [`*filt*`, ][`*e*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the general orthogonal group GO(e, d, q) of those $d \times d$ matrices over the field with q elements that respect a non-singular quadratic form (see 42.4.5) specified by e, in the category given by the filter *filt*.

The value of e must be 0 for odd d (and can optionally be omitted in this case), respectively one of 1 or -1 for even d. If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the general orthogonal group itself.

Note that in [KL90], GO is defined as the stabilizer $\Delta(V, F, \kappa)$ of the quadratic form, up to scalars, whereas our GO is called $I(V, F, \kappa)$ there.

7 ▶ `SpecialOrthogonalGroup( [`*filt*`, ][`*e*`, ]`*d*`, `*q*` )` F
▶ `SO( [`*filt*`, ][`*e*`, ]`*d*`, `*q*` )` F

`SpecialOrthogonalGroup` returns a group isomorphic to the special orthogonal group SO(e, d, q), which is the subgroup of all those matrices in the general orthogonal group (see 48.2.6) that have determinant one, in the category given by the filter *filt*. (The index of SO(e, d, q) in GO(e, d, q) is 2 if q is odd, and 1 if q is even.)

If *filt* is not given it defaults to `IsMatrixGroup`, and the returned group is the special orthogonal group itself.

```
gap> GeneralOrthogonalGroup( 3, 7 );
GO(0,3,7)
gap> GeneralOrthogonalGroup( -1, 4, 3 );
GO(-1,4,3)
gap> SpecialOrthogonalGroup( 1, 4, 4 );
GO(+1,4,4)
```

8 ▶ `ProjectiveGeneralLinearGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `PGL( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the projective general linear group PGL(*d*, *q*) of those $d \times d$ matrices over the field with q elements, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsPermGroup`, and the returned group is the action on lines of the underlying vector space.

9 ▶ `ProjectiveSpecialLinearGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `PSL( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the projective special linear group PSL(*d*, *q*) of those $d \times d$ matrices over the field with q elements whose determinant is the identity of the field, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsPermGroup`, and the returned group is the action on lines of the underlying vector space.

10 ▶ `ProjectiveGeneralUnitaryGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `PGU( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the projective general unitary group PGU(*d*, *q*) of those $d \times d$ matrices over the field with q^2 elements that respect a fixed nondegenerate sesquilinear form, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsPermGroup`, and the returned group is the action on lines of the underlying vector space.

11 ▶ `ProjectiveSpecialUnitaryGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `PSU( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the projective special unitary group PSU(*d*, *q*) of those $d \times d$ matrices over the field with q^2 elements that respect a fixed nondegenerate sesquilinear form and have determinant 1, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsPermGroup`, and the returned group is the action on lines of the underlying vector space.

12 ▶ `ProjectiveSymplecticGroup( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `PSP( [`*filt*`, ]`*d*`, `*q*` )` F
▶ `PSp( [`*filt*`, ]`*d*`, `*q*` )` F

constructs a group isomorphic to the projective symplectic group PSp(d,q) of those $d \times d$ matrices over the field with q elements that respect a fixed nondegenerate symplectic form, modulo the centre, in the category given by the filter *filt*.

If *filt* is not given it defaults to `IsPermGroup`, and the returned group is the action on lines of the underlying vector space.

48.3 Conjugacy Classes in Classical Groups

For general and special linear groups (see 48.2.1 and 48.2.2) GAP has an efficient method to generate representatives of the conjugacy classes. This uses results from linear algebra on normal forms of matrices. If you know how to do this for other types of classical groups, please, tell us.

```
gap> g := SL(4,9);
SL(4,9)
gap> NrConjugacyClasses(g);
861
gap> cl := ConjugacyClasses(g);;
gap> Length(cl);
861
```

1 ▶ `NrConjugacyClassesGL( n, q )` F
▶ `NrConjugacyClassesGU( n, q )` F
▶ `NrConjugacyClassesSL( n, q )` F
▶ `NrConjugacyClassesSU( n, q )` F
▶ `NrConjugacyClassesPGL( n, q )` F
▶ `NrConjugacyClassesPGU( n, q )` F
▶ `NrConjugacyClassesPSL( n, q )` F
▶ `NrConjugacyClassesPSU( n, q )` F
▶ `NrConjugacyClassesSLIsogeneous( n, q, f )` F
▶ `NrConjugacyClassesSUIsogeneous( n, q, f )` F

The first of these functions compute for given $n \in N$ and prime power q the number of conjugacy classes in the classical groups $GL(n,q)$, $GU(n,q)$, $SL(n,q)$, $SU(n,q)$, $PGL(n,q)$, $PGU(n,q)$, $PSL(n,q)$, $PSL(n,q)$, respectively. (See also 37.10.2 and Section 48.2.)

For each divisor f of n there is a group of Lie type with the same order as $SL(n,q)$, such that its derived subgroup modulo its center is isomorphic to $PSL(n,q)$. The various such groups with fixed n and q are called **isogeneous**. (Depending on congruence conditions on q and n several of these groups may actually be isomorphic.) The function `NrConjugacyClassesSLIsogeneous` computes the number of conjugacy classes in this group. The extreme cases $f = 1$ and $f = n$ lead to the groups $SL(n,q)$ and $PGL(n,q)$, respectively.

The function `NrConjugacyClassesSUIsogeneous` is the analogous one for the corresponding unitary groups.

The formulae for the number of conjugacy classes are taken from [Mac81].

```
gap> NrConjugacyClassesGL(24,27);
225283995449391744060672885806099 52
gap> NrConjugacyClassesPSU(19,17);
15052300411163848367708
gap> NrConjugacyClasses(SL(16,16));
1229782938228219920
```

48.4 Constructors for Basic Groups

All functions described in the previous sections call constructor operations to do the work. The names of the constructors are obtained from the names of the functions by appending `Cons`, so for example `CyclicGroup` calls the constructor

▶ `CyclicGroupCons( cat, n )` O

The first argument *cat* for each method of this constructor must be the category for which the method is installed. For example the method for constructing a cyclic permutation group is installed as follows (see 2.2.1 in "Programming in GAP" for the meaning of the arguments of `InstallMethod`).

```
InstallMethod( CyclicGroupCons,
    "regular perm group",
    true,
    [ IsPermGroup and IsRegularProp and IsFinite, IsInt and IsPosRat ], 0,
    function( filter, n )

    ...

    end );
```

48.5 Selection Functions

1 ▶ `All`*Library*`Groups(` *fun1*`,` *val1*`, ... )`

For a number of the group libraries two **selection functions** are provided. Each `All`*Library*`Groups` selection function permits one to select **all** groups from the library *Library* that have a given set of properties. Currently, the library selection functions provided, of this type, are `AllSmallGroups`, `AllIrreducibleSolvableGroups`, `AllTransitiveGroups`, and `AllPrimitiveGroups`. Corresponding to each of these there is a `One`*Library*`Group` function (see 48.5.2) which returns at most one group.

These functions take an arbitrary number of pairs (but at least one pair) of arguments. The first argument in such a pair is a function that can be applied to the groups in the library, and the second argument is either a single value that this function must return in order to have this group included in the selection, or a list of such values. For the function `AllSmallGroups` the first such function must be `Size`, and, unlike the other library selection functions, it supports an alternative syntax where `Size` is omitted (see 48.7.2). Also, see 48.11.3, for details pertaining to `AllIrreducibleSolvableGroups`.

For an example, let us consider the selection function for the library of transitive groups (also see 48.6). The command,

```
gap> AllTransitiveGroups(NrMovedPoints,[10..15],
>                        Size,         [1..100],
>                        IsAbelian,    false    );
```

returns a list of all transitive groups with degree between 10 and 15 and size less than 100 that are not abelian.

Thus the `AllTransitiveGroups` behaves as if it was implemented by a function similar to the one defined below, where `TransitiveGroupsList` is a list of all transitive groups. (Note that in the definition below we assume for simplicity that `AllTransitiveGroups` accepts exactly 4 arguments. It is of course obvious how to change this definition so that the function would accept a variable number of arguments.)

```
AllTransitiveGroups := function( fun1, val1, fun2, val2 )
local    groups, g, i;
  groups := [];
  for i  in [ 1 .. Length( TransitiveGroupsList ) ] do
    g := TransitiveGroupsList[i];
    if      fun1(g) = val1  or IsList(val1) and fun1(g) in val1
        and fun2(g) = val2  or IsList(val2) and fun2(g) in val2
     then
      Add( groups, g );
    fi;
  od;
  return groups;
end;
```

Note that the real selection functions are considerably more difficult, to improve the efficiency. Most important, each recognizes a certain set of properties which are precomputed for the library without having to

compute them anew for each group. This will substantially speed up the selection process. In the description of each library we will list the properties that are stored for this library.

2 ▶ `One`*Library*`Group(` *fun1*`,` *val1*`, ... )`

For each `All`*Library*`Groups` function (see 48.5.1) there is a corresponding function `One`*Library*`Group` on exactly the same arguments, i.e. there are `OneSmallGroup`, `OneIrreducibleSolvableGroup`, `OneTransitiveGroup`, and `OnePrimitiveGroup`. Each function simply returns the **first** group in the library (in the stored order) that has the prescribed properties, instead of **all** such groups. It returns `fail` if no such group exists in the library.

48.6 Transitive Permutation Groups

The transitive groups library currently contains representatives for all transitive permutation groups of degree at most 30. Two permutations groups of the same degree are considered to be equivalent, if there is a renumbering of points, which maps one group into the other one. In other words, if they lie in the same conjugacy class under operation of the full symmetric group by conjugation.

1 ▶ `TransitiveGroup(` *deg*`,` *nr* `)` F

returns the *nr*-th transitive group of degree *deg*. Both *deg* and *nr* must be positive integers. The transitive groups of equal degree are sorted with respect to their size, so for example `TransitiveGroup(` *deg*`, 1 )` is a transitive group of degree and size *deg*, e.g, the cyclic group of size *deg*, if *deg* is a prime.

2 ▶ `NrTransitiveGroups(` *deg* `)` F

returns the number of transitive groups of degree *deg* stored in the library of transitive groups. The function returns `fail` if *deg* is beyond the range of the library.

The selection functions (see 48.5) for the transitive groups library are `AllTransitiveGroups` and `OneTransitiveGroup`. They obtain the following properties from the database without having to compute them anew:

`NrMovedPoints`, `Size`, `Transitivity`, and `IsPrimitive`.

This library was computed by Gregory Butler, John McKay, Gordon Royle and Alexander Hulpke. The list of transitive groups up to degree 11 was published in [BM83], the list of degree 12 was published in [Roy87], degree 14 and 15 were published in [But93] and degrees 16–30 were published in [Hul96] and [Hular]. (Groups of prime degree of course are primitive and were known long before.)

The arrangement and the names of the groups of degree up to 15 is the same as given in [CHM98]. With the exception of the symmetric and alternating group (which are represented as `SymmetricGroup` and `AlternatingGroup`) the generators for these groups also conform to this paper with the only difference that 0 (which is not permitted in GAP for permutations to act on) is always replaced by the degree.

```
gap> TransitiveGroup(10,22);
S(5)[x]2
gap> l:=AllTransitiveGroups(NrMovedPoints,12,Size,1440,IsSolvable,false);
[ S(6)[x]2, M_10.2(12)=A_6.E_4(12)=[S_6[1/720]{M_10}S_6]2 ]
gap> List(l,IsSolvable);
[ false, false ]
```

3 ▶ `TransitiveIdentification(` *G* `)` A

Let *G* be a permutation group, acting transitively on a set of up to 30 points. Then `TransitiveIdentification` will return the position of this group in the transitive groups library. This means, if *G* acts on *m* points and `TransitiveIdentification` returns *n*, then *G* is permutation isomorphic to the group `TransitiveGroup(m,n)`.

Note: The points moved do **not** need to be $[1..n]$, the group $\langle (2,3,4),(2,3)\rangle$ is considered to be transitive on 3 points. If the group has several orbits on the points moved by it the result of `TransitiveIdentification` is undefined.

```
gap> TransitiveIdentification(Group((1,2),(1,2,3)));
2
```

48.7 Small Groups

The Small Groups library gives access to all groups of certain "small" orders. The groups are sorted by their orders and they are listed up to isomorphism; that is, for each of the available orders a complete and irredundant list of isomorphism type representatives of groups is given. Currently, the library contains the following groups:

- those of order at most 2000 except 1024 (423 164 062 groups);
- those of order 5^5 and 7^4 (92 groups);
- those of order $q^n \cdot p$ where q^n divides 2^8, 3^6, 5^5 or 7^4 and p is an arbitrary prime with $p \neq q$;
- those whose order factorises into at most 3 primes.

The first two items in this list cover an explicit range of orders; the last two provide access to infinite families of groups having orders of certain types.

The library also has an identification function: it returns the library number of a given group. This function determines library numbers using invariants of groups. The function is available for all orders in the library except 512 and 1536.

The library is organised in 8 layers. Each layer contains the groups of certain orders and their corresponding group identification routines. It is possible to install the first n layers of the group library and the first m layers of the group identification for each $1 \leq m \leq n \leq 8$. This might be useful to save disk space. There is an extensive `README` file for the Small Groups library available in the `small` directory of the GAP distribution containing detailed information on the layers. In summary, the layers are:

(1) the groups whose order factorises into at most 3 primes.

(2) the remaining groups of order at most 1000 except 512 and 768.

(3) the remaining groups of order $2^n * p$ with $n \leq 8$ and p an odd prime.

(4) the groups of order 5^5 and 7^4 and the remaining groups of order $q^n * p$ where q^n divides 3^6, 5^5 or 7^4 and p is a prime with $p \neq q$.

(5) the remaining groups of order at most 2000 except 1024, 1152, 1536 and 1920.

(6) the groups of orders 1152 and 1920.

(7) the groups of order 512.

(8) the groups of order 1536.

The data in this library has been carefully checked and cross-checked. It is believed to be reliable. However, no absolute guarantees are given and users should, as always, make their own checks in critical cases.

The data occupies about 26 MB (storing over 400 million groups in about 200 megabits). The group identification occupies about 42 MB of which 18 MB is used for the groups in layer (6). More information on the Small Groups library can be found on

`http://www.math.rwth-aachen.de/~Hans-Ulrich.Besche/small.html`

This group library has been computed by Hans Ulrich Besche, Bettina Eick and E. A. O'Brien. A survey on this topic and an account of the history of group constructions can be found in [BEOon]. For further detailed

information on the construction of this library we refer to [New77], [O'B90], [O'B91], [BE99a], [BE99b] and [BE01].

The Small Groups library incorporates the GAP 3 libraries `TwoGroup` and `ThreeGroup`. The data from these libraries was directly included into the Small Groups library, and the ordering there was preserved. The Small Groups library replaces the Gap 3 library of solvable groups of order at most 100. However, both the organisation and data descriptions of these groups has changed in the Small Groups library.

1 ▶ `SmallGroup(` *order*`,` *i* `)` F
▶ `SmallGroup( [`*order*`,` *i*`] )` F

returns the i-th group of order *order* in the catalogue. If the group is solvable, it will be given as a PcGroup; otherwise it will be given as a permutation group. If the groups of order *order* are not installed, the function reports an error and enters a break loop.

2 ▶ `AllSmallGroups(` *arg* `)` F

returns all groups with certain properties as specified by *arg*. If *arg* is a number n, then this function returns all groups of order n. However, the function can also take several arguments which then must be organized in pairs `function` and `value`. In this case the first function must be `Size` and the first value an order or a range of orders. If value is a list then it is considered a list of possible function values to include. The function returns those groups of the specified orders having those properties specified by the remaining functions and their values.

Precomputed information is stored for the properties `IsAbelian`, `IsNilpotentGroup`, `IsSupersolvableGroup`, `IsSolvableGroup`, `RankPGroup`, `PClassPGroup`, `LGLength`, `FrattinifactorSize` and `FrattinifactorId` for the groups of order at most 2000 which have more than three prime factors, except those of order 512, 768, 1024, 1152, 1536, 1920 and those of order $p^n \cdot q > 1000$ with $n > 2$.

3 ▶ `OneSmallGroup(` *arg* `)` F

returns one group with certain properties as specified by *arg*. The permitted arguments are those supported by `AllSmallGroups`.

4 ▶ `NumberSmallGroups(` *order* `)` F

returns the number of groups of order *order*.

5 ▶ `IdSmallGroup(` *G* `)` A
▶ `IdGroup(` *G* `)` A

returns the library number of G; that is, the function returns a pair `[`*order*`,` *i*`]` where G is isomorphic to `SmallGroup(` *order*`,` *i* `)`.

6 ▶ `IdsOfAllSmallGroups(` *arg* `)` F

similar to `AllSmallGroups` but returns ids instead of groups. This may prevent workspace overflows, if a large number of groups are expected in the output.

7 ▶ `IdGap3SolvableGroup(` *G* `)` A
▶ `Gap3CatalogueIdGroup(` *G* `)` A

returns the catalogue number of G in the GAP 3 catalogue of solvable groups; that is, the function returns a pair `[`*order*`,` *i*`]` meaning that G is isomorphic to the group `SolvableGroup(` *order*`,` *i* `)` in GAP 3.

8 ▶ `SmallGroupsInformation(` *order* `)` F

prints information on the groups of the specified order.

9 ▶ `UnloadSmallGroupsData( )` F

GAP loads all necessary data from the library automatically, but it does not delete the data from the workspace again. Usually, this will be not necessary, since the data is stored in a compressed format. However, if a large number of groups from the library have been loaded, then the user might wish to remove the data from the workspace and this can be done by the above function call.

```
gap> G := SmallGroup( 768, 1000000 );
<pc group of size 768 with 9 generators>
gap> G := SmallGroup( [768, 1000000] );
<pc group of size 768 with 9 generators>

gap> AllSmallGroups( 6 );
[ <pc group of size 6 with 2 generators>,
  <pc group of size 6 with 2 generators> ]
gap> AllSmallGroups( Size, 120, IsSolvableGroup, false );
[ Group([ (1,2,4,8)(3,6,9,5)(7,12,13,17)(10,14,11,15)(16,20,21,24)(18,22,19,
        23), (1,3,7)(2,5,10)(4,9,13)(6,11,8)(12,16,20)(14,18,22)(15,19,23)(17,
        21,24) ]), Group([ (1,2,3,4,5), (1,2) ]),
  Group([ (1,2,3,5,4), (1,3)(2,4)(6,7) ]) ]

gap> G := OneSmallGroup( 120, IsNilpotentGroup, false );
<pc group of size 120 with 5 generators>
gap> IdSmallGroup(G);
[ 120, 1 ]
gap> G := OneSmallGroup( Size, [1..1000], IsSolvableGroup, false );
Group([ (1,2,3,4,5), (1,2,3) ])
gap> IdSmallGroup(G);
[ 60, 5 ]
gap> UnloadSmallGroupsData();

gap> IdSmallGroup( GL( 2,3 ) );
[ 48, 29 ]
gap> IdSmallGroup( Group( (1,2,3,4),(4,5) ) );
[ 120, 34 ]
gap> IdsOfAllSmallGroups( Size, 60, IsSupersolvableGroup, true );
[ [ 60, 1 ], [ 60, 2 ], [ 60, 3 ], [ 60, 4 ], [ 60, 6 ], [ 60, 7 ],
  [ 60, 8 ], [ 60, 10 ], [ 60, 11 ], [ 60, 12 ], [ 60, 13 ] ]

gap> NumberSmallGroups( 512 );
10494213
gap> NumberSmallGroups( 2^8 * 23 );
1083472

gap> NumberSmallGroups( 2^9 * 23 );
Error, the library of groups of size 11776 is not available called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
gap>

gap> SmallGroupsInformation( 32 );
```

```
There are 51 groups of order 32.
They are sorted by their ranks.
   1 is cyclic.
   2 - 20 have rank 2.
   21 - 44 have rank 3.
   45 - 50 have rank 4.
   51 is elementary abelian.

For the selection functions the values of the following attributes
are precomputed and stored:
   IsAbelian, PClassPGroup, RankPGroup, FrattinifactorSize and
   FrattinifactorId.

This size belongs to layer 2 of the SmallGroups library.
IdSmallGroup is available for this size.
```

48.8 Finite Perfect Groups

The GAP library of finite perfect groups provides, up to isomorphism, a list of all perfect groups whose sizes are less than 10^6 excluding the following sizes:

- For $n = 61440$, 122880, 172032, 245760, 344064, 491520, 688128, or 983040, the perfect groups of size n have not completely been determined yet. The library neither provides the number of these groups nor the groups themselves.
- For $n = 86016$, 368640, or 737280, the library does not yet contain the perfect groups of size n, it only provides their numbers which are 52, 46, and 54, respectively.

Except for these eleven sizes, the list of altogether 1096 perfect groups in the library is complete. It relies on results of Derek F. Holt and Wilhelm Plesken which are published in their book "Perfect Groups" [HP89]. Moreover, they have supplied us with files with presentations of 488 of the groups. In terms of these, the remaining 607 nontrivial groups in the library can be described as 276 direct products, 107 central products, and 224 subdirect products. They are computed automatically by suitable GAP functions whenever they are needed.

We are grateful to Derek Holt and Wilhelm Plesken for making their groups available to the GAP community by contributing their files. It should be noted that their book contains a lot of further information for many of the library groups. So we would like to recommend it to any GAP user who is interested in the groups.

The library has been brought into GAP format by Volkmar Felsch.

1 ▶ `SizesPerfectGroups( )` F

2 ▶ `PerfectGroup( [`*filt*`, ]`*size*`[,` *n*`] )` F
▶ `PerfectGroup( [`*filt*`, ]`*sizenumberpair* `)` F

returns a group which is isomorphic to the library group specified by the size number `[` *size*`,` *n* `]` or by the two separate arguments *size* and *n*, assuming a default value of $n = 1$. The optional argument *filt* defines the filter in which the group is returned. Possible filters so far are `IsPermGroup` and `IsSubgroupFpGroup`. In the latter case, the generators and relators used coincide with those given in [HP89].

```
gap> G := PerfectGroup(IsPermGroup,6048,1);
U3(3)
```

As all groups are stored by presentations, a permutation representation is obtained by coset enumeration. Note that some of the library groups do not have a faithful permutation representation of small degree. Computations in these groups may be rather time consuming.

```
gap> G:=PerfectGroup(IsPermGroup,129024,2);
L2(8) N ( 2^6 E 2^1 A ) C 2^1
gap> NrMovedPoints(G);
14336
```

3 ▶ `PerfectIdentification( G )` A

This attribute is set for all groups obtained from the perfect groups library and has the value `[`*size*`,`*nr*`]` if the group is obtained with these parameters from the library.

4 ▶ `NumberPerfectGroups( size )` F

returns the number of non-isomorphic perfect groups of size *size* for each positive integer *size* up to 10^6 except for the eight sizes listed at the beginning of this section for which the number is not yet known. For these values as well as for any argument out of range it returns `fail`.

5 ▶ `NumberPerfectLibraryGroups( size )` F

returns the number of perfect groups of size *size* which are available in the library of finite perfect groups. (The purpose of the function is to provide a simple way to formulate a loop over all library groups of a given size.)

6 ▶ `SizeNumbersPerfectGroups( factor1, factor2, ... )` F

`SizeNumbersPerfectGroups` returns a list of pairs, each entry consisting of a group order and the number of those groups in the library of perfect groups that contain the specified factors *factor1*, *factor2*, ... among their composition factors.

Each argument must either be the name of a simple group or an integer which stands for the product of the sizes of one or more cyclic factors. (In fact, the function replaces all integers among the arguments by their product.)

The following text strings are accepted as simple group names.

- `A`*n* or `A(`*n*`)` for the alternating groups A_n, $5 \leq n \leq 9$, for example `A5` or `A(6)`.
- `L`*n*`(`*q*`)` or `L(`*n*`,`*q*`)` for $PSL(n, q)$, where $n \in \{2, 3\}$ and q a prime power, ranging
  - for $n = 2$ from 4 to 125
  - for $n = 3$ from 2 to 5
- `U`*n*`(`*q*`)` or `U(`*n*`,`*q*`)` for $PSU(n, q)$, where $n \in \{3, 4\}$ and q a prime power, ranging
  - for $n = 3$ from 3 to 5
  - for $n = 4$ from 2 to 2
- `Sp4(4)` or `S(4,4)` for the symplectic group $S(4, 4)$,
- `Sz(8)` for the Suzuki group $Sz(8)$,
- `M`*n* or `M(`*n*`)` for the Mathieu groups M_{11}, M_{12}, and M_{22}, and
- `J`*n* or `J(`*n*`)` for the Janko groups J_1 and J_2.

Note that, for most of the groups, the preceding list offers two different names in order to be consistent with the notation used in [HP89] as well as with the notation used in the `DisplayCompositionSeries` command

of GAP. However, as the names are compared as text strings, you are restricted to the above choice. Even expressions like `L2(2^5)` are not accepted.

As the use of the term $PSU(n, q)$ is not unique in the literature, we mention that in this library it denotes the factor group of $SU(n, q)$ by its centre, where $SU(n, q)$ is the group of all $n \times n$ unitary matrices with entries in $GF(q^2)$ and determinant 1.

The purpose of the function is to provide a simple way to formulate a loop over all library groups which contain certain composition factors.

7 ▶ `DisplayInformationPerfectGroups(` *size* `)` F
▶ `DisplayInformationPerfectGroups(` *size*`,` *n* `)` F
▶ `DisplayInformationPerfectGroups( [` *size*`,` *n* `] )` F

`DisplayInformationPerfectGroups` displays some invariants of the n-th group of order *size* from the perfect groups library.

If no value of n has been specified, the invariants will be displayed for all groups of size *size* available in the library. The information provided for G includes the following items:

- a headline containing the size number `[` *size*`,` *n* `]` of G in the form *size*`.`*n* (the suffix `.`*n* will be suppressed if, up to isomorphism, G is the only perfect group of order *size*),
- a message if G is simple or quasisimple, i.e., if the factor group of G by its centre is simple,
- the "description" of the structure of G as it is given by Holt and Plesken in [HP89] (see below),
- the size of the centre of G (suppressed, if G is simple),
- the prime decomposition of the size of G,
- orbit sizes for a faithful permutation representation of G which is provided by the library (see below),
- a reference to each occurrence of G in the tables of section 5.3 of [HP89]. Each of these references consists of a class number and an internal number (i, j) under which G is listed in that class. For some groups, there is more than one reference because these groups belong to more than one of the classes in the book.

```
gap> DisplayInformationPerfectGroups( 30720, 3 );
#I Perfect group 30720:  A5 ( 2^4 E N 2^1 E 2^4 ) A
#I   size = 2^11*3*5  orbit size = 240
#I   Holt-Plesken class 1 (9,3)
gap> DisplayInformationPerfectGroups( 30720, 6 );
#I Perfect group 30720:  A5 ( 2^4 x 2^4 ) C N 2^1
#I   centre = 2  size = 2^11*3*5  orbit size = 384
#I   Holt-Plesken class 1 (9,6)
gap> DisplayInformationPerfectGroups( Factorial( 8 ) / 2 );
#I Perfect group 20160.1:  A5 x L3(2) 2^1
#I   centre = 2  size = 2^6*3^2*5*7  orbit sizes = 5 + 16
#I   Holt-Plesken class 31 (1,1) (occurs also in class 32)
#I Perfect group 20160.2:  A5 2^1 x L3(2)
#I   centre = 2  size = 2^6*3^2*5*7  orbit sizes = 7 + 24
#I   Holt-Plesken class 31 (1,2) (occurs also in class 32)
#I Perfect group 20160.3:  ( A5 x L3(2) ) 2^1
#I   centre = 2  size = 2^6*3^2*5*7  orbit size = 192
#I   Holt-Plesken class 31 (1,3)
#I Perfect group 20160.4:  simple group  A8
#I   size = 2^6*3^2*5*7  orbit size = 8
#I   Holt-Plesken class 26 (0,1)
```

```
#I Perfect group 20160.5:  simple group  L3(4)
#I   size = 2^6*3^2*5*7  orbit size = 21
#I   Holt-Plesken class 27 (0,1)
```

For any library group G, the library files do not only provide a presentation, but, in addition, a list of one or more subgroups $S_1, \ldots, S_r$ of G such that there is a faithful permutation representation of G of degree $\sum_{i=1}^{r}[G : S_i]$ on the set $\{S_i g \mid 1 \leq i \leq r,\ g \in G\}$ of the cosets of the S_i. This allows one to construct the groups as permutation groups. The `DisplayInformationPerfectGroups` function displays only the available degree. The message

```
orbit size = 8
```

in the above example means that the available permutation representation is transitive and of degree 8, whereas the message

```
orbit sizes = 5 + 16
```

means that a nontransitive permutation representation is available which acts on two orbits of size 5 and 16 respectively.

The notation used in the "description" of a group is explained in section 5.1.2 of [HP89]. We quote the respective page from there:

> Within a class $Q \# p$, an isomorphism type of groups will be denoted by an ordered pair of integers (r, n), where $r \geq 0$ and $n > 0$. More precisely, the isomorphism types in $Q\#p$ of order $p^r|Q|$ will be denoted by $(r, 1), (r, 2), (r, 3), \ldots$. Thus Q will always get the size number $(0, 1)$.
>
> In addition to the symbol (r, n), the groups in $Q \# p$ will also be given a more descriptive name. The purpose of this is to provide a very rough idea of the structure of the group. The names are derived in the following manner. First of all, the isomorphism classes of irreducible $F_p Q$-modules M with $|Q| \cdot |M| \leq 10^6$, where F_p is the field of order p, are assigned symbols. These will either be simply p^x, where x is the dimension of the module, or, if there is more than one isomorphism class of irreducible modules having the same dimension, they will be denoted by $p^x, p^{x'}$, etc. The one-dimensional module with trivial Q-action will therefore be denoted by p^1. These symbols will be listed under the description of Q. The group name consists essentially of a list of the composition factors working from the top of the group downwards; hence it always starts with the name of Q itself. (This convention is the most convenient in our context, but it is different from that adopted in the ATLAS [CCN+85], for example, where composition factors are listed in the reverse order. For example, we denote a group isomorphic to $SL(2, 5)$ by $A_5 2^1$ rather than $2 \cdot A_5$.)
>
> Some other symbols are used in the name, in order to give some idea of the relationship between these composition factors, and splitting properties. We shall now list these additional symbols.
>
> - $\times$ between two factors denotes a direct product of $F_p Q$-modules or groups.
> - C (for "commutator") between two factors means that the second lies in the commutator subgroup of the first. Similarly, a segment of the form $(f_1 \times f_2)Cf_3$ would mean that the factors f_1 and f_2 commute modulo f_3 and f_3 lies in $[f_1, f_2]$.
> - A (for "abelian") between two factors indicates that the second is in the pth power (but not the commutator subgroup) of the first. "A" may also follow the factors, if bracketed.
> - E (for "elementary abelian") between two factors indicates that together they generate an elementary abelian group (modulo subsequent factors), but that the resulting $F_p Q$-module extension does not split.

N (for "nonsplit") before a factor indicates that Q (or possibly its covering group) splits down as far at this factor but not over the factor itself. So "Qf_1Nf_2" means that the normal subgroup f_1f_2 of the group has no complement but, modulo f_2, f_1, does have a complement.

Brackets have their obvious meaning. Summarizing, we have:

$\times$ = direct product;

C = commutator subgroup;

A = abelian;

E = elementary abelian; and

N = nonsplit.

Here are some examples.

(i) $A_5(2^4E2^1E2^4)A$ means that the pairs 2^4E2^1 and 2^1E2^4 are both elementary abelian of exponent 4.

(ii) $A_5(2^4E2^1A)C2^1$ means that $O_2(G)$ is of symplectic type 2^{1+5}, with Frattini factor group of type 2^4E2^1. The "A" after the 2^1 indicates that G has a central cyclic subgroup 2^1A2^1 of order 4.

(iii) $L_3(2)((2^1E)\times(N2^3E2^{3'}A)C)2^{3'}$ means that the $2^{3'}$ factor at the bottom lies in the commutator subgroup of the pair $2^3E2^{3'}$ in the middle, but the lower pair $2^{3'}A2^{3'}$ is abelian of exponent 4. There is also a submodule $2^1E2^{3'}$, and the covering group $L_3(2)2^1$ of $L_3(2)$ does not split over the 2^3 factor. (Since G is perfect, it goes without saying that the extension $L_3(2)2^1$ cannot split itself.)

We must stress that this notation does not always succeed in being precise or even unambiguous, and the reader is free to ignore it if it does not seem helpful.

If such a group description has been given in the book for G (and, in fact, this is the case for most of the library groups), it is displayed by the `DisplayInformationPerfectGroups` function. Otherwise the function provides a less explicit description of the (in these cases unique) Holt-Plesken class to which G belongs, together with a serial number if this is necessary to make it unique.

48.9 Primitive Permutation Groups

GAP contains a library of primitive permutation groups which includes the following permutation groups up to permutation isomorphism (i.e., up to conjugacy in the corresponding symmetric group)

- the non-affine primitive permutation groups of degree $\leq$ 999, described in [DM88], with generators calculated in [The97],
- all primitive permutation groups of degree $<$ 256, in particular,
  - the primitive permutation groups up to degree 50, calculated by C. Sims,
  - the solvable (hence affine) primitive permutation groups of degree $<$ 256 as calculated by M. Short [Sho92],
  - some insolvable affine primitive permutation groups of degree $<$ 256 as calculated in [The97].
  - The solvable primitive groups of degree up to 999 as calculated in [EH02].

Note that the nonsolvable affine primitive permutation groups of degrees 256,512,625 and 729 are not included.

For degree up to 50, the names used are as given by Buekenhout and Leemans [BL96].

The names for the groups of higher degree are as chosen by [The97] and reflect the cohort structure in [DM88]. They do **not** conform to the usual naming for extensions. That is `l3,4.3` is the third (in some arbitrary ordering) group in a cohort for socle l3,4, but the socle factor is **not necessarily** of order 3. Names of groups of affine type are not necessarily unique!

The work in [The97] is known to have ommissions. Because of this we do not guarantee completeness of the lists beyond degree 50, though we have corrected errors as far as we know of them. When preparing the library it also has been ensured that the groups in it are all primitive and not conjugate. Also as far as included, the groups agree with the list prepared independently in [RDU02]

In detail, we guarantee the following properties for this and further versions (but **not** versions which came before GAP 4.2) of the library:

- All groups in the library are primitive permutation groups of the indicated degree.
- The positions of the groups in the library are stable. That is `PrimitiveGroup(`*n*`,`*nr*`)` will always give you a permutation isomorphic group. Note however that we do not guarantee to keep the chosen S_n-representative, the generating set or the name for eternity.
- Different groups in the library are not conjugate in S_n.
- If a group in the library has a primitive subgroup with the same socle, this group is in the library as well.

(Note that the arrangement of groups is not guaranteed to be in increasing size, though it holds for many degrees.)

1 ▶ `PrimitiveGroup(` *deg*`,` *nr* `)` F

returns the primitive permutation group of degree *deg* with number *nr* from the list.

The arrangement of the groups differs from the arrangement of primitive groups in the list of C. Sims, which was used in GAP 3. See `SimsNo` (48.10.2).

2 ▶ `NrPrimitiveGroups(` *deg* `)` F

returns the number of primitive permutation groups of degree *deg* in the library.

```
gap> NrPrimitiveGroups(25);
28
gap> PrimitiveGroup(25,19);
5^2:((Q_8:3)‘4
gap> PrimitiveGroup(25,20);
ASL(2,5)
gap> PrimitiveGroup(25,22);
AGL(2,5)
gap> PrimitiveGroup(25,23);
(A_5xA_5):2
```

The selection functions (see 48.5) for the primitive groups library are `AllPrimitiveGroups` and `OnePrimitiveGroup`. They obtain the following properties from the database without having to compute them anew:

`NrMovedPoints`, `Size`, `Transitivity`, `ONanScottType`, `IsSimpleGroup`, `IsSolvableGroup`, and `SocleTypePrimitiveGroup`.

(Note, that for groups of degree up to 999, O'Nan-Scott types 4a, 4b and 5 cannot occur.)

3 ▶ `PrimitiveGroupsIterator(` *attr1*`,` *val1*`,` *attr2*`,` *val2*`, ... )` F

returns an iterator through `AllPrimitiveGroups(`*attr1*`,`*val1*`,`*attr2*`,`*val2*`,...)` without creating all these groups at the same time.

4 ► `COHORTS_PRIMITIVE_GROUPS` V

In [DM88] the primitive groups are sorted in "cohorts" according to their socle. For each degree, the variable `COHORTS_PRIMITIVE_GROUPS` contains a list of the cohorts for the primitive groups of this degree. Each cohort is represented by a list of length 2, the first entry specifies the socle type (see `SocleTypePrimitiveGroup`, section 41.4.2), the second entry listing the index numbers of the groups in this degree.

For example in degree 49, we have four cohorts with socles $\mathbb{F}_7^2$, $L_2(7)^2$, A_7^2 and A_{49} respectively. the group `PrimitiveGroup(49,36)`, which is isomorphic to $(A_7 \times A_7) : 2^2$, lies in the third cohort with socle $(A_7 \times A_7)$.

```
gap> COHORTS_PRIMITIVE_GROUPS[49];
[ [ rec( series := "Z", width := 2, parameter := 7 ),
      [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
          20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33 ] ],
  [ rec( series := "L", width := 2, parameter := [ 2, 7 ] ), [ 34 ] ],
  [ rec( series := "A", width := 2, parameter := 7 ), [ 35, 36, 37, 38 ] ],
  [ rec( series := "A", parameter := 49, width := 1 ), [ 39, 40 ] ] ]
```

48.10 Index numbers of primitive groups

1 ► `PrimitiveIdentification( G )` A

For a primitive permutation group for which an S_n-conjugate exists in the library of primitive permutation groups (see 48.9), this attribute returns the index position. That is G is conjugate to `PrimitiveGroup(NrMovedPoints(`G`),PrimitiveIdentification(`G`)`.

Methods only exist if the primitive groups library is installed.

Note: As this function uses the primitive groups library, the result is only guaranteed to the same extent as this library. If it is incomplete, `PrimitiveIdentification` might return an existing index number for a group not in the library.

```
gap> PrimitiveIdentification(Group((1,2),(1,2,3)));
2
```

2 ► `SimsNo( G )` A

If G is a primitive group obtained by `PrimitiveGroup` (respectively one of the selection functions) this attribute contains the number of the isomorphic group in the original list of C. Sims. (this is the arrangement as it was used in GAP 3.

```
gap> g:=PrimitiveGroup(25,2);
5^2:S_3
gap> SimsNo(g);
3
```

As mentioned in the previous section, the index numbers of primitive groups in GAP are guaranteed to remain stable. (Thus, missing groups will be added to the library at the end of each degree.) In particular, it is safe to refer to a primitive group of type *deg*,*nr* in the GAP library.

The system Magma also provides a list of primitive groups (see [RDU02]). For historical reasons, its indexing however differs from the one used by GAP. The variable

3 ► `PRIMITIVE_INDICES_MAGMA` V

can be used to obtain this correspondence. The magma index number of the GAP group `PrimitiveGroup(`*deg*`,`*nr*`)` is stored in the entry `PRIMITIVE_INDICES_MAGMA[`*deg*`][`*nr*`]`.

Vice versa, the group of degree *deg* with Magma index number *nr* has the GAP index `Position(PRIMITIVE_INDICES_MAGMA[`*deg*`],`*nr*`)`, in particular it can be obtained by the GAP command `PrimitiveGroup(`*deg*`,Position(PRIMITIVE_INDICES_MAGMA[`*deg*`],`*nr*`));`

48.11 Irreducible Solvable Matrix Groups

1 ▶ `IrreducibleSolvableGroupMS( n, p, i )` F

This function returns a representative of the i-th conjugacy class of irreducible solvable subgroup of *GL(n,p)*, where n is an integer > 1, p is a prime, and $p^n < 256$.

The numbering of the representatives should be considered arbitrary. However, it is guaranteed that the i-th group on this list will lie in the same conjugacy class in all future versions of GAP, unless two (or more) groups on the list are discovered to be duplicates, in which case `IrreducibleSolvableMatrixGroup` will return `fail` for all but one of the duplicates.

For values of n, p, and i admissible to `IrreducibleSolvableGroup`, `IrreducibleSolvableMatrixGroup` returns a representative of the same conjugacy class of subgroups of *GL(n,p)* as `IrreducibleSolvableGroup`. Note that it currently adds two more groups (missing from the original list by Mark Short) for $n = 2$, $p = 13$.

2 ▶ `NumberIrreducibleSolvableGroups( n, p )` F

This function returns the number of conjugacy classes of irreducible solvable subgroup of *GL(n,p)*.

3 ▶ `AllIrreducibleSolvableGroups( func_1, val_1, func_2, val_2, ... )` F

This function returns a list of conjugacy class representatives G of matrix groups over a prime field such that $func\_i(G) = val\_i$ or $func\_i(G) \in val\_i$. The following possibilities for *func_i* are particularly efficient, because the values can be read off the information in the data base: `DegreeOfMatrixGroup` (or `Dimension` or `DimensionOfMatrixGroup`) for the linear degree, `Characteristic` for the field characteristic, `Size`, `IsPrimitiveMatrixGroup` (or `IsLinearlyPrimitive`), and `MinimalBlockDimension`.

4 ▶ `OneIrreducibleSolvableGroup( func1, val1, func2, val2, ... )` F

This function returns one solvable subgroup G of a matrix group over a prime field such that $func\_i(G) = val\_i$ or $func\_i(G) \in val\_i$ for all i. The following possibilities for *func_i* are particularly efficient, because the values can be read off the information in the data base: `DegreeOfMatrixGroup` (or `Dimension` or `DimensionOfMatrixGroup`) for the linear degree, `Characteristic` for the field characteristic, `Size`, `IsPrimitiveMatrixGroup` (or `IsLinearlyPrimitive`), and `MinimalBlockDimension`.

5 ▶ `PrimitiveIndexIrreducibleSolvableGroup` V

This variable provides a way to get from irreducible solvable groups to primitive groups and vice versa. For the group G=`IrreducibleSolvableGroup( n, p, k )` and $d = p^n$, the entry `PrimitiveIndexIrreducibleSolvableGroup[d][i]` gives the index number of the semidirect product $p^n : G$ in the library of primitive groups.

Searching for an index `Position` in this list gives the translation in the other direction.

6 ▶ `IrreducibleSolvableGroup( n, p, i )` F

This function is obsolete, because for $n = 2$, $p = 13$, two groups were missing from the underlying database. It has been replaced by the function `IrreducibleSolvableGroupMS` (see 48.11.1). Please note that the latter function does not guarantee any ordering of the groups in the database. However, for values of n, p, and i admissible to `IrreducibleSolvableGroup`, `IrreducibleSolvableGroupMS` returns a representative of the same conjugacy class of subgroups of *GL(n,p)* as `IrreducibleSolvableGroup` did before.

48.12 Irreducible Maximal Finite Integral Matrix Groups

A library of irreducible maximal finite integral matrix groups is provided with GAP. It contains $\mathbb{Q}$-class representatives for all of these groups of dimension at most 31, and $\mathbb{Z}$-class representatives for those of dimension at most 11 or of dimension 13, 17, 19, or 23.

The groups provided in this library have been determined by Wilhelm Plesken, partially as joint work with Michael Pohst, or by members of his institute (Lehrstuhl B für Mathematik, RWTH Aachen). In particular, the data for the groups of dimensions 2 to 9 have been taken from the output of computer calculations which they performed in 1979 (see [PP77], [PP80]). The $\mathbb{Z}$-class representatives of the groups of dimension 10 have been determined and computed by Bernd Souvignier ([Sou94]), and those of dimensions 11, 13, and 17 have been recomputed for this library from the circulant Gram matrices given in [Ple85], using the stand-alone programs for the computation of short vectors and Bravais groups which have been developed in Plesken's institute. The $\mathbb{Z}$-class representatives of the groups of dimensions 19 and 23 had already been determined in [Ple85]. Gabriele Nebe has recomputed them for us. Her main contribution to this library, however, is that she has determined and computed the $\mathbb{Q}$-class representatives of the groups of non-prime dimensions between 12 and 24 and the groups of dimensions 25 to 31 (see [PN95], [NP95b], [Neb95], [Neb96]).

The library has been brought into GAP format by Volkmar Felsch. He has applied several GAP routines to check certain consistency of the data. However, the credit and responsibility for the lists remain with the authors. We are grateful to Wilhelm Plesken, Gabriele Nebe, and Bernd Souvignier for supplying their results to GAP.

In the preceding acknowledgement, we used some notations that will also be needed in the sequel. We first define these.

Any integral matrix group G of dimension n is a subgroup of $GL_n(\mathbb{Z})$ as well as of $GL_n(\mathbb{Q})$ and hence lies in some conjugacy class of integral matrix groups under $GL_n(\mathbb{Z})$ and also in some conjugacy class of rational matrix groups under $GL_n(\mathbb{Q})$. As usual, we call these classes the $\mathbb{Z}$-class and the $\mathbb{Q}$-class of G, respectively. Note that any conjugacy class of subgroups of $GL_n(\mathbb{Q})$ contains at least one $\mathbb{Z}$-class of subgroups of $GL_n(\mathbb{Z})$ and hence can be considered as the $\mathbb{Q}$-class of some integral matrix group.

In the context of this library we are only concerned with $\mathbb{Z}$-classes and $\mathbb{Q}$-classes of subgroups of $GL_n(\mathbb{Z})$ which are irreducible and maximal finite in $GL_n(\mathbb{Z})$ (we will call them **i.m.f.** subgroups of $GL_n(\mathbb{Z})$). We can distinguish two types of these groups:

First, there are those i.m.f. subgroups of $GL_n(\mathbb{Z})$ which are also maximal finite subgroups of $GL_n(\mathbb{Q})$. Let us denote the set of their $\mathbb{Q}$-classes by $Q_1(n)$. It is clear from the above remark that $Q_1(n)$ just consists of the $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Q})$.

Secondly, there is the set $Q_2(n)$ of the $\mathbb{Q}$-classes of the remaining i.m.f. subgroups of $GL_n(\mathbb{Z})$, i.e., of those which are not maximal finite subgroups of $GL_n(\mathbb{Q})$. For any such group G, say, there is at least one class $C \in Q_1(n)$ such that G is conjugate under $\mathbb{Q}$ to a proper subgroup of some group $H \in C$. In fact, the class C is uniquely determined for any group G occurring in the library (though there seems to be no reason to assume that this property should hold in general). Hence we may call C the **rational i.m.f. class** of G. Finally, we will denote the number of classes in $Q_1(n)$ and $Q_2(n)$ by $q_1(n)$ and $q_2(n)$, respectively.

As an example, let us consider the case $n = 4$. There are 6 $\mathbb{Z}$-classes of i.m.f. subgroups of $GL_4(\mathbb{Z})$ with representative subgroups $G_1, \ldots, G_6$ of isomorphism types $G_1 \cong W(F_4)$, $G_2 \cong D_{12} \wr C_2$, $G_3 \cong G_4 \cong C_2 \times S_5$, $G_5 \cong W(B_4)$, and $G_6 \cong (D_{12} \Upsilon D_{12}) : C_2$. The corresponding $\mathbb{Q}$-classes, $R_1, \ldots, R_6$, say, are pairwise different except that R_3 coincides with R_4. The groups G_1, G_2, and G_3 are i.m.f. subgroups of $GL_4(\mathbb{Q})$, but G_5 and G_6 are not because they are conjugate under $GL_4(\mathbb{Q})$ to proper subgroups of G_1 and G_2, respectively. So we have $Q_1(4) = \{R_1, R_2, R_3\}$, $Q_2(4) = \{R_5, R_6\}$, $q_1(4) = 3$, and $q_2(4) = 2$.

The $q_1(n)$ $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Q})$ have been determined for each dimension $n \leq 31$. The current GAP library provides integral representative groups for all these classes. Moreover, all $\mathbb{Z}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Z})$ are known for $n \leq 11$ and for $n \in \{13, 17, 19, 23\}$. For these dimensions, the

library offers integral representative groups for all $\mathbb{Q}$-classes in $Q_1(n)$ and $Q_2(n)$ as well as for all $\mathbb{Z}$-classes of i.m.f. subgroups of $GL_n(\mathbb{Z})$.

Any group G of dimension n given in the library is represented as the automorphism group $G = \mathrm{Aut}(F, L) = \{g \in GL_n(\mathbb{Z}) \mid Lg = L \text{ and } gFg^{\mathrm{tr}} = F\}$ of a positive definite symmetric $n \times n$ matrix $F \in \mathbb{Z}^{n \times n}$ on an n-dimensional lattice $L \cong \mathbb{Z}^{1 \times n}$ (for details see e.g. [PN95]). GAP provides for G a list of matrix generators and the **Gram matrix** F.

The positive definite quadratic form defined by F defines a **norm** vFv^{tr} for each vector $v \in L$, and there is only a finite set of vectors of minimal norm. These vectors are often simply called the **short vectors**. Their set splits into orbits under G, and G being irreducible acts faithfully on each of these orbits by multiplication from the right. GAP provides for each of these orbits the orbit size and a representative vector.

Like most of the other GAP libraries, the library of i.m.f. integral matrix groups supplies an extraction function, `ImfMatrixGroup`. However, as the library involves only 525 different groups, there is no need for a selection or an example function. Instead, there are two functions, `ImfInvariants` and `DisplayImfInvariants`, which provide some $\mathbb{Z}$-class invariants that can be extracted from the library without actually constructing the representative groups themselves. The difference between these two functions is that the latter one displays the resulting data in some easily readable format, whereas the first one returns them as record components so that you can properly access them.

We shall give an individual description of each of the library functions, but first we would like to insert a short remark concerning their names: Any self-explaining name of a function handling **irreducible maximal finite integral matrix groups** would have to include this term in full length and hence would grow extremely long. Therefore we have decided to use the abbreviation `Imf` instead in order to restrict the names to some reasonable length.

The first three functions can be used to formulate loops over the classes.

1 ▶ `ImfNumberQQClasses(` *dim* `)` F
▶ `ImfNumberQClasses(` *dim* `)` F
▶ `ImfNumberZClasses(` *dim*`,` *q* `)` F

`ImfNumberQQClasses` returns the number $q_1(dim)$ of $\mathbb{Q}$-classes of i.m.f. rational matrix groups of dimension *dim*. Valid values of *dim* are all positive integers up to 31.

Note: In order to enable you to loop just over the classes belonging to $Q_1(dim)$, we have arranged the list of $\mathbb{Q}$-classes of dimension *dim* for any dimension *dim* in the library such that, whenever the classes of $Q_2(dim)$ are known, too, i.e., in the cases $dim \leq 11$ or $dim \in \{13, 17, 19, 23\}$, the classes of $Q_1(dim)$ precede those of $Q_2(dim)$ and hence are numbered from 1 to $q_1(dim)$.

`ImfNumberQClasses` returns the number of $\mathbb{Q}$-classes of groups of dimension *dim* which are available in the library. If $dim \leq 11$ or $dim \in \{13, 17, 19, 23\}$, this is the number $q_1(dim) + q_2(dim)$ of $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_{dim}(\mathbb{Z})$. Otherwise, it is just the number $q_1(dim)$ of $\mathbb{Q}$-classes of i.m.f. subgroups of $GL_{dim}(\mathbb{Q})$. Valid values of *dim* are all positive integers up to 31.

`ImfNumberZClasses` returns the number of $\mathbb{Z}$-classes in the qth $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension *dim*. Valid values of *dim* are all positive integers up to 11 and all primes up to 23.

2 ▶ `DisplayImfInvariants(` *dim*`,` *q* `)` F
▶ `DisplayImfInvariants(` *dim*`,` *q*`,` *z* `)` F

`DisplayImfInvariants` displays the following $\mathbb{Z}$-class invariants of the groups in the zth $\mathbb{Z}$-class in the qth $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension *dim*:

- its $\mathbb{Z}$-class number in the form $dim.q.z$, if *dim* is at most 11 or a prime at most 23, or its $\mathbb{Q}$-class number in the form $dim.q$, else,
- a message if the group is solvable,
- the size of the group,

- the isomorphism type of the group,
- the elementary divisors of the associated quadratic form,
- the sizes of the orbits of short vectors (these sizes are the degrees of the faithful permutation representations which you may construct using the functions `IsomorphismPermGroup` or `IsomorphismPermGroupImfGroup` below),
- the norm of the associated short vectors,
- only in case that the group is not an i.m.f. group in $GL_n(\mathbb{Q})$: an appropriate message, including the $\mathbb{Q}$-class number of the corresponding rational i.m.f. class.

If you specify the value 0 for any of the parameters *dim*, *q*, or *z*, the command will loop over all available dimensions, $\mathbb{Q}$-classes of given dimension, or $\mathbb{Z}$-classes within the given $\mathbb{Q}$-class, respectively. Otherwise, the values of the arguments must be in range. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i.e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of z is 1. This value of z will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the qth $\mathbb{Q}$-class. The greatest legal value of *dim* is 31.

```
gap> DisplayImfInvariants( 3, 1, 0 );
#I Z-class 3.1.1:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = W(B3)
#I   elementary divisors = 1^3
#I   orbit size = 6, minimal norm = 1
#I Z-class 3.1.2:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = C2 x W(A3)
#I   elementary divisors = 1*4^2
#I   orbit size = 8, minimal norm = 3
#I Z-class 3.1.3:  Solvable, size = 2^4*3
#I   isomorphism type = C2 wr S3 = C2 x S4 = C2 x W(A3)
#I   elementary divisors = 1^2*4
#I   orbit size = 12, minimal norm = 2
gap> DisplayImfInvariants( 8, 15, 1 );
#I Z-class 8.15.1:  Solvable, size = 2^5*3^4
#I   isomorphism type = C2 x (S3 wr S3)
#I   elementary divisors = 1*3^3*9^3*27
#I   orbit size = 54, minimal norm = 8
#I   not maximal finite in GL(8,Q), rational imf class is 8.5
gap> DisplayImfInvariants( 20, 23 );
#I Q-class 20.23:  Size = 2^5*3^2*5*11
#I   isomorphism type = (PSL(2,11) x D12).C2
#I   elementary divisors = 1^18*11^2
#I   orbit size = 3*660 + 2*1980 + 2640 + 3960, minimal norm = 4
```

Note that the function `DisplayImfInvariants` uses a kind of shorthand to display the elementary divisors. E. g., the expression `1*3^3*9^3*27` in the preceding example stands for the elementary divisors $1, 3, 3, 3, 9, 9, 9, 27$. (See also the next example which shows that the function `ImfInvariants` provides the elementary divisors in form of an ordinary GAP list.)

In the description of the isomorphism types the following notations are used:

A `x` *B*
: denotes a direct product of a group *A* by a group *B*,

A `subd` *B*
: denotes a subdirect product of *A* by *B*,

A `Y` *B*
: denotes a central product of A by B,

A `wr` *B*
: denotes a wreath product of A by B,

A:*B*
: denotes a split extension of A by B,

A·*B*
: denotes just an extension of A by B (split or nonsplit).

The groups involved are

- the cyclic groups C_n, dihedral groups D_n, and generalized quaternion groups Q_n of order n, denoted by `C`n, `D`n, and `Q`n, respectively,
- the alternating groups A_n and symmetric groups S_n of degree n, denoted by `A`n and `S`n, respectively,
- the linear groups $GL_n(q)$, $PGL_n(q)$, $SL_n(q)$, and $PSL_n(q)$, denoted by `GL`(n,q), `PGL`(n,q), `SL`(n,q), and `PSL`(n,q), respectively,
- the unitary groups $SU_n(q)$ and $PSU_n(q)$, denoted by `SU`(n,q) and `PSU`(n,q), respectively,
- the symplectic groups $Sp(n, q)$ and $PSp(n, q)$, denoted by `Sp`(n,q) and `PSp`(n,q), respectively,
- the orthogonal groups $O_8^+(2)$ and $PO_8^+(2)$, denoted by `O+`(8,2) and `PO+`(8,2), respectively,
- the extraspecial groups 2_+^{1+8}, 3_+^{1+2}, 3_+^{1+4}, and 5_+^{1+2}, denoted by `2+^(1+8)`, `3+^(1+2)`, `3+^(1+4)`, and `5+^(1+2)`, respectively,
- the Chevalley group $G_2(3)$, denoted by `G2(3)`,
- the twisted Chevalley group ${}^3D_4(2)$, denoted by `3D4(2)`,
- the Suzuki group $Sz(8)$, denoted by `Sz(8)`,
- the Weyl groups $W(A_n)$, $W(B_n)$, $W(D_n)$, $W(E_n)$, and $W(F_4)$, denoted by `W(A`n`)`, `W(B`n`)`, `W(D`n`)`, `W(E`n`)`, and `W(F4)`, respectively,
- the sporadic simple groups Co_1, Co_2, Co_3, HS, J_2, M_{12}, M_{22}, M_{23}, M_{24}, and Mc, denoted by `Co1`, `Co2`, `Co3`, `HS`, `J2`, `M12`, `M22`, `M23`, `M24`, and `Mc`, respectively,
- a point stabilizer of index 11 in M_{11}, denoted by `M10`.

As mentioned above, the data assembled by the function `DisplayImfInvariants` are "cheap data" in the sense that they can be provided by the library without loading any of its large matrix files or performing any matrix calculations. The following function allows you to get proper access to these cheap data instead of just displaying them.

3 ▶ `ImfInvariants(` *dim*`,` *q* `)` F
▶ `ImfInvariants(` *dim*`,` *q*`,` *z* `)` F

`ImfInvariants` returns a record which provides some $\mathbb{Z}$-class invariants of the groups in the zth $\mathbb{Z}$-class in the qth $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension *dim*. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i.e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of z is 1. This value of z will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the qth $\mathbb{Q}$-class. The greatest legal value of *dim* is 31.

The resulting record contains six or seven components:

`size`
: the size of any representative group G,

`isSolvable`
: is `true` if G is solvable,

`isomorphismType`
: a text string describing the isomorphism type of G (in the same notation as used by the function `DisplayImfInvariants` above),

`elementaryDivisors`
: the elementary divisors of the associated Gram matrix F (in the same format as the result of the function `ElementaryDivisorsMat`, see 24.8.1),

`minimalNorm`
: the norm of the associated short vectors,

`sizesOrbitsShortVectors`
: the sizes of the orbits of short vectors under F,

`maximalQClass`
: the $\mathbb{Q}$-class number of an i.m.f. group in $GL_n(\mathbb{Q})$ that contains G as a subgroup (only in case that not G itself is an i.m.f. subgroup of $GL_n(\mathbb{Q})$).

Note that four of these data, namely the group size, the solvability, the isomorphism type, and the corresponding rational i.m.f. class, are not only $\mathbb{Z}$-class invariants, but also $\mathbb{Q}$-class invariants.

Note further that, though the isomorphism type is a $\mathbb{Q}$-class invariant, you will sometimes get different descriptions for different $\mathbb{Z}$-classes of the same $\mathbb{Q}$-class (as, e.g., for the classes 3.1.1 and 3.1.2 in the last example above). The purpose of this behaviour is to provide some more information about the underlying lattices.

```
gap> ImfInvariants( 8, 15, 1 );
rec( size := 2592, isSolvable := true, isomorphismType := "C2 x (S3 wr S3)",
  elementaryDivisors := [ 1, 3, 3, 3, 9, 9, 9, 27 ], minimalNorm := 8,
  sizesOrbitsShortVectors := [ 54 ], maximalQClass := 5 )
gap> ImfInvariants( 24, 1 ).size;
1040939685273333245386162176000 0
gap> ImfInvariants( 23, 5, 2 ).sizesOrbitsShortVectors;
[ 552, 53130 ]
gap> for i in [ 1 .. ImfNumberQClasses( 22 ) ] do
>    Print( ImfInvariants( 22, i ).isomorphismType, "\n" ); od;
C2 wr S22 = W(B22)
(C2 x PSU(6,2)).S3
(C2 x S3) wr S11 = (C2 x W(A2)) wr S11
(C2 x S12) wr C2 = (C2 x W(A11)) wr C2
C2 x S3 x S12 = C2 x W(A2) x W(A11)
(C2 x HS).C2
(C2 x Mc).C2
C2 x S23 = C2 x W(A22)
C2 x PSL(2,23)
C2 x PSL(2,23)
C2 x PGL(2,23)
C2 x PGL(2,23)
```

4 ▶ `ImfMatrixGroup(` *dim*, *q* `)` F
▶ `ImfMatrixGroup(` *dim*, *q*, *z* `)` F

`ImfMatrixGroup` is the essential extraction function of this library (note that its name has been changed from `ImfMatGroup` in GAP 3 to `ImfMatrixGroup` in GAP 4). It returns a representative group, G say, of the

z^{th} $\mathbb{Z}$-class in the qth $\mathbb{Q}$-class of i.m.f. integral matrix groups of dimension *dim*. A value $z \neq 1$ must not be specified if the $\mathbb{Z}$-classes are not known for the given dimension, i.e., if $dim > 11$ and $dim \notin \{13, 17, 19, 23\}$. The default value of z is 1. This value of z will be accepted even if the $\mathbb{Z}$-classes are not known. Then it specifies the only representative group which is available for the qth $\mathbb{Q}$-class. The greatest legal value of *dim* is 31.

```
gap> G := ImfMatrixGroup( 5, 1, 3 );
ImfMatrixGroup(5,1,3)
gap> for m in GeneratorsOfGroup( G ) do PrintArray( m ); od;
[ [  -1,   0,   0,   0,   0 ],
  [   0,   1,   0,   0,   0 ],
  [   0,   0,   0,   1,   0 ],
  [  -1,  -1,  -1,  -1,   2 ],
  [  -1,   0,   0,   0,   1 ] ]
[ [  0,  1,  0,  0,  0 ],
  [  0,  0,  1,  0,  0 ],
  [  0,  0,  0,  1,  0 ],
  [  1,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  1 ] ]
```

The attributes `Size` and `IsSolvable` will be properly set in the resulting matrix group G. In addition, it has two attributes `IsImfMatrixGroup` and `ImfRecord` where the first one is just a logical flag set to true and the latter one is a record. Except for the group size and the solvability flag, this record contains the same components as the resulting record of the function `ImfInvariants` described above (see 48.12.3), namely the components `isomorphismType`, `elementaryDivisors`, `minimalNorm`, and `sizesOrbitsShortVectors` and, if G is not a rational i.m.f. group, `maximalQClass`. Moreover, it has the two components

`form`
: the associated Gram matrix F, and

`repsOrbitsShortVectors`
: representatives of the orbits of short vectors under F.

The last one of these components will be required by the function `IsomorphismPermGroup` below.

Example:

```
gap> Size( G );
3840
gap> imf := ImfRecord( G );;
gap> imf.isomorphismType;
"C2 wr S5 = C2 x W(D5)"
gap> PrintArray( imf.form );
[ [  4,  0,  0,  0,  2 ],
  [  0,  4,  0,  0,  2 ],
  [  0,  0,  4,  0,  2 ],
  [  0,  0,  0,  4,  2 ],
  [  2,  2,  2,  2,  5 ] ]
gap> imf.elementaryDivisors;
[ 1, 4, 4, 4, 4 ]
gap> imf.minimalNorm;
4
```

If you want to perform calculations in such a matrix group G you should be aware of the fact that the permutation group routines of GAP are much more efficient than the matrix group routines. Hence we

recommend that you do your computations, whenever possible, in the isomorphic permutation group which is induced by the action of G on one of the orbits of the associated short vectors. You may call one of the following functions `IsomorphismPermGroup` or `IsomorphismPermGroupImfGroup` to get an isomorphism to such a permutation group (note that these GAP 4 functions have replaced the GAP 3 functions `PermGroup` and `PermGroupImfGroup`).

5 ► `IsomorphismPermGroup( G )` M

returns an isomorphism, φ say, from the given i.m.f. integral matrix group G to a permutation group $P := \varphi(G)$ acting on a minimal orbit, S say, of short vectors of G such that each matrix $m \in G$ is mapped to the permutation induced by its action on S.

Note that in case of a large orbit the construction of φ may be space and time consuming. Fortunately, there are only six $\mathbb{Q}$-classes in the library for which the smallest orbit of short vectors is of size greater than 20000, the worst case being the orbit of size 196560 for the Leech lattice ($dim = 24$, $q = 3$).

The inverse isomorphism φ^{-1} from P to G is constructed by determining a $\mathbb{Q}$-base $B \subset S$ of $\mathbb{Q}^{1\times dim}$ in S and, in addition, the associated base change matrix M which transforms B into the standard base of $\mathbb{Z}^{1\times dim}$. This allows a simple computation of the preimage $\varphi^{-1}(p)$ of any permutation $p \in P$ as follows. If, for $1 \le i \le dim$, b_i is the position number in S of the ith base vector in B, it suffices to look up the vector whose position number in S is the image of b_i under p and to multiply this vector by M to get the ith row of $\varphi^{-1}(p)$.

You may use the functions `Image` and `PreImage` (see 31.3.6 and 31.4.6) to switch from G to P and back from P to G.

As an example, let us continue the preceding example and compute the solvable residuum of the group G.

```
gap> # Perform the computations in an isomorphic permutation group.
gap> phi := IsomorphismPermGroup( G );;
gap> P := Image( phi );
Group([ (1,7,6)(2,9)(4,5,10), (2,3,4,5)(6,9,8,7) ])
gap> D := DerivedSubgroup( P );
Group([ (1,2,10,9)(3,8)(4,5)(6,7), (1,3,10,8)(2,5)(4,7)(6,9),
  (1,2,8,7,5)(3,4,6,10,9) ])
gap> Size( D );
960
gap> IsPerfectGroup( D );
true
gap> # We have found the solvable residuum of P,
gap> # now move the results back to the matrix group G.
gap> R := PreImage( phi, D );
<matrix group of size 960 with 3 generators>
gap> for m in GeneratorsOfGroup( R ) do PrintArray( m ); od;
[ [  -1,  -1,  -1,  -1,   2 ],
  [   0,  -1,   0,   0,   0 ],
  [   0,   0,   0,   1,   0 ],
  [   0,   0,   1,   0,   0 ],
  [  -1,  -1,   0,   0,   1 ] ]
[ [   0,   0,   0,   1,   0 ],
  [  -1,  -1,  -1,  -1,   2 ],
  [   0,   0,  -1,   0,   0 ],
  [   1,   0,   0,   0,   0 ],
  [   0,  -1,  -1,   0,   1 ] ]
[ [   0,  -1,   0,   0,   0 ],
  [   0,   0,   1,   0,   0 ],
```

```
[   0,   0,   0,  -1,   0 ],
[   1,   1,   1,   1,  -2 ],
[   0,   0,   1,   0,  -1 ] ]
```

6 ▶ `IsomorphismPermGroupImfGroup( G, n )` F

`IsomorphismPermGroupImfGroup` returns an isomorphism, φ say, from the given i.m.f. integral matrix group G to a permutation group P acting on the nth orbit, S say, of short vectors of G such that each matrix $m \in G$ is mapped to the permutation induced by its action on S.

The only difference to the above function `IsomorphismPermGroup` is that you can specify the orbit to be used. In fact, as the orbits of short vectors are sorted by increasing sizes, the function `IsomorphismPermGroup( G )` has been implemented such that it is equivalent to `IsomorphismPermGroupImfGroup( G, 1 )`.

```
gap> ImfInvariants( 12, 9 ).sizesOrbitsShortVectors;
[ 120, 300 ]
gap> G := ImfMatrixGroup( 12, 9 );
ImfMatrixGroup(12,9)
gap> phi1 := IsomorphismPermGroupImfGroup( G, 1 );;
gap> P1 := Image( phi1 );
<permutation group of size 2400 with 2 generators>
gap> LargestMovedPoint( P1 );
120
gap> phi2 := IsomorphismPermGroupImfGroup( G, 2 );;
gap> P2 := Image( phi2 );
<permutation group of size 2400 with 2 generators>
gap> LargestMovedPoint( P2 );
300
```

49 Semigroups

This chapter describes functions for creating semigroups and determining information about them.

1 ▶ `IsSemigroup( D )` P

returns `true` if the object D is a semigroup. A **semigroup** is a magma (see 33) with associative multiplication.

2 ▶ `Semigroup( gen1, gen2 ... )` F
▶ `Semigroup( gens )` F

In the first form, `Semigroup` returns the semigroup generated by the arguments *gen1*, *gen2*, ..., that is, the closure of these elements under multiplication. In the second form, `Semigroup` returns the semigroup generated by the elements in the homogeneous list *gens*; a square matrix as only argument is treated as one generator, not as a list of generators.

It is **not** checked whether the underlying multiplication is associative, use `Magma` (see 33.2.1) and `IsAssociative` (see 33.4.7) if you want to check whether a magma is in fact a semigroup.

```
gap> a:= Transformation([2, 3, 4, 1]);
Transformation( [ 2, 3, 4, 1 ] )
gap> b:= Transformation([2, 2, 3, 4]);
Transformation( [ 2, 2, 3, 4 ] )
gap> s:= Semigroup(a, b);
<semigroup with 2 generators>
```

3 ▶ `Subsemigroup( S, gens )` F
▶ `SubsemigroupNC( S, gens )` F

are just synonyms of `Submagma` and `SubmagmaNC`, respectively (see 33.2.7).

```
gap> a:=GeneratorsOfSemigroup(s)[1];
Transformation( [ 2, 3, 4, 1 ] )
gap> t:=Subsemigroup(s,[a]);
<semigroup with 1 generator>
```

4 ▶ `SemigroupByGenerators( gens )` O

is the underlying operation of `Semigroup` (see 49).

5 ▶ `AsSemigroup( C )` A

If C is a collection whose elements form a semigroup (see 49) then `AsSemigroup` returns this semigroup. Otherwise `fail` is returned.

6 ▶ `AsSubsemigroup( D, C )` O

Let D be a domain and C a collection. If C is a subset of D that forms a semigroup then `AsSubsemigroup` returns this semigroup, with parent D. Otherwise `fail` is returned.

7 ► GeneratorsOfSemigroup(*S*) A

Semigroup generators of a semigroup D are the same as magma generators (see 33.4.1).

```
gap> GeneratorsOfSemigroup(s);
[ Transformation( [ 2, 3, 4, 1 ] ), Transformation( [ 2, 2, 3, 4 ] ) ]
gap> GeneratorsOfSemigroup(t);
[ Transformation( [ 2, 3, 4, 1 ] ) ]
```

8 ► FreeSemigroup([*wfilt*,]*rank*) F
► FreeSemigroup([*wfilt*,]*rank*, *name*) F
► FreeSemigroup([*wfilt*,]*name1*, *name2*, ...) F
► FreeSemigroup([*wfilt*,]*names*) F
► FreeSemigroup([*wfilt*,]infinity, *name*, *init*) F

Called in the first form, FreeSemigroup returns a free semigroup on *rank* generators. Called in the second form, FreeSemigroup returns a free semigroup on *rank* generators, printed as *name*1, *name*2 etc., that is, each name is the concatenation of the string *name* and an integer from 1 to *range*. Called in the third form, FreeSemigroup returns a free semigroup on as many generators as arguments, printed as *name1*, *name2* etc. Called in the fourth form, FreeSemigroup returns a free semigroup on as many generators as the length of the list *names*, the i-th generator being printed as *names*[*i*]. Called in the fifth form, FreeSemigroup returns a free semigroup on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either IsSyllableWordsFamily or IsLetterWordsFamily or IsWLetterWordsFamily or IsBLetterWordsFamily. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

```
gap> f1 := FreeSemigroup( 3 );
<free semigroup on the generators [ s1, s2, s3 ]>
gap> f2 := FreeSemigroup( 3 , "generator" );
<free semigroup on the generators [ generator1, generator2, generator3 ]>
gap> f3 := FreeSemigroup( "gen1" , "gen2" );
<free semigroup on the generators [ gen1, gen2 ]>
gap> f4 := FreeSemigroup( ["gen1" , "gen2"] );
<free semigroup on the generators [ gen1, gen2 ]>
```

9 ► SemigroupByMultiplicationTable(*A*) F

returns the semigroup whose multiplication is defined by the square matrix A (see 33.3.1) if such a semigroup exists. Otherwise fail is returned.

The following functions determine information about semigroups:

10 ► IsRegularSemigroup(*S*) P

returns true if S is regular—i.e. if every D class of S is regular.

11 ► IsRegularSemigroupElement(*S*, *x*) O

returns true if x has a general inverse in S—i.e. there is an element $y \in S$ such that $xyx = x$ and $yxy = y$.

12 ► IsSimpleSemigroup(*S*) P

is true if and only if the semigroup has no proper ideals.

13 ► IsZeroSimpleSemigroup(*S*) P

is true if and only if the semigroup has no proper ideals except for 0, where S is a semigroup with zero. If the semigroup does not find its zero, then a break-loop is entered.

14 ▶ `IsZeroGroup( S )` P

is `true` if and only if the semigroup is a group with zero adjoined.

15 ▶ `IsReesCongruenceSemigroup( S )` P

returns `true` if S is a Rees Congruence semigroup, that is, if all congruences of S are Rees Congruences.

49.1 Making transformation semigroups

Cayley's Theorem gives special status to semigroups of transformations, and accordingly there are special functions to deal with them, and to create them from other finite semigroups.

1 ▶ `IsTransformationSemigroup( obj )` P
▶ `IsTransformationMonoid( obj )` P

A transformation semigroup (resp. monoid) is a subsemigroup (resp. submonoid) of the full transformation monoid. Note that for a transformation semigroup to be a transformation monoid we necessarily require the identity transformation to be an element.

2 ▶ `DegreeOfTransformationSemigroup( S )` A

The number of points the semigroup acts on.

3 ▶ `IsomorphismTransformationSemigroup( S )` A
▶ `HomomorphismTransformationSemigroup( S, r )` O

IsomorphismTransformationSemigroup is a generic attribute which is a transformation semigroup isomorphic to S (if such can be computed). In the case of an fp- semigroup, a todd-coxeter will be attempted. For a semigroup of endomorphisms of a finite domain of n elements, it will be to a semigroup of transformations of $\{1, \ldots, n\}$. Otherwise, it will be the right regular representation on S or S^1 if S has no MultiplicativeNeutralElement.

HomomorphismTransformationSemigroup finds a representation of S as transformations of the set of equivalence classes of the right congruence r.

4 ▶ `IsFullTransformationSemigroup( obj )` P

5 ▶ `FullTransformationSemigroup( degree )` F

Returns the full transformation semigroup of degree *degree*.

49.2 Ideals of semigroups

Ideals of semigroups are the same as ideals of the semigroup when considered as a magma. For documentation on ideals for magmas, see `Magma` (33.2.1).

1 ▶ `SemigroupIdealByGenerators( S, gens )` O

S is a semigroup, *gens* is a list of elements of S. Returns the two-sided ideal of S generated by *gens*.

2 ▶ `ReesCongruenceOfSemigroupIdeal( I )` A

A two sided ideal I of a semigroup S defines a congruence on S given by $\Delta \cup I \times I$.

3 ▶ `IsLeftSemigroupIdeal( I )` P
▶ `IsRightSemigroupIdeal( I )` P
▶ `IsSemigroupIdeal( I )` P

Categories of semigroup ideals.

49.3 Congruences for semigroups

An equivalence or a congruence on a semigroup is the equivalence or congruence on the semigroup considered as a magma. So, to deal with equivalences and congruences on semigroups, magma functions are used. For documentation on equivalences and congruences for magmas, see `Magma` (33.2.1).

1 ▶ `IsSemigroupCongruence( c )` P

a magma congruence c on a semigroup.

2 ▶ `IsReesCongruence( c )` P

returns true precisely when the congruence c has at most one nonsingleton congruence class.

49.4 Quotients

Given a semigroup and a congruence on the semigroup, one can construct a new semigroup: the quotient semigroup. The following functions deal with quotient semigroups in GAP.

1 ▶ `IsQuotientSemigroup( S )` C

is the category of semigroups constructed from another semigroup and a congruence on it

Elements of a quotient semigroup are equivalence classes of elements of `QuotientSemigroupPreimage(`S`)` under the congruence `QuotientSemigroupCongruence(`S`)`.

It is probably most useful for calculating the elements of the equivalence classes by using Elements or by looking at the images of elements of the `QuotientSemigroupPreimage(`S`)` under `QuotientSemigroupHomomorphism(`S`):QuotientSemigroupPreimage(`S`)` $\rightarrow S$.

For intensive computations in a quotient semigroup, it is probably worthwhile finding another representation as the equality test could involve enumeration of the elements of the congruence classes being compared.

2 ▶ `HomomorphismQuotientSemigroup( cong )` F

for a congruence *cong* and a semigroup S. Returns the homomorphism from S to the quotient of S by *cong*.

3 ▶ `QuotientSemigroupPreimage( S )` A
▶ `QuotientSemigroupCongruence( S )` A
▶ `QuotientSemigroupHomomorphism( S )` A

for a quotient semigroup S.

49.5 Green's Relations

Green's equivalence relations play a very important role in semigroup theory. In this section we describe how they can be used in GAP.

The five Green's relations are R, L, J, H, D: two elements x, y from S are R-related if and only if $xS^1 = yS^1$, L-related if and only if $S^1x = S^1y$ and J-related if and only if $S^1xS^1 = S^1yS^1$; finally, $H = R \wedge L$, and $D = R \circ L$.

Recall that relations R, L and J induce a partial order among the elements of the semigroup: for two elements x, y from S, we say that x is less than or equal to y in the order on R if $xS^1 \subseteq yS^1$; similarly, x is less than or equal to y under L if $S^1x \subseteq S^1y$; finally x is less than or equal to y under J if $S^1xS^1 \subseteq S^1tS^1$. We extend this preorder to a partial order on equivalence classes in the natural way.

1 ▶ `GreensRRelation(` *semigroup* `)` A
▶ `GreensLRelation(` *semigroup* `)` A
▶ `GreensJRelation(` *semigroup* `)` A
▶ `GreensDRelation(` *semigroup* `)` A
▶ `GreensHRelation(` *semigroup* `)` A

The Green's relations (which are equivalence relations) are attributes of the semigroup *semigroup*.

2 ▶ `IsGreensRelation(` *equiv-relation* `)` P
▶ `IsGreensRRelation(` *equiv-relation* `)` P
▶ `IsGreensLRelation(` *equiv-relation* `)` P
▶ `IsGreensJRelation(` *equiv-relation* `)` P
▶ `IsGreensHRelation(` *equiv-relation* `)` P
▶ `IsGreensDRelation(` *equiv-relation* `)` P

return `true` if the equivalence relation *equiv-relation* is a Green's relation of any type, or of R, L, J, H, D type, respectively, or `false` otherwise.

3 ▶ `IsGreensClass(` *equiv-class* `)` P
▶ `IsGreensRClass(` *equiv-class* `)` P
▶ `IsGreensLClass(` *equiv-class* `)` P
▶ `IsGreensJClass(` *equiv-class* `)` P
▶ `IsGreensHClass(` *equiv-class* `)` P
▶ `IsGreensDClass(` *equiv-class* `)` P

return `true` if the equivalence class *equiv-class* is a Green's class of any type, or of R, L, J, H, D type, respectively, or `false` otherwise.

4 ▶ `IsGreensLessThanOrEqual(` *C1*`,` *C2* `)` O

returns `true` if the greens class *C1* is less than or equal to *C2* under the respective ordering (as defined above), and `false` otherwise.

Only defined for R, L and J classes.

5 ▶ `RClassOfHClass(` *H* `)` A
▶ `LClassOfHClass(` *H* `)` A

are attributes reflecting the natural ordering over the various Green's classes. `RClassOfHClass` and `LClassOfHClass` return the R and L classes respectively in which an H class is contained.

6 ▶ `EggBoxOfDClass(` *Dclass* `)` A

returns for a Green's D class *Dclass* a matrix whose rows represent R classes and columns represent L classes. The entries are the H classes.

7 ▶ `DisplayEggBoxOfDClass(` *Dclass* `)` F

displays a "picture" of the D class *Dclass*, as an array of 1s and 0s. A 1 represents a group H class.

8 ▶ `GreensRClassOfElement(` *S*`,` *a* `)` O
▶ `GreensLClassOfElement(` *S*`,` *a* `)` O
▶ `GreensDClassOfElement(` *S*`,` *a* `)` O
▶ `GreensJClassOfElement(` *S*`,` *a* `)` O
▶ `GreensHClassOfElement(` *S*`,` *a* `)` O

Creates the X class of the element a in the semigroup S where X is one of L, R, D, J or H.

9 ▶ `GreensRClasses( semigroup )` O
▶ `GreensLClasses( semigroup )` O
▶ `GreensJClasses( semigroup )` O
▶ `GreensDClasses( semigroup )` O
▶ `GreensHClasses( semigroup )` O

return the R, L, J, H, or D Green's classes, respectively for semigroup *semigroup*.

10 ▶ `GroupHClassOfGreensDClass( Dclass )` A

for a D class *Dclass* of a semigroup, returns a group H class of the D class, or `fail` if there is no group H class.

11 ▶ `IsGroupHClass( Hclass )` P

returns `true` if the Greens H class *Hclass* is a group, which in turn is true if and only if *Hclass*^2 intersects *Hclass*.

12 ▶ `IsRegularDClass( Dclass )` P

returns `true` if the Greens D class *Dclass* is regular. A D class is regular if and only if each of its elements is regular, which in turn is true if and only if any one element of *Dclass* is regular. Idempotents are regular since $eee = e$ so it follows that a Greens D class containing an idempotent is regular. Conversely, it is true that a regular D class must contain at least one idempotent. (See [How76], Prop. 3.2).

49.6 Rees Matrix Semigroups

In this section we describe GAP functions for Rees matrix semigroups and Rees 0-matrix semigroups. The importance of this construction is that Rees Matrix semigroups over groups are exactly the completely simple semigroups, and Rees 0-matrix semigroups over groups are the completely 0-simple semigroups

Recall that a Rees Matrix semigroup is constructed from a semigroup (the underlying semigroup), and a matrix. A Rees Matrix semigroup element is a triple (s, i, *lambda*) where s is an element of the underlying semigroup S and i, *lambda* are indices. This can be thought of as a matrix with zero everywhere except for an occurrence of s at row i and column *lambda*. The multiplication is defined by $(s, i, \lambda) * (t, j, \mu) = (sP_{\lambda j}t, i, \mu)$ where P is the defining matrix of the semigroup. In the case that the underlying semigroup has a zero we can make the ReesZeroMatrixSemigroup, wherein all elements whose s entry is the zero of the underlying semigroup are identified to the unique zero of the Rees 0-matrix semigroup.

1 ▶ `ReesMatrixSemigroup( S, matrix )` F

for a semigroup S and *matrix* whose entries are in S. Returns the Rees Matrix semigroup with multiplication defined by *matrix*.

2 ▶ `ReesZeroMatrixSemigroup( S, matrix )` F

for a semigroup S with zero, and *matrix* over S returns the Rees 0-Matrix semigroup such that all elements $(i, 0, \lambda)$ are identified to zero.

The zero in S is found automatically. If one cannot be found, an error is signalled.

3 ▶ `IsReesMatrixSemigroup( T )` P

returns `true` if the object T is a (whole) Rees matrix semigroup.

4 ▶ `IsReesZeroMatrixSemigroup( T )` P

returns `true` if the object T is a (whole) Rees 0-matrix semigroup.

5 ▶ `ReesMatrixSemigroupElement( R, a, i, lambda )` F
▶ `ReesZeroMatrixSemigroupElement( R, a, i, lambda )` F

for a Rees matrix semigroup R, a in `UnderlyingSemigroup(`R`)`, i and *lambda* in the row (resp. column) ranges of R, returns the element of R corresponding to the matrix with zero everywhere and a in row i and column x.

6 ▶ `IsReesMatrixSemigroupElement( e )` C
▶ `IsReesZeroMatrixSemigroupElement( e )` C

is the category of elements of a Rees (0-) matrix semigroup. Returns true if e is an element of a Rees Matrix semigroup.

7 ▶ `SandwichMatrixOfReesMatrixSemigroup( R )` A
▶ `SandwichMatrixOfReesZeroMatrixSemigroup( R )` A

each return the defining matrix of the Rees (0-) matrix semigroup.

8 ▶ `RowIndexOfReesMatrixSemigroupElement( x )` A
▶ `RowIndexOfReesZeroMatrixSemigroupElement( x )` A
▶ `ColumnIndexOfReesMatrixSemigroupElement( x )` A
▶ `ColumnIndexOfReesZeroMatrixSemigroupElement( x )` A
▶ `UnderlyingElementOfReesMatrixSemigroupElement( x )` A
▶ `UnderlyingElementOfReesZeroMatrixSemigroupElement( x )` A

For an element x of a Rees Matrix semigroup, of the form `(`s`, `i`, `*lambda*`)`, the row index is i, the column index is *lambda* and the underlying element is s. If we think of an element as a matrix then this corresponds to the row where the non-zero entry is, the column where the non-zero entry is and the entry at that position, respectively.

9 ▶ `ReesZeroMatrixSemigroupElementIsZero( x )` P

returns `true` if x is the zero of the Rees 0-matrix semigroup.

10 ▶ `AssociatedReesMatrixSemigroupOfDClass( D )` A

Given a regular D class of a finite semigroup, it can be viewed as a Rees matrix semigroup by identifying products which do not lie in the D class with zero, and this is what it is returned.

Formally, let I_1 be the ideal of all J classes less than or equal to D, I_2 the ideal of all J classes **strictly** less than D, and ρ the Rees congruence associated with I_2. Then I/ρ is zero-simple. Then `AssociatedReesMatrixSemigroupOfDClass(` D `)` returns this zero-simple semigroup as a Rees matrix semigroup.

11 ▶ `IsomorphismReesMatrixSemigroup( obj )` A

an isomorphism to a Rees matrix semigroup over a group (resp. zero group).

50 Monoids

This chapter describes functions for monoids. Currently there are only few of them. More general functions for magmas and semigroups can be found in Chapters 33 and 49.

1 ▶ `IsMonoid(` *D* `)` P

A **monoid** is a magma-with-one (see 33) with associative multiplication.

2 ▶ `Monoid(` *gen1*`,` *gen2* `...)` F
▶ `Monoid(` *gens* `)` F
▶ `Monoid(` *gens*`,` *id* `)` F

In the first form, `Monoid` returns the monoid generated by the arguments *gen1*, *gen2* ..., that is, the closure of these elements under multiplication and taking the 0-th power. In the second form, `Monoid` returns the monoid generated by the elements in the homogeneous list *gens*; a square matrix as only argument is treated as one generator, not as a list of generators. In the third form, `Monoid` returns the monoid generated by the elements in the homogeneous list *gens*, with identity *id*.

It is **not** checked whether the underlying multiplication is associative, use `MagmaWithOne` (see 33.2.2) and `IsAssociative` (see 33.4.7) if you want to check whether a magma-with-one is in fact a monoid.

3 ▶ `Submonoid(` *M*`,` *gens* `)` F
▶ `SubmonoidNC(` *M*`,` *gens* `)` F

are just synonyms of `SubmagmaWithOne` and `SubmagmaWithOneNC`, respectively (see 33.2.8).

4 ▶ `MonoidByGenerators(` *gens* `)` O
▶ `MonoidByGenerators(` *gens*`,` *one* `)` O

is the underlying operation of `Monoid` (see 50).

5 ▶ `AsMonoid(` *C* `)` A

If *C* is a collection whose elements form a monoid (see 50) then `AsMonoid` returns this monoid. Otherwise `fail` is returned.

6 ▶ `AsSubmonoid(` *D*`,` *C* `)` O

Let *D* be a domain and *C* a collection. If *C* is a subset of *D* that forms a monoid then `AsSubmonoid` returns this monoid, with parent *D*. Otherwise `fail` is returned.

7 ▶ `GeneratorsOfMonoid(` *M* `)` A

Monoid generators of a monoid *M* are the same as magma-with-one generators (see 33.4.2).

8 ▶ `TrivialSubmonoid(` *M* `)` A

is just a synonym for `TrivialSubmagmaWithOne` (see 33.4.14).

9 ▶ `FreeMonoid( [`*wfilt*`, ]`*rank*` )` F
▶ `FreeMonoid( [`*wfilt*`, ]`*rank*`, `*name*` )` F
▶ `FreeMonoid( [`*wfilt*`, ]`*name1*`, `*name2*`, ... )` F
▶ `FreeMonoid( [`*wfilt*`, ]`*names*` )` F
▶ `FreeMonoid( [`*wfilt*`, ]infinity, `*name*`, `*init*` )` F

Called in the first form, `FreeMonoid` returns a free monoid on *rank* generators. Called in the second form, `FreeMonoid` returns a free monoid on *rank* generators, printed as *name*`1`, *name*`2` etc., that is, each name is the concatenation of the string *name* and an integer from `1` to *range*. Called in the third form, `FreeMonoid` returns a free monoid on as many generators as arguments, printed as *name1*, *name2* etc. Called in the fourth form, `FreeMonoid` returns a free monoid on as many generators as the length of the list *names*, the i-th generator being printed as *names*`[`i`]`. Called in the fifth form, `FreeMonoid` returns a free monoid on infinitely many generators, where the first generators are printed by the names in the list *init*, and the other generators by *name* and an appended number.

If the extra argument *wfilt* is given, it must be either `IsSyllableWordsFamily` or `IsLetterWordsFamily` or `IsWLetterWordsFamily` or `IsBLetterWordsFamily`. The filter then specifies the representation used for the elements of the free group (see 35.6). If no such filter is given, a letter representation is used.

10 ▶ `MonoidByMultiplicationTable( `A` )` F

returns the monoid whose multiplication is defined by the square matrix A (see 33.3.1) if such a monoid exists. Otherwise `fail` is returned.

51 Finitely Presented Semigroups and Monoids

A **finitely presented semigroup** (resp. **finitely presented monoid**) is a quotient of a free semigroup (resp. free monoid) on a finite number of generators over a finitely generated congruence on the free semigroup (resp. free monoid).

Finitely presented semigroups are obtained by factoring a free semigroup by a set of relations (a generating set for the congruence), ie, a set of pairs of words in the free semigroup.

```
gap> f:=FreeSemigroup("a","b");;
gap> x:=GeneratorsOfSemigroup(f);;
gap> s:=f/[ [x[1]*x[2],x[2]*x[1]] ];
<fp semigroup on the generators [ a, b ]>
gap> GeneratorsOfSemigroup(s);
[ a, b ]
gap> RelationsOfFpSemigroup(s);
[ [ a*b, b*a ] ]
```

Finitely presented monoids are obtained by factoring a free monoid by a set of relations, i.e. a set of pairs of words in the free monoid.

```
gap> f:=FreeMonoid("a","b");;
gap> x:=GeneratorsOfMonoid(f);
[ a, b ]
gap> e:=Identity(f);
<identity ...>
gap> m:=f/[ [x[1]*x[2],e] ];
<fp monoid on the generators [ a, b ]>
gap> RelationsOfFpMonoid(m);
[ [ a*b, <identity ...> ] ]
```

Notice that for GAP a finitely presented monoid is not a finitely presented semigroup.

```
gap> IsFpSemigroup(m);
false
```

However, one can build a finitely presented semigroup isomorphic to that finitely presented monoid (see 51.1.3).

Also note that is not possible to refer to the generators by their names. These names are not variables, but just display figures. So, if one wants to access the generators by their names, one first has to introduce the respective variables and to assign the generators to them.

```
gap> f:=FreeSemigroup("a","b");;
gap> x:=GeneratorsOfSemigroup(f);;
gap> s:=f/[ [x[1]*x[2],x[2]*x[1]] ];;
gap> a;
Variable: 'a' must have a value

gap> a:=GeneratorsOfSemigroup(s)[1];
a
gap> b:=GeneratorsOfSemigroup(s)[2];
b
gap> a in f;
false
gap> a in s;
true
```

The generators of the free semigroup (resp. free monoid) are different from the generators of the finitely presented semigroup (resp. finitely presented monoid) (even though they are displayed by the same names). This means that words in the generators of the free semigroup (resp. free monoid) are not elements of the finitely presented semigroup (resp. finitely presented monoid). Conversely elements of the finitely presented semigroup (resp. finitely presented monoid) are not words of the free semigroup (resp. free monoid).

Calculations comparing elements of an finitely presented semigroup may run into problems: there are finitely presented semigroups for which no algorithm exists (it is known that no such algorithm can exist) that will tell for two arbitrary words in the generators whether the corresponding elements in the finitely presented semigroup are equal. Therefore the methods used by GAP to compute in finitely presented semigroups may run into warning errors, run out of memory or run forever. If the finitely presented semigroup is (by theory) known to be finite the algorithms are guaranteed to terminate (if there is sufficient memory available), but the time needed for the calculation cannot be bounded a priori. The same can be said for monoids. (See 51.5.)

```
gap> a*b=a^5;
false
gap> a^5*b^2*a=a^6*b^2;
true
```

Note than elements of a finitely presented semigroup (or monoid) are not printed in a unique way:

```
gap> a^5*b^2*a;
a^5*b^2*a
gap> a^6*b^2;
a^6*b^2
```

1 ▶ `IsSubsemigroupFpSemigroup( `*t*` )` A

true if *t* is a finitely presented semigroup or a subsemigroup of a finitely presented semigroup (generally speaking, such a subsemigroup can be constructed with `Semigroup(`*gens*`)`, where *gens* is a list of elements of a finitely presented semigroup).

2 ▶ `IsSubmonoidFpMonoid( `*t*` )` A

true if *t* is a finitely presented monoid or a submonoid of a finitely presented monoid (generally speaking, such a semigroup can be constructed with `Monoid(`*gens*`)`, where *gens* is a list of elements of a finitely presented monoid).

A submonoid of a monoid has the same identity as the monoid.

3 ▶ `IsFpSemigroup( s )` P

is a synonym for `IsSubsemigroupFpSemigroup(`s`)` and `IsWholeFamily(`s`)` (this is because a subsemigroup of a finitely presented semigroup is not necessarily finitely presented).

4 ▶ `IsFpMonoid( m )` P

is a synonym for `IsSubmonoidFpMonoid(`m`)` and `IsWholeFamily(`m`)` (this is because a submonoid of a finitely presented monoid is not necessarily finitely presented).

5 ▶ `IsElementOfFpSemigroup( elm )` C

returns true if *elm* is an element of a finitely presented semigroup.

6 ▶ `IsElementOfFpMonoid( elm )` C

returns true if *elm* is an element of a finitely presented monoid.

7 ▶ `FpGrpMonSmgOfFpGrpMonSmgElement( elm )` O

returns the finitely presented group, monoid or semigroup to which *elm* belongs

```
gap> f := FreeSemigroup("a","b");;
gap> a := GeneratorsOfSemigroup( f )[ 1 ];;
gap> b := GeneratorsOfSemigroup( f )[ 2 ];;
gap> s := f / [ [ a^2 , a*b ] ];;
gap> IsFpSemigroup( s );
true
gap> t := Semigroup( [ GeneratorsOfSemigroup( s )[ 1 ] ]);
<semigroup with 1 generator>
gap> IsSubsemigroupFpSemigroup( t );
true
gap> IsElementOfFpSemigroup( GeneratorsOfSemigroup( t )[ 1 ] );
true
```

51.1 Creating Finitely Presented Semigroups

1 ▶ $F/rels$

creates a finitely presented semigroup given by the presentation $\langle gens \mid rels \rangle$ where *gens* are the generators of the free semigroup F, and the relations *rels* are entered as pairs of words in the generators of the free semigroup.

```
gap> f:=FreeSemigroup(3);;
gap> s:=GeneratorsOfSemigroup(f);;
gap> f/[ [s[1]*s[2]*s[1],s[1]] , [s[2]^4,s[1]] ];
<fp semigroup on the generators [ s1, s2, s3 ]>
```

One may also call the following functions to construct finitely presented semigroups:

2 ▶ `FactorFreeSemigroupByRelations( f, rels )` F

for a free semigroup f and *rels* is a list of pairs of elements of f. Returns the finitely presented semigroup which is the quotient of f by the least congruence on f generated by the pairs in *rels*.

```
gap> FactorFreeSemigroupByRelations(f,[[s[1]*s[2]*s[1],s[1]],[s[2]^4,s[1]]]);
<fp semigroup on the generators [ s1, s2, s3 ]>
```

Finally, if one has a finitely presented group or a finitely presented monoid, to find an isomorphic finitely presented semigroup use

3 ▶ `IsomorphismFpSemigroup( s )` A

for a semigroup s returns an isomorphism from s to a finitely presented semigroup

```
gap> f := FreeGroup(2);;
gap> g := f/[f.1^4,f.2^5];
<fp group on the generators [ f1, f2 ]>
gap> phi := IsomorphismFpSemigroup(g);
MappingByFunction( <fp group on the generators
[ f1, f2 ]>, <fp semigroup on the generators
[ <identity ...>, f1^-1, f1, f2^-1, f2
 ]>, function( x ) ... end, function( x ) ... end )
gap> s := Range(phi);
<fp semigroup on the generators [ <identity ...>, f1^-1, f1, f2^-1, f2 ]>
```

51.2 Comparison of Elements of Finitely Presented Semigroups

1 ▶ a `=` b

Two elements of a finitely presented semigroup are equal if they are equal in the semigroup. Nevertheless they may be represented as different words in the generators. Because of the fundamental problems mentioned in the introduction to this chapter such a test may take a very long time and cannot be guaranteed to finish (see 51.5).

51.3 Preimages in the Free Semigroup

1 ▶ `FreeSemigroupOfFpSemigroup( s )` A

returns the underlying free semigroup for the finitely presented semigroup s, ie, the free semigroup over which s is defined as a quotient (this is the free semigroup generated by the free generators provided by `FreeGeneratorsOfFpSemigroup(`s`)`).

2 ▶ `FreeGeneratorsOfFpSemigroup( s )` A

returns the underlying free generators corresponding to the generators of the finitely presented semigroup s.

3 ▶ `RelationsOfFpSemigroup( s )` A

returns the relations of the finitely presented semigroup s as pairs of words in the free generators provided by `FreeGeneratorsOfFpSemigroup(`s`)`.

```
gap> f := FreeSemigroup( "a" , "b" );;
gap> a := GeneratorsOfSemigroup( f )[ 1 ];;
gap> b := GeneratorsOfSemigroup( f )[ 2 ];;
gap> s := f / [ [ a^3 , a ] , [ b^3 , b ] , [ a*b , b*a ] ];
<fp semigroup on the generators [ a, b ]>
gap> Size( s );
8
gap> fs := FreeSemigroupOfFpSemigroup( s );;
gap> f = fs;
true
gap> FreeGeneratorsOfFpSemigroup( s );
[ a, b ]
gap> RelationsOfFpSemigroup( s );
[ [ a^3, a ], [ b^3, b ], [ a*b, b*a ] ]
```

Elements of a finitely presented semigroup are not words, but are represented using a word from the free semigroup as representative.

4 ► `UnderlyingElement( elm )` O

for an element *elm* of a finitely presented semigroup, it returns the word from the free semigroup that is used as a representative for *elm*.

```
gap> w := GeneratorsOfSemigroup(s)[1] * GeneratorsOfSemigroup(s)[2];
a*b
gap> IsWord (w );
false
gap> ue := UnderlyingElement( w );
a*b
gap> IsWord( ue );
true
```

5 ► `ElementOfFpSemigroup( fam, w )` O

for a family *fam* of elements of a finitely presented semigroup and a word w in the free generators underlying this finitely presented semigroup, this operation creates the element of the finitely presented semigroup with the representative w in the free semigroup.

```
gap> fam := FamilyObj( GeneratorsOfSemigroup(s)[1] );;
gap> ge := ElementOfFpSemigroup( fam, a*b );
a*b
gap> ge in f;
false
gap> ge in s;
true
```

51.4 Finitely presented monoids

1 ► *F*/*rels*

creates a finitely presented monoid given by the monoid presentation $\langle gens \mid rels \rangle$ where *gens* are the generators of the free monoid F, and the relations *rel* are entered as pairs of words in both the identity and the generators of the free monoid.

```
gap> f := FreeMonoid( 3 );
<free monoid on the generators [ m1, m2, m3 ]>
gap> x := GeneratorsOfMonoid( f );
[ m1, m2, m3 ]
gap> e:= Identity ( f );
<identity ...>
gap> m := f/[ [x[1]^3,e] , [x[1]*x[2],x[2] ]];
<fp monoid on the generators [ m1, m2, m3 ]>
```

The functionality available for finitely presented monoids is essentially the same as that available for finitely presented semigroups, and thus the previous sections apply (with the obvious changes) to finitely presented monoids.

51.5 Rewriting Systems and the Knuth-Bendix Procedure

If a finitely presented semigroup has a confluent rewriting system then it has a solvable word problem, that is, there is an algorithm to decide when two words in the free underlying semigroup represent the same element of the finitely presented semigroup. Indeed, once we have a confluent rewriting system, it is possible to successfully test that two words represent the same element in the semigroup, by reducing both words using the rewriting system rules. This is, at the moment, the method that GAP uses to check equality in finitely presented semigroups and monoids.

1 ▶ `ReducedConfluentRewritingSystem( `*S*` )` A
▶ `ReducedConfluentRewritingSystem( `*S*` , `*ordering*` )` A

in the first form returns a reduced confluent rewriting system of the finitely presented semigroup or monoid S with respect to the length plus lexicographic ordering on words (also called the shortlex ordering; for the definition see for example Sims [Sim94]).

In the second form it returns a reduced confluent rewriting system of the finitely presented semigroup or monoid S with respect to the reduction ordering *ordering* (see 29).

Notice that this might not terminate. In particular, if the semigroup or monoid S does not have a solvable word problem then it this will certainly never end. Also, in this case, the object returned is an immutable rewriting system, because once we have a confluent rewriting system for a finitely presented semigroup or monoid we do not want to allow it to change (as it was most probably very time consuming to get it in the first place). Furthermore, this is also an attribute storing object (see 13.4).

```
gap> f := FreeSemigroup( "a" , "b" );;
gap> a := GeneratorsOfSemigroup( f )[ 1 ];;
gap> b := GeneratorsOfSemigroup( f )[ 2 ];;
gap> g := f /  [ [ a^2 , a*b ] , [ a^4 , b] ];;
gap> rws := ReducedConfluentRewritingSystem(g);
Rewriting System for Semigroup( [ a, b ] ) with rules
[ [ a*b, a^2 ], [ a^4, b ], [ b*a, a^2 ], [ b^2, a^2 ] ]
```

The creation of a reduced confluent rewriting system for a semigroup or for a monoid, in GAP, uses the Knuth-Bendix procedure for strings, which manipulates a rewriting system of the semigroup or monoid and attempts to make it confluent (See 36. See also Sims [Sim94]). (Since the word problem for semigroups/monoids is not solvable in general, Knuth-Bendix procedure cannot always terminate).

In order to apply this procedure we will build a rewriting system for the semigroup or monoid, which we will call a **Knuth-Bendix Rewriting System** (we need to define this because we need the rewriting system to store some information needed for the implementation of the Knuth-Bendix procedure).

Actually, Knuth-Bendix Rewriting Systems do not only serve this purpose. Indeed these are objects which are mutable and which can be manipulated (see 36).

Note that the implemented version of the Knuth-Bendix procedure, in GAP returns, if it terminates, a confluent rewriting system which is reduced. Also, a reduction ordering has to be specified when building a rewriting system. If none is specified, the shortlex ordering is assumed (note that the procedure may terminate with a certain ordering and not with another one).

On Unix systems it is possible to replace the built-in Knuth-Bendix by other routines, for example the package `kbmag` offers such a possibility.

2 ▶ `KB_REW` V
▶ `GAPKB_REW` V

KB_REW is a global record variable whose components contain functions used for Knuth-Bendix. By default `KB_REW` is assigned to `GAPKB_REW`, which contains the KB functions provided by the GAP library.

3 ▶ KnuthBendixRewritingSystem(*s*,*wordord*)
▶ KnuthBendixRewritingSystem(*m*,*wordord*)

in the first form, for a semigroup s and a reduction ordering for the underlying free semigroup, it returns the Knuth-Bendix rewriting system of the finitely presented semigroup s using the reduction ordering *wordord*. In the second form, for a monoid m and a reduction ordering for the underlying free semigroup, it returns the Knuth-Bendix rewriting system of the finitely presented semigroup s using the reduction ordering *wordord*.

4 ▶ SemigroupOfRewritingSystem(*rws*) A

returns the semigroup over which *rws* is a rewriting system

5 ▶ MonoidOfRewritingSystem(*rws*) A

returns the monoid over which *rws* is a rewriting system

6 ▶ FreeSemigroupOfRewritingSystem(*rws*) A

returns the free semigroup over which *rws* is a rewriting system

7 ▶ FreeMonoidOfRewritingSystem(*rws*) A

returns the free monoid over which *rws* is a rewriting system

```
gap> f1 := FreeSemigroupOfRewritingSystem(rws);
<free semigroup on the generators [ a, b ]>
gap> f1=f;
true
gap> g1 := SemigroupOfRewritingSystem(rws);
<fp semigroup on the generators [ a, b ]>
gap> g1=g;
true
```

As mentioned before, having a confluent rewriting system, one can decide whether two words represent the same element of a finitely presented semigroup (or finitely presented monoid).

```
gap> a := GeneratorsOfSemigroup( g )[ 1 ];
a
gap> b := GeneratorsOfSemigroup( g )[ 2 ];
b
gap> a*b*a=a^3;
true
gap> ReducedForm(rws,UnderlyingElement(a*b*a));
a^3
gap> ReducedForm(rws,UnderlyingElement(a^3));
a^3
```

51.6 Todd-Coxeter Procedure

This procedure gives a standard way of finding a transformation representation of a finitely presented semigroup. Usually one does not explicitly call this procedure but uses IsomorphismTransformationSemigroup or HomomorphismTransformationSemigroup (see 49.1.3).

1 ▶ CosetTableOfFpSemigroup(*r*) A

r is a right congruence of an fp-semigroup S. This attribute is the coset table of FP semigroup S on a right congruence r. Given a right congruence r we represent S as a set of transformations of the congruence classes of r.

The images of the cosets under the generators are compiled in a list *table* such that *table[i][s]* contains the image of coset s under generator i.

52 Transformations

This chapter describes functions for transformations.

A **transformation** in GAP is an endomorphism of a set of integers of the form $\{1, \ldots, n\}$. Transformations are taken to act on the right, which defines the composition $i^{(\alpha\beta)} = (i^\alpha)^\beta$ for i in $\{1, \ldots, n\}$.

For a transformation α on the set $\{1, \ldots, n\}$, we define its **degree** to be n, its **image list** to be the list, $[1\alpha, \ldots, n\alpha]$, its **image** to be the image list considered as a set, and its **rank** to be the size of the image. We also define the **kernel** of α to be the equivalence relation containing the pair (i, j) if and only if $i^\alpha = j^\alpha$.

Note that unlike permutations, we do not consider unspecified points to be fixed by a transformation. Therefore multiplication is only defined on two transformations of the same degree.

1 ► `IsTransformation( obj )` C
► `IsTransformationCollection( obj )` C

We declare it as `IsMultiplicativeElementWithOne` since the identity automorphism of $\{1, \ldots, n\}$ is a multiplicative two sided identity for any transformation on the same set.

2 ► `TransformationFamily( n )` F
► `TransformationType( n )` F
► `TransformationData( n )` F

For each n `>` `0` there is a single family and type of transformations on n points. To speed things up, we store these in a database of types. The three functions above a then access functions. If the nth entry isn't yet created, they trigger creation as well.

For n `>` `0`, element n of the type database is `[TransformationFamily(n), TransformationType(n)]`

3 ► `Transformation( images )` F
► `TransformationNC( images )` F

both return a transformation with the image list *images*. The normal version checks that the all the elements of the given list lie within the range $\{1, \ldots, n\}$ where n is the length of *images*, but for speed purposes, a non-checking version is also supplied.

4 ► `IdentityTransformation( n )` F

return the identity transformation of degree n

5 ► `RandomTransformation( n )` F

returns a random transformation of degree n

6 ► `DegreeOfTransformation( trans )` A

returns the degree of *trans*.

```
gap> t:= Transformation([2, 3, 4, 2, 4]);
Transformation( [ 2, 3, 4, 2, 4 ] )
gap> DegreeOfTransformation(t);
5
```

7 ▶ `ImageListOfTransformation( trans )` A

returns the image list of *trans*.

```
gap> ImageListOfTransformation(t);
[ 2, 3, 4, 2, 4 ]
```

8 ▶ `ImageSetOfTransformation( trans )` A

returns the image of *trans* as a set.

```
gap> ImageSetOfTransformation(t);
[ 2, 3, 4 ]
```

9 ▶ `RankOfTransformation( trans )` A

returns the rank of *trans*.

```
gap> RankOfTransformation(t);
3
```

10 ▶ `KernelOfTransformation( trans )` A

Returns the kernel of *trans* as an equivalence relation (See 32.1).

```
gap> KernelOfTransformation(t);
<equivalence relation on <object> >
gap> EquivalenceRelationPartition(last);
[ [ 1, 4 ], [ 3, 5 ] ]
```

11 ▶ `PreimagesOfTransformation( trans, i )` O

returns the subset of $\{1, \ldots, n\}$ which maps to i under *trans*.

```
gap> PreimagesOfTransformation(t, 2);
[ 1, 4 ]
```

12 ▶ `RestrictedTransformation( trans, alpha )` O

The transformation *trans* is restricted to only those points of *alpha*.

13 ▶ `AsTransformation( O )` O
▶ `AsTransformation( O, n )` O
▶ `AsTransformationNC( O, n )` O

returns the object O as a transformation. Supported objects are permutations and binary relations on points. In the second form, the operation returns a transformation of degree n, signalling an error if such a representation is not possible. `AsTransformationNC` does not perform this check.

```
gap> AsTransformation((1, 3)(2, 4));
Transformation( [ 3, 4, 1, 2 ] )
gap> AsTransformation((1, 3)(2, 4), 10);
Transformation( [ 3, 4, 1, 2, 5, 6, 7, 8, 9, 10 ] )

gap> AsTransformation((1, 3)(2, 4), 3);
Error, Permutation moves points over the degree specified called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can 'return;' to continue
brk> quit;
```

14 ▶ `PermLeftQuoTransformation( `*tr1*`, `*tr2*` )` O

Given transformations *tr1* and *tr2* with equal kernel and image, we compute the permutation induced by $(tr1)^{-1} * tr2$ on the set of images of *tr1*. If the kernels and images are not equal, an error is signaled.

15 ▶ `BinaryRelationTransformation( `*trans*` )` O

returns *trans* when considered as a binary relation.

16 ▶ `TransformationRelation( `*R*` )` O

returns the binary relation *R* when considered as a transformation. Only makes sense for injective binary relations over `[1..n]`. Returns an error if the relation is not over `[1..n]`, and `fail` if it is not injective.

53 Additive Magmas (preliminary)

This chapter deals with domains that are closed under addition +, which are called **near-additive magmas** in GAP. Together with the domains closed under multiplication *, (see 33), they are the basic algebraic structures. In many cases, the addition is commutative (see 53.3.1), the domain is called an **additive magma** then; every module (see 55), vector space (see 59), ring (see 54), or field (see 56) is an additive magma. In the cases of al **(near-)additive magma-with-zero** or **(near-)additive magma-with-inverses**, additional additive structure is present (see 53.1).

53.1 (Near-)Additive Magma Categories

1 ▶ `IsNearAdditiveMagma( obj )` C

A **near-additive magma** in GAP is a domain A with an associative but not necessarily commutative addition +: $A \times A \to A$.

2 ▶ `IsNearAdditiveMagmaWithZero( obj )` C

A **near-additive magma-with-zero** in GAP is a near-additive magma A with an operation `0*` (or `Zero`) that yields the zero of A.

So a near-additive magma-with-zero A does always contain a unique additively neutral element z, i.e., $z + a = a = a + z$ holds for all $a \in A$ (see 53.3.5). This element z can be computed with the operation `Zero` (see 30.10.3) as `Zero( A )`, and z is also equal to `Zero( elm )` and to `0*`*elm* for each element *elm* in A.

Note that a near-additive magma containing a zero may **not** lie in the category `IsNearAdditiveMagmaWithZero` (see 30.6).

3 ▶ `IsNearAdditiveGroup( obj )` C
▶ `IsNearAdditiveMagmaWithInverses( obj )` C

A **near-additive group** in GAP is a near-additive magma-with-zero A with an operation `-1*`: $A \to A$ that maps each element a of A to its additive inverse `-1*`a (or `AdditiveInverse( a )`, see 30.10.9).

The addition + of A is assumed to be associative, so a near-additive group is not more than a **near-additive magma-with-inverses**. `IsNearAdditiveMagmaWithInverses` is just a synonym for `IsNearAdditiveGroup`, and can be used alternatively in all function names involving `NearAdditiveGroup`.

Note that not every trivial near-additive magma is a near-additive magma-with-zero, but every trivial near-additive magma-with-zero is a near-additive group.

4 ▶ `IsAdditiveMagma( obj )` C

An **additive magma** in GAP is a domain A with an associative and commutative addition +: $A \times A \to A$, see 53.1.1 and 53.3.1.

5 ▶ `IsAdditiveMagmaWithZero( obj )` C

An **additive magma-with-zero** in GAP is an additive magma A with an operation `0*` (or `Zero`) that yields the zero of A.

So an additive magma-with-zero A does always contain a unique additively neutral element z, i.e., $z + a = a = a + z$ holds for all $a \in A$ (see 53.3.5). This element z can be computed with the operation `Zero` (see 30.10.3) as `Zero( A )`, and z is also equal to `Zero( elm )` and to `0*`*elm* for each element *elm* in A.

Note that an additive magma containing a zero may **not** lie in the category `IsAdditiveMagmaWithZero` (see 30.6).

6 ▶ `IsAdditiveGroup( obj )` C
▶ `IsAdditiveMagmaWithInverses( obj )` C

An **additive group** in GAP is an additive magma-with-zero A with an operation `-1*`: $A \to A$ that maps each element a of A to its additive inverse `-1*`a (or `AdditiveInverse( a )`, see 30.10.9).

The addition `+` of A is assumed to be commutative and associative, so an additive group is not more than an **additive magma-with-inverses**. `IsAdditiveMagmaWithInverses` is just a synonym for `IsAdditiveGroup`, and can be used alternatively in all function names involving `AdditiveGroup`.

Note that not every trivial additive magma is an additive magma-with-zero, but every trivial additive magma-with-zero is an additive group.

53.2 (Near-)Additive Magma Generation

1 ▶ `NearAdditiveMagma( gens )` F
▶ `NearAdditiveMagma( Fam, gens )` F

returns the (near-)additive magma A that is generated by the elements in the list *gens*, that is, the closure of *gens* under addition `+`. The family *Fam* of A can be entered as first argument; this is obligatory if *gens* is empty (and hence also A is empty).

2 ▶ `NearAdditiveMagmaWithZero( gens )` F
▶ `NearAdditiveMagmaWithZero( Fam, gens )` F

returns the (near-)additive magma-with-zero A that is generated by the elements in the list *gens*, that is, the closure of *gens* under addition `+` and `Zero`. The family *Fam* of A can be entered as first argument; this is obligatory if *gens* is empty (and hence A is trivial).

3 ▶ `NearAdditiveGroup( gens )` F
▶ `NearAdditiveGroup( Fam, gens )` F

returns the (near-)additive group A that is generated by the elements in the list *gens*, that is, the closure of *gens* under addition `+`, `Zero`, and `AdditiveInverse`. The family *Fam* of A can be entered as first argument; this is obligatory if *gens* is empty (and hence A is trivial).

The underlying operations for which methods can be installed are the following.

4 ▶ `NearAdditiveMagmaByGenerators( gens )` O
▶ `NearAdditiveMagmaByGenerators( Fam, gens )` O

5 ▶ `NearAdditiveMagmaWithZeroByGenerators( gens )` O
▶ `NearAdditiveMagmaWithZeroByGenerators( Fam, gens )` O

6 ▶ `NearAdditiveGroupByGenerators( gens )` O
▶ `NearAdditiveGroupByGenerators( Fam, gens )` O

Substructures of an additive magma can be formed as follows.

7 ▶ `SubnearAdditiveMagma( D, gens )` F
▶ `SubnearAdditiveMagmaNC( D, gens )` F

`SubadditiveMagma` returns the near-additive magma generated by the elements in the list *gens*, with parent the domain D. `SubadditiveMagmaNC` does the same, except that it is not checked whether the elements of *gens* lie in D.

8 ▶ `SubnearAdditiveMagmaWithZero( D, gens )` F
▶ `SubnearAdditiveMagmaWithZeroNC( D, gens )` F

`SubadditiveMagmaWithZero` returns the near-additive magma-with-zero generated by the elements in the list *gens*, with parent the domain D. `SubadditiveMagmaWithZeroNC` does the same, except that it is not checked whether the elements of *gens* lie in D.

9 ▶ `SubnearAdditiveGroup( D, gens )` F
▶ `SubnearAdditiveGroupNC( D, gens )` F

`SubadditiveGroup` returns the near-additive group generated by the elements in the list *gens*, with parent the domain D. `SubadditiveGroupNC` does the same, except that it is not checked whether the elements of *gens* lie in D.

53.3 Attributes and Properties for (Near-)Additive Magmas

1 ▶ `IsAdditivelyCommutative( A )` P

A near-additive magma A in GAP is **additively commutative** if for all elements $a, b \in A$ the equality $a + b = b + a$ holds.

Note that the commutativity of the **multiplication** `*` in a multiplicative structure can be tested with `IsCommutative`, (see 33.4.9).

2 ▶ `GeneratorsOfNearAdditiveMagma( A )` A
▶ `GeneratorsOfAdditiveMagma( A )` A

is a list *gens* of elements of the near-additive magma A that generates A as a near-additive magma, that is, the closure of *gens* under addition is A.

3 ▶ `GeneratorsOfNearAdditiveMagmaWithZero( A )` A
▶ `GeneratorsOfAdditiveMagmaWithZero( A )` A

is a list *gens* of elements of the near-additive magma-with-zero A that generates A as a near-additive magma-with-zero, that is, the closure of *gens* under addition and `Zero` (see 30.10.3) is A.

4 ▶ `GeneratorsOfNearAdditiveGroup( A )` A
▶ `GeneratorsOfAdditiveGroup( A )` A

is a list *gens* of elements of the near-additive group A that generates A as a near-additive group, that is, the closure of *gens* under addition, taking the zero element, and taking additive inverses (see 30.10.9) is A.

5 ▶ `AdditiveNeutralElement( A )` A

returns the element z in the near-additive magma A with the property that $z + a = a = a + z$ holds for all $a \in A$, if such an element exists. Otherwise `fail` is returned.

A near-additive magma that is not a near-additive magma-with-zero can have an additive neutral element z; in this case, z **cannot** be obtained as `Zero( A )` or as `0*`*elm* for an element *elm* in A, see 30.10.3.

6 ▶ `TrivialSubnearAdditiveMagmaWithZero( A )` A

is the additive magma-with-zero that has the zero of the near-additive magma-with-zero A as only element.

53.4 Operations for (Near-)Additive Magmas

1 ▶ `ClosureNearAdditiveGroup( A, a )` O
▶ `ClosureNearAdditiveGroup( A, B )` O

returns the closure of the near-additive magma A with the element a or the near-additive magma B, w.r.t. addition, taking the zero element, and taking additive inverses.

54 Rings

This chapter deals with domains that are additive groups closed under multiplication *. Such a domain, if * and + are distributive, is called a **ring** in GAP. Each division ring, field (see 56), or algebra (see 60) is a ring, important examples are the integers (see 14) and matrix rings.

In the case of a **ring-with-one**, additional multiplicative structure is present, see 54.3.1.

Several functions for ring elements, such as `IsPrime` (54.5.7) and `Factors` (54.5.8), are defined only relative to a ring R, which can be entered as an optional argument; if R is omitted then a **default ring** is formed from the ring elements given as arguments, see 54.1.3.

54.1 Generating Rings

1 ▶ `IsRing(` *R* `)` P

A **ring** in GAP is an additive group (see 53.1.6) that is also a magma (see 33.1.1), such that addition + and multiplication * are distributive.

The multiplication need **not** be associative (see 33.4.7). For example, a Lie algebra (see 61) is regarded as a ring in GAP.

2 ▶ `Ring(` *r* `,` *s*`, ... )` F
▶ `Ring(` *coll* `)` F

In the first form `Ring` returns the smallest ring that contains all the elements r, s... etc. In the second form `Ring` returns the smallest ring that contains all the elements in the collection *coll*. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

`Ring` differs from `DefaultRing` (see 54.1.3) in that it returns the smallest ring in which the elements lie, while `DefaultRing` may return a larger ring if that makes sense.

```
gap> Ring( 2, E(4) );
<ring with 2 generators>
```

3 ▶ `DefaultRing(` *r* `,` *s*`, ... )` F
▶ `DefaultRing(` *coll* `)` F

In the first form `DefaultRing` returns a ring that contains all the elements r, s, ... etc. In the second form `DefaultRing` returns a ring that contains all the elements in the collection *coll*. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

The ring returned by `DefaultRing` need not be the smallest ring in which the elements lie. For example for elements from cyclotomic fields, `DefaultRing` may return the ring of integers of the smallest cyclotomic field in which the elements lie, which need not be the smallest ring overall, because the elements may in fact lie in a smaller number field which is itself not a cyclotomic field.

(For the exact definition of the default ring of a certain type of elements, look at the corresponding method installation.)

`DefaultRing` is used by the ring functions like `Quotient`, `IsPrime`, `Factors`, or `Gcd` if no explicit ring is given.

`Ring` (see 54.1.2) differs from `DefaultRing` in that it returns the smallest ring in which the elements lie, while `DefaultRing` may return a larger ring if that makes sense.

```
gap> DefaultRing( 2, E(4) );
GaussianIntegers
```

4 ▶ `RingByGenerators(` *C* `)` O

`RingByGenerators` returns the ring generated by the elements in the collection C, i. e., the closure of C under addition, multiplication, and taking additive inverses.

```
gap> RingByGenerators([ 2, E(4) ]);
<ring with 2 generators>
```

5 ▶ `DefaultRingByGenerators(` *coll* `)` O

```
gap> DefaultRingByGenerators([ 2, E(4) ]);
GaussianIntegers
```

6 ▶ `GeneratorsOfRing(` *R* `)` A

`GeneratorsOfRing` returns a list of elements such that the ring R is the closure of these elements under addition, multiplication, and taking additive inverses.

```
gap> R:=Ring( 2, 1/2 );
<ring with 2 generators>
gap> GeneratorsOfRing( R );
[ 2, 1/2 ]
```

7 ▶ `AsRing(` *C* `)` A

If the elements in the collection C form a ring then `AsRing` returns this ring, otherwise `fail` is returned.

8 ▶ `Subring(` *R*, *gens* `)` F
▶ `SubringNC(` *R*, *gens* `)` F

returns the ring with parent R generated by the elements in *gens*. When the second form, `SubringNC` is used, it is **not** checked whether all elements in *gens* lie in R.

```
gap> R:= Integers;
Integers
gap> S:= Subring( R, [ 4, 6 ] );
<ring with 2 generators>
gap> Parent( S );
Integers
```

9 ▶ `ClosureRing(` *R*, *r* `)` O
▶ `ClosureRing(` *R*, *S* `)` O

For a ring R and either an element r of its elements family or a ring S, `ClosureRing` returns the ring generated by both arguments.

```
gap> ClosureRing( Integers, E(4) );
<ring-with-one, with 2 generators>
```

10 ▶ `Quotient(` *R*, *r*, *s* `)` O
▶ `Quotient(` *r*, *s* `)` O

In the first form `Quotient` returns the quotient of the two ring elements r and s in the ring R. In the second form `Quotient` returns the quotient of the two ring elements r and s in their default ring. It returns `fail` if the quotient does not exist in the respective ring.

(To perform the division in the quotient field of a ring, use the quotient operator `/`.)

```
gap> Quotient( 2, 3 );
fail
gap> Quotient( 6, 3 );
2
```

54.2 Ideals in Rings

A **left ideal** in a ring R is a subring of R that is closed under multiplication with elements of R from the left.

A **right ideal** in a ring R is a subring of R that is closed under multiplication with elements of R from the right.

A **two-sided ideal** or simply **ideal** in a ring R is both a left ideal and a right ideal in R.

So being a (left/right/two-sided) ideal is not a property of a domain but refers to the acting ring(s). Hence we must ask, e. g., `IsIdeal( R, I )` if we want to know whether the ring I is an ideal in the ring R. The property `IsIdealInParent` can be used to store whether a ring is an ideal in its parent.

(Whenever the term `Ideal` occurs without specifying prefix `Left` or `Right`, this means the same as `TwoSidedIdeal`. Conversely, any occurrence of `TwoSidedIdeal` can be substituted by `Ideal`.)

For any of the above kinds of ideals, there is a notion of generators, namely `GeneratorsOfLeftIdeal`, `GeneratorsOfRightIdeal`, and `GeneratorsOfTwoSidedIdeal`. The acting rings can be accessed as `LeftActingRingOfIdeal` and `RightActingRingOfIdeal`, respectively. Note that ideals are detected from known values of these attributes, especially it is assumed that whenever a domain has both a left and a right acting ring then these two are equal.

Note that we cannot use `LeftActingDomain` and `RightActingDomain` here, since ideals in algebras are themselves vector spaces, and such a space can of course also be a module for an action from the right. In order to make the usual vector space functionality automatically available for ideals, we have to distinguish the left and right module structure from the additional closure properties of the ideal.

Further note that the attributes denoting ideal generators and acting ring are used to create ideals if this is explicitly wanted, but the ideal relation in the sense of `IsIdeal` is of course independent of the presence of the attribute values.

Ideals are constructed with `LeftIdeal`, `RightIdeal`, `TwoSidedIdeal`. Principal ideals of the form $x * R$, $R * x$, $R * x * R$ can also be constructed with a simple multiplication.

Currently many methods for dealing with ideals need linear algebra to work, so they are mainly applicable to ideals in algebras.

1 ▶ `TwoSidedIdeal( R, gens[, "basis"] )` F
▶ `Ideal( R, gens[, "basis"] )` F
▶ `LeftIdeal( R, gens[, "basis"] )` F
▶ `RightIdeal( R, gens[, "basis"] )` F

Let R be a ring, and *gens* a list of collection of elements in R. `TwoSidedIdeal`, `LeftIdeal`, and `RightIdeal` return the two-sided, left, or right ideal, respectively, I in R that is generated by *gens*. The ring R can be accessed as `LeftActingRingOfIdeal` or `RightActingRingOfIdeal` (or both) of I.

If R is a left F-module then also I is a left F-module, in particular the `LeftActingDomain` (see 55.1.11) values of R and I are equal.

If the optional argument `"basis"` is given then *gens* are assumed to be a list of basis vectors of I viewed as a free F-module. (This is mainly applicable to ideals in algebras.) In this case, it is **not** checked whether *gens* really is linearly independent and whether *gens* is a subset of R.

`Ideal` is simply a synonym of `TwoSidedIdeal`.

```
gap> R:= Integers;;
gap> I:= Ideal( R, [ 2 ] );
<two-sided ideal in Integers, (1 generators)>
```

2 ▶ `TwoSidedIdealNC( `*R*`, `*gens*`[, "basis"] )` F
▶ `IdealNC( `*R*`, `*gens*`[, "basis"] )` F
▶ `LeftIdealNC( `*R*`, `*gens*`[, "basis"] )` F
▶ `RightIdealNC( `*R*`, `*gens*`[, "basis"] )` F

The effects of `TwoSidedIdealNC`, `LeftIdealNC`, and `RightIdealNC` are the same as `TwoSidedIdeal`, `LeftIdeal`, and `RightIdeal`, respectively (see 54.2.1), but they do not check whether all entries of *gens* lie in *R*.

3 ▶ `IsTwoSidedIdeal( `*R*`, `*I*` )` O
▶ `IsLeftIdeal( `*R*`, `*I*` )` O
▶ `IsRightIdeal( `*R*`, `*I*` )` O
▶ `IsTwoSidedIdealInParent( `*I*` )` P
▶ `IsLeftIdealInParent( `*I*` )` P
▶ `IsRightIdealInParent( `*I*` )` P

The properties `IsTwoSidedIdealInParent` etc., are attributes of the ideal, and once known they are stored in the ideal.

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );
( Rationals^[ 3, 3 ] )
gap> I:= Ideal( A, [ Random( A ) ] );
<two-sided ideal in ( Rationals^[ 3, 3 ] ), (1 generators)>
gap> IsTwoSidedIdeal( A, I );
true
```

4 ▶ `TwoSidedIdealByGenerators( `*R*`, `*gens*` )` O
▶ `IdealByGenerators( `*R*`, `*gens*` )` O

`TwoSidedIdealByGenerators` returns the ring that is generated by the elements of the collection *gens* under addition, multiplication, and multiplication with elements of the ring *R* from the left and from the right.

R can be accessed by `LeftActingRingOfIdeal` or `RightActingRingOfIdeal`, *gens* can be accessed by `GeneratorsOfTwoSidedIdeal`.

5 ▶ `LeftIdealByGenerators( `*R*`, `*gens*` )` O

`LeftIdealByGenerators` returns the ring that is generated by the elements of the collection *gens* under addition, multiplication, and multiplication with elements of the ring *R* from the left.

R can be accessed by `LeftActingRingOfIdeal`, *gens* can be accessed by `GeneratorsOfLeftIdeal`.

6 ▶ `RightIdealByGenerators( `*R*`, `*gens*` )` O

`RightIdealByGenerators` returns the ring that is generated by the elements of the collection *gens* under addition, multiplication, and multiplication with elements of the ring *R* from the right.

R can be accessed by `RightActingRingOfIdeal`, *gens* can be accessed by `GeneratorsOfRightIdeal`.

7 ▶ `GeneratorsOfTwoSidedIdeal( `*I*` )` A
▶ `GeneratorsOfIdeal( `*I*` )` A

is a list of generators for the bi-ideal *I*, with respect to the action of `LeftActingRingOfIdeal( `*I*` )` from the left and the action of `RightActingRingOfIdeal( `*I*` )`from the right.

Note that LeftActingRingOfIdeal(I) and RightActingRingOfIdeal(I) coincide if I is a two-sided ideal.

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );;
gap> I:= Ideal( A, [ One( A ) ] );;
gap> GeneratorsOfIdeal( I );
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] ]
```

8 ▶ GeneratorsOfLeftIdeal(I) A

is a list of generators for the left ideal I, with respect to the action of LeftActingRingOfIdeal(I) from the left.

9 ▶ GeneratorsOfRightIdeal(I) A

is a list of generators for the right ideal I, with respect to the action of RightActingRingOfIdeal(I) from the right.

10 ▶ LeftActingRingOfIdeal(I) A
▶ RightActingRingOfIdeal(I) A

11 ▶ AsLeftIdeal(R, S) O
▶ AsRightIdeal(R, S) O
▶ AsTwoSidedIdeal(R, S) O

Let S be a subring of R.

If S is a left ideal in R then AsLeftIdeal returns this left ideal, otherwise fail is returned. If S is a right ideal in R then AsRightIdeal returns this right ideal, otherwise fail is returned. If S is a two-sided ideal in R then AsTwoSidedIdeal returns this two-sided ideal, otherwise fail is returned.

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );;
gap> B:= DirectSumOfAlgebras( A, A );
<algebra over Rationals, with 6 generators>
gap> C:= Subalgebra( B, Basis( B ){[1..9]} );
<algebra over Rationals, with 9 generators>
gap> I:= AsTwoSidedIdeal( B, C );
<two-sided ideal in <algebra of dimension 18 over Rationals>, (9 generators)>
```

54.3 Rings With One

1 ▶ IsRingWithOne(R) P

A **ring-with-one** in GAP is a ring (see 54.1.1) that is also a magma-with-one (see 33.1.2).

Note that the identity and the zero of a ring-with-one need **not** be distinct. This means that a ring that consists only of its zero element can be regarded as a ring-with-one.

This is especially useful in the case of finitely presented rings, in the sense that each factor of a ring-with-one is again a ring-with-one.

2 ▶ RingWithOne(r, s, ...) F
▶ RingWithOne(C) F

In the first form RingWithOne returns the smallest ring with one that contains all the elements r, s... etc. In the second form RingWithOne returns the smallest ring with one that contains all the elements in the collection C. If any element is not an element of a ring or if the elements lie in no common ring an error is raised.

```
gap> RingWithOne( [ 4, 6 ] );
<ring-with-one, with 2 generators>
```

3 ► `RingWithOneByGenerators(` *coll* `)` O

`RingWithOneByGenerators` returns the ring-with-one generated by the elements in the collection *coll*, i. e., the closure of *coll* under addition, multiplication, taking additive inverses, and taking the identity of an element.

4 ► `GeneratorsOfRingWithOne(` *R* `)` A

`GeneratorsOfRingWithOne` returns a list of elements such that the ring *R* is the closure of these elements under addition, multiplication, taking additive inverses, and taking the identity element `One(` *R* `)`.

R itself need **not** be known to be a ring-with-one.

```
gap> R:= RingWithOne( [ 4, 6 ] );
<ring-with-one, with 2 generators>
gap> GeneratorsOfRingWithOne( R );
[ 4, 6 ]
```

5 ► `SubringWithOne(` *R*, *gens* `)` F
► `SubringWithOneNC(` *R*, *gens* `)` F

returns the ring with one with parent *R* generated by the elements in *gens*. When the second form, `SubringNC` is used, it is **not** checked whether all elements in *gens* lie in *R*.

```
gap> R:= SubringWithOne( Integers, [ 4, 6 ] );
<ring-with-one, with 2 generators>
gap> Parent( R );
Integers
```

54.4 Properties of Rings

1 ► `IsIntegralRing(` *R* `)` P

A ring-with-one *R* is integral if it is commutative, contains no nontrivial zero divisors, and if its identity is distinct from its zero.

```
gap> IsIntegralRing( Integers );
true
```

2 ► `IsUniqueFactorizationRing(` *R* `)` C

A ring *R* is called a **unique factorization ring** if it is an integral ring (see 54.4.1), and every element has a unique factorization into irreducible elements, i.e., a unique representation as product of irreducibles (see 54.5.6). Unique in this context means unique up to permutations of the factors and up to multiplication of the factors by units (see 54.5.2).

Mathematically, a field should therefore also be a unique factorization ring, since every element is a unit. In GAP, however, at least at present fields do not lie in the filter `IsUniqueFactorizationRing` (see 54.4.2), since Operations such as `Factors`, `Gcd`, `StandardAssociate` and so on do not apply to fields (the results would be trivial, and not especially useful) and Methods which require their arguments to lie in `IsUnique-FactorizationRing` expect these Operations to work.

(Note that we cannot install a subset maintained method for this category since the factorization of an element needs not exist in a subring. As an example, consider the subring $4\mathbb{N}+1$ of the ring $4\mathbb{Z}+1$; in the subring, the element $3 \cdot 3 \cdot 11 \cdot 7$ has the two factorizations $33 \cdot 21 = 9 \cdot 77$, but in the large ring there is the

unique factorization $(-3)\cdot(-3)\cdot(-11)\cdot(-7)$, and it is easy to see that every element in $4\mathbb{Z}+1$ has a unique factorization.)

```
gap> IsUniqueFactorizationRing( PolynomialRing( Rationals, 1 ) );
true
```

3 ▶ `IsLDistributive( C )` P

is `true` if the relation $a*(b+c)=(a*b)+(a*c)$ holds for all elements a, b, c in the collection C, and `false` otherwise.

4 ▶ `IsRDistributive( C )` P

is `true` if the relation $(a+b)*c=(a*c)+(b*c)$ holds for all elements a, b, c in the collection C, and `false` otherwise.

5 ▶ `IsDistributive( C )` P

is `true` if the collection C is both left and right distributive, and `false` otherwise.

```
gap> IsDistributive( Integers );
true
```

6 ▶ `IsAnticommutative( R )` P

is `true` if the relation $a*b=-b*a$ holds for all elements a, b in the ring R, and `false` otherwise.

7 ▶ `IsZeroSquaredRing( R )` P

is `true` if $a*a$ is the zero element of the ring R for all a in R, and `false` otherwise.

8 ▶ `IsJacobianRing( R )` P

is `true` if the Jacobi identity holds in R, and `false` otherwise. The Jacobi identity means that $x*(y*z)+z*(x*y)+y*(z*x)$ is the zero element of R, for all elements x, y, z in R.

```
gap> L:= FullMatrixLieAlgebra( GF( 5 ), 7 );
<Lie algebra over GF(5), with 13 generators>
gap> IsJacobianRing( L );
true
```

54.5 Units and Factorizations

1 ▶ `IsUnit( R, r )` O
▶ `IsUnit( r )` O

In the first form `IsUnit` returns `true` if r is a unit in the ring R. In the second form `IsUnit` returns `true` if the ring element r is a unit in its default ring (see 54.1.3).

An element r is called a **unit** in a ring R, if r has an inverse in R.

`IsUnit` may call `Quotient`.

2 ▶ `Units( R )` A

`Units` returns the group of units of the ring R. This may either be returned as a list or as a group.

An element r is called a **unit** of a ring R, if r has an inverse in R. It is easy to see that the set of units forms a multiplicative group.

```
gap> Units( GaussianIntegers );
[ -1, 1, -E(4), E(4) ]
gap> Units( GF( 16 ) );
<group with 1 generators>
```

3 ▶ `IsAssociated( R, r, s )` O
▶ `IsAssociated( r, s )` O

In the first form `IsAssociated` returns `true` if the two ring elements r and s are associated in the ring R and `false` otherwise. In the second form `IsAssociated` returns `true` if the two ring elements r and s are associated in their default ring (see 54.1.3) and `false` otherwise.

Two elements r and s of a ring R are called **associated** if there is a unit u of R such that $ru = s$.

4 ▶ `Associates( R, r )` O
▶ `Associates( r )` O

In the first form `Associates` returns the set of associates of r in the ring R. In the second form `Associates` returns the set of associates of the ring element r in its default ring (see 54.1.3).

Two elements r and s of a ring R are called **associated** if there is a unit u of R such that $ru = s$.

```
gap> Associates( Integers, 2 );
[ -2, 2 ]
gap> Associates( GaussianIntegers, 2 );
[ -2, 2, -2*E(4), 2*E(4) ]
```

5 ▶ `StandardAssociate( R, r )` O
▶ `StandardAssociate( r )` O

In the first form `StandardAssociate` returns the standard associate of the ring element r in the ring R. In the second form `StandardAssociate` returns the standard associate of the ring element r in its default ring (see 54.1.3).

The **standard associate** of a ring element r of R is an associated element of r which is, in a ring dependent way, distinguished among the set of associates of r. For example, in the ring of integers the standard associate is the absolute value.

```
gap> x:= Indeterminate( Rationals, "x" );;
gap> StandardAssociate( -x^2-x+1 );
x^2+x-1
```

6 ▶ `IsIrreducibleRingElement( R, r )` O
▶ `IsIrreducibleRingElement( r )` O

In the first form `IsIrreducibleRingElement` returns `true` if the ring element r is irreducible in the ring R and `false` otherwise. In the second form `IsIrreducibleRingElement` returns `true` if the ring element r is irreducible in its default ring (see 54.1.3) and `false` otherwise.

An element r of a ring R is called **irreducible** if r is not a unit in R and if there is no nontrivial factorization of r in R, i.e., if there is no representation of r as product st such that neither s nor t is a unit (see 54.5.1). Each prime element (see 54.5.7) is irreducible.

```
gap> IsIrreducibleRingElement( Integers, 2 );
true
```

7 ▶ `IsPrime( R, r )` O
▶ `IsPrime( r )` O

In the first form `IsPrime` returns `true` if the ring element r is a prime in the ring R and `false` otherwise. In the second form `IsPrime` returns `true` if the ring element r is a prime in its default ring (see 54.1.3) and `false` otherwise.

An element r of a ring R is called **prime** if for each pair s and t such that r divides st the element r divides either s or t. Note that there are rings where not every irreducible element (see 54.5.6) is a prime.

8 ▶ `Factors( R, r )` O
▶ `Factors( r )` O

In the first form `Factors` returns the factorization of the ring element r in the ring R. In the second form `Factors` returns the factorization of the ring element r in its default ring (see 54.1.3). The factorization is returned as a list of primes (see 54.5.7). Each element in the list is a standard associate (see 54.5.5) except the first one, which is multiplied by a unit as necessary to have `Product( Factors( R, r ) ) = r`. This list is usually also sorted, thus smallest prime factors come first. If r is a unit or zero, `Factors( R, r ) = [ r ]`.

```
gap> x:= Indeterminate( GF(2), "x" );;
gap> pol:= x^2+x+1;
x^2+x+Z(2)^0
gap> Factors( pol );
[ x^2+x+Z(2)^0 ]
gap> Factors( PolynomialRing( GF(4) ), pol );
[ x+Z(2^2), x+Z(2^2)^2 ]
```

54.6 Euclidean Rings

1 ▶ `IsEuclideanRing( R )` C

A ring R is called a Euclidean ring if it is an integral ring and there exists a function δ, called the Euclidean degree, from $R - \{0_R\}$ to the nonnegative integers, such that for every pair $r \in R$ and $s \in R - \{0_R\}$ there exists an element q such that either $r - qs = 0_R$ or $\delta(r - qs) < \delta(s)$. In GAP the Euclidean degree δ is implicitly built into an ring and cannot be changed. The existence of this division with remainder implies that the Euclidean algorithm can be applied to compute a greatest common divisor of two elements, which in turn implies that R is a unique factorization ring.

```
gap> IsEuclideanRing( GaussianIntegers );
true
```

2 ▶ `EuclideanDegree( R, r )` O
▶ `EuclideanDegree( r )` O

In the first form `EuclideanDegree` returns the Euclidean degree of the ring element in the ring R. In the second form `EuclideanDegree` returns the Euclidean degree of the ring element r in its default ring. R must of course be a Euclidean ring (see 54.6.1).

```
gap> EuclideanDegree( GaussianIntegers, 3 );
9
```

3 ▶ `EuclideanQuotient( R, r, m )` O
▶ `EuclideanQuotient( r, m )` O

In the first form `EuclideanQuotient` returns the Euclidean quotient of the ring elements r and m in the ring R. In the second form `EuclideanQuotient` returns the Euclidean quotient of the ring elements r and m in their default ring. The ring R must be a Euclidean ring (see 54.6.1) otherwise an error is signalled.

```
gap> EuclideanQuotient( 8, 3 );
2
```

4 ▶ `EuclideanRemainder( R, r, m )` O
▶ `EuclideanRemainder( r, m )` O

In the first form `EuclideanRemainder` returns the remainder of the ring element r modulo the ring element m in the ring R. In the second form `EuclideanRemainder` returns the remainder of the ring element r

modulo the ring element m in their default ring. The ring R must be a Euclidean ring (see 54.6.1) otherwise an error is signalled.

```
gap> EuclideanRemainder( 8, 3 );
2
```

5 ▶ `QuotientRemainder( R, r, s )` O
▶ `QuotientRemainder( r, s )` O

In the first form `QuotientRemainder` returns the Euclidean quotient and the Euclidean remainder of the ring elements r and m in the ring R. In the second form `QuotientRemainder` returns the Euclidean quotient and the Euclidean remainder of the ring elements r and m in their default ring as pair of ring elements. The ring R must be a Euclidean ring (see 54.6.1) otherwise an error is signalled.

```
gap> QuotientRemainder( GaussianIntegers, 8, 3 );
[ 3, -1 ]
```

54.7 Gcd and Lcm

1 ▶ `Gcd( R, r1, r2, ... )` F
▶ `Gcd( R, list )` F
▶ `Gcd( r1, r2, ... )` F
▶ `Gcd( list )` F

In the first two forms `Gcd` returns the greatest common divisor of the ring elements *r1*, *r2*, ... resp. of the ring elements in the list *list* in the ring R. In the second two forms `Gcd` returns the greatest common divisor of the ring elements *r1*, *r2*, ... resp. of the ring elements in the list *list* in their default ring (see 54.1.3). R must be a Euclidean ring (see 54.6.1) so that `QuotientRemainder` (see 54.6.5) can be applied to its elements. `Gcd` returns the standard associate (see 54.5.5) of the greatest common divisors.

A greatest common divisor of the elements $r_1, r_2, \ldots$ of the ring R is an element of largest Euclidean degree (see 54.6.2) that is a divisor of $r_1, r_2, \ldots$.

We define `Gcd( r, 0`$_R$`, r ) = StandardAssociate( r )` and `Gcd(` 0_R`, 0`$_R$`, r ) =` 0_R.

```
gap> Gcd( Integers, [ 10, 15 ] );
5
```

2 ▶ `GcdOp( R, r, s )` O
▶ `GcdOp( r, s )` O

`GcdOp` is the operation to compute the greatest common divisor of two ring elements r, s in the ring R or in their default ring.

3 ▶ `GcdRepresentation( R, r1, r2, ... )` F
▶ `GcdRepresentation( R, list )` F
▶ `GcdRepresentation( r1, r2, ... )` F
▶ `GcdRepresentation( list )` F

In the first two forms `GcdRepresentation` returns the representation of the greatest common divisor of the ring elements *r1*, *r2*, ... resp. of the ring elements in the list *list* in the ring R. In the second two forms `GcdRepresentation` returns the representation of the greatest common divisor of the ring elements *r1*, *r2*, ... resp. of the ring elements in the list *list* in their default ring (see 54.1.3). R must be a Euclidean ring (see 54.6.1) so that `Gcd` (see 54.7.1) can be applied to its elements.

The representation of the gcd g of the elements $r_1, r_2, \ldots$ of a ring R is a list of ring elements $s_1, s_2, \ldots$ of R, such that $g = s_1 r_1 + s_2 r_2 + \cdots$. That this representation exists can be shown using the Euclidean algorithm, which in fact can compute those coefficients.

```
gap> x:= Indeterminate( Rationals, "x" );;
gap> GcdRepresentation( x^2+1, x^3+1 );
[ -1/2*x^2-1/2*x+1/2, 1/2*x+1/2 ]
```

4 ▶ `GcdRepresentationOp( R, r, s )` O
▶ `GcdRepresentationOp( r, s )` O

`GcdRepresentationOp` is the operation to compute the representation of the greatest common divisor of two ring elements r, s in the ring R or in their default ring, respectively.

5 ▶ `Lcm( R, r1, r2, ... )` F
▶ `Lcm( R, list )` F
▶ `Lcm( r1, r2, ... )` F
▶ `Lcm( list )` F

In the first two forms `Lcm` returns the least common multiple of the ring elements *r1*, *r2*, ... resp. of the ring elements in the list *list* in the ring R. In the second two forms `Lcm` returns the least common multiple of the ring elements *r1*, *r2*, ... resp. of the ring elements in the list *list* in their default ring (see 54.1.3).

R must be a Euclidean ring (see 54.6.1) so that `Gcd` (see 54.7.1) can be applied to its elements. `Lcm` returns the standard associate (see 54.5.5) of the least common multiples.

A least common multiple of the elements $r_1, r_2, \ldots$ of the ring R is an element of smallest Euclidean degree (see 54.6.2) that is a multiple of $r_1, r_2, \ldots$.

We define `Lcm( r, 0`$_R$` ) = Lcm( 0`$_R$`, r ) = StandardAssociate( r )` and `Lcm( 0`$_R$`, 0`$_R$` ) = 0`$_R$.

`Lcm` uses the equality $lcm(m,n) = m * n/gcd(m,n)$ (see 54.7.1).

6 ▶ `LcmOp( R, r, s )` O
▶ `LcmOp( r, s )` O

`LcmOp` is the operation to compute the least common multiple of two ring elements r, s in the ring R or in their default ring, respectively.

7 ▶ `QuotientMod( R, r, s, m )` O
▶ `QuotientMod( r, s, m )` O

In the first form `QuotientMod` returns the quotient of the ring elements r and s modulo the ring element m in the ring R. In the second form `QuotientMod` returns the quotient of the ring elements r and s modulo the ring element m in their default ring (see 54.1.3). R must be a Euclidean ring (see 54.6.1) so that `EuclideanRemainder` (see 54.6.4) can be applied. If the modular quotient does not exist, `fail` is returned.

The quotient q of r and s modulo m is an element of R such that $qs = r$ modulo m, i.e., such that $qs - r$ is divisible by m in R and that q is either 0 (if r is divisible by m) or the Euclidean degree of q is strictly smaller than the Euclidean degree of m.

```
gap> QuotientMod( 7, 2, 3 );
2
```

8 ▶ `PowerMod( R, r, e, m )` O
▶ `PowerMod( r, e, m )` O

In the first form `PowerMod` returns the e-th power of the ring element r modulo the ring element m in the ring R. In the second form `PowerMod` returns the e-th power of the ring element r modulo the ring element m in their default ring (see 54.1.3). e must be an integer. R must be a Euclidean ring (see 54.6.1) so that `EuclideanRemainder` (see 54.6.4) can be applied to its elements.

If e is positive the result is r^e modulo m. If e is negative then `PowerMod` first tries to find the inverse of r modulo m, i.e., i such that $ir = 1$ modulo m. If the inverse does not exist an error is signalled. If the inverse does exist `PowerMod` returns `PowerMod( R, i, -e, m )`.

`PowerMod` reduces the intermediate values modulo m, improving performance drastically when e is large and m small.

```
gap> PowerMod( 12, 100000, 7 );
2
```

9 ▶ `InterpolatedPolynomial( R, x, y )` O

`InterpolatedPolynomial` returns, for given lists x, y of elements in a ring R of the same length n, say, the unique polynomial of degree less than n which has value $y[i]$ at $x[i]$, for all $i \in \{1, \ldots, n\}$. Note that the elements in x must be distinct.

```
gap>  InterpolatedPolynomial( Integers, [ 1, 2, 3 ], [ 5, 7, 0 ] );
-9/2*x^2+31/2*x-6
```

55 Modules (preliminary)

55.1 Generating modules

1 ▶ `IsLeftOperatorAdditiveGroup( D )` C

A domain D lies in `IsLeftOperatorAdditiveGroup` if it is an additive group that is closed under scalar multplication from the left, and such that $\lambda * (x + y) = \lambda * x + \lambda * y$ for all scalars λ and elements $x, y \in D$.

2 ▶ `IsLeftModule( M )` C

A domain M lies in `IsLeftModule` if it lies in `IsLeftOperatorAdditiveGroup`, *and* the set of scalars forms a ring, *and* $(\lambda + \mu) * x = \lambda * x + \mu * x$ for scalars λ, μ and $x \in M$, *and* scalar multiplication satisfies $\lambda * (\mu * x) = (\lambda * \mu) * x$ for scalars λ, μ and $x \in M$.

```
gap> V:= FullRowSpace( Rationals, 3 );
( Rationals^3 )
gap> IsLeftModule( V );
true
```

3 ▶ `GeneratorsOfLeftOperatorAdditiveGroup( D )` A

returns a list of elements of D that generates D as a left operator additive group.

4 ▶ `GeneratorsOfLeftModule( M )` A

returns a list of elements of M that generate M as a left module.

```
gap> V:= FullRowSpace( Rationals, 3 );;
gap> GeneratorsOfLeftModule( V );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
```

5 ▶ `AsLeftModule( R, D )` O

if the domain D forms an additive group and is closed under left multiplication by the elements of R, then `AsLeftModule( R, D )` returns the domain D viewed as a left module.

```
gap> coll:= [ [0*Z(2),0*Z(2)], [Z(2),0*Z(2)], [0*Z(2),Z(2)], [Z(2),Z(2)] ];
[ [ 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ],
  [ Z(2)^0, Z(2)^0 ] ]
gap> AsLeftModule( GF(2), coll );
<vector space of dimension 2 over GF(2)>
```

6 ▶ `IsRightOperatorAdditiveGroup( D )` C

A domain D lies in `IsRightOperatorAdditiveGroup` if it is an additive group that is closed under scalar multplication from the right, and such that $(x + y) * \lambda = x * \lambda + y * \lambda$ for all scalars λ and elements $x, y \in D$.

7 ▶ `IsRightModule( M )` C

A domain M lies in `IsRightModule` if it lies in `IsRightOperatorAdditiveGroup`, *and* the set of scalars forms a ring, *and* $x * (\lambda + \mu) = x * \lambda + x * \mu$ for scalars λ, μ and $x \in M$, *and* scalar multiplication satisfies $(x * \mu) * \lambda = x * (\mu * \lambda)$ for scalars λ, μ and $x \in M$.

8 ▶ `GeneratorsOfRightOperatorAdditiveGroup( D )` A

returns a list of elements of D that generates D as a right operator additive group.

9 ▶ `GeneratorsOfRightModule( M )` A

returns a list of elements of M that generate M as a left module.

10 ▶ `LeftModuleByGenerators( R, gens )` O
▶ `LeftModuleByGenerators( R, gens, zero )` O

returns the left module over R generated by *gens*.

```
gap> coll:= [ [Z(2),0*Z(2)], [0*Z(2),Z(2)], [Z(2),Z(2)] ];;
gap> V:= LeftModuleByGenerators( GF(16), coll );
<vector space over GF(2^4), with 3 generators>
```

11 ▶ `LeftActingDomain( D )` A

Let D be an external left set, that is, D is closed under the action of a domain L by multiplication from the left. Then L can be accessed as value of `LeftActingDomain` for D.

55.2 Submodules

1 ▶ `Submodule( M, gens )` F
▶ `Submodule( M, gens, "basis" )` F

is the left module generated by the collection *gens*, with parent module M. The second form generates the submodule of M for that the list *gens* is known to be a list of basis vectors; in this case, it is **not** checked whether *gens* really are linearly independent and whether all in *gens* lie in M.

```
gap> coll:= [ [Z(2),0*Z(2)], [0*Z(2),Z(2)], [Z(2),Z(2)] ];;
gap> V:= LeftModuleByGenerators( GF(16), coll );;
gap> W:= Submodule( V, [ coll[1], coll[2] ] );
<vector space over GF(2^4), with 2 generators>
gap> Parent( W ) = V;
true
```

2 ▶ `SubmoduleNC( M, gens )` F
▶ `SubmoduleNC( M, gens, "basis" )` F

`SubmoduleNC` does the same as `Submodule`, except that it does not check whether all in *gens* lie in M.

3 ▶ `ClosureLeftModule( M, m )` O

is the left module generated by the left module generators of M and the element m.

```
gap> V:= LeftModuleByGenerators( Rationals, [ [ 1, 0, 0 ], [ 0, 1, 0 ] ] );
<vector space over Rationals, with 2 generators>
gap> ClosureLeftModule( V, [ 1, 1, 1 ] );
<vector space over Rationals, with 3 generators>
```

4 ▶ `TrivialSubmodule( M )` A

returns the zero submodule of M.

```
gap> V:= LeftModuleByGenerators( Rationals, [ [ 1, 0, 0 ], [ 0, 1, 0 ] ] );;
gap> TrivialSubmodule( V );
<vector space over Rationals, with 0 generators>
```

55.3 Free Modules

1 ► `IsFreeLeftModule( M )` C

A left module is free as module if it is isomorphic to a direct sum of copies of its left acting domain.

Free left modules can have bases.

The characteristic (see 30.10.1) of a free left module is defined as the characteristic of its left acting domain (see 55.1.11).

2 ► `FreeLeftModule( R, gens )` F
► `FreeLeftModule( R, gens, zero )` F
► `FreeLeftModule( R, gens, "basis" )` F
► `FreeLeftModule( R, gens, zero, "basis" )` F

`FreeLeftModule( R, gens )` is the free left module over the ring R, generated by the vectors in the collection *gens*.

If there are three arguments, a ring R and a collection *gens* and an element *zero*, then `FreeLeftModule( R, gens, zero )` is the R-free left module generated by *gens*, with zero element *zero*.

If the last argument is the string `"basis"` then the vectors in *gens* are known to form a basis of the free module.

It should be noted that the generators *gens* must be vectors, that is, they must support an addition and a scalar action of R via left multiplication. (See also Section 30.3 for the general meaning of "generators" in GAP.) In particular, `FreeLeftModule` is **not** an equivalent of commands such as `FreeGroup` (see 35.2.1) in the sense of a constructor of a free group on abstract generators; Such a construction seems to be unnecessary for vector spaces, for that one can use for example row spaces (see 59.8.4) in the finite dimensional case and polynomial rings (see 64.14.1) in the infinite dimensional case. Moreover, the definition of a "natural" addition for elements of a given magma (for example a permutation group) is possible via the construction of magma rings (see Chapter 63).

```
gap> V:= FreeLeftModule( Rationals, [ [ 1, 0, 0 ], [ 0, 1, 0 ] ], "basis" );
<vector space of dimension 2 over Rationals>
```

3 ► `AsFreeLeftModule( F, D )` O

if the domain D is a free left module over F, then `AsFreeLeftModule( F, D )` returns the domain D viewed as free left module over F.

4 ► `Dimension( M )` A

A free left module has dimension n if it is isomorphic to a direct sum of n copies of its left acting domain.

(We do **not** mark `Dimension` as invariant under isomorphisms since we want to call `UseIsomorphismRelation` also for free left modules over different left acting domains.)

```
gap> V:= FreeLeftModule( Rationals, [ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ] );;
gap> Dimension( V );
2
```

5 ► `IsFiniteDimensional( M )` P

is `true` if M is a free left module that is finite dimensional over its left acting domain, and `false` otherwise.

```
gap> V:= FreeLeftModule( Rationals, [ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ] );;
gap> IsFiniteDimensional( V );
true
```

6 ▶ `UseBasis( V, gens )` O

The vectors in the list *gens* are known to form a basis of the free left module V. `UseBasis` stores information in V that can be derived form this fact, namely

- *gens* are stored as left module generators if no such generators were bound (this is useful especially if V is an algebra),
- the dimension of V is stored.

```
gap> V:= FreeLeftModule( Rationals, [ [ 1, 0 ], [ 0, 1 ], [ 1, 1 ] ] );;
gap> UseBasis( V, [ [ 1, 0 ], [ 1, 1 ] ] );
gap> V;  # now V knows its dimension
<vector space of dimension 2 over Rationals>
```

7 ▶ `IsRowModule( V )` P

A **row module** is a free left module whose elements are row vectors.

8 ▶ `IsMatrixModule( V )` P

A **matrix module** is a free left module whose elements are matrices.

9 ▶ `IsFullRowModule( M )` P

A **full row module** is a module R^n, for a ring R and a nonnegative integer n.

More precisely, a full row module is a free left module over a ring R such that the elements are row vectors with entries in R and such that the dimension is equal to the length of the row vectors.

Several functions delegate their tasks to full row modules, for example `Iterator` and `Enumerator`.

10 ▶ `FullRowModule( R, n )` F

is the row module `R^n`, for a ring R and a nonnegative integer n.

```
gap> V:= FullRowModule( Integers, 5 );
( Integers^5 )
```

11 ▶ `IsFullMatrixModule( M )` P

A **full matrix module** is a module $R^{[m,n]}$, for a ring R and two nonnegative integers m, n.

More precisely, a full matrix module is a free left module over a ring R such that the elements are matrices with entries in R and such that the dimension is equal to the number of entries in each matrix.

12 ▶ `FullMatrixModule( R, m, n )` F

is the row module `R^[m,n]`, for a ring R and nonnegative integers m and n.

```
gap> FullMatrixModule( GaussianIntegers, 3, 6 );
( GaussianIntegers^[ 3, 6 ] )
```

13 ▶ `IsHandledByNiceBasis( M )` C

For a free left module M in this category, essentially all operations are performed using a "nicer" free left module, which is usually a row module.

56 Fields and Division Rings

A **division ring** is a ring (see Chapter 54) in which every non-zero element has an inverse. The most important class of division rings are the commutative ones, which are called **fields**.

GAP supports finite fields (see Chapter 57) and abelian number fields (see Chapter 58), in particular the field of rationals (see Chapter 16).

This chapter describes the general GAP functions for fields and division rings.

If a field F is a subfield of a commutative ring C, C can be considered as a vector space over the (left) acting domain F (see Chapter 59). In this situation, we call F the **field of definition** of C.

Each field in GAP is represented as a vector space over a subfield (see 56.1.2), thus each field is in fact a field extension in a natural way, which is used by functions such as `Norm` and `Trace` (see 56.3).

56.1 Generating Fields

1 ▶ `IsDivisionRing( D )` C

A **division ring** in GAP is a nontrivial associative algebra D with a multiplicative inverse for each nonzero element. In GAP every division ring is a vector space over a division ring (possibly over itself). Note that being a division ring is thus not a property that a ring can get, because a ring is usually not represented as a vector space.

The field of coefficients is stored as `LeftActingDomain( D )`.

2 ▶ `IsField( D )` P

A **field** is a commutative division ring (see 56.1.1 and 33.4.9).

```
gap> IsField( GaloisField(16) );           # the field with 16 elements
true
gap> IsField( Rationals );                 # the field of rationals
true
gap> q:= QuaternionAlgebra( Rationals );;  # a noncommutative division ring
gap> IsField( q );  IsDivisionRing( q );
false
true
gap> mat:= [ [ 1 ] ];;  a:= Algebra( Rationals, [ mat ] );;
gap> IsDivisionRing( a );   # an algebra not constructed as a division ring
false
```

3 ▶ `Field( z, ... )` F
▶ `Field( list )` F
▶ `Field( F, list )` F

`Field` returns the smallest field K that contains all the elements $z, \ldots$, or the smallest field K that contains all elements in the list *list*. If no subfield F is given, K is constructed as a field over itself, i.e. the left acting

domain of K is K. In the third form, `Field` constructs the field generated by the field F and the elements in the list *list*, as a vector space over F.

4 ► `DefaultField( z, ... )` F
► `DefaultField( list )` F

`DefaultField` returns a field K that contains all the elements $z, \ldots$, or a field K that contains all elements in the list *list*.

This field need not be the smallest field in which the elements lie, cf. `Field` (see 56.1.3). For example, for elements from cyclotomic fields `DefaultField` returns the smallest cyclotomic field in which the elements lie, but the elements may lie in a smaller number field which is not a cyclotomic field.

```
gap> Field( Z(4) );  Field( [ Z(4), Z(8) ] );  # finite fields
GF(2^2)
GF(2^6)
gap> Field( E(9) );  Field( CF(4), [ E(9) ] ); # abelian number fields
CF(9)
AsField( GaussianRationals, CF(36) )
gap> f1:= Field( EB(5) );  f2:= DefaultField( EB(5) );
NF(5,[ 1, 4 ])
CF(5)
gap> f1 = f2;  IsSubset( f2, f1 );
false
true
```

5 ► `DefaultFieldByGenerators( [ z, ... ] )` O

returns the default field containing the elements z,.... This field may be bigger than the smallest field containing these elements.

6 ► `GeneratorsOfDivisionRing( D )` A

generators with respect to addition, multiplication, and taking inverses (the identity cannot be omitted ...)

7 ► `GeneratorsOfField( F )` A

generators with respect to addition, multiplication, and taking inverses. This attribute is the same as `GeneratorsOfDivisionRing` (see 56.1.6).

8 ► `DivisionRingByGenerators( [ z, ... ] )` O
► `DivisionRingByGenerators( F, [ z, ... ] )` O

The first version returns a division ring as vector space over `FieldOverItselfByGenerators( gens )`.

9 ► `AsDivisionRing( C )` O
► `AsDivisionRing( F, C )` O
► `AsField( C )` O
► `AsField( F, C )` O

If the collection C can be regarded as a division ring then `AsDivisionRing( C )` is the division ring that consists of the elements of C, viewed as a vector space over its prime field; otherwise `fail` is returned.

In the second form, if F is a division ring contained in C then the returned division ring is viewed as a vector space over F.

`AsField` is just a synonym for `AsDivisionRing`.

56.2 Subfields of Fields

1 ▶ `Subfield( F, gens )` F
▶ `SubfieldNC( F, gens )` F

Constructs the subfield of F generated by *gens*.

2 ▶ `FieldOverItselfByGenerators( [ z, ... ] )` O

This operation is needed for the call of `Field` or `FieldByGenerators` without explicitly given subfield, in order to construct a left acting domain for such a field.

3 ▶ `PrimitiveElement( D )` A

is an element of D that generates D as a division ring together with the left acting domain.

4 ▶ `PrimeField( D )` A

The **prime field** of a division ring D is the smallest field which is contained in D. For example, the prime field of any field in characteristic zero is isomorphic to the field of rational numbers.

5 ▶ `IsPrimeField( D )` P

A division ring is a prime field if it is equal to its prime field (see 56.2.4).

6 ▶ `DegreeOverPrimeField( F )` A

is the degree of the field F over its prime field (see 56.2.4).

7 ▶ `DefiningPolynomial( F )` A

is the defining polynomial of the field F as a field extension over the left acting domain of F. A root of the defining polynomial can be computed with `RootOfDefiningPolynomial` (see 56.2.8).

8 ▶ `RootOfDefiningPolynomial( F )` A

is a root in the field F of its defining polynomial as a field extension over the left acting domain of F. The defining polynomial can be computed with `DefiningPolynomial` (see 56.2.7).

9 ▶ `FieldExtension( F, poly )` O

is the field obtained on adjoining a root of the irreducible polynomial *poly* to the field F.

10 ▶ `Subfields( F )` A

is the set of all subfields of the field F.

56.3 Galois Action

Let $L > K$ be a field extension of finite degree. Then to each element $\alpha \in L$, we can associate a K-linear mapping φ_α on L, and for a fixed K-basis of L, we can associate to α the matrix M_α (over K) of this mapping.

The **norm** of α is defined as the determinant of M_α, the **trace** of α is defined as the trace of M_α, the **minimal polynomial** μ_α and the **trace polynomial** χ_α of α are defined as the minimal polynomial (see 56.3.2) and the characteristic polynomial (see 24.12.1 and 56.3.3) of M_α. (Note that μ_α depends only on K whereas χ_α depends on both L and K.)

Thus norm and trace of α are elements of K, and μ_α and χ_α are polynomials over K, χ_α being a power of μ_α, and the degree of χ_α equals the degree of the field extension $L > K$.

The **conjugates** of α in L are those roots of χ_α (with multiplicity) that lie in L; note that if only L is given, there is in general no way to access the roots outside L.

Analogously, the **Galois group** of the extension $L > K$ is defined as the group of all those field automorphisms of L that fix K pointwise.

If $L > K$ is a Galois extension then the conjugates of α are all roots of χ_α (with multiplicity), the set of conjugates equals the roots of μ_α, the norm of α equals the product and the trace of α equals the sum of the conjugates of α, and the Galois group in the sense of the above definition equals the usual Galois group,

Note that `MinimalPolynomial( F, z )` is a polynomial **over** F, whereas `Norm( F, z )` is the norm of the element z **in** F w.r.t. the field extension $F >$ `LeftActingDomain`(F).

1 ▶ `GaloisGroup( F )` A

The **Galois group** of a field F is the group of all field automorphisms of F that fix the subfield $K =$ `LeftActingDomain`(F) pointwise.

Note that the field extension $F > K$ need **not** be a Galois extension.

```
gap> g:= GaloisGroup( AsField( GF(2^2), GF(2^12) ) );;
gap> Size( g );  IsCyclic( g );
6
true
gap> h:= GaloisGroup( CF(60) );;
gap> Size( h );  IsAbelian( h );
16
true
```

2 ▶ `MinimalPolynomial( F, z[, ind] )` O

returns the minimal polynomial of z over the field F. This is a generator of the ideal in $F[x]$ of all polynomials which vanish on z. (This definition is consistent with the general definition of `MinimalPolynomial` for rings, see 64.8.1.)

```
gap> MinimalPolynomial( Rationals, E(8) );
x_1^4+1
gap> MinimalPolynomial( CF(4), E(8) );
x_1^2+-E(4)
gap> MinimalPolynomial( CF(8), E(8) );
x_1+-E(8)
```

3 ▶ `TracePolynomial( L, K, z[, inum] )` O

returns the polynomial that is the product of $(X - c)$ where c runs over the conjugates of z in the field extension L over K. The polynomial is returned as a univariate polynomial over K in the indeterminate number *inum* (defaulting to 1).

This polynomial is sometimes also called the **characteristic polynomial** of z w.r.t. the field extension $L > K$. Therefore methods are installed for `CharacteristicPolynomial` (see 24.12.1) that call `TracePolynomial` in the case of field extensions.

```
gap> TracePolynomial( CF(8), Rationals, E(8) );
x_1^4+1
gap> TracePolynomial( CF(16), Rationals, E(8) );
x_1^8+2*x_1^4+1
```

4 ▶ `Norm( z )` A
▶ `Norm( L, z )` O
▶ `Norm( L, K, z )` O

`Norm` returns the norm of the field element z. If two fields L and K are given then the norm is computed w.r.t. the field extension $L > K$, if only one field L is given then `LeftActingDomain( L )` is taken as default for the subfield K, and if no field is given then `DefaultField( z )` is taken as default for L.

5 ▶ `Trace( z )` A
▶ `Trace( mat )` A
▶ `Trace( L, z )` O
▶ `Trace( L, K, z )` O

`Trace` returns the trace of the field element z. If two fields L and K are given then the trace is computed w.r.t. the field extension $L > K$, if only one field L is given then `LeftActingDomain( L )` is taken as default for the subfield K, and if no field is given then `DefaultField( z )` is taken as default for L.

The **trace of a matrix** is the sum of its diagonal entries. Note that this is **not** compatible with the definition of `Trace` for field elements, so the one-argument version is not suitable when matrices shall be regarded as field elements.

6 ▶ `Conjugates( z )` A
▶ `Conjugates( L, z )` O
▶ `Conjugates( L, K, z )` O

`Conjugates` returns the list of **conjugates** of the field element z. If two fields L and K are given then the conjugates are computed w.r.t. the field extension $L > K$, if only one field L is given then `LeftActingDomain( L )` is taken as default for the subfield K, and if no field is given then `DefaultField( z )` is taken as default for L.

The result list will contain duplicates if z lies in a proper subfield of L, respectively of the default field of z. The result list need not be sorted.

```
gap> Norm( E(8) );  Norm( CF(8), E(8) );
1
1
gap> Norm( CF(8), CF(4), E(8) );
-E(4)
gap> Norm( AsField( CF(4), CF(8) ), E(8) );
-E(4)
gap> Trace( E(8) );  Trace( CF(8), CF(8), E(8) );
0
E(8)
gap> Conjugates( CF(8), E(8) );
[ E(8), E(8)^3, -E(8), -E(8)^3 ]
gap> Conjugates( CF(8), CF(4), E(8) );
[ E(8), -E(8) ]
gap> Conjugates( CF(16), E(8) );
[ E(8), E(8)^3, -E(8), -E(8)^3, E(8), E(8)^3, -E(8), -E(8)^3 ]
```

The default methods for field elements are as follows. `MinimalPolynomial` solves a system of linear equations, `TracePolynomial` computes the appropriate power of the minimal polynomial, `Norm` and `Trace` values are obtained as coefficients of the characteristic polynomial, and `Conjugates` uses the factorization of the characteristic polynomial.

For elements in finite fields and cyclotomic fields, one wants to do the computations in a different way since the field extensions in question are Galois extensions, and the Galois groups are well-known in these cases. More general, if a field is in the category `IsFieldControlledByGaloisGroup` then the default methods are the following. `Conjugates` returns the sorted list of images (with multiplicity) of the element under the Galois group, `Norm` computes the product of the conjugates, `Trace` computes the sum of the conjugates, `TracePolynomial` and `MinimalPolynomial` compute the product of linear factors $x - c$ with c ranging over the conjugates and the set of conjugates, respectively.

7 ▶ `NormalBase( F )` A
▶ `NormalBase( F, elm )` O

Let F be a field that is a Galois extension of its subfield `LeftActingDomain( F )`. Then `NormalBase` returns a list of elements in F that form a normal basis of F, that is, a vector space basis that is closed under the action of the Galois group (see 56.3.1) of F.

If a second argument *elm* is given, it is used as a hint for the algorithm to find a normal basis with the algorithm described in [Art68].

```
gap> NormalBase( CF(5) );
[ -E(5), -E(5)^2, -E(5)^3, -E(5)^4 ]
gap> NormalBase( CF(4) );
[ 1/2-1/2*E(4), 1/2+1/2*E(4) ]
gap> NormalBase( GF(3^6) );
[ Z(3^6)^2, Z(3^6)^6, Z(3^6)^18, Z(3^6)^54, Z(3^6)^162, Z(3^6)^486 ]
gap> NormalBase( GF( GF(8), 2 ) );
[ Z(2^6), Z(2^6)^8 ]
```

57 Finite Fields

This chapter describes the special functionality which exists in GAP for finite fields and their elements. Of course the general functionality for fields (see Chapter 56) also applies to finite fields.

In the following, the term **finite field element** is used to denote GAP objects in the category `IsFFE` (see 57.1.1), and **finite field** means a field consisting of such elements. Note that in principle we must distinguish these fields from (abstract) finite fields. For example, the image of the embedding of a finite field into a field of rational functions in the same characteristic is of course a finite field but its elements are not in `IsFFE`, and in fact GAP does currently not support such fields.

Special representations exist for row vectors and matrices over small finite fields (see sections 23.2 and 24.13).

57.1 Finite Field Elements

1 ▶ `IsFFE(` *obj* `)` C
▶ `IsFFECollection(` *obj* `)` C
▶ `IsFFECollColl(` *obj* `)` C

Objects in the category `IsFFE` are used to implement elements of finite fields. In this manual, the term **finite field element** always means an object in `IsFFE`. All finite field elements of the same characteristic form a family in GAP (see 13.1). Any collection of finite field elements (see 28) lies in `IsFFECollection`, and a collection of such collections (e.g., a matrix) lies in `IsFFECollColl`.

2 ▶ `Z(` *p*`^`*d* `)` F

For creating elements of a finite field the function `Z` can be used. The call `Z(` *p*`^`*d* `)` returns the designated generator of the multiplicative group of the finite field with *p*`^`*d* elements. *p* must be a prime and *p*`^`*d* must be less than or equal to $2^{16} = 65536$.

The root returned by `Z` is a generator of the multiplicative group of the finite field with p^d elements, which is cyclic. The order of the element is of course $p^d - 1$. The $p^d - 1$ different powers of the root are exactly the nonzero elements of the finite field.

Thus all nonzero elements of the finite field with *p*`^`*d* elements can be entered as `Z(`*p*`^`*d*`)^`*i*. Note that this is also the form that GAP uses to output those elements.

The additive neutral element is `0*Z(`*p*`)`. It is different from the integer `0` in subtle ways. First `IsInt( 0*Z(`*p*`) )` (see 14.1.1) is `false` and `IsFFE( 0*Z(`*p*`) )` (see 57.1.1) is `true`, whereas it is just the other way around for the integer `0`.

The multiplicative neutral element is `Z(`*p*`)^0`. It is different from the integer `1` in subtle ways. First `IsInt( Z(`*p*`)^0 )` (see 14.1.1) is `false` and `IsFFE( Z(`*p*`)^0 )` (see 57.1.1) is `true`, whereas it is just the other way around for the integer `1`. Also `1+1` is `2`, whereas, e.g., `Z(2)^0 + Z(2)^0` is `0*Z(2)`.

The various roots returned by `Z` for finite fields of the same characteristic are compatible in the following sense. If the field $GF(p^n)$ is a subfield of the field $GF(p^m)$, i.e., n divides m, then $Z(p^n) = Z(p^m)^{(p^m-1)/(p^n-1)}$. Note that this is the simplest relation that may hold between a generator of $GF(p^n)$ and $GF(p^m)$, since $Z(p^n)$ is an element of order p^m-1 and $Z(p^m)$ is an element of order p^n-1. This is achieved by choosing $Z(p)$ as the smallest primitive root modulo p and $Z(p^n)$ as a root of the n-th **Conway polynomial** (see 57.5.1)

of characteristic p. Those polynomials were defined by J. H. Conway, and many of them were computed by R. A. Parker.

Elements of prime fields of order larger than 2^{16} can be handled using the machinery of Residue Class Rings (see section 14.4).

```
gap> a:= Z( 32 );
Z(2^5)
gap> a+a;
0*Z(2)
gap> a*a;
Z(2^5)^2
```

Elements of finite fields can be compared using the operators `=` and `<`. The call a `=` b returns `true` if and only if the finite field elements a and b are equal. Furthermore a `<` b tests whether a is smaller than b. Finite field elements are ordered in the following way. If the two elements lie in fields of different characteristics the one that lies in the field with the smaller characteristic is smaller. If the two elements lie in different fields of the same characteristic the one that lies in the smaller field is smaller. If the two elements lie in the same field and one is the zero and the other is not, the zero element is smaller. If the two elements lie in the same field and both are nonzero, and are represented as $Z(p^d)^{i_1}$ and $Z(p^d)^{i_2}$ respectively, then the one with the smaller i is smaller.

For the comparison of finite field elements with other GAP objects, see 4.11.

```
gap> Z( 16 )^10 = Z( 4 )^2;  # this illustrates the embedding of GF(4) in GF(16)
true
gap> 0 < 0*Z(101);
true
gap> Z(256) > Z(101);
false
```

57.2 Operations for Finite Field Elements

Since finite field elements are scalars, the operations `Characteristic`, `One`, `Zero`, `Inverse`, `AdditiveInverse`, `Order` can be applied to then (see 30.10). Contrary to the situation with other scalars, `Order` is defined also for the zero element in a finite field, with value `0`.

```
gap> Characteristic( Z( 16 )^10 );  Characteristic( Z( 9 )^2 );
2
3
gap> Characteristic( [ Z(4), Z(8) ] );
2
gap> One( Z(9) );  One( 0*Z(4) );
Z(3)^0
Z(2)^0
gap> Inverse( Z(9) );  AdditiveInverse( Z(9) );
Z(3^2)^7
Z(3^2)^5
gap> Order( Z(9)^7 );
8
```

1 ▶ `DegreeFFE(` *z* `)` O
▶ `DegreeFFE(` *vec* `)` O
▶ `DegreeFFE(` *mat* `)` O

`DegreeFFE` returns the degree of the smallest finite field F containing the element z, respectively all elements of the vector *vec* over a finite field (see 23), or matrix *mat* over a finite field (see 24).

```
gap> DegreeFFE( Z( 16 )^10 );
2
gap> DegreeFFE( Z( 16 )^11 );
4
gap> DegreeFFE( [ Z(2^13), Z(2^10) ] );
130
```

2 ▶ `LogFFE( z, r )` O

`LogFFE` returns the discrete logarithm of the element z in a finite field with respect to the root r. An error is signalled if z is zero, or if z is not a power of r.

The **discrete logarithm** of an element z with respect to a root r is the smallest nonnegative integer i such that $r^i = z$.

```
gap> LogFFE( Z(409)^116, Z(409) );  LogFFE( Z(409)^116, Z(409)^2 );
116
58
```

3 ▶ `IntFFE( z )` O

`IntFFE` returns the integer corresponding to the element z, which must lie in a finite prime field. That is `IntFFE` returns the smallest nonnegative integer i such that i `* One( z ) =` z.

The correspondence between elements from a finite prime field of characteristic p (for $p < 2^{16}$) and the integers between 0 and $p-1$ is defined by choosing `Z(`p`)` the element corresponding to the smallest primitive root mod p (see 15.2.3).

`IntFFE` is installed as a method for the operation `Int` (see 14.1.3) with argument a finite field element.

```
gap> IntFFE( Z(13) );  PrimitiveRootMod( 13 );
2
2
gap> IntFFE( Z(409) );
21
gap> IntFFE( Z(409)^116 );  21^116 mod 409;
311
311
```

4 ▶ `IntFFESymm( z )` O
▶ `IntFFESymm( vec )` O

For a finite prime field element z, `IntFFESymm` returns the corresponding integer of smallest absolute value. That is `IntFFESymm` returns the integer i of smallest absolute value that i `* One( z ) =` z.

For a vector *vec*, the operation returns the result if applying `IntFFESymm` to every entry of the vector.

The correspondence between elements from a finite prime field of characteristic p (for $p < 2^{16}$) and the integers between $-p/2$ and $p/2$ is defined by choosing `Z(`p`)` the element corresponding to the smallest positive primitive root mod p (see 15.2.3) and reducing results to the $-p/2 \cdot \cdot p/2$ range.

```
gap> IntFFE(Z(13)^2);IntFFE(Z(13)^3);
4
8
gap> IntFFESymm(Z(13)^2);IntFFESymm(Z(13)^3);
4
-5
```

5 ▶ `IntVecFFE( vecffe )` O

is the list of integers corresponding to the vector *vecffe* of finite field elements in a prime field (see 57.2.3).

57.3 Creating Finite Fields

`DefaultField` (see 56.1.4) and `DefaultRing` (see 54.1.3) for finite field elements are defined to return the **smallest** field containing the given elements.

```
gap> DefaultField( [ Z(4), Z(4)^2 ] );  DefaultField( [ Z(4), Z(8) ] );
GF(2^2)
GF(2^6)
```

1 ▶ `GaloisField( p^d )` F
▶ `GF( p^d )` F
▶ `GaloisField( p, d )` F
▶ `GF( p, d )` F
▶ `GaloisField( subfield, d )` F
▶ `GF( subfield, d )` F
▶ `GaloisField( p, pol )` F
▶ `GF( p, pol )` F
▶ `GaloisField( subfield, pol )` F
▶ `GF( subfield, pol )` F

`GaloisField` returns a finite field. It takes two arguments. The form `GaloisField( p, d )`, where p, d are integers, can also be given as `GaloisField( p^d )`. `GF` is an abbreviation for `GaloisField`.

The first argument specifies the subfield S over which the new field F is to be taken. It can be a prime or a finite field. If it is a prime p, the subfield is the prime field of this characteristic.

The second argument specifies the extension. It can be an integer or an irreducible polynomial over the field S. If it is an integer d, the new field is constructed as the polynomial extension with the Conway polynomial (see 57.5.1) of degree d over the subfield S. If it is an irreducible polynomial *pol* over S, the new field is constructed as polynomial extension of the subfield S with this polynomial; in this case, *pol* is accessible as the value of `DefiningPolynomial` (see 56.2.7) for the new field, and a root of *pol* in the new field is accessible as the value of `RootOfDefiningPolynomial` (see 56.2.8).

Note that the subfield over which a field was constructed determines over which field the Galois group, conjugates, norm, trace, minimal polynomial, and trace polynomial are computed (see 56.3.1, 56.3.6, 56.3.4, 56.3.5, 56.3.2, 56.3.3).

The field is regarded as a vector space (see 59) over the given subfield, so this determines the dimension and the canonical basis of the field.

```
gap> f1:= GF( 2^4 );
GF(2^4)
gap> Size( GaloisGroup ( f1 ) );
4
gap> BasisVectors( Basis( f1 ) );
[ Z(2)^0, Z(2^4), Z(2^4)^2, Z(2^4)^3 ]
gap> f2:= GF( GF(4), 2 );
AsField( GF(2^2), GF(2^4) )
gap> Size( GaloisGroup( f2 ) );
2
gap> BasisVectors( Basis( f2 ) );
[ Z(2)^0, Z(2^4) ]
```

2 ▶ `PrimitiveRoot( F )` A

A **primitive root** of a finite field is a generator of its multiplicative group. A primitive root is always a primitive element (see 56.2.3), the converse is in general not true.

```
gap> f:= GF( 3^5 );
GF(3^5)
gap> PrimitiveRoot( f );
Z(3^5)
```

57.4 FrobeniusAutomorphism

1 ▶ `FrobeniusAutomorphism( F )` A

returns the Frobenius automorphism of the finite field F as a field homomorphism (see 31.11).

The **Frobenius automorphism** f of a finite field F of characteristic p is the function that takes each element z of F to its p-th power. Each automorphism of F is a power of f. Thus f is a generator for the Galois group of F relative to the prime field of F, and an appropriate power of f is a generator of the Galois group of F over a subfield (see 56.3.1).

```
gap> f := GF(16);
GF(2^4)
gap> x := FrobeniusAutomorphism( f );
FrobeniusAutomorphism( GF(2^4) )
gap> Z(16) ^ x;
Z(2^4)^2
gap> x^2;
FrobeniusAutomorphism( GF(2^4) )^2
```

The image of an element z under the i-th power of f is computed as the p^i-th power of z. The product of the i-th power and the j-th power of f is the k-th power of f, where k is $ij \pmod{\texttt{Size}(F)-1}$. The zeroth power of f is `IdentityMapping( F )`.

57.5 Conway Polynomials

1 ▶ `ConwayPolynomial( p, n )` F

is the Conway polynomial of the finite field $GF(p^n)$ as polynomial over the prime field in characteristic p.

The **Conway polynomial** $\Phi_{n,p}$ of $GF(p^n)$ is defined by the following properties.

First define an ordering of polynomials of degree n over $GF(p)$ as follows. $f = \sum_{i=0}^{n}(-1)^i f_i x^i$ is smaller than $g = \sum_{i=0}^{n}(-1)^i g_i x^i$ if and only if there is an index $m \leq n$ such that $f_i = g_i$ for all $i > m$, and $\tilde{f_m} < \tilde{g_m}$, where $\tilde{c}$ denotes the integer value in $\{0, 1, \ldots, p-1\}$ that is mapped to $c \in GF(p)$ under the canonical epimorphism that maps the integers onto $GF(p)$.

$\Phi_{n,p}$ is **primitive** over $GF(p)$ (see 64.4.12). That is, $\Phi_{n,p}$ is irreducible, monic, and is the minimal polynomial of a primitive root of $GF(p^n)$.

For all divisors d of n the compatibility condition $\Phi_{d,p}(x^{\frac{p^n-1}{p^m-1}}) \equiv 0 \pmod{\Phi_{n,p}(x)}$ holds. (That is, the appropriate power of a zero of $\Phi_{n,p}$ is a zero of the Conway polynomial $\Phi_{d,p}$.)

With respect to the ordering defined above, $\Phi_{n,p}$ shall be minimal.

The computation of Conway polynomials can be time consuming. Therefore, GAP comes with a list of precomputed polynomials. If a requested polynomial is not stored then GAP prints a warning and computes it by checking all polynomials in the order defined above for the defining conditions. If n is not a prime this is probably a very long computation. (Some previously known polynomials with prime n are not stored in GAP because they are quickly recomputed.) Use the function 57.5.2 to check in advance if `ConwayPolynomial` will give a result after a short time.

Note that primitivity of a polynomial can only be checked if GAP can factorize $p^n - 1$. A sufficiently new version of the FactInt package contains many precomputed factors of such numbers from various factorization projects.

See [Lüb] for further information on known Conway polynomials.

If *pol* is a result returned by `ConwayPolynomial` the command `Print( InfoText( pol ) );` will print some info on the origin of that particular polynomial.

For some purposes it may be enough to have any primitive polynomial for an extension of a finite field instead of the Conway polynomial, see 57.5.3 below.

```
gap> ConwayPolynomial( 2, 5 );  ConwayPolynomial( 3, 7 );
x_1^5+x_1^2+Z(2)^0
x_1^7-x_1^2+Z(3)^0
```

2 ► `IsCheapConwayPolynomial( p, n )` F

Returns `true` if `ConwayPolynomial( p, n )` will give a result in **reasonable** time. This is either the case when this polynomial is pre-computed, or if n is a not too big prime.

3 ► `RandomPrimitivePolynomial( F, n[, i ] )` F

For a finite field F and a positive integer n this function returns a primitive polynomial of degree n over F, that is a zero of this polynomial has maximal multiplicative order $|F|^n - 1$. If i is given then the polynomial is written in variable number i over F (see 64.1.1), the default for i is 1.

Alternatively, F can be a prime power q, then $F = \mathrm{GF}(q)$ is assumed. And i can be a univariate polynomial over F, then the result is a polynomial in the same variable.

This function can work for much larger fields than those for which Conway polynomials are available, of course GAP must be able to factorize $|F|^n - 1$.

58 Abelian Number Fields

An **abelian number field** is a field in characteristic zero that is a finite dimensional normal extension of its prime field such that the Galois group is abelian. In GAP, one implementation of abelian number fields is given by fields of cyclotomic numbers (see Chapter 18). Note that abelian number fields can also be constructed with the more general `AlgebraicExtension` (see 65.1.1), a discussion of advantages and disadvantages can be found in 18.6. The functions described in this chapter have been developed for fields whose elements are in the filter `IsCyclotomic` (see 18.1.3), they may or may not work well for abelian number fields consisting of other kinds of elements.

Throughout this chapter, $\mathbb{Q}_n$ will denote the cyclotomic field generated by the field $\mathbb{Q}$ of rationals together with n-th roots of unity.

In 58.1, constructors for abelian number fields are described, 58.2 introduces operations for abelian number fields, 58.3 deals with the vector space structure of abelian number fields, and 58.4 describes field automorphisms of abelian number fields,

58.1 Construction of Abelian Number Fields

Besides the usual construction using `Field` or `DefaultField` (see 58.2), abelian number fields consisting of cyclotomics can be created with `CyclotomicField` and `AbelianNumberField`.

1 ▶ `CyclotomicField(` *n* `)` F
▶ `CyclotomicField(` *gens* `)` F
▶ `CyclotomicField(` *subfield*`,` *n* `)` F
▶ `CyclotomicField(` *subfield*`,` *gens* `)` F

The first version creates the n-th cyclotomic field $\mathbb{Q}_n$. The second version creates the smallest cyclotomic field containing the elements in the list *gens*. In both cases the field can be generated as an extension of a designated subfield *subfield* (cf. 58.3).

`CyclotomicField` can be abbreviated to `CF`, this form is used also when GAP prints cyclotomic fields.

Fields constructed with the one argument version of `CF` are stored in the global list `CYCLOTOMIC_FIELDS`, so repeated calls of `CF` just fetch these field objects after they have been created once.

```
gap> CyclotomicField( 5 );  CyclotomicField( [ Sqrt(3) ] );
CF(5)
CF(12)
gap> CF( CF(3), 12 );  CF( CF(4), [ Sqrt(7) ] );
AsField( CF(3), CF(12) )
AsField( GaussianRationals, CF(28) )
```

2 ▶ `AbelianNumberField(` *n*`,` *stab* `)` F

For a positive integer n and a list *stab* of prime residues modulo n, `AbelianNumberField` returns the fixed field of the group described by *stab* (cf. 58.2.4), in the n-th cyclotomic field. `AbelianNumberField` is mainly

thought for internal use and for printing fields in a standard way; `Field` (see 56.1.3, cf. also 58.2) is probably more suitable if one knows generators of the field in question.

`AbelianNumberField` can be abbreviated to `NF`, this form is used also when GAP prints abelian number fields.

Fields constructed with `NF` are stored in the global list `ABELIAN_NUMBER_FIELDS`, so repeated calls of `NF` just fetch these field objects after they have been created once.

```
gap> NF( 7, [ 1 ] );
CF(7)
gap> f:= NF( 7, [ 1, 2 ] );  Sqrt(-7); Sqrt(-7) in f;
NF(7,[ 1, 2, 4 ])
E(7)+E(7)^2-E(7)^3+E(7)^4-E(7)^5-E(7)^6
true
```

3 ▶ `GaussianRationals` V
▶ `IsGaussianRationals( obj )` C

`GaussianRationals` is the field $\mathbb{Q}_4 = \mathbb{Q}(\sqrt{-1})$ of Gaussian rationals, as a set of cyclotomic numbers, see Chapter 18 for basic operations. This field can also be obtained as `CF(4)` (see 58.1.1).

The filter `IsGaussianRationals` returns `true` for the GAP object `GaussianRationals`, and `false` for all other GAP objects.

(For details about the field of rationals, see Chapter 16.)

```
gap> CF(4) = GaussianRationals;
true
gap> Sqrt(-1) in GaussianRationals;
true
```

58.2 Operations for Abelian Number Fields

For operations for elements of abelian number fields, e.g., `Conductor` (see 18.1.5) or `ComplexConjugate` (see 18.5.2), see Chapter 18.

For a dense list l of cyclotomics, `DefaultField` (see 56.1.4) returns the smallest cyclotomic field containing all entries of l, `Field` (see 56.1.3) returns the smallest field containing all entries of l, which need not be a cyclotomic field. In both cases, the fields represent vector spaces over the rationals (see 58.3).

```
gap> DefaultField( [ E(5) ] ); DefaultField( [ E(3), ER(6) ] );
CF(5)
CF(24)
gap> Field( [ E(5) ] ); Field( [ E(3), ER(6) ] );
CF(5)
NF(24,[ 1, 19 ])
```

Factoring of polynomials over abelian number fields consisting of cyclotomics works in principle but is not very efficient if the degree of the field extension is large.

```
gap> x:= Indeterminate( CF(5) );
x_1
gap> Factors( PolynomialRing( Rationals ), x^5-1 );
[ x_1-1, x_1^4+x_1^3+x_1^2+x_1+1 ]
gap> Factors( PolynomialRing( CF(5) ), x^5-1 );
[ x_1-1, x_1+-E(5), x_1+-E(5)^2, x_1+-E(5)^3, x_1+-E(5)^4 ]
```

1 ▶ `IsNumberField( F )` P

returns `true` if the field F is a finite dimensional extension of a prime field in characteristic zero, and `false` otherwise.

2 ▶ `IsAbelianNumberField( F )` P

returns `true` if the field F is a number field (see 58.2.1) that is a Galois extension of the prime field, with abelian Galois group (see 56.3.1).

3 ▶ `IsCyclotomicField( F )` P

returns `true` if the field F is a **cyclotomic field**, i.e., an abelian number field (see 58.2.2) that can be generated by roots of unity.

```
gap> IsNumberField( CF(9) ); IsAbelianNumberField( Field( [ ER(3) ] ) );
true
true
gap> IsNumberField( GF(2) );
false
gap> IsCyclotomicField( CF(9) );
true
gap> IsCyclotomicField( Field( [ Sqrt(-3) ] ) );
true
gap> IsCyclotomicField( Field( [ Sqrt(3) ] ) );
false
```

4 ▶ `GaloisStabilizer( F )` A

Let F be an abelian number field (see 58.2.2) with conductor n, say. (This means that the n-th cyclotomic field is the smallest cyclotomic field containing F, see 18.1.5.) `GaloisStabilizer` returns the set of all those integers k in the range from 1 to n such that the field automorphism induced by raising n-th roots of unity to the k-th power acts trivially on F.

```
gap> r5:= Sqrt(5);
E(5)-E(5)^2-E(5)^3+E(5)^4
gap> GaloisCyc( r5, 4 ) = r5;  GaloisCyc( r5, 2 ) = r5;
true
false
gap> GaloisStabilizer( Field( [ r5 ] ) );
[ 1, 4 ]
```

58.3 Integral Bases of Abelian Number Fields

Each abelian number field is naturally a vector space over $\mathbb{Q}$. Moreover, if the abelian number field F contains the n-th cyclotomic field $\mathbb{Q}_n$ then F is a vector space over $\mathbb{Q}_n$. In GAP, each field object represents a vector space object over a certain subfield S, which depends on the way F was constructed. The subfield S can be accessed as the value of the attribute `LeftActingDomain` (see 55.1.11).

The return values of `NF` (see 58.1.2) and of the one argument versions of `CF` (see 58.1.1) represent vector spaces over $\mathbb{Q}$, and the return values of the two argument version of `CF` represent vector spaces over the field that is given as the first argument. For an abelian number field F and a subfield S of F, a GAP object representing F as a vector space over S can be constructed using `AsField` (see 56.1.9).

Let F be the cyclotomic field $\mathbb{Q}_n$, represented as a vector space over the subfield S. If S is the cyclotomic field $\mathbb{Q}_m$, with m a divisor of n, then `CanonicalBasis( F )` returns the Zumbroich basis of F relative to S, which consists of the roots of unity $\texttt{E}(n)^i$ where i is an element of the list `ZumbroichBase( n, m )` (see 58.3.1). If S is an abelian number field that is not a cyclotomic field then `CanonicalBasis( F )` returns a normal S-basis of F, i.e., a basis that is closed under the field automorphisms of F.

Let F be the abelian number field `NF( n, stab )`, with conductor n, that is itself not a cyclotomic field, represented as a vector space over the subfield S. If S is the cyclotomic field $\mathbb{Q}_m$, with m a divisor of n, then `CanonicalBasis( F )` returns the Lenstra basis of F relative to S that consists of the sums of roots of unity described by `LenstraBase( n, stab, stab, m )` (see 58.3.2). If S is an abelian number field that is not a cyclotomic field then `CanonicalBasis( F )` returns a normal S-basis of F.

```
gap> f:= CF(8);;   # a cycl. field over the rationals
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1, E(8), E(4), E(8)^3 ]
gap> Coefficients( b, Sqrt(-2) );
[ 0, 1, 0, 1 ]
gap> f:= AsField( CF(4), CF(8) );;  # a cycl. field over a cycl. field
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1, E(8) ]
gap> Coefficients( b, Sqrt(-2) );
[ 0, 1+E(4) ]
gap> f:= AsField( Field( [ Sqrt(-2) ] ), CF(8) );;
gap> # a cycl. field over a non-cycl. field
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1/2+1/2*E(8)-1/2*E(8)^2-1/2*E(8)^3, 1/2-1/2*E(8)+1/2*E(8)^2+1/2*E(8)^3 ]
gap> Coefficients( b, Sqrt(-2) );
[ E(8)+E(8)^3, E(8)+E(8)^3 ]
gap> f:= Field( [ Sqrt(-2) ] );   # a non-cycl. field over the rationals
NF(8,[ 1, 3 ])
gap> b:= CanonicalBasis( f );;  BasisVectors( b );
[ 1, E(8)+E(8)^3 ]
gap> Coefficients( b, Sqrt(-2) );
[ 0, 1 ]
```

1 ▶ `ZumbroichBase( n, m )` F

Let n and m be positive integers, such that m divides n. `ZumbroichBase` returns the set of exponents i for which `E(n)^i` belongs to the (generalized) Zumbroich basis of the cyclotomic field $\mathbb{Q}_n$, viewed as a vector space over $\mathbb{Q}_m$.

This basis is defined as follows. Let P denote the set of prime divisors of n, $n = \prod_{p \in P} p^{\nu_p}$, and $m = \prod_{p \in P} p^{\mu_p}$ with $\mu_p \leq \nu_p$. Let $e_n = \texttt{E}(n)$, and $\{e_{n_1}^j\}_{j \in J} \otimes \{e_{n_2}^k\}_{k \in K} = \{e_{n_1}^j \cdot e_{n_2}^k\}_{j \in J, k \in K}$.

Then the basis is

$$B_{n,m} = \bigotimes_{p \in P} \bigotimes_{k=\mu_p}^{\nu_p - 1} \{ e_{p^{k+1}}^{j} \}_{j \in J_{k,p}} \quad \text{where} \quad J_{k,p} = \begin{cases} \{0\} & ; \quad k = 0, p = 2 \\ \{0, 1\} & ; \quad k > 0, p = 2 \\ \{1, \ldots, p-1\} & ; \quad k = 0, p \neq 2 \\ \{-\frac{p-1}{2}, \ldots, \frac{p-1}{2}\} & ; \quad k > 0, p \neq 2 \end{cases}$$

$B_{n,1}$ is equal to the basis of $\mathbb{Q}_n$ over the rationals which is introduced in [Zum89]. Also the conversion of arbitrary sums of roots of unity into its basis representation, and the reduction to the minimal cyclotomic field are described in this thesis. (Note that the notation here is slightly different from that there.)

$B_{n,m}$ consists of roots of unity, it is an integral basis (that is, exactly the integral elements in $\mathbb{Q}_n$ have integral coefficients w.r.t. $B_{n,m}$, cf. 18.1.4), it is a normal basis for squarefree n and closed under complex conjugation for odd n.

Note: For $n \equiv 2 \pmod 4$, we have `ZumbroichBase(`n`, 1) = 2 * ZumbroichBase(`$n/2$`, 1)` and `List( ZumbroichBase(`n`, 1), x -> E(`n`)^x ) = List( ZumbroichBase(`$n/2$`, 1), x -> E(`$n/2$`)^x )`.

```
gap> ZumbroichBase( 15, 1 ); ZumbroichBase( 12, 3 );
[ 1, 2, 4, 7, 8, 11, 13, 14 ]
[ 0, 3 ]
gap> ZumbroichBase( 10, 2 ); ZumbroichBase( 32, 4 );
[ 2, 4, 6, 8 ]
[ 0, 1, 2, 3, 4, 5, 6, 7 ]
```

2 ► `LenstraBase(` *n*`,` *stabilizer*`,` *super*`,` *m* `)` F

Let n and m be positive integers, such that m divides n, *stabilizer* be a list of prime residues modulo n, which describes a subfield of the n-th cyclotomic field (see 58.2.4), and *super* be a list representing a supergroup of the group given by *stabilizer*.

`LenstraBase` returns a list $[b_1, b_2, \ldots, b_k]$ of lists, each b_i consisting of integers such that the elements $\sum_{j \in b_i} \texttt{E(n)}^j$ form a basis of the abelian number field `NF(` *n*`,` *stabilizer* `)`, as a vector space over the m-th cyclotomic field (see 58.1.2).

This basis is an integral basis, that is, exactly the integral elements in `NF(` *n*`,` *stabilizer* `)` have integral coefficients. (For details about this basis, see [Bre97].)

If possible then the result is chosen such that the group described by *super* acts on it, consistently with the action of *stabilizer*, i.e., each orbit of *super* is a union of orbits of *stabilizer*. (A usual case is *super* `=` *stabilizer*, so there is no additional condition.

Note: The b_i are in general not sets, since for *stabilizer* `=` *super*, the first entry is always an element of `ZumbroichBase(` *n*`,` *m* `)`; this property is used by `NF` (see 58.1.2) and `Coefficients` (see 58.3).

stabilizer must not contain the stabilizer of a proper cyclotomic subfield of the n-th cyclotomic field, i.e., the result must describe a basis for a field with conductor n.

```
gap> LenstraBase( 24, [ 1, 19 ], [ 1, 19 ], 1 );
[ [ 1, 19 ], [ 8 ], [ 11, 17 ], [ 16 ] ]
gap> LenstraBase( 24, [ 1, 19 ], [ 1, 5, 19, 23 ], 1 );
[ [ 1, 19 ], [ 5, 23 ], [ 8 ], [ 16 ] ]
gap> LenstraBase( 15, [ 1, 4 ], PrimeResidues( 15 ), 1 );
[ [ 1, 4 ], [ 2, 8 ], [ 7, 13 ], [ 11, 14 ] ]
```

The first two results describe two bases of the field $\mathbb{Q}_3(\sqrt{6})$, the third result describes a normal basis of $\mathbb{Q}_3(\sqrt{5})$.

58.4 Galois Groups of Abelian Number Fields

The field automorphisms of the cyclotomic field $\mathbb{Q}_n$ (see Chapter 18) are given by the linear maps $*k$ on $\mathbb{Q}_n$ that are defined by $\texttt{E}(n)^{*k} = \texttt{E}(n)^k$, where $1 \leq k < n$ and $\texttt{Gcd}(n,k) = 1$ hold (see 18.5.1). Note that this action is **not** equal to exponentiation of cyclotomics, i.e., for general cyclotomics z, z^{*k} is different from z^k. (In GAP, the image of a cyclotomic z under $*k$ can be computed as `GaloisCyc`(z,k).)

```
gap> ( E(5) + E(5)^4 )^2; GaloisCyc( E(5) + E(5)^4, 2 );
-2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
E(5)^2+E(5)^3
```

For $\texttt{Gcd}(n,k) \neq 1$, the map $\texttt{E}(n) \mapsto \texttt{E}(n)^k$ does **not** define a field automorphism of $\mathbb{Q}_n$ but only a $\mathbb{Q}$-linear map.

```
gap> GaloisCyc( E(5)+E(5)^4, 5 ); GaloisCyc( ( E(5)+E(5)^4 )^2, 5 );
2
-6
```

1 ▶ `ANFAutomorphism(` F`,` k `)` F

Let F be an abelian number field and k an integer that is coprime to the conductor (see 18.1.5) of F. Then `ANFAutomorphism` returns the automorphism of F that is defined as the linear extension of the map that raises each root of unity in F to its k-th power.

```
gap> f:= CF(25);
CF(25)
gap> alpha:= ANFAutomorphism( f, 2 );
ANFAutomorphism( CF(25), 2 )
gap> alpha^2;
ANFAutomorphism( CF(25), 4 )
gap> Order( alpha );
20
gap> E(5)^alpha;
E(5)^2
```

The Galois group $Gal(\mathbb{Q}_n, \mathbb{Q})$ of the field extension $\mathbb{Q}_n/\mathbb{Q}$ is isomorphic to the group $(\mathbb{Z}/n\mathbb{Z})^*$ of prime residues modulo n, via the isomorphism $(\mathbb{Z}/n\mathbb{Z})^* \to Gal(\mathbb{Q}_n, \mathbb{Q})$ that is defined by $k + n\mathbb{Z} \mapsto (z \mapsto z^{*k})$.

The Galois group of the field extension $\mathbb{Q}_n/L$ with any abelian number field $L \subseteq \mathbb{Q}_n$ is simply the factor group of $Gal(\mathbb{Q}_n, \mathbb{Q})$ modulo the stabilizer of L, and the Galois group of L/L', with L' an abelian number field contained in L, is the subgroup in this group that stabilizes L'. These groups are easily described in terms of $(\mathbb{Z}/n\mathbb{Z})^*$. Generators of $(\mathbb{Z}/n\mathbb{Z})^*$ can be computed using `GeneratorsPrimeResidues` (see 15.1.4).

In GAP, a field extension L/L' is given by the field object L with `LeftActingDomain` value L' (see 58.3).

```
gap> f:= CF(15);
CF(15)
gap> g:= GaloisGroup( f );
<group with 2 generators>
gap> Size( g ); IsCyclic( g ); IsAbelian( g );
8
false
true
gap> Action( g, NormalBase( f ), OnPoints );
Group([ (1,6)(2,4)(3,8)(5,7), (1,4,3,7)(2,8,5,6) ])
```

The following example shows Galois groups of a cyclotomic field and of a proper subfield that is not a cyclotomic field.

```
gap> gens1:= GeneratorsOfGroup( GaloisGroup( CF(5) ) );
[ ANFAutomorphism( CF(5), 2 ) ]
gap> gens2:= GeneratorsOfGroup( GaloisGroup( Field( Sqrt(5) ) ) );
[ ANFAutomorphism( NF(5,[ 1, 4 ]), 2 ) ]
gap> Order( gens1[1] );  Order( gens2[1] );
4
2
gap> Sqrt(5)^gens1[1] = Sqrt(5)^gens2[1];
true
```

The following example shows the Galois group of a cyclotomic field over a non-cyclotomic field.

```
gap> g:= GaloisGroup( AsField( Field( [ Sqrt(5) ] ), CF(5) ) );
<group with 1 generators>
gap> gens:= GeneratorsOfGroup( g );
[ ANFAutomorphism( AsField( NF(5,[ 1, 4 ]), CF(5) ), 4 ) ]
gap> x:= last[1];;  x^2;
IdentityMapping( AsField( NF(5,[ 1, 4 ]), CF(5) ) )
```

58.5 Gaussians

1 ▶ `GaussianIntegers` V

`GaussianIntegers` is the ring $\mathbb{Z}[\sqrt{-1}]$ of Gaussian integers. This is a subring of the cyclotomic field `GaussianRationals`, see 58.1.3.

2 ▶ `IsGaussianIntegers(` *obj* `)` C

is the defining category for the domain `GaussianIntegers`.

59 Vector Spaces

1 ▶ `IsLeftVectorSpace( V )` C
▶ `IsVectorSpace( V )` C

A **vector space** in GAP is a free left module (see 55.3.1) over a division ring (see Chapter 56).

Whenever we talk about an F-vector space V then V is an additive group (see 53.1.6) on which the division ring F acts via multiplication from the left such that this action and the addition in V are left and right distributive. The division ring F can be accessed as value of the attribute `LeftActingDomain` (see 55.1.11).

The characteristic (see 30.10.1) of a vector space is equal to the characteristic of its left acting domain.

Vector spaces in GAP are always **left** vector spaces, `IsLeftVectorSpace` and `IsVectorSpace` are synonyms.

59.1 Constructing Vector Spaces

1 ▶ `VectorSpace( F, gens[, zero][, "basis"] )` F

For a field F and a collection *gens* of vectors, `VectorSpace` returns the F-vector space spanned by the elements in *gens*.

The optional argument *zero* can be used to specify the zero element of the space; *zero* **must** be given if *gens* is empty. The optional string `"basis"` indicates that *gens* is known to be linearly independent over F, in particular the dimension of the vector space is immediately set; note that `Basis` (see 59.4.2) need **not** return the basis formed by *gens* if the argument `"basis"` is given.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );
<vector space over Rationals, with 2 generators>
```

2 ▶ `Subspace( V, gens[, "basis"] )` F
▶ `SubspaceNC( V, gens[, "basis"] )` F

For an F-vector space V and a list or collection *gens* that is a subset of V, `Subspace` returns the F-vector space spanned by *gens*; if *gens* is empty then the trivial subspace (see 59.2.2) of V is returned. The parent (see 30.7) of the returned vector space is set to V.

`SubspaceNC` does the same as `Subspace`, except that it omits the check whether *gens* is a subset of V.

The optional string `"basis"` indicates that *gens* is known to be linearly independent over F. In this case the dimension of the subspace is immediately set, and both `Subspace` and `SubspaceNC` do **not** check whether *gens* really is linearly independent and whether *gens* is a subset of V.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );;
gap> W:= Subspace( V, [ [ 0, 1, 2 ] ] );
<vector space over Rationals, with 1 generators>
```

3 ▶ `AsVectorSpace( F, D )` O

Let F be a division ring and D a domain. If the elements in D form an F-vector space then `AsVectorSpace` returns this F-vector space, otherwise `fail` is returned.

`AsVectorSpace` can be used for example to view a given vector space as a vector space over a smaller or larger division ring.

```
gap> V:= FullRowSpace( GF( 27 ), 3 );
( GF(3^3)^3 )
gap> Dimension( V );  LeftActingDomain( V );
3
GF(3^3)
gap> W:= AsVectorSpace( GF( 3 ), V );
<vector space over GF(3), with 9 generators>
gap> Dimension( W );  LeftActingDomain( W );
9
GF(3)
gap> AsVectorSpace( GF( 9 ), V );
fail
```

4 ▶ `AsSubspace( V, U )` O

Let V be an F-vector space, and U a collection. If U is a subset of V such that the elements of U form an F-vector space then `AsSubspace` returns this vector space, with parent set to V (see 59.1.3). Otherwise `fail` is returned.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 3 ], [ 1, 1, 1 ] ] );;
gap> W:= VectorSpace( Rationals, [ [ 1/2, 1/2, 1/2 ] ] );;
gap> U:= AsSubspace( V, W );
<vector space over Rationals, with 1 generators>
gap> Parent( U ) = V;
true
gap> AsSubspace( V, [ [ 1, 1, 1 ] ] );
fail
```

59.2 Operations and Attributes for Vector Spaces

1 ▶ `GeneratorsOfLeftVectorSpace( V )` A
▶ `GeneratorsOfVectorSpace( V )` A

For an F-vector space V, `GeneratorsOfLeftVectorSpace` returns a list of vectors in V that generate V as an F-vector space.

```
gap> GeneratorsOfVectorSpace( FullRowSpace( Rationals, 3 ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
```

2 ▶ `TrivialSubspace( V )` A

For a vector space V, `TrivialSubspace` returns the subspace of V that consists of the zero vector in V.

```
gap> V:= GF(3)^3;;
gap> triv:= TrivialSubspace( V );
<vector space over GF(3), with 0 generators>
gap> AsSet( triv );
[ [ 0*Z(3), 0*Z(3), 0*Z(3) ] ]
```

59.3 Domains of Subspaces of Vector Spaces

1 ▶ `Subspaces( V )` A
▶ `Subspaces( V, k )` O

Let V be a finite vector space. In the first form, `Subspaces` returns the domain of all subspaces of V. In the second form, k must be a nonnegative integer, and `Subspaces` returns the domain of all k-dimensional subspaces of V.

Special `Size` and `Iterator` methods are provided for these domains.

2 ▶ `IsSubspacesVectorSpace( D )` C

The domain of all subspaces of a (finite) vector space or of all subspaces of fixed dimension, as returned by `Subspaces` (see 59.3.1) lies in the category `IsSubspacesVectorSpace`.

```
gap> D:= Subspaces( GF(3)^3 );
Subspaces( ( GF(3)^3 ) )
gap> Size( D );
28
gap> iter:= Iterator( D );;
gap> NextIterator( iter );
<vector space over GF(3), with 0 generators>
gap> NextIterator( iter );
<vector space of dimension 1 over GF(3)>
gap> IsSubspacesVectorSpace( D );
true
```

59.4 Bases of Vector Spaces

In GAP, a **basis** of a free left F-module V is a list of vectors $B = [v_1, v_2, \ldots, v_n]$ in V such that V is generated as a left F-module by these vectors and such that B is linearly independent over F. The integer n is the dimension of V (see 55.3.4). In particular, as each basis is a list (see Chapter 21), it has a length (see 21.17.5), and the i-th vector of B can be accessed as $B[i]$.

```
gap> V:= Rationals^3;
( Rationals^3 )
gap> B:= Basis( V );
CanonicalBasis( ( Rationals^3 ) )
gap> Length( B );
3
gap> B[1];
[ 1, 0, 0 ]
```

The operations described below make sense only for bases of **finite** dimensional vector spaces. (In practice this means that the vector spaces must be **low** dimensional, that is, the dimension should not exceed a few hundred.)

Besides the basic operations for lists (see 21.2), the **basic operations for bases** are `BasisVectors` (see 59.5.1), `Coefficients` (see 59.5.3), `LinearCombination` (see 59.5.4), and `UnderlyingLeftModule` (see 59.5.2). These and other operations for arbitrary bases are described in 59.5.

For special kinds of bases, further operations are defined (see 59.6).

GAP supports the following three kinds of bases.

Relative bases delegate the work to other bases of the same free left module, via basechange matrices (see 59.4.4).

Bases handled by nice bases delegate the work to bases of isomorphic left modules over the same left acting domain (see 59.10).

Finally, of course there must be bases in GAP that really do the work.

For example, in the case of a Gaussian row or matrix space V (see 59.8), `Basis( V )` is a semi-echelonized basis (see 59.8.7) that uses Gaussian elimination; such a basis is of the third kind. `Basis( V, vectors )` is either semi-echelonized or a relative basis. Other examples of bases of the third kind are canonical bases of finite fields and of abelian number fields.

Bases handled by nice bases are described in 59.10. Examples are non-Gaussian row and matrix spaces, and subspaces of finite fields and abelian number fields that are themselves not fields.

1 ▶ `IsBasis( obj )` C

In GAP, a **basis** of a free left module is an object that knows how to compute coefficients w.r.t. its basis vectors (see 59.5.3). Bases are constructed by `Basis` (see 59.4.2). Each basis is an immutable list, the i-th entry being the i-th basis vector.

(See 59.7 for mutable bases.)

```
gap> V:= GF(2)^2;;
gap> B:= Basis( V );;
gap> IsBasis( B );
true
gap> IsBasis( [ [ 1, 0 ], [ 0, 1 ] ] );
false
gap> IsBasis( Basis( Rationals^2, [ [ 1, 0 ], [ 0, 1 ] ] ) );
true
```

2 ▶ `Basis( V )` A
▶ `Basis( V, vectors )` O
▶ `BasisNC( V, vectors )` O

Called with a free left F-module V as the only argument, `Basis` returns an F-basis of V whose vectors are not further specified.

If additionally a list *vectors* of vectors in V is given that forms an F-basis of V then `Basis` returns this basis; if *vectors* is not linearly independent over F or does not generate V as a free left F-module then `fail` is returned.

`BasisNC` does the same as `Basis` for two arguments, except that it does not check whether *vectors* form a basis.

If no basis vectors are prescribed then `Basis` need not compute basis vectors; in this case, the vectors are computed in the first call to `BasisVectors`.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V );
SemiEchelonBasis( <vector space over Rationals, with 2 generators>, ... )
gap> BasisVectors( B );
[ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ]
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 3, 2, 30 ] ] );
Basis( <vector space over Rationals, with 2 generators>,
[ [ 1, 2, 7 ], [ 3, 2, 30 ] ] )
gap> Basis( V, [ [ 1, 2, 3 ] ] );
fail
```

3 ▶ `CanonicalBasis( V )` A

If the vector space V supports a **canonical basis** then `CanonicalBasis` returns this basis, otherwise `fail` is returned.

The defining property of a canonical basis is that its vectors are uniquely determined by the vector space. If canonical bases exist for two vector spaces over the same left acting domain (see 55.1.11) then the equality of these vector spaces can be decided by comparing the canonical bases.

The exact meaning of a canonical basis depends on the type of V. Canonical bases are defined for example for Gaussian row and matrix spaces (see 59.8).

If one designs a new kind of vector spaces (see 59.11) and defines a canonical basis for these spaces then the `CanonicalBasis` method one installs (see 2.2.1 in "Programming in GAP") must **not** call `Basis`. On the other hand, one probably should install a `Basis` method that simply calls `CanonicalBasis`, the value of the method (see 2.2 and 2.3 in "Programming in GAP") being `CANONICAL_BASIS_FLAGS`.

```
gap> vecs:= [ [ 1, 2, 3 ], [ 1, 1, 1 ], [ 1, 1, 1 ] ];;
gap> V:= VectorSpace( Rationals, vecs );;
gap> B:= CanonicalBasis( V );
CanonicalBasis( <vector space over Rationals, with 3 generators> )
gap> BasisVectors( B );
[ [ 1, 0, -1 ], [ 0, 1, 2 ] ]
```

4 ▶ `RelativeBasis( `*B*`, `*vectors*` )` O
▶ `RelativeBasisNC( `*B*`, `*vectors*` )` O

A relative basis is a basis of the free left module V that delegates the computation of coefficients etc. to another basis of V via a basechange matrix.

Let B be a basis of the free left module V, and *vectors* a list of vectors in V.

`RelativeBasis` checks whether *vectors* form a basis of V, and in this case a basis is returned in which *vectors* are the basis vectors; otherwise `fail` is returned.

`RelativeBasisNC` does the same, except that it omits the check.

59.5 Operations for Vector Space Bases

1 ▶ `BasisVectors( `*B*` )` A

For a vector space basis B, `BasisVectors` returns the list of basis vectors of B. The lists B and `BasisVectors( `*B*` )` are equal; the main purpose of `BasisVectors` is to provide access to a list of vectors that does **not** know about an underlying vector space.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ] );;
gap> BasisVectors( B );
[ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ]
```

2 ▶ `UnderlyingLeftModule( `*B*` )` A

For a basis B of a free left module V, say, `UnderlyingLeftModule` returns V.

The reason why a basis stores a free left module is that otherwise one would have to store the basis vectors and the coefficient domain separately. Storing the module allows one for example to deal with bases whose basis vectors have not yet been computed yet (see 59.4.2); furthermore, in some cases it is convenient to test membership of a vector in the module before computing coefficients w.r.t. a basis.

```
gap> B:= Basis( GF(2)^6 );;  UnderlyingLeftModule( B );
( GF(2)^6 )
```

3 ▶ `Coefficients( `*B*`, `*v*` )` O

Let V be the underlying left module of the basis B, and v a vector such that the family of v is the elements family of the family of V. Then `Coefficients( `*B*`, `*v*` )` is the list of coefficients of v w.r.t. B if v lies in V, and `fail` otherwise.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ] );;
gap> Coefficients( B, [ 1/2, 1/3, 5 ] );
[ 1/2, -2/3 ]
gap> Coefficients( B, [ 1, 0, 0 ] );
fail
```

4 ▶ `LinearCombination( B, coeff )` O
▶ `LinearCombination( vectors, coeff )` O

If B is a basis of length n, say, and *coeff* is a row vector of the same length as B, `LinearCombination` returns the vector $\sum_{i=1}^{n} coeff[i] * B[i]$.

If *vectors* and *coeff* are homogeneous lists of the same length n, say, `LinearCombination` returns the vector $\sum_{i=1}^{n} coeff[i] * vectors[i]$. Perhaps the most important usage is the case where *vectors* forms a basis.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V, [ [ 1, 2, 7 ], [ 0, 1, -9/4 ] ] );;
gap> LinearCombination( B, [ 1/2, -2/3 ] );
[ 1/2, 1/3, 5 ]
```

5 ▶ `EnumeratorByBasis( B )` A

For a basis B of the free left F-module V of dimension n, say, `EnumeratorByBasis` returns an enumerator that loops over the elements of V as linear combinations of the vectors of B with coefficients the row vectors in the full row space (see 59.8.4) of dimension n over F, in the succession given by the default enumerator of this row space.

```
gap> V:= GF(2)^3;;
gap> enum:= EnumeratorByBasis( CanonicalBasis( V ) );;
gap> Print( enum{ [ 1 .. 4 ] }, "\n" );
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ 0*Z(2), 0*Z(2), Z(2)^0 ],
  [ 0*Z(2), Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, Z(2)^0 ] ]
gap> B:= Basis( V, [ [ 1, 1, 1 ], [ 1, 1, 0 ], [ 1, 0, 0 ] ] * Z(2) );;
gap> enum:= EnumeratorByBasis( B );;
gap> Print( enum{ [ 1 .. 4 ] }, "\n" );
[ [ 0*Z(2), 0*Z(2), 0*Z(2) ], [ Z(2)^0, 0*Z(2), 0*Z(2) ],
  [ Z(2)^0, Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0, 0*Z(2) ] ]
```

6 ▶ `IteratorByBasis( B )` O

For a basis B of the free left F-module V of dimension n, say, `IteratorByBasis` returns an iterator that loops over the elements of V as linear combinations of the vectors of B with coefficients the row vectors in the full row space (see 59.8.4) of dimension n over F, in the succession given by the default enumerator of this row space.

```
gap> V:= GF(2)^3;;
gap> iter:= IteratorByBasis( CanonicalBasis( V ) );;
gap> for i in [ 1 .. 4 ] do Print( NextIterator( iter ), "\n" ); od;
[ 0*Z(2), 0*Z(2), 0*Z(2) ]
[ 0*Z(2), 0*Z(2), Z(2)^0 ]
[ 0*Z(2), Z(2)^0, 0*Z(2) ]
[ 0*Z(2), Z(2)^0, Z(2)^0 ]
gap> B:= Basis( V, [ [ 1, 1, 1 ], [ 1, 1, 0 ], [ 1, 0, 0 ] ] * Z(2) );;
gap> iter:= IteratorByBasis( B );;
gap> for i in [ 1 .. 4 ] do Print( NextIterator( iter ), "\n" ); od;
```

```
[ 0*Z(2), 0*Z(2), 0*Z(2) ]
[ Z(2)^0, 0*Z(2), 0*Z(2) ]
[ Z(2)^0, Z(2)^0, 0*Z(2) ]
[ 0*Z(2), Z(2)^0, 0*Z(2) ]
```

59.6 Operations for Special Kinds of Bases

1 ▶ `IsCanonicalBasis( B )` P

If the underlying free left module V of the basis B supports a canonical basis (see 59.4.3) then `IsCanonicalBasis` returns `true` if B is equal to the canonical basis of V, and `false` otherwise.

2 ▶ `IsIntegralBasis( B )` P

Let B be an S-basis of a **field** F, say, for a subfield S of F, and let R and M be the rings of algebraic integers in S and F, respectively. `IsIntegralBasis` returns `true` if B is also an R-basis of M, and `false` otherwise.

3 ▶ `IsNormalBasis( B )` P

Let B be an S-basis of a **field** F, say, for a subfield S of F. `IsNormalBasis` returns `true` if B is invariant under the Galois group (see 56.3.1) of the field extension F/S, and `false` otherwise.

```
gap> B:= CanonicalBasis( GaussianRationals );
CanonicalBasis( GaussianRationals )
gap> IsIntegralBasis( B );  IsNormalBasis( B );
true
false
```

4 ▶ `StructureConstantsTable( B )` A

Let B be a basis of a free left module R, say, that is also a ring. In this case `StructureConstantsTable` returns a structure constants table T in sparse representation, as used for structure constants algebras (see Section 6.2 of the GAP User's Tutorial).

If B has length n then T is a list of length $n+2$. The first n entries of T are lists of length n. $T[n+1]$ is one of 1, -1, or 0; in the case of 1 the table is known to be symmetric, in the case of -1 it is known to be antisymmetric, and 0 occurs in all other cases. $T[n+2]$ is the zero element of the coefficient domain.

The coefficients w.r.t. B of the product of the i-th and j-th basis vector of B are stored in $T[i][j]$ as a list of length 2; its first entry is the list of positions of nonzero coefficients, the second entry is the list of these coefficients themselves.

The multiplication in an algebra A with vector space basis B with basis vectors $[v_1, \ldots, v_n]$ is determined by the so-called structure matrices $M_k = [m_{ijk}]_{ij}, 1 \leq k \leq n$. The M_k are defined by $v_i v_j = \sum_k m_{i,j,k} v_k$. Let $a = [a_1, \ldots, a_n]$ and $b = [b_1, \ldots, b_n]$. Then

$$(\sum_i a_i v_i)(\sum_j b_j v_j) = \sum_{i,j} a_i b_j (v_i v_j) = \sum_k (\sum_j (\sum_i a_i m_{i,j,k}) b_j) v_k = \sum_k (a M_k b^{tr}) v_k.$$

In the following example we temporarily increase the line length limit from its default value 80 to 83 in order to get a nicer output format.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> SizeScreen([ 83, ]);;
gap> StructureConstantsTable( Basis( A ) );
[ [ [ [ 1 ], [ 1 ] ], [ [ 2 ], [ 1 ] ], [ [ 3 ], [ 1 ] ], [ [ 4 ], [ 1 ] ] ],
  [ [ [ 2 ], [ 1 ] ], [ [ 1 ], [ -1 ] ], [ [ 4 ], [ 1 ] ], [ [ 3 ], [ -1 ] ] ],
  [ [ [ 3 ], [ 1 ] ], [ [ 4 ], [ -1 ] ], [ [ 1 ], [ -1 ] ], [ [ 2 ], [ 1 ] ] ],
  [ [ [ 4 ], [ 1 ] ], [ [ 3 ], [ 1 ] ], [ [ 2 ], [ -1 ] ], [ [ 1 ], [ -1 ] ] ],
  0, 0 ]
gap> SizeScreen([ 80, ]);;
```

59.7 Mutable Bases

It is useful to have a **mutable basis** of a free module when successively closures with new vectors are formed, since one does not want to create a new module and a corresponding basis for each step.

Note that the situation here is different from the situation with stabilizer chains, which are (mutable or immutable) records that do not need to know about the groups they describe, whereas each (immutable) basis stores the underlying left module (see 59.5.2).

So immutable bases and mutable bases are different categories of objects. The only thing they have in common is that one can ask both for their basis vectors and for the coefficients of a given vector.

Since `Immutable` produces an immutable copy of any GAP object, it would in principle be possible to construct a mutable basis that is in fact immutable. In the sequel, we will deal only with mutable bases that are in fact **mutable** GAP objects, hence these objects are unable to store attribute values.

Basic operations for immutable bases are `NrBasisVectors` (see 59.7.3), `IsContainedInSpan` (see 59.7.5), `CloseMutableBasis` (see 59.7.6), `ImmutableBasis` (see 59.7.4), `Coefficients` (see 59.5.3), and `BasisVectors` (see 59.5.1). `ShallowCopy` (see 12.7.1) for a mutable basis returns a mutable plain list containing the current basis vectors.

Since mutable bases do not admit arbitrary changes of their lists of basis vectors, a mutable basis is **not** a list. It is, however, a collection, more precisely its family (see 13.1) equals the family of its collection of basis vectors.

Mutable bases can be constructed with `MutableBasis`.

Similar to the situation with bases (cf. 59.4), GAP supports the following three kinds of mutable bases.

The **generic method** of `MutableBasis` returns a mutable basis that simply stores an immutable basis; clearly one wants to avoid this whenever possible with reasonable effort.

There are mutable bases that store a mutable basis for a nicer module. Note that this is meaningful only if the mechanism of computing nice and ugly vectors (see 59.10) is invariant under closures of the basis; this is the case for example if the vectors are matrices, Lie objects, or elements of structure constants algebras.

There are mutable bases that use special information to perform their tasks; examples are mutable bases of Gaussian row and matrix spaces.

1 ▶ `IsMutableBasis(` *MB* `)` C

Every mutable basis lies in the category `IsMutableBasis`.

2 ▶ `MutableBasis(` *R*`,` *vectors*`[,` *zero*`] )` O

`MutableBasis` returns a mutable basis for the R-free module generated by the vectors in the list *vectors*. The optional argument *zero* is the zero vector of the module; it must be given if *vectors* is empty.

Note that *vectors* will in general **not** be the basis vectors of the mutable basis!

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 2, 3 ], [ 0, 1, 0 ] ] );
<mutable basis over Rationals, 2 vectors>
```

3 ▶ `NrBasisVectors(` *MB* `)` O

For a mutable basis *MB*, `NrBasisVectors` returns the current number of basis vectors of *MB*. Note that this operation is **not** an attribute, as it makes no sense to store the value. `NrBasisVectors` is used mainly as an equivalent of `Dimension` for the underlying left module in the case of immutable bases.

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 1], [ 2, 2 ] ] );;
gap> NrBasisVectors( MB );
1
```

4 ▶ `ImmutableBasis(` *MB*`[,` *V*`] )` O

`ImmutableBasis` returns the immutable basis B, say, with the same basis vectors as in the mutable basis *MB*.

If the second argument V is present then V is the value of `UnderlyingLeftModule` (see 59.5.2) for B. The second variant is used mainly for the case that one knows the module for the desired basis in advance, and if it has a nicer structure than the module known to *MB*, for example if it is an algebra.

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 1 ], [ 2, 2 ] ] );;
gap> B:= ImmutableBasis( MB );
SemiEchelonBasis( <vector space of dimension 1 over Rationals>, [ [ 1, 1 ] ] )
gap> UnderlyingLeftModule( B );
<vector space of dimension 1 over Rationals>
```

5 ▶ `IsContainedInSpan(` *MB*`,` *v* `)` O

For a mutable basis *MB* over the coefficient ring R, say, and a vector v, `IsContainedInSpan` returns `true` is v lies in the R-span of the current basis vectors of *MB*, and `false` otherwise.

6 ▶ `CloseMutableBasis(` *MB*`,` *v* `)` O

For a mutable basis *MB* over the coefficient ring R, say, and a vector v, `CloseMutableBasis` changes *MB* such that afterwards it describes the R-span of the former basis vectors together with v.

Note that if v enlarges the dimension then this does in general **not** mean that v is simply added to the basis vectors of *MB*. Usually a linear combination of v and the other basis vectors is added, and also the old basis vectors may be modified, for example in order to keep the list of basis vectors echelonized (see 59.8.7).

```
gap> MB:= MutableBasis( Rationals, [ [ 1, 1, 3 ], [ 2, 2, 1 ] ] );
<mutable basis over Rationals, 2 vectors>
gap> IsContainedInSpan( MB, [ 1, 0, 0 ] );
false
gap> CloseMutableBasis( MB, [ 1, 0, 0 ] );
gap> MB;
<mutable basis over Rationals, 3 vectors>
gap> IsContainedInSpan( MB, [ 1, 0, 0 ] );
true
```

59.8 Row and Matrix Spaces

1 ▶ `IsRowSpace(` *V* `)` F

A **row space** in GAP is a vector space that consists of row vectors (see Chapter 23).

2 ▶ `IsMatrixSpace(` *V* `)` F

A **matrix space** in GAP is a vector space that consists of matrices (see Chapter 24).

3 ▶ `IsGaussianSpace(` *V* `)` F

The filter `IsGaussianSpace` (see 13.2) for the row space (see 59.8.1) or matrix space (see 59.8.2) V over the field F, say, indicates that the entries of all row vectors or matrices in V, respectively, are all contained in F. In this case, V is called a **Gaussian** vector space. Bases for Gaussian spaces can be computed using Gaussian elimination for a given list of vector space generators.

```
gap> mats:= [ [[1,1],[2,2]], [[3,4],[0,1]] ];;
gap> V:= VectorSpace( Rationals, mats );;
gap> IsGaussianSpace( V );
true
gap> mats[1][1][1]:= E(4);;   # an element in an extension field
gap> V:= VectorSpace( Rationals, mats );;
gap> IsGaussianSpace( V );
false
gap> V:= VectorSpace( Field( Rationals, [ E(4) ] ), mats );;
gap> IsGaussianSpace( V );
true
```

4 ▶ `FullRowSpace( F, n )` F

For a field F and a nonnegative integer n, `FullRowSpace` returns the F-vector space that consists of all row vectors (see 23) of length n with entries in F.

An alternative to construct this vector space is via F`^`n.

```
gap> FullRowSpace( GF( 9 ), 3 );
( GF(3^2)^3 )
gap> GF(9)^3;           # the same as above
( GF(3^2)^3 )
```

5 ▶ `FullMatrixSpace( F, m, n )` F

For a field F and two positive integers m and n, `FullMatrixSpace` returns the F-vector space that consists of all m by n matrices (see 24.1.1) with entries in F.

If m `=` n then the result is in fact an algebra (see 60.4.4).

An alternative to construct this vector space is via F`^[`m`,`n`]`.

```
gap> FullMatrixSpace( GF(2), 4, 5 );
( GF(2)^[ 4, 5 ] )
gap> GF(2)^[ 4, 5 ];    # the same as above
( GF(2)^[ 4, 5 ] )
```

6 ▶ `DimensionOfVectors( M )` A

For a left module M that consists of row vectors (see 55.3.7), `DimensionOfVectors` returns the common length of all row vectors in M. For a left module M that consists of matrices (see 55.3.8), `DimensionOfVectors` returns the common matrix dimensions (see 24.3.1) of all matrices in M.

```
gap> DimensionOfVectors( GF(2)^5 );
5
gap> DimensionOfVectors( GF(2)^[2,3] );
[ 2, 3 ]
```

7 ▶ `IsSemiEchelonized( B )` P

Let B be a basis of a Gaussian row or matrix space V, say (see 59.8.3) over the field F.

If V is a row space then B is semi-echelonized if the matrix formed by its basis vectors has the property that the first nonzero element in each row is the identity of F, and all values exactly below these pivot elements are the zero of F (cf. 24.9.1).

If V is a matrix space then B is semi-echelonized if the matrix obtained by replacing each basis vector by the concatenation of its rows is semi-echelonized (see above, cf. 24.9.4).

```
gap> V:= GF(2)^2;;
gap> B1:= Basis( V, [ [ 0, 1 ], [ 1, 0 ] ] * Z(2) );;
gap> IsSemiEchelonized( B1 );
true
gap> B2:= Basis( V, [ [ 0, 1 ], [ 1, 1 ] ] * Z(2) );;
gap> IsSemiEchelonized( B2 );
false
```

8 ▶ `SemiEchelonBasis(` *V* `)` A
▶ `SemiEchelonBasis(` *V*`,` *vectors* `)` O
▶ `SemiEchelonBasisNC(` *V*`,` *vectors* `)` O

Let V be a Gaussian row or matrix vector space over the field F (see 59.8.3, 59.8.1, 59.8.2).

Called with V as the only argument, `SemiEchelonBasis` returns a basis of V that has the property `IsSemiEchelonized` (see 59.8.7).

If additionally a list *vectors* of vectors in V is given that forms a semi-echelonized basis of V then `SemiEchelonBasis` returns this basis; if *vectors* do not form a basis of V then `fail` is returned.

`SemiEchelonBasisNC` does the same as `SemiEchelonBasis` for two arguments, except that it is not checked whether *vectors* form a semi-echelonized basis.

```
gap> V:= GF(2)^2;;
gap> B:= SemiEchelonBasis( V );
SemiEchelonBasis( ( GF(2)^2 ), ... )
gap> Print( BasisVectors( B ), "\n" );
[ [ Z(2)^0, 0*Z(2) ], [ 0*Z(2), Z(2)^0 ] ]
gap> B:= SemiEchelonBasis( V, [ [ 1, 1 ], [ 0, 1 ] ] * Z(2) );
SemiEchelonBasis( ( GF(2)^2 ), <an immutable 2x2 matrix over GF2> )
gap> Print( BasisVectors( B ), "\n" );
[ [ Z(2)^0, Z(2)^0 ], [ 0*Z(2), Z(2)^0 ] ]
gap> Coefficients( B, [ 0, 1 ] * Z(2) );
[ 0*Z(2), Z(2)^0 ]
gap> Coefficients( B, [ 1, 0 ] * Z(2) );
[ Z(2)^0, Z(2)^0 ]
gap> SemiEchelonBasis( V, [ [ 0, 1 ], [ 1, 1 ] ] * Z(2) );
fail
```

9 ▶ `IsCanonicalBasisFullRowModule(` *B* `)` P

`IsCanonicalBasisFullRowModule` returns `true` if B is the canonical basis (see 59.6.1) of a full row module (see 55.3.9), and `false` otherwise.

The **canonical basis** of a Gaussian row space is defined as the unique semi-echelonized (see 59.8.7) basis with the additional property that for $j > i$ the position of the pivot of row j is bigger than the position of the pivot of row i, and that each pivot column contains exactly one nonzero entry.

10 ▶ `IsCanonicalBasisFullMatrixModule(` *B* `)` P

`IsCanonicalBasisFullMatrixModule` returns `true` if B is the canonical basis (see 59.6.1) of a full matrix module (see 55.3.11), and `false` otherwise.

The **canonical basis** of a Gaussian matrix space is defined as the unique semi-echelonized (see 59.8.7) basis for which the list of concatenations of the basis vectors forms the canonical basis of the corresponding Gaussian row space.

11 ▶ `NormedRowVectors( V )` A

For a finite Gaussian row space V (see 59.8.1, 59.8.3), `NormedRowVectors` returns a list of those nonzero vectors in V that have a one in the first nonzero component.

The result list can be used as action domain for the action of a matrix group via `OnLines` (see 39.2.12), which yields the natural action on one-dimensional subspaces of V (see also 59.3.1).

```
gap> vecs:= NormedRowVectors( GF(3)^2 );
[ [ 0*Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3) ], [ Z(3)^0, Z(3)^0 ],
  [ Z(3)^0, Z(3) ] ]
gap> Action( GL(2,3), vecs, OnLines );
Group([ (3,4), (1,2,4) ])
```

12 ▶ `SiftedVector( B, v )` O

Let B be a semi-echelonized basis (see 59.8.7) of a Gaussian row or matrix space V (see 59.8.3), and v a row vector or matrix, respectively, of the same dimension as the elements in V. `SiftedVector` returns the **residuum** of v with respect to B, which is obtained by successively cleaning the pivot positions in v by subtracting multiples of the basis vectors in B. So the result is the zero vector in V if and only if v lies in V.

B may also be a mutable basis (see 59.7) of a Gaussian row or matrix space.

```
gap> V:= VectorSpace( Rationals, [ [ 1, 2, 7 ], [ 1/2, 1/3, 5 ] ] );;
gap> B:= Basis( V );;
gap> SiftedVector( B, [ 1, 2, 8 ] );
[ 0, 0, 1 ]
```

59.9 Vector Space Homomorphisms

Vector space homomorphisms (or **linear mappings**) are defined in Section 31.10. GAP provides special functions to construct a particular linear mapping from images of given elements in the source, from a matrix of coefficients, or as a natural epimorphism.

F-linear mappings with same source and same range can be added, so one can form vector spaces of linear mappings.

1 ▶ `LeftModuleGeneralMappingByImages( V, W, gens, imgs )` O

Let V and W be two left modules over the same left acting domain R, say, and *gens* and *imgs* lists (of the same length) of elements in V and W, respectively. `LeftModuleGeneralMappingByImages` returns the general mapping with source V and range W that is defined by mapping the elements in *gens* to the corresponding elements in *imgs*, and taking the R-linear closure.

gens need not generate V as a left R-module, and if the specification does not define a linear mapping then the result will be multi-valued; hence in general it is not a mapping (see 31.2.3).

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [1,2,3], [1,0,1] ] );;
gap> f:= LeftModuleGeneralMappingByImages( V, W,
>                               [[1,0],[2,0]], [[1,0,1],[1,0,1] ] );
[ [ 1, 0 ], [ 2, 0 ] ] -> [ [ 1, 0, 1 ], [ 1, 0, 1 ] ]
gap> IsMapping( f );
false
```

2 ▶ `LeftModuleHomomorphismByImages( V, W, gens, imgs )` F
▶ `LeftModuleHomomorphismByImagesNC( V, W, gens, imgs )` O

Let V and W be two left modules over the same left acting domain R, say, and *gens* and *imgs* lists (of the same length) of elements in V and W, respectively. `LeftModuleHomomorphismByImages` returns the left

R-module homomorphism with source V and range W that is defined by mapping the elements in *gens* to the corresponding elements in *imgs*.

If *gens* does not generate V or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then `fail` is returned. For creating a possibly multi-valued mapping from V to W that respects addition, multiplication, and scalar multiplication, `LeftModuleGeneralMappingByImages` can be used.

`LeftModuleHomomorphismByImagesNC` does the same as `LeftModuleHomomorphismByImages`, except that it omits all checks.

```
gap> V:=Rationals^2;;
gap> W:=VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> f:=LeftModuleHomomorphismByImages( V, W,
> [ [ 1, 0 ], [ 0, 1 ] ], [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );
[ [ 1, 0 ], [ 0, 1 ] ] -> [ [ 1, 0, 1 ], [ 1, 2, 3 ] ]
gap> Image( f, [1,1] );
[ 2, 2, 4 ]
```

3 ▶ `LeftModuleHomomorphismByMatrix(` *BS*, *matrix*, *BR* `)` O

Let *BS* and *BR* be bases of the left R-modules V and W, respectively. `LeftModuleHomomorphismByMatrix` returns the R-linear mapping from V to W that is defined by the matrix *matrix* as follows. The image of the i-th basis vector of *BS* is the linear combination of the basis vectors of *BR* with coefficients the i-th row of *matrix*.

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> f:= LeftModuleHomomorphismByMatrix( Basis( V ),
> [ [ 1, 2 ], [ 3, 1 ] ], Basis( W ) );
<linear mapping by matrix, ( Rationals^
2 ) -> <vector space over Rationals, with 2 generators>>
```

4 ▶ `NaturalHomomorphismBySubspace(` *V*, *W* `)` O

For an R-vector space V and a subspace W of V, `NaturalHomomorphismBySubspace` returns the R-linear mapping that is the natural projection of V onto the factor space *V* `/` *W*.

```
gap> V:= Rationals^3;;
gap> W:= VectorSpace( Rationals, [ [ 1, 1, 1 ] ] );;
gap> f:= NaturalHomomorphismBySubspace( V, W );
<linear mapping by matrix, ( Rationals^3 ) -> ( Rationals^2 )>
```

5 ▶ `Hom(` *F*, *V*, *W* `)` O

For a field F and two vector spaces V and W that can be regarded as F-modules (see 55.1.5), `Hom` returns the F-vector space of all F-linear mappings from V to W.

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> H:= Hom( Rationals, V, W );
Hom( Rationals, ( Rationals^2 ), <vector space over Rationals, with
2 generators> )
gap> Dimension( H );
4
```

6 ▶ `End(` *F*, *V* `)` O

For a field F and a vector space V that can be regarded as an F-module (see 55.1.5), `End` returns the F-algebra of all F-linear mappings from V to V.

```
gap> A:= End( Rationals, Rationals^2 );
End( Rationals, ( Rationals^2 ) )
gap> Dimension( A );
4
```

7 ▶ `IsFullHomModule( M )` P

A **full hom module** is a module of all R-linear mappings between two left R-modules. The function `Hom` (see 59.9.5) can be used to construct a full hom module.

```
gap> V:= Rationals^2;;
gap> W:= VectorSpace( Rationals, [ [ 1, 0, 1 ], [ 1, 2, 3 ] ] );;
gap> H:= Hom( Rationals, V, W );;
gap> IsFullHomModule( H );
true
```

8 ▶ `IsPseudoCanonicalBasisFullHomModule( B )` P

A basis of a full hom module is called pseudo canonical basis if the matrices of its basis vectors w.r.t. the stored bases of source and range contain exactly one identity entry and otherwise zeros.

Note that this is not a canonical basis (see 59.4.3) because it depends on the stored bases of source and range.

```
gap> IsPseudoCanonicalBasisFullHomModule( Basis( H ) );
true
```

9 ▶ `IsLinearMappingsModule( V )` F

If an F-vector space V is in the filter `IsLinearMappingsModule` then this expresses that V consists of linear mappings, and that V is handled via the mechanism of nice bases (see 59.10) in the following way. Let S and R be the source and the range, respectively, of each mapping in V. Then the `NiceFreeLeftModuleInfo` value of V is a record with the components `basissource` (a basis B_S of S) and `basisrange` (a basis B_R of R), and the `NiceVector` value of $v \in V$ is defined as the matrix of the F-linear mapping v w.r.t. the bases B_S and B_R.

59.10 Vector Spaces Handled By Nice Bases

There are kinds of free R-modules for which efficient computations are possible because the elements are "nice", for example subspaces of full row modules or of full matrix modules. In other cases, a "nice" canonical basis is known that allows one to do the necessary computations in the corresponding row module, for example algebras given by structure constants.

In many other situations, one knows at least an isomorphism from the given module V to a "nicer" free left module W, in the sense that for each vector in V, the image in W can easily be computed, and analogously for each vector in W, one can compute the preimage in V.

This allows one to delegate computations w.r.t. a basis B, say, of V to the corresponding basis C, say, of W. We call W the **nice free left module** of V, and C the **nice basis** of B. (Note that it may happen that also C delegates questions to a "nicer" basis.) The basis B indicates the intended behaviour by the filter `IsBasisByNiceBasis` (see 59.10.5), and stores C as value of the attribute `NiceBasis` (see 59.10.4). V indicates the intended behaviour by the filter `IsHandledByNiceBasis` (see 59.10.6), and stores W as value of the attribute `NiceFreeLeftModule` (see 59.10.1).

The bijection between V and W is implemented by the functions `NiceVector` (see 59.10.2) and `UglyVector` (see 59.10.2); additional data needed to compute images and preimages can be stored as value of `NiceFreeLeftModuleInfo` (see 59.10.3).

1 ▶ `NiceFreeLeftModule( V )` A

For a free left module V that is handled via the mechanism of nice bases, this attribute stores the associated free left module to which the tasks are delegated.

2 ▶ `NiceVector( V, v )` O
▶ `UglyVector( V, r )` O

`NiceVector` and `UglyVector` provide the linear bijection between the free left module V and W`:= NiceFreeLeftModule( V )`.

If v lies in the elements family of the family of V then `NiceVector( v )` is either `fail` or an element in the elements family of the family of W.

If r lies in the elements family of the family of W then `UglyVector( r )` is either `fail` or an element in the elements family of the family of V.

If v lies in V (which usually **cannot** be checked without using W) then `UglyVector( V, NiceVector( V, v ) ) = v`. If r lies in W (which usually **can** be checked) then `NiceVector( V, UglyVector( V, r ) ) = r`.

(This allows one to implement for example a membership test for V using the membership test in W.)

3 ▶ `NiceFreeLeftModuleInfo( V )` A

For a free left module V that is handled via the mechanism of nice bases, this operation has to provide the necessary information (if any) for calls of `NiceVector` and `UglyVector` (see 59.10.2).

4 ▶ `NiceBasis( B )` A

Let B be a basis of a free left module V that is handled via nice bases. If B has no basis vectors stored at the time of the first call to `NiceBasis` then `NiceBasis( B )` is obtained as `Basis( NiceFreeLeftModule( V ) )`. If basis vectors are stored then `NiceBasis( B )` is the result of the call of `Basis` with arguments `NiceFreeLeftModule( V )` and the `NiceVector` values of the basis vectors of B.

Note that the result is `fail` if and only if the "basis vectors" stored in B are in fact not basis vectors.

The attributes `GeneratorsOfLeftModule` of the underlying left modules of B and the result of `NiceBasis` correspond via `NiceVector` and `UglyVector`.

5 ▶ `IsBasisByNiceBasis( B )` C

This filter indicates that the basis B delegates tasks such as the computation of coefficients (see 59.5.3) to a basis of an isomorphisc "nicer" free left module.

6 ▶ `IsHandledByNiceBasis( M )` C

For a free left module M in this category, essentially all operations are performed using a "nicer" free left module, which is usually a row module.

59.11 How to Implement New Kinds of Vector Spaces

1 ▶ `DeclareHandlingByNiceBasis( name, info )` F
▶ `InstallHandlingByNiceBasis( name, record )` F

These functions are used to implement a new kind of free left modules that shall be handled via the mechanism of nice bases (see 59.10).

name must be a string, a filter f with this name is created, and a logical implication from f to `IsHandledByNiceBasis` (see 59.10.6) is installed.

record must be a record with the following components.

`detect`
: a function of four arguments R, l, V, and z, where V is a free left module over the ring R with generators the list or collection l, and z is either the zero element of V or `false` (then l is nonempty); the function returns `true` if V shall lie in the filter f, and `false` otherwise; the return value may also be `fail`, which indicates that V is **not** to be handled via the mechanism of nice bases at all,

`NiceFreeLeftModuleInfo`
: the `NiceFreeLeftModuleInfo` method for left modules in f,

`NiceVector`
: the `NiceVector` method for left modules V in f; called with V and a vector $v \in V$, this function returns the nice vector r associated with v, and

`UglyVector`
: the `UglyVector` method for left modules V in f; called with V and a vector r in the `NiceFreeLeftModule` value of V, this function returns the vector $v \in V$ to which r is associated.

The idea is that all one has to do for implementing a new kind of free left modules handled by the mechanism of nice bases is to call `DeclareHandlingByNiceBasis` and `InstallHandlingByNiceBasis`, which causes the installation of the necessary methods and adds the pair $[f, record$`.detect`$]$ to the global list `NiceBasisFiltersInfo`. The `LeftModuleByGenerators` methods call `CheckForHandlingByNiceBasis` (see 59.11.3), which sets the appropriate filter for the desired left module if applicable.

2 ▶ `NiceBasisFiltersInfo` V

An overview of all kinds of vector spaces that are currently handled by nice bases is given by the global list `NiceBasisFiltersInfo`. Examples of such vector spaces are vector spaces of field elements (but not the fields themselves) and non-Gaussian row and matrix spaces (see 59.8.3).

3 ▶ `CheckForHandlingByNiceBasis(` R`,` *gens*`,` M`,` *zero* `)` F

Whenever a free left module is constructed for which the filter `IsHandledByNiceBasis` may be useful, `CheckForHandlingByNiceBasis` should be called. (This is done in the methods for `VectorSpaceByGenerators`, `AlgebraByGenerators`, `IdealByGenerators` etc. in the GAP library.)

The arguments of this function are the coefficient ring R, the list *gens* of generators, the constructed module M itself, and the zero element *zero* of M; if *gens* is nonempty then the *zero* value may also be `false`.

60 Algebras

An algebra is a vector space equipped with a bilinear map (multiplication). This chapter describes the functions in GAP that deal with general algebras and associative algebras.

Algebras in GAP are vector spaces in a natural way. So all the functionality for vector spaces (see Chapter 59) is also applicable to algebras.

1 ► `InfoAlgebra` V

is the info class for the functions dealing with algebras (see 7.4).

60.1 Constructing Algebras by Generators

1 ► `Algebra(` *F*`,` *gens* `)` F
► `Algebra(` *F*`,` *gens*`,` *zero* `)` F
► `Algebra(` *F*`,` *gens*`, "basis" )` F
► `Algebra(` *F*`,` *gens*`,` *zero*`, "basis" )` F

`Algebra(` *F*`,` *gens* `)` is the algebra over the division ring *F*, generated by the vectors in the list *gens*.

If there are three arguments, a division ring *F* and a list *gens* and an element *zero*, then `Algebra(` *F*`,` *gens*`,` *zero* `)` is the *F*-algebra generated by *gens*, with zero element *zero*.

If the last argument is the string `"basis"` then the vectors in *gens* are known to form a basis of the algebra (as an *F*-vector space).

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= Algebra( Rationals, [ m ] );
<algebra over Rationals, with 1 generators>
gap> Dimension( A );
2
```

2 ► `AlgebraWithOne(` *F*`,` *gens* `)` F
► `AlgebraWithOne(` *F*`,` *gens*`,` *zero* `)` F
► `AlgebraWithOne(` *F*`,` *gens*`, "basis" )` F
► `AlgebraWithOne(` *F*`,` *gens*`,` *zero*`, "basis" )` F

`AlgebraWithOne(` *F*`,` *gens* `)` is the algebra-with-one over the division ring *F*, generated by the vectors in the list *gens*.

If there are three arguments, a division ring *F* and a list *gens* and an element *zero*, then `AlgebraWithOne(` *F*`,` *gens*`,` *zero* `)` is the *F*-algebra-with-one generated by *gens*, with zero element *zero*.

If the last argument is the string `"basis"` then the vectors in *gens* are known to form a basis of the algebra (as an *F*-vector space).

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> Dimension( A );
3
gap> One(A);
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
```

60.2 Constructing Algebras as Free Algebras

1 ▶ `FreeAlgebra(` *R*, *rank* `)` F
▶ `FreeAlgebra(` *R*, *rank*, *name* `)` F
▶ `FreeAlgebra(` *R*, *name1*, *name2*, ... `)` F

is a free (nonassociative) algebra of rank *rank* over the ring *R*. Here *name*, and *name1*, *name2*,... are optional strings that can be used to provide names for the generators.

```
gap> A:= FreeAlgebra( Rationals, "a", "b" );
<algebra over Rationals, with 2 generators>
gap> g:= GeneratorsOfAlgebra( A );
[ (1)*a, (1)*b ]
gap> (g[1]*g[2])*((g[2]*g[1])*g[1]);
(1)*((a*b)*((b*a)*a))
```

2 ▶ `FreeAlgebraWithOne(` *R*, *rank* `)` F
▶ `FreeAlgebraWithOne(` *R*, *rank*, *name* `)` F
▶ `FreeAlgebraWithOne(` *R*, *name1*, *name2*, ... `)` F

is a free (nonassociative) algebra-with-one of rank *rank* over the ring *R*. Here *name*, and *name1*, *name2*,... are optional strings that can be used to provide names for the generators.

```
gap> A:= FreeAlgebraWithOne( Rationals, 4, "q" );
<algebra-with-one over Rationals, with 4 generators>
gap> GeneratorsOfAlgebra( A );
[ (1)*<identity ...>, (1)*q.1, (1)*q.2, (1)*q.3, (1)*q.4 ]
gap> One( A );
(1)*<identity ...>
```

3 ▶ `FreeAssociativeAlgebra(` *R*, *rank* `)` F
▶ `FreeAssociativeAlgebra(` *R*, *rank*, *name* `)` F
▶ `FreeAssociativeAlgebra(` *R*, *name1*, *name2*, ... `)` F

is a free associative algebra of rank *rank* over the ring *R*. Here *name*, and *name1*, *name2*,... are optional strings that can be used to provide names for the generators.

```
gap> A:= FreeAssociativeAlgebra( GF( 5 ), 4, "a" );
<algebra over GF(5), with 4 generators>
```

4 ▶ `FreeAssociativeAlgebraWithOne(` *R*, *rank* `)` F
▶ `FreeAssociativeAlgebraWithOne(` *R*, *rank*, *name* `)` F
▶ `FreeAssociativeAlgebraWithOne(` *R*, *name1*, *name2*, ... `)` F

is a free associative algebra-with-one of rank *rank* over the ring *R*. Here *name*, and *name1*, *name2*,... are optional strings that can be used to provide names for the generators.

```
gap> A:= FreeAssociativeAlgebraWithOne( Rationals, "a", "b", "c" );
<algebra-with-one over Rationals, with 3 generators>
gap> GeneratorsOfAlgebra( A );
[ (1)*<identity ...>, (1)*a, (1)*b, (1)*c ]
gap> One( A );
(1)*<identity ...>
```

60.3 Constructing Algebras by Structure Constants

For an introduction into structure constants and how they are handled by GAP, we refer to Section 6.2 of the user's tutorial.

1 ▶ `EmptySCTable( `*dim*`, `*zero*` )` F
▶ `EmptySCTable( `*dim*`, `*zero*`, "symmetric" )` F
▶ `EmptySCTable( `*dim*`, `*zero*`, "antisymmetric" )` F

`EmptySCTable` returns a structure constants table for an algebra of dimension *dim*, describing trivial multiplication. *zero* must be the zero of the coefficients domain. If the multiplication is known to be (anti)commutative then this can be indicated by the optional third argument.

For filling up the structure constants table, see 60.3.2.

```
gap> EmptySCTable( 2, Zero( GF(5) ), "antisymmetric" );
[ [ [ [  ], [  ] ], [ [  ], [  ] ] ], [ [ [  ], [  ] ], [ [  ], [  ] ] ], -1,
  0*Z(5) ]
```

2 ▶ `SetEntrySCTable( `T`, `i`, `j`, `*list*` )` F

sets the entry of the structure constants table T that describes the product of the i-th basis element with the j-th basis element to the value given by the list *list*.

If T is known to be antisymmetric or symmetric then also the value T`[`j`][`i`]` is set.

list must be of the form $[c_{ij}^{k_1}, k_1, c_{ij}^{k_2}, k_2, \ldots]$.

The entries at the odd positions of *list* must be compatible with the zero element stored in T. For convenience, these entries may also be rational numbers that are automatically replaced by the corresponding elements in the appropriate prime field in finite characteristic if necessary.

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1/2, 1, 2/3, 2 ] );
gap> T;
[ [ [ [ 1, 2 ], [ 1/2, 2/3 ] ], [ [  ], [  ] ] ],
  [ [ [  ], [  ] ], [ [  ], [  ] ] ], 0, 0 ]
```

3 ▶ `GapInputSCTable( `T`, `*varname*` )` F

is a string that describes the structure constants table T in terms of `EmptySCTable` and `SetEntrySCTable`. The assignments are made to the variable *varname*.

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );
gap> SetEntrySCTable( T, 2, 1, [ 1, 2 ] );
gap> GapInputSCTable( T, "T" );
"T:= EmptySCTable( 2, 0 );\nSetEntrySCTable( T, 1, 2, [1,2] );\nSetEntrySCTabl\
e( T, 2, 1, [1,2] );\n"
```

4 ▶ `TestJacobi( `T` )` F

tests whether the structure constants table T satisfies the Jacobi identity $v_i*(v_j*v_k)+v_j*(v_k*v_i)+v_k*(v_i*v_j)=0$ for all basis vectors v_i of the underlying algebra, where $i \leq j \leq k$. (Thus antisymmetry is assumed.)

The function returns `true` if the Jacobi identity is satisfied, and a failing triple `[ i, j, k ]` otherwise.

```
gap> T:= EmptySCTable( 2, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );;
gap> TestJacobi( T );
true
```

5 ▶ `AlgebraByStructureConstants(` *R*, *sctable* `)` F
▶ `AlgebraByStructureConstants(` *R*, *sctable*, *name* `)` F
▶ `AlgebraByStructureConstants(` *R*, *sctable*, *names* `)` F
▶ `AlgebraByStructureConstants(` *R*, *sctable*, *name1*, *name2*, ... `)` F

returns a free left module A over the ring R, with multiplication defined by the structure constants table *sctable*. Here *name* and *name1*, *name2*, ... are optional strings that can be used to provide names for the elements of the canonical basis of A. *names* is a list of strings that can be entered instead of the specific names *name1*, *name2*, ... The vectors of the canonical basis of A correspond to the vectors of the basis given by *sctable*.

It is **not** checked whether the coefficients in *sctable* are really elements in R.

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1/2, 1, 2/3, 2 ] );
gap> A:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 2 over Rationals>
gap> b:= BasisVectors( Basis( A ) );;
gap> b[1]^2;
(1/2)*v.1+(2/3)*v.2
gap> b[1]*b[2];
0*v.1
```

6 ▶ `IdentityFromSCTable(` *T* `)` F

Let T be a structure constants table of an algebra A of dimension n. `IdentityFromSCTable(` *T* `)` is either `fail` or the vector of length n that contains the coefficients of the multiplicative identity of A with respect to the basis that belongs to T.

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1, 1 ] );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );;
gap> SetEntrySCTable( T, 2, 1, [ 1, 2 ] );;
gap> IdentityFromSCTable( T );
[ 1, 0 ]
```

7 ▶ `QuotientFromSCTable(` *T*, *num*, *den* `)` F

Let T be a structure constants table of an algebra A of dimension n. `QuotientFromSCTable(` *T* `)` is either `fail` or the vector of length n that contains the coefficients of the quotient of *num* and *den* with respect to the basis that belongs to T.

We solve the equation system $num = x\,den$. If no solution exists, `fail` is returned.

In terms of the basis B with vectors $b_1, \ldots, b_n$ this means for $num = \sum_{i=1}^{n} a_i b_i$, $den = \sum_{i=1}^{n} c_i b_i$, $x = \sum_{i=1}^{n} x_i b_i$ that $a_k = \sum_{i,j} c_i x_j c_{ijk}$ for all k. Here c_{ijk} denotes the structure constants with respect to B. This means that (as a vector) $a = xM$ with $M_{jk} = \sum_{i=1}^{n} c_{ijk} c_i$.

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1, 1 ] );;
gap> SetEntrySCTable( T, 2, 1, [ 1, 2 ] );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 2 ] );;
gap> QuotientFromSCTable( T, [0,1], [1,0] );
[ 0, 1 ]
```

60.4 Some Special Algebras

1 ▶ `QuaternionAlgebra( F )` F
▶ `QuaternionAlgebra( F, a, b )` F

is a quaternion algebra over the field F with parameters a and b in F, i.e., a four-dimensional associative F-algebra with basis (e, i, j, k) and multiplication defined by $ee = e$, $ei = ie = i$, $ej = je = j$, $ek = ke = k$, $ii = ae$, $ij = -ji = k$, $ik = -ki = aj$, $jj = be$, $jk = -kj = bi$, $kk = -abe$. The default value for both a and b is $-1 \in F$.

The `GeneratorsOfAlgebra` (see 60.8.1) and `CanonicalBasis` (see 59.4.3) value of an algebra constructed with `QuaternionAlgebra` is the list $[e, i, j, k]$.

The embedding of the field `GaussianRationals` into a quaternion algebra A over `Rationals` is not uniquely determined. One can specify one as a vector space homomorphism that maps `1` to the first algebra generator of A, and `E(4)` to one of the others.

```
gap> QuaternionAlgebra( Rationals );
<algebra-with-one of dimension 4 over Rationals>
```

2 ▶ `ComplexificationQuat( vector )` F
▶ `ComplexificationQuat( matrix )` F

Let $A = eF \oplus iF \oplus jF \oplus kF$ be a quaternion algebra over the field F of cyclotomics, with basis (e, i, j, k).

If $v = v_1 + v_2 j$ is a row vector over A with $v_1 = ew_1 + iw_2$ and $v_2 = ew_3 + iw_4$ then `ComplexificationQuat( v )` is the concatenation of $w_1 + \mathtt{E(4)}w_2$ and $w_3 + \mathtt{E(4)}w_4$.

If $M = M_1 + M_2 j$ is a matrix over A with $M_1 = eN_1 + iN_2$ and $M_2 = eN_3 + iN_4$ then `ComplexificationQuat( M )` is the block matrix A over $eF \oplus iF$ such that $A(1,1) = N_1 + \mathtt{E(4)}N_2$, $A(2,2) = N_1 - \mathtt{E(4)}N_2$, $A(1,2) = N_3 + \mathtt{E(4)}N_4$ and $A(2,1) = -N_3 + \mathtt{E(4)}N_4$.

Then `ComplexificationQuat(`v`)*ComplexificationQuat(`M`)= ComplexificationQuat(`v`*`M`)`, since

$$vM = v_1 M_1 + v_2 j M_1 + v_1 M_2 j + v_2 j M_2 j = (v_1 M_1 - v_2 \overline{M_2}) + (v_1 M_2 + v_2 \overline{M_1}) j.$$

3 ▶ `OctaveAlgebra( F )` F

The algebra of octonions over F.

```
gap> OctaveAlgebra( Rationals );
<algebra of dimension 8 over Rationals>
```

4 ▶ `FullMatrixAlgebra( R, n )` F
▶ `MatrixAlgebra( R, n )` F
▶ `MatAlgebra( R, n )` F

is the full matrix algebra $R^{n \times n}$, for a ring R and a nonnegative integer n.

```
gap> A:=FullMatrixAlgebra( Rationals, 20 );
( Rationals^[ 20, 20 ] )
gap> Dimension( A );
400
```

5 ► `NullAlgebra( R )` A

The zero-dimensional algebra over R.

```
gap> A:= NullAlgebra( Rationals );
<algebra over Rationals>
gap> Dimension( A );
0
```

60.5 Subalgebras

1 ► `Subalgebra( A, gens )` F
► `Subalgebra( A, gens, "basis" )` F

is the F-algebra generated by *gens*, with parent algebra A, where F is the left acting domain of A.

Note that being a subalgebra of A means to be an algebra, to be contained in A, **and** to have the same left acting domain as A.

An optional argument `"basis"` may be added if it is known that the generators already form a basis of the algebra. Then it is **not** checked whether *gens* really are linearly independent and whether all elements in *gens* lie in A.

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= Algebra( Rationals, [ m ] );
<algebra over Rationals, with 1 generators>
gap> B:= Subalgebra( A, [ m^2 ] );
<algebra over Rationals, with 1 generators>
```

2 ► `SubalgebraNC( A, gens )` F
► `SubalgebraNC( A, gens, "basis" )` F

`SubalgebraNC` constructs the subalgebra generated by *gens*, only it does not check whether all elements in *gens* lie in A.

```
gap> m:= RandomMat( 3, 3 );;
gap> A:= Algebra( Rationals, [ m ] );
<algebra over Rationals, with 1 generators>
gap> SubalgebraNC( A, [ IdentityMat( 3, 3 ) ], "basis" );
<algebra of dimension 1 over Rationals>
```

3 ► `SubalgebraWithOne( A, gens )` F
► `SubalgebraWithOne( A, gens, "basis" )` F

is the algebra-with-one generated by *gens*, with parent algebra A.

The optional third argument `"basis"` may be added if it is known that the elements from *gens* are linearly independent. Then it is **not** checked whether *gens* really are linearly independent and whether all elements in *gens* lie in A.

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> B1:= SubalgebraWithOne( A, [ m ] );;
gap> B2:= Subalgebra( A, [ m ] );;
gap> Dimension( B1 );
3
gap> Dimension( B2 );
2
```

4 ▶ `SubalgebraWithOneNC( `*A*`, `*gens*` )` F
▶ `SubalgebraWithOneNC( `*A*`, `*gens*`, "basis" )` F

`SubalgebraWithOneNC` does not check whether all elements in *gens* lie in *A*.

```
gap> m:= RandomMat( 3, 3 );; A:= Algebra( Rationals, [ m ] );;
gap> SubalgebraWithOneNC( A, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
```

5 ▶ `TrivialSubalgebra( `*A*` )` A

The zero dimensional subalgebra of the algebra *A*.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= TrivialSubalgebra( A );
<algebra over Rationals>
gap> Dimension( B );
0
```

60.6 Ideals

For constructing and working with ideals in algebras the same functions are available as for ideals in rings. So for the precise description of these functions we refer to Chapter 54. Here we give examples demonstrating the use of ideals in algebras. For an introduction into the concstruction of quotient algebras we refer to Chapter 6.2 of the user's tutorial.

```
gap> m:= [ [ 0, 2, 3 ], [ 0, 0, 4 ], [ 0, 0, 0] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );;
gap> I:= Ideal( A, [ m ] );  # i.e., the two-sided ideal of ‘A’ generated by ‘m’.
<two-sided ideal in <algebra-with-one of dimension 3 over Rationals>,
  (1 generators)>
gap> Dimension( I );
2
gap> GeneratorsOfIdeal( I );
[ [ [ 0, 2, 3 ], [ 0, 0, 4 ], [ 0, 0, 0 ] ] ]
gap> BasisVectors( Basis( I ) );
[ [ [ 0, 1, 3/2 ], [ 0, 0, 2 ], [ 0, 0, 0 ] ],
  [ [ 0, 0, 1 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ] ]

gap> A:= FullMatrixAlgebra( Rationals, 4 );;
gap> m:= NullMat( 4, 4 );; m[1][4]:=1;;
gap> I:= LeftIdeal( A, [ m ] );
<left ideal in ( Rationals^[ 4, 4 ] ), (1 generators)>
gap> Dimension( I );
4
```

```
gap> GeneratorsOfLeftIdeal( I );
[ [ [ 0, 0, 0, 1 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ], [ 0, 0, 0, 0 ] ] ]

gap> mats:= [ [[1,0],[0,0]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> A:= Algebra( Rationals, mats );;
gap> # Form the two-sided ideal for which 'mats[2]' is known to be
gap> # the unique basis element.
gap> I:= Ideal( A, [ mats[2] ], "basis" );
<two-sided ideal in <algebra of dimension 3 over Rationals>, (dimension 1)>
```

60.7 Categories and Properties of Algebras

1 ▶ `IsFLMLOR( obj )` C

A FLMLOR ("free left module left operator ring") in GAP is a ring that is also a free left module.

Note that this means that being a FLMLOR is not a property a ring can get, since a ring is usually not represented as an external left set.

Examples are magma rings (e.g. over the integers) or algebras.

```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> IsFLMLOR ( A );
true
```

2 ▶ `IsFLMLORWithOne( obj )` C

A FLMLOR-with-one in GAP is a ring-with-one that is also a free left module.

Note that this means that being a FLMLOR-with-one is not a property a ring-with-one can get, since a ring-with-one is usually not represented as an external left set.

Examples are magma rings-with-one or algebras-with-one (but also over the integers).

```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> IsFLMLORWithOne ( A );
true
```

3 ▶ `IsAlgebra( obj )` C

An algebra in GAP is a ring that is also a left vector space. Note that this means that being an algebra is not a property a ring can get, since a ring is usually not represented as an external left set.

```
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsAlgebra( A );
true
```

4 ▶ `IsAlgebraWithOne( obj )` C

An algebra-with-one in GAP is a ring-with-one that is also a left vector space. Note that this means that being an algebra-with-one is not a property a ring-with-one can get, since a ring-with-one is usually not represented as an external left set.

```
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsAlgebraWithOne( A );
true
```

5 ▶ `IsLieAlgebra( A )` P

An algebra A is called Lie algebra if $a * a = 0$ for all a in A and $(a * (b * c)) + (b * (c * a)) + (c * (a * b)) = 0$ for all a, b, c in A (Jacobi identity).

```
gap> A:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> IsLieAlgebra( A );
true
```

6 ▶ `IsSimpleAlgebra( A )` P

is `true` if the algebra A is simple, and `false` otherwise. This function is only implemented for the cases where A is an associative or a Lie algebra. And for Lie algebras it is only implemented fot the case where the ground field is of characteristic 0.

```
gap> A:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> IsSimpleAlgebra( A );
false
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsSimpleAlgebra( A );
true
```

7 ▶ `IsFiniteDimensional(`*matalg*`)` O

returns `true` (always) for a matrix algebra *matalg*, since matrix algebras are always finite dimensional.

```
gap> A:= MatAlgebra( Rationals, 3 );;
gap> IsFiniteDimensional( A );
true
```

8 ▶ `IsQuaternion( obj )` C
▶ `IsQuaternionCollection( obj )` C
▶ `IsQuaternionCollColl( obj )` C

`IsQuaternion` is the category of elements in an algebra constructed by `QuaternionAlgebra`. A collection of quaternions lies in the category `IsQuaternionCollection`. Finally, a collection of quaternion collections (e.g., a matrix of quaternions) lies in the category `IsQuaternionCollColl`.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> b:= BasisVectors( Basis( A ) );
[ e, i, j, k ]
gap> IsQuaternion( b[1] );
true
gap> IsQuaternionCollColl( [ [ b[1], b[2] ], [ b[3], b[4] ] ] );
true
```

60.8 Attributes and Operations for Algebras

1 ▶ `GeneratorsOfAlgebra( A )` A

returns a list of elements that generate A as an algebra.

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> GeneratorsOfAlgebra( A );
[ [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
  [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ] ]
```

2 ▶ `GeneratorsOfAlgebraWithOne( A )` A

returns a list of elements of A that generate A as an algebra with one.

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOne( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> GeneratorsOfAlgebraWithOne( A );
[ [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ] ]
```

3 ► `ProductSpace( U, V )` O

is the vector space $\langle u * v; u \in U, v \in V \rangle$, where U and V are subspaces of the same algebra.

If $U = V$ is known to be an algebra then the product space is also an algebra, moreover it is an ideal in U. If U and V are known to be ideals in an algebra A then the product space is known to be an algebra and an ideal in A.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> b:= BasisVectors( Basis( A ) );;
gap> B:= Subalgebra( A, [ b[4] ] );
<algebra over Rationals, with 1 generators>
gap> ProductSpace( A, B );
<vector space of dimension 4 over Rationals>
```

4 ► `PowerSubalgebraSeries( A )` A

returns a list of subalgebras of A, the first term of which is A; and every next term is the product space of the previous term with itself.

```
gap> A:= QuaternionAlgebra( Rationals );
<algebra-with-one of dimension 4 over Rationals>
gap> PowerSubalgebraSeries( A );
[ <algebra-with-one of dimension 4 over Rationals> ]
```

5 ► `AdjointBasis( B )` A

Let x be an element of an algebra A. Then the adjoint map of x is the left multiplication by x. It is a linear map of A. For the basis B of an algebra A, this function returns a particular basis C of the matrix space generated by adA, (the matrix spaces spanned by the matrices of the left multiplication); namely a basis consisting of elements of the form adx_i, where x_i is a basis element of B.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> AdjointBasis( Basis( A ) );
Basis( <vector space over Rationals, with 4 generators>,
[ [ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 0, 0, 1 ] ],
  [ [ 0, -1, 0, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, -1 ], [ 0, 0, 1, 0 ] ],
  [ [ 0, 0, -1, 0 ], [ 0, 0, 0, 1 ], [ 1, 0, 0, 0 ], [ 0, -1, 0, 0 ] ],
  [ [ 0, 0, 0, -1 ], [ 0, 0, -1, 0 ], [ 0, 1, 0, 0 ], [ 1, 0, 0, 0 ] ] ] )
```

6 ► `IndicesOfAdjointBasis( B )` A

Let A be an algebra and let B be the basis that is output by `AdjointBasis( Basis( A ) )`. This function returns a list of indices. If i is an index belonging to this list, then adx_i is a basis vector of the matrix space spanned by adA, where x_i is the i-th basis vector of the basis B.

```
gap> L:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> B:= AdjointBasis( Basis( L ) );;
gap> IndicesOfAdjointBasis( B );
[ 1, 2, 3, 4, 5, 6, 7, 8 ]
```

7 ► `AsAlgebra( F, A )` O

Returns the algebra over F generated by A.

```
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> AsAlgebra( Rationals, V );
<algebra of dimension 1 over Rationals>
```

8 ▶ `AsAlgebraWithOne( F, A )` O

If the algebra A has an identity, then it can be viewed as an algebra with one over F. This function returns this algebra with one.

```
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> A:= AsAlgebra( Rationals, V );;
gap> AsAlgebraWithOne( Rationals, A );
<algebra-with-one over Rationals, with 1 generators>
```

9 ▶ `AsSubalgebra( A, B )` O

If all elements of the algebra B happen to be contained in the algebra A, then B can be viewed as a subalgebra of A. This function returns this subalgebra.

```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> B:= AsAlgebra( Rationals, V );;
gap> BA:= AsSubalgebra( A, B );
<algebra of dimension 1 over Rationals>
```

10 ▶ `AsSubalgebraWithOne( A, B )` O

If B is an algebra with one, all elements of which happen to be contained in the algebra with one A, then B can be viewed as a subalgebra with one of A. This function returns this subalgebra with one.

```
gap> A:= FullMatrixAlgebra( Rationals, 2 );;
gap> V:= VectorSpace( Rationals, [ IdentityMat( 2 ) ] );;
gap> B:= AsAlgebra( Rationals, V );;
gap> C:= AsAlgebraWithOne( Rationals, B );;
gap> AC:= AsSubalgebraWithOne( A, C );
<algebra-with-one over Rationals, with 1 generators>
```

11 ▶ `MutableBasisOfClosureUnderAction( F, Agens, from, init, opr, zero, maxdim )` F

Let F be a ring, *Agens* a list of generators for an F-algebra A, and *from* one of `"left"`, `"right"`, `"both"`; (this means that elements of A act via multiplication from the respective side(s).) *init* must be a list of initial generating vectors, and *opr* the operation (a function of two arguments).

`MutableBasisOfClosureUnderAction` returns a mutable basis of the F-free left module generated by the vectors in *init* and their images under the action of *Agens* from the respective side(s).

zero is the zero element of the desired module. *maxdim* is an upper bound for the dimension of the closure; if no such upper bound is known then the value of *maxdim* must be `infinity`.

`MutableBasisOfClosureUnderAction` can be used to compute a basis of an **associative** algebra generated by the elements in *Agens*. In this case *from* may be `"left"` or `"right"`, *opr* is the multiplication `*`, and *init* is a list containing either the identity of the algebra or a list of algebra generators. (Note that if the algebra has an identity then it is in general not sufficient to take algebra-with-one generators as *init*, whereas of course *Agens* need not contain the identity.)

(Note that bases of **not** necessarily associative algebras can be computed using `MutableBasisOfNonassociativeAlgebra`.)

Other applications of `MutableBasisOfClosureUnderAction` are the computations of bases for (left/ right/ two-sided) ideals I in an **associative** algebra A from ideal generators of I; in these cases *Agens* is a list of

algebra generators of A, *from* denotes the appropriate side(s), *init* is a list of ideal generators of I, and *opr* is again `*`.

(Note that bases of ideals in **not** necessarily associative algebras can be computed using `MutableBasisOfIdealInNonassociativeAlgebra`.)

Finally, bases of right A-modules also can be computed using `MutableBasisOfClosureUnderAction`. The only difference to the ideal case is that *init* is now a list of right module generators, and *opr* is the operation of the module.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> g:= GeneratorsOfAlgebra( A );;
gap> B:= MutableBasisOfClosureUnderAction( Rationals, g, "left", [ g[1] ], \*, Zero(A), 4 );
<mutable basis over Rationals, 4 vectors>
gap> BasisVectors( B );
[ e, i, j, k ]
```

12 ▶ `MutableBasisOfNonassociativeAlgebra(` *F*, *Agens*, *zero*, *maxdim* `)` F

is a mutable basis of the (not necessarily associative) F-algebra that is generated by *Agens*, has zero element *zero*, and has dimension at most *maxdim*. If no finite bound for the dimension is known then `infinity` must be the value of *maxdim*.

The difference to `MutableBasisOfClosureUnderAction` is that in general it is not sufficient to multiply just with algebra generators. (For special cases of nonassociative algebras, especially for Lie algebras, multiplying with algebra generators suffices.)

```
gap> L:= FullMatrixLieAlgebra( Rationals, 4 );;
gap> m1:= Random( L );;
gap> m2:= Random( L );;
gap> MutableBasisOfNonassociativeAlgebra( Rationals, [ m1, m2 ], Zero( L ),
> 16 );
<mutable basis over Rationals, 16 vectors>
```

13 ▶ `MutableBasisOfIdealInNonassociativeAlgebra(` *F*, *Vgens*, *Igens*, *zero*, *from*, *maxdim* `)` F

is a mutable basis of the ideal generated by *Igens* under the action of the (not necessarily associative) F-algebra with vector space generators *Vgens*. The zero element of the ideal is *zero*, *from* is one of `"left"`, `"right"`, `"both"` (with the same meaning as in `MutableBasisOfClosureUnderAction`), and *maxdim* is a known upper bound on the dimension of the ideal; if no finite bound for the dimension is known then `infinity` must be the value of *maxdim*.

The difference to `MutableBasisOfClosureUnderAction` is that in general it is not sufficient to multiply just with algebra generators. (For special cases of nonassociative algebras, especially for Lie algebras, multiplying with algebra generators suffices.)

```
gap> mats:= [  [[ 1, 0 ], [ 0, -1 ]], [[0,1],[0,0]] ];;
gap> A:= Algebra( Rationals, mats );;
gap> basA:= BasisVectors( Basis( A ) );;
gap> B:= MutableBasisOfIdealInNonassociativeAlgebra( Rationals, basA,
> [ mats[2] ], 0*mats[1], "both", infinity );
<mutable basis over Rationals, 1 vectors>
gap> BasisVectors( B );
[ [ [ 0, 1 ], [ 0, 0 ] ] ]
```

14 ▶ `DirectSumOfAlgebras(` *A1*, *A2* `)` O
▶ `DirectSumOfAlgebras(` *list* `)` O

is the direct sum of the two algebras *A1* and *A2* respectively of the algebras in the list *list*.

If all involved algebras are associative algebras then the result is also known to be associative. If all involved algebras are Lie algebras then the result is also known to be a Lie algebra.

All involved algebras must have the same left acting domain.

The default case is that the result is a structure constants algebra. If all involved algebras are matrix algebras, and either both are Lie algebras or both are associative then the result is again a matrix algebra of the appropriate type.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> DirectSumOfAlgebras( [A, A, A] );
<algebra of dimension 12 over Rationals>
```

15 ▶ `FullMatrixAlgebraCentralizer( `*F*`, `*lst*` )` F

Compute the centralizer of the list of matrices in the list *lst* in the full matrix algebra over the ring *F*.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> mats:= List(BasisVectors(Basis( A ) ), x -> AdjointMatrix(Basis(A), x ));;
gap> FullMatrixAlgebraCentralizer( Rationals, mats );
<algebra-with-one of dimension 4 over Rationals>
```

16 ▶ `RadicalOfAlgebra( `*A*` )` A

is the maximal nilpotent ideal of *A*, where *A* is an associative algebra.

```
gap> m:= [ [ 0, 1, 2 ], [ 0, 0, 3 ], [ 0, 0, 0 ] ];;
gap> A:= AlgebraWithOneByGenerators( Rationals, [ m ] );
<algebra-with-one over Rationals, with 1 generators>
gap> RadicalOfAlgebra( A );
<algebra of dimension 2 over Rationals>
```

17 ▶ `CentralIdempotentsOfAlgebra( `*A*` )` A

For an associative algebra *A*, this function returns a list of central primitive idempotents such that their sum is the identity element of *A*. Therefore *A* is required to have an identity.

(This is a synonym of `CentralIdempotentsOfSemiring`.)

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= DirectSumOfAlgebras( [A, A, A] );
<algebra of dimension 12 over Rationals>
gap> CentralIdempotentsOfAlgebra( B );
[ v.9, v.5, v.1 ]
```

18 ▶ `DirectSumDecomposition( `*L*` )` A

This function calculates a list of ideals of the algebra *L* such that *L* is equal to their direct sum. Currently this is only implemented for semisimple associative algebras, and Lie algebras (semisimple or not).

```
gap> G:= SymmetricGroup( 4 );;
gap> A:= GroupRing( Rationals, G );
<algebra-with-one over Rationals, with 2 generators>
gap> dd:= DirectSumDecomposition( A );
[ <two-sided ideal in <algebra-with-one of dimension 24 over Rationals>,
      (1 generators)>,
  <two-sided ideal in <algebra-with-one of dimension 24 over Rationals>,
      (1 generators)>,
  <two-sided ideal in <algebra-with-one of dimension 24 over Rationals>,
      (1 generators)>,
```

```
    <two-sided ideal in <algebra-with-one of dimension 24 over Rationals>,
        (1 generators)>,
    <two-sided ideal in <algebra-with-one of dimension 24 over Rationals>,
        (1 generators)> ]
gap> List( dd, Dimension );
[ 1, 1, 4, 9, 9 ]
```

19 ▶ `LeviMalcevDecomposition( L )` A

A Levi-Malcev subalgebra of the algebra L is a semisimple subalgebra complementary to the radical of L. This function returns a list with two components. The first component is a Levi-Malcev subalgebra, the second the radical. This function is implemented for associative and Lie algebras.

```
gap> m:= [ [ 1, 2, 0 ], [ 0, 1, 3 ], [ 0, 0, 1] ];;
gap> A:= Algebra( Rationals, [ m ] );;
gap> LeviMalcevDecomposition( A );
[ <algebra of dimension 1 over Rationals>,
  <algebra of dimension 2 over Rationals> ]
```

20 ▶ `Grading( A )` A

Let G be an Abelian group and A an algebra. Then A is said to be graded over G if for every $g \in G$ there is a subspace A_g of A such that $A_g \cdot A_h \subset A_{g+h}$ for $g, h \in G$. In GAP 4 a **grading** of an algebra is a record containing the following components:

`source`
: the Abelian group over which the algebra is graded. `hom_components`
: a function assigning to each element from the source a subspace of the algebra. `min_degree`
: in the case where the algebra is graded over the integers this is the minimum number for which `hom_components` returns a nonzero subspace. `max_degree`
: is analogous to `min_degree`.

We note that there are no methods to compute a grading of an arbitrary algebra; however some algebras get a natural grading when they are constructed (see 61.8.4, 61.11.2).

We note also that these components may be not enough to handle the grading efficiently, and another record component may be needed. For instance in a Lie algebra L constructed by `JenningsLieAlgebra`, the length of the of the range `[ Grading(L)!.min_degree .. Grading(L)!.max_degree ]` may be non-polynomial in the dimension of L. To handle efficiently this situation, an optional component can be used:

`non_zero_hom_components`
: the subset of `source` for which `hom_components` returns a nonzero subspace.

```
gap> G:= SmallGroup(3^6, 100 );
<pc group of size 729 with 6 generators>
gap> L:= JenningsLieAlgebra( G );
<Lie algebra of dimension 6 over GF(3)>
gap> g:= Grading( L );
rec( min_degree := 1, max_degree := 9, source := Integers,
  hom_components := function( d ) ... end )
gap> g.hom_components( 3 );
<vector space over GF(3), with 1 generators>
gap> g.hom_components( 14 );
<vector space over GF(3), with 0 generators>
```

60.9 Homomorphisms of Algebras

Algebra homomorphisms are vector space homomorphisms that preserve the multiplication. So the default methods for vector space homomorphisms work, and in fact there is not much use of the fact that source and range are algebras, except that preimages and images are algebras (or even ideals) in certain cases.

1 ▶ `AlgebraGeneralMappingByImages( A, B, gens, imgs )` O

is a general mapping from the F-algebra A to the F-algebra B. This general mapping is defined by mapping the entries in the list *gens* (elements of A) to the entries in the list *imgs* (elements of B), and taking the F-linear and multiplicative closure.

gens need not generate A as an F-algebra, and if the specification does not define a linear and multiplicative mapping then the result will be multivalued. Hence, in general it is not a mapping. For constructing a linear map that is not necessarily multiplicative, we refer to `LeftModuleHomomorphismByImages` (59.9.2).

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= FullMatrixAlgebra( Rationals, 2 );;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraGeneralMappingByImages( A, B, bA, bB );
[ e, i, j, k ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 1 ], [ 0, 0 ] ],
  [ [ 0, 0 ], [ 1, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ]
gap> Images( f, bA[1] );
<add. coset of <algebra over Rationals, with 60 generators>>
```

2 ▶ `AlgebraHomomorphismByImages( A, B, gens, imgs )` F

`AlgebraHomomorphismByImages` returns the algebra homomorphism with source A and range B that is defined by mapping the list *gens* of generators of A to the list *imgs* of images in B.

If *gens* does not generate A or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then `fail` is returned.

One can avoid the checks by calling `AlgebraHomomorphismByImagesNC`, and one can construct multi-valued mappings with `AlgebraGeneralMappingByImages`.

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] ); SetEntrySCTable( T, 2, 2, [1,2] );
gap> A:= AlgebraByStructureConstants( Rationals, T );;
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> B:= AlgebraByGenerators( Rationals, [ m1, m2 ] );;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraHomomorphismByImages( A, B, bA, bB );
[ v.1, v.2 ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ]
gap> Image( f, bA[1]+bA[2] );
[ [ 1, 0 ], [ 0, 1 ] ]
```

3 ▶ `AlgebraHomomorphismByImagesNC( A, B, gens, imgs )` O

`AlgebraHomomorphismByImagesNC` is the operation that is called by the function `AlgebraHomomorphism-ByImages`. Its methods may assume that *gens* generates A and that the mapping of *gens* to *imgs* defines an algebra homomorphism. Results are unpredictable if these conditions do not hold.

For creating a possibly multi-valued mapping from A to B that respects addition, multiplication, and scalar multiplication, `AlgebraGeneralMappingByImages` can be used.

For the definitions of the algebras `A` and `B` in the next example we refer to the previous example.

```
gap> f:= AlgebraHomomorphismByImagesNC( A, B, bA, bB );
[ v.1, v.2 ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ]
```

4 ▶ `AlgebraWithOneGeneralMappingByImages( `*A*`, `*B*`, `*gens*`, `*imgs*` )` O

This function is analogous to 60.9.1; the only difference being that the identity of A is automatically mapped to the identity of B.

```
gap> A:= QuaternionAlgebra( Rationals );;
gap> B:= FullMatrixAlgebra( Rationals, 2 );;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraWithOneGeneralMappingByImages(A,B,bA{[2,3,4]},bB{[1,2,3]});
[ i, j, k, e ] -> [ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 1 ], [ 0, 0 ] ],
  [ [ 0, 0 ], [ 1, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ]
```

5 ▶ `AlgebraWithOneHomomorphismByImages( `*A*`, `*B*`, `*gens*`, `*imgs*` )` F

`AlgebraWithOneHomomorphismByImages` returns the algebra-with-one homomorphism with source A and range B that is defined by mapping the list *gens* of generators of A to the list *imgs* of images in B.

The difference between an algebra homomorphism and an algebra-with-one homomorphism is that in the latter case, it is assumed that the identity of A is mapped to the identity of B, and therefore *gens* needs to generate A only as an algebra-with-one.

If *gens* does not generate A or if the homomorphism does not exist (i.e., if mapping the generators describes only a multi-valued mapping) then `fail` is returned.

One can avoid the checks by calling `AlgebraWithOneHomomorphismByImagesNC`, and one can construct multi-valued mappings with `AlgebraWithOneGeneralMappingByImages`.

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:=1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:=1;;
gap> A:= AlgebraByGenerators( Rationals, [m1,m2] );;
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] );
gap> SetEntrySCTable( T, 2, 2, [1,2] );
gap> B:= AlgebraByStructureConstants(Rationals, T);;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
gap> f:= AlgebraWithOneHomomorphismByImages( A, B, bA{[1]}, bB{[1]} );
[ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ] -> [ v.1, v.1+v.2 ]
```

6 ▶ `AlgebraWithOneHomomorphismByImagesNC( `*A*`, `*B*`, `*gens*`, `*imgs*` )` O

`AlgebraWithOneHomomorphismByImagesNC` is the operation that is called by the function `AlgebraWithOneHomomorphismByImages`. Its methods may assume that *gens* generates A and that the mapping of *gens* to *imgs* defines an algebra-with-one homomorphism. Results are unpredictable if these conditions do not hold.

For creating a possibly multi-valued mapping from A to B that respects addition, multiplication, identity, and scalar multiplication, `AlgebraWithOneGeneralMappingByImages` can be used.

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:=1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:=1;;
gap> A:= AlgebraByGenerators( Rationals, [m1,m2] );;
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] );
gap> SetEntrySCTable( T, 2, 2, [1,2] );
gap> B:= AlgebraByStructureConstants( Rationals, T);;
gap> bA:= BasisVectors( Basis( A ) );; bB:= BasisVectors( Basis( B ) );;
```

```
gap> f:= AlgebraWithOneHomomorphismByImagesNC( A, B, bA{[1]}, bB{[1]} );
[ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 1, 0 ], [ 0, 1 ] ] ] -> [ v.1, v.1+v.2 ]
```

7 ► `NaturalHomomorphismByIdeal( `*A*`, `*I*` )` O

is the homomorphism of algebras provided by the natural projection map of A onto the quotient algebra A/I. This map can be used to take pre-images in the original algebra from elements in the quotient.

```
gap> L:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> C:= LieCentre( L );
<two-sided ideal in <Lie algebra of dimension 9 over Rationals>, (dimension 1
 )>
gap> hom:= NaturalHomomorphismByIdeal( L, C );
<linear mapping by matrix, <Lie algebra of dimension
9 over Rationals> -> <Lie algebra of dimension 8 over Rationals>>
gap> ImagesSource( hom );
<Lie algebra of dimension 8 over Rationals>
```

8 ► `OperationAlgebraHomomorphism( `*A*`, `*B*`[, `*opr*`] )` O
► `OperationAlgebraHomomorphism( `*A*`, `*V*`[, `*opr*`] )` O

`OperationAlgebraHomomorphism` returns an algebra homomorphism from the F-algebra A into a matrix algebra over F that describes the F-linear action of A on the basis B of a free left module respectively on the free left module V (in which case some basis of V is chosen), via the operation *opr*.

The homomorphism need not be surjective. The default value for *opr* is `OnRight`.

If A is an algebra-with-one then the operation homomorphism is an algebra-with-one homomorphism because the identity of A must act as the identity.

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> B:= AlgebraByGenerators( Rationals, [ m1, m2 ] );;
gap> V:= FullRowSpace( Rationals, 2 );
( Rationals^2 )
gap> f:=OperationAlgebraHomomorphism( B, Basis( V ), OnRight );
<op. hom. Algebra( Rationals,
[ [ [ 1, 0 ], [ 0, 0 ] ], [ [ 0, 0 ], [ 0, 1 ] ] ] ) -> matrices of dim. 2>
gap> Image( f, m1 );
[ [ 1, 0 ], [ 0, 0 ] ]
```

9 ► `IsomorphismFpAlgebra( `*A*` )` A

isomorphism from the algebra A onto a finitely presented algebra. Currently this is only implemented for associative algebras with one.

```
gap> A:= QuaternionAlgebra( Rationals );
<algebra-with-one of dimension 4 over Rationals>
gap> f:= IsomorphismFpAlgebra( A );
[ e, i, j, k, e ] -> [ [(1)*x.1], [(1)*x.2], [(1)*x.3], [(1)*x.4],
  [(1)*<identity ...>] ]
```

10 ► `IsomorphismMatrixAlgebra( `*A*` )` A

isomorphism from the algebra A onto a matrix algebra. Currently this is only implemented for associative algebras with one.

```
gap> T:= EmptySCTable( 2, 0 );;
gap> SetEntrySCTable( T, 1, 1, [1,1] ); SetEntrySCTable( T, 2, 2, [1,2] );
gap> A:= AlgebraByStructureConstants( Rationals, T );;
gap> A:= AsAlgebraWithOne( Rationals, A );;
gap> f:=IsomorphismMatrixAlgebra( A );
<op. hom. AlgebraWithOne( Rationals, ... ) -> matrices of dim. 2>
gap> Image( f, BasisVectors( Basis( A ) )[1] );
[ [ 1, 0 ], [ 0, 0 ] ]
```

11 ▶ `IsomorphismSCAlgebra( B )` A
▶ `IsomorphismSCAlgebra( A )` A

For a basis B of an algebra A, say, `IsomorphismSCAlgebra` returns an algebra isomorphism from A to an algebra S given by structure constants (see 60.3), such that the canonical basis of S is the image of B.

For an algebra A, `IsomorphismSCAlgebra` chooses a basis of A and returns the `IsomorphismSCAlgebra` value for that basis.

```
gap> IsomorphismSCAlgebra( GF(8) );
CanonicalBasis( GF(2^3) ) -> CanonicalBasis( <algebra of dimension 3 over GF(
2)> )
gap> IsomorphismSCAlgebra( GF(2)^[2,2] );
CanonicalBasis( ( GF(2)^[ 2, 2 ] ) ) -> CanonicalBasis( <algebra of dimension
4 over GF(2)> )
```

12 ▶ `RepresentativeLinearOperation( A, v, w, opr )` O

is an element of the algebra A that maps the vector v to the vector w under the linear operation described by the function *opr*. If no such element exists then `fail` is returned.

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> B:= AlgebraByGenerators( Rationals, [ m1, m2 ] );;
gap> RepresentativeLinearOperation( B, [1,0], [1,0], OnRight );
[ [ 1, 0 ], [ 0, 0 ] ]
gap> RepresentativeLinearOperation( B, [1,0], [0,1], OnRight );
fail
```

60.10 Representations of Algebras

An algebra module is a vector space together with an action of an algebra. So a module over an algebra is constructed by giving generators of a vector space, and a function for calculating the action of algebra elements on elements of the vector space. When creating an algebra module, the generators of the vector space are wrapped up and given the category `IsLeftAlgebraModuleElement` or `IsRightModuleElement` if the algebra acts from the left, or right respectively. (So in the case of a bi-module the elements get both categories.) Most linear algebra computations are delegated to the original vector space.

The transition between the original vector space and the corresponding algebra module is handled by `ExtRepOfObj` and `ObjByExtRep`. For an element `v` of the algebra module, `ExtRepOfObj( v )` returns the underlying element of the original vector space. Furthermore, if `vec` is an element of the original vector space, and `fam` the elements family of the corresponding algebra module, then `ObjByExtRep( fam, vec )` returns the corresponding element of the algebra module. Below is an example of this.

The action of the algebra on elements of the algebra module is constructed by using the operator `^`. If `x` is an element of an algebra `A`, and `v` an element of a left `A`-module, then `x^v` calculates the result of the action of `x` on `v`. Similarly, if `v` is an element of a right `A`-module, then `v^x` calculates the action of `x` on `v`.

1 ▶ `LeftAlgebraModuleByGenerators( A, op, gens )` O

Constructs the left algebra module over A generated by the list of vectors *gens*. The action of A is described by the function *op*. This must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector; it outputs the result of applying the algebra element to the vector.

2 ▶ `RightAlgebraModuleByGenerators( A, op, gens )` O

Constructs the right algebra module over A generated by the list of vectors *gens*. The action of A is described by the function *op*. This must be a function of two arguments; the first argument is a vector, and the second argument is the algebra element; it outputs the result of applying the algebra element to the vector.

3 ▶ `BiAlgebraModuleByGenerators( A, B, opl, opr, gens )` O

Constructs the algebra bi-module over A and B generated by the list of vectors *gens*. The left action of A is described by the function *opl*, and the right action of B by the function *opr*. *opl* must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector; it outputs the result of applying the algebra element on the left to the vector. *opr* must be a function of two arguments; the first argument is a vector, and the second argument is the algebra element; it outputs the result of applying the algebra element on the right to the vector.

```
gap> A:= Rationals^[3,3];
( Rationals^[ 3, 3 ] )
gap> V:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> W:= RightAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );
<right-module over ( Rationals^[ 3, 3 ] )>
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );
<bi-module over ( Rationals^[ 3, 3 ] ) (left) and ( Rationals^
[ 3, 3 ] ) (right)>
```

In the above examples, the modules `V`, `W`, and `M` are 3-dimensional vector spaces over the rationals. The algebra `A` acts from the left on `V`, from the right on `W`, and from the left and from the right on `M`.

4 ▶ `LeftAlgebraModule( A, op, V )` O

Constructs the left algebra module over A with underlying space V. The action of A is described by the function *op*. This must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector from V; it outputs the result of applying the algebra element to the vector.

5 ▶ `RightAlgebraModule( A, op, V )` O

Constructs the right algebra module over A with underlying space V. The action of A is described by the function *op*. This must be a function of two arguments; the first argument is a vector, from V and the second argument is the algebra element; it outputs the result of applying the algebra element to the vector.

6 ▶ `BiAlgebraModule( A, B, opl, opr, V )` O

Constructs the algebra bi-module over A and B with underlying space V. The left action of A is described by the function *opl*, and the right action of B by the function *opr*. *opl* must be a function of two arguments; the first argument is the algebra element, and the second argument is a vector from V; it outputs the result of applying the algebra element on the left to the vector. *opr* must be a function of two arguments; the first argument is a vector from V, and the second argument is the algebra element; it outputs the result of applying the algebra element on the right to the vector.

```
gap> A:= Rationals^[3,3];;
gap> V:= Rationals^3;
( Rationals^3 )
gap> V:= Rationals^3;;
gap> M:= BiAlgebraModule( A, A, \*, \*, V );
<bi-module over ( Rationals^[ 3, 3 ] ) (left) and ( Rationals^
[ 3, 3 ] ) (right)>
gap> Dimension( M );
3
```

7 ▶ `GeneratorsOfAlgebraModule( M )` A

A list of elements of M that generate M as an algebra module.

```
gap> A:= Rationals^[3,3];;
gap> V:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> GeneratorsOfAlgebraModule( V );
[ [ 1, 0, 0 ] ]
```

8 ▶ `IsAlgebraModuleElement( obj )` C
▶ `IsAlgebraModuleElementCollection( obj )` C
▶ `IsAlgebraModuleElementFamily( fam )` C

Category of algebra module elements. If an object has `IsAlgebraModuleElementCollection`, then it is an algebra module. If a family has `IsAlgebraModuleElementFamily`, then it is a family of algebra module elements (every algebra module has its own elements family).

9 ▶ `IsLeftAlgebraModuleElement( obj )` C
▶ `IsLeftAlgebraModuleElementCollection( obj )` C

Category of left algebra module elements. If an object has `IsLeftAlgebraModuleElementCollection`, then it is a left-algebra module.

10 ▶ `IsRightAlgebraModuleElement( obj )` C
▶ `IsRightAlgebraModuleElementCollection( obj )` C

Category of right algebra module elements. If an object has `IsRightAlgebraModuleElementCollection`, then it is a right-algebra module.

```
gap> A:= Rationals^[3,3];
( Rationals^[ 3, 3 ] )
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );
<bi-module over ( Rationals^[ 3, 3 ] ) (left) and ( Rationals^
[ 3, 3 ] ) (right)>
gap> vv:= BasisVectors( Basis( M ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> IsLeftAlgebraModuleElement( vv[1] );
true
gap> IsRightAlgebraModuleElement( vv[1] );
true
gap> vv[1] = [ 1, 0, 0 ];
false
gap> ExtRepOfObj( vv[1] ) = [ 1, 0, 0 ];
true
gap> ObjByExtRep( ElementsFamily( FamilyObj( M ) ), [ 1, 0, 0 ] ) in M;
true
```

```
gap> xx:= BasisVectors( Basis( A ) );;
gap> xx[4]^vv[1];  # left action
[ 0, 1, 0 ]
gap> vv[1]^xx[2];  # right action
[ 0, 1, 0 ]
```

11 ▶ `LeftActingAlgebra( V )` A

Here V is a left-algebra module; this function returns the algebra that acts from the left on V.

12 ▶ `RightActingAlgebra( V )` A

Here V is a right-algebra module; this function returns the algebra that acts from the right on V.

13 ▶ `ActingAlgebra( V )` O

Here V is an algebra module; this function returns the algebra that acts on V (this is the same as `LeftActingAlgebra( V )` if V is a left module, and `RightActingAlgebra( V )` if V is a right module; it will signal an error if V is a bi-module).

```
gap> A:= Rationals^[3,3];;
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );;
gap> LeftActingAlgebra( M );
( Rationals^[ 3, 3 ] )
gap> RightActingAlgebra( M );
( Rationals^[ 3, 3 ] )
gap> V:= RightAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> ActingAlgebra( V );
( Rationals^[ 3, 3 ] )
```

14 ▶ `IsBasisOfAlgebraModuleElementSpace( B )` C

If a basis B lies in the category `IsBasisOfAlgebraModuleElementSpace`, then B is a basis of a subspace of an algebra module. This means that B has the record field B`!.delegateBasis` set. This last object is a basis of the corresponding subspace of the vector space underlying the algebra module (i.e., the vector space spanned by all `ExtRepOfObj( v )` for `v` in the algebra module).

```
gap> A:= Rationals^[3,3];;
gap> M:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [ 1, 0, 0 ] ] );;
gap> B:= Basis( M );
Basis( <3-dimensional bi-module over ( Rationals^
[ 3, 3 ] ) (left) and ( Rationals^[ 3, 3 ] ) (right)>,
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] )
gap> IsBasisOfAlgebraModuleElementSpace( B );
true
gap> B!.delegateBasis;
SemiEchelonBasis( <vector space of dimension 3 over Rationals>,
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ] )
```

15 ▶ `MatrixOfAction( B, x )` O
▶ `MatrixOfAction( B, x, side )` O

Here B is a basis of an algebra module and x is an element of the algebra that acts on this module. This function returns the matrix of the action of x with respect to B. If x acts from the left, then the coefficients of the images of the basis elements of B (under the action of x) are the columns of the output. If x acts from the right, then they are the rows of the output.

If the module is a bi-module, then the third parameter *side* must be specified. This is the string `left`, or `right` depending whether x acts from the left or the right.

```
gap> M:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> x:= Basis(A)[3];
[ [ 0, 0, 1 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ]
gap> MatrixOfAction( Basis( M ), x );
[ [ 0, 0, 1 ], [ 0, 0, 0 ], [ 0, 0, 0 ] ]
```

16 ▶ `SubAlgebraModule(` *M*, *gens* `[,` *"basis"*`] )` O

is the sub-module of the algebra module M, generated by the vectors in *gens*. If as an optional argument the string `basis` is added, then it is assumed that the vectors in *gens* form a basis of the submodule.

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> A:= Algebra( Rationals, [ m1, m2 ] );;
gap> M:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0 ], [ 0, 1 ] ] );
<left-module over <algebra over Rationals, with 2 generators>>
gap> bb:= BasisVectors( Basis( M ) );
[ [ 1, 0 ], [ 0, 1 ] ]
gap> V:= SubAlgebraModule( M, [ bb[1] ] );
<left-module over <algebra over Rationals, with 2 generators>>
gap> Dimension( V );
1
```

17 ▶ `LeftModuleByHomomorphismToMatAlg(` *A*, *hom* `)` O

Here A is an algebra and *hom* a homomorphism from A into a matrix algebra. This function returns the left A-module defined by the homomorphism *hom*.

18 ▶ `RightModuleByHomomorphismToMatAlg(` *A*, *hom* `)` O

Here A is an algebra and *hom* a homomorphism from A into a matrix algebra. This function returns the right A-module defined by the homomorphism *hom*.

First we produce a structure constants algebra with basis elements x, y, z such that $x^2 = x$, $y^2 = y$, $xz = z$, $zy = z$ and all other products are zero.

```
gap> T:= EmptySCTable( 3, 0 );;
gap> SetEntrySCTable( T, 1, 1, [ 1, 1 ]);
gap> SetEntrySCTable( T, 2, 2, [ 1, 2 ]);
gap> SetEntrySCTable( T, 1, 3, [ 1, 3 ]);
gap> SetEntrySCTable( T, 3, 2, [ 1, 3 ]);
gap> A:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 3 over Rationals>
```

Now we construct an isomorphic matrix algebra.

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> m3:= NullMat( 2, 2 );; m3[1][2]:= 1;;
gap> B:= Algebra( Rationals, [ m1, m2, m3 ] );
<algebra over Rationals, with 3 generators>
```

Finally we construct the homomorphism and the corresponding right module.

```
gap> f:= AlgebraHomomorphismByImages( A, B, Basis(A), [ m1, m2, m3 ] );;
gap> RightModuleByHomomorphismToMatAlg( A, f );
<right-module over <algebra of dimension 3 over Rationals>>
```

19 ▶ `AdjointModule( A )` A

returns the A-module defined by the left action of A on itself.

```
gap> m1:= NullMat( 2, 2 );; m1[1][1]:= 1;;
gap> m2:= NullMat( 2, 2 );; m2[2][2]:= 1;;
gap> m3:= NullMat( 2, 2 );; m3[1][2]:= 1;;
gap> A:= Algebra( Rationals, [ m1, m2, m3 ] );
<algebra over Rationals, with 3 generators>
gap> V:= AdjointModule( A );
<3-dimensional left-module over <algebra of dimension 3 over Rationals>>
gap> v:= Basis( V )[3];
[ [ 0, 1 ], [ 0, 0 ] ]
gap> W:= SubAlgebraModule( V, [ v ] );
<left-module over <algebra of dimension 3 over Rationals>>
gap> Dimension( W );
1
```

20 ▶ `FaithfulModule( A )` A

returns a faithful finite-dimensional left-module over the algebra A. This is only implemented for associative algebras, and for Lie algebras of characteristic 0. (It may also work for certain Lie algebras of characteristic $p > 0$.)

```
gap> T:= EmptySCTable( 2, 0 );;
gap> A:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 2 over Rationals>
```

`A` is a 2-dimensional algebra where all products are zero.

```
gap> V:= FaithfulModule( A );
<left-module over <algebra of dimension 2 over Rationals>>
gap> vv:= BasisVectors( Basis( V ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> xx:= BasisVectors( Basis( A ) );
[ v.1, v.2 ]
gap> xx[1]^vv[3];
[ 1, 0, 0 ]
```

21 ▶ `ModuleByRestriction( V, sub )` O
▶ `ModuleByRestriction( V, subl, subr )` O

Here V is an algebra module and *sub* is a subalgebra of the acting algebra of V. This function returns the module that is the restriction of V to *sub*. So it has the same underlying vector space as V, but the acting algebra is *sub*. If two subalgebras are given then V is assumed to be a bi-module, and *subl* a subalgebra of the algebra acting on the left, and *subr* a subalgebra of the algebra acting on the right.

```
gap> A:= Rationals^[3,3];;
gap> V:= LeftAlgebraModuleByGenerators( A, \*, [ [ 1, 0, 0 ] ] );;
gap> B:= Subalgebra( A, [ Basis(A)[1] ] );
<algebra over Rationals, with 1 generators>
gap> W:= ModuleByRestriction( V, B );
<left-module over <algebra over Rationals, with 1 generators>>
```

22 ▶ `NaturalHomomorphismBySubAlgebraModule( `*V*`, `*W*` )` O

Here V must be a sub-algebra module of V. This function returns the projection from V onto V/W. It is a linear map, that is also a module homomorphism. As usual images can be formed with `Image( f, v )` and pre-images with `PreImagesRepresentative( f, u )`.

The quotient module can also be formed by entering V/W.

```
gap> A:= Rationals^[3,3];;
gap> B:= DirectSumOfAlgebras( A, A );
<algebra over Rationals, with 6 generators>
gap> T:= StructureConstantsTable( Basis( B ) );;
gap> C:= AlgebraByStructureConstants( Rationals, T );
<algebra of dimension 18 over Rationals>
gap> V:= AdjointModule( C );
<left-module over <algebra of dimension 18 over Rationals>>
gap> W:= SubAlgebraModule( V, [ Basis(V)[1] ] );
<left-module over <algebra of dimension 18 over Rationals>>
gap> f:= NaturalHomomorphismBySubAlgebraModule( V, W );
<linear mapping by matrix, <
18-dimensional left-module over <algebra of dimension 18 over Rationals>> -> <
9-dimensional left-module over <algebra of dimension 18 over Rationals>>>
gap> quo:= ImagesSource( f );  # i.e., the quotient module
<9-dimensional left-module over <algebra of dimension 18 over Rationals>>
gap> v:= Basis( quo )[1];
[ 1, 0, 0, 0, 0, 0, 0, 0, 0 ]
gap> PreImagesRepresentative( f, v );
v.4
gap> Basis( C )[4]^v;
[ 1, 0, 0, 0, 0, 0, 0, 0, 0 ]
```

23 ▶ `DirectSumOfAlgebraModules( `*list*` )` O
▶ `DirectSumOfAlgebraModules( `*V*`, `*W*` )` O

Here *list* must be a list of algebra modules. This function returns the direct sum of the elements in the list (as an algebra module). The modules must be defined over the same algebras.

In the second form is short for `DirectSumOfAlgebraModules( [ `*V*`, `*W*` ] )`

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );;
gap> V:= BiAlgebraModuleByGenerators( A, A, \*, \*, [ [1,0,0] ] );;
gap> W:= DirectSumOfAlgebraModules( V, V );
<6-dimensional left-module over ( Rationals^[ 3, 3 ] )>
gap> BasisVectors( Basis( W ) );
[ ( [ 1, 0, 0 ] )(+)( [ 0, 0, 0 ] ), ( [ 0, 1, 0 ] )(+)( [ 0, 0, 0 ] ),
  ( [ 0, 0, 1 ] )(+)( [ 0, 0, 0 ] ), ( [ 0, 0, 0 ] )(+)( [ 1, 0, 0 ] ),
  ( [ 0, 0, 0 ] )(+)( [ 0, 1, 0 ] ), ( [ 0, 0, 0 ] )(+)( [ 0, 0, 1 ] ) ]
```

24 ▶ `TranslatorSubalgebra( M, U, W )` O

Here M is an algebra module, and U and W are two subspaces of M. Let A be the algebra acting on M. This function returns the subspace of elements of A that map U into W. If W is a sub-algebra-module (i.e., closed under the action of A), then this space is a subalgebra of A.

This function works for left, or right modules over a finite-dimensional algebra. We stress that it is not checked whether U and W are indeed subspaces of M. If this is not the case nothing is guaranteed about the behaviour of the function.

```
gap> A:= FullMatrixAlgebra( Rationals, 3 );
( Rationals^[ 3, 3 ] )
gap> V:= Rationals^[3,2];
( Rationals^[ 3, 2 ] )
gap> M:= LeftAlgebraModule( A, \*, V );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> bm:= Basis(M);;
gap> U:= SubAlgebraModule( M, [ bm[1] ] );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> TranslatorSubalgebra( M, U, M );
<algebra of dimension 9 over Rationals>
gap> W:= SubAlgebraModule( M, [ bm[4] ] );
<left-module over ( Rationals^[ 3, 3 ] )>
gap> T:=TranslatorSubalgebra( M, U, W );
<algebra of dimension 0 over Rationals>
```

61 Lie Algebras

A Lie algebra L is an algebra such that $xx = 0$ and $x(yz) + y(zx) + z(xy) = 0$ for all $x, y, z \in L$. A common way of creating a Lie algebra is by taking an associative algebra together with the commutator as product. Therefore the product of two elements x, y of a Lie algebra is usually denoted by $[x, y]$, but in GAP this denotes the list of the elements x and y; hence the product of elements is made by the usual `*`. This gives no problems when dealing with Lie algebras given by a table of structure constants. However, for matrix Lie algebras the situation is not so easy as `*` denotes the ordinary (associative) matrix multiplication. In GAPthis problem is solved by wrapping elements of a matrix Lie algebra up as LieObjects, and then define the `*` for LieObjects to be the commutator (see 61.1);

61.1 Lie objects

Let x be a ring element, then `LieObject(x)` wraps x up into an object that contains the same data (namely x). The multiplication `*` for Lie objects is formed by taking the commutator. More exactly, if l_1 and l_2 are the Lie objects corresponding to the ring elements r_1 and r_2, then $l_1 * l_2$ is equal to the Lie object corresponding to $r_1 * r_2 - r_2 * r_2$. Two rules for Lie objects are worth noting:

- An element is **not** equal to its Lie element.
- If we take the Lie object of an ordinary (associative) matrix then this is again a matrix; it is therefore a collection (of its rows) and a list. But it is **not** a collection of collections of its entries, and its family is **not** a collections family.

1 ▶ `LieObject(` *obj* `)` A

Let *obj* be a ring element. Then `LieObject(` *obj* `)` is the corresponding Lie object. If *obj* lies in the family F, then `LieObject(` *obj* `)` lies in the family LieFamily(F) (see 61.1.3).

```
gap> m:= [ [ 1, 0 ], [ 0, 1 ] ];;
gap> lo:= LieObject( m );
LieObject( [ [ 1, 0 ], [ 0, 1 ] ] )
gap> m*m;
[ [ 1, 0 ], [ 0, 1 ] ]
gap> lo*lo;
LieObject( [ [ 0, 0 ], [ 0, 0 ] ] )
```

2 ▶ `IsLieObject(` *obj* `)` C
▶ `IsLieObjectCollection(` *obj* `)` C

An object lies in `IsLieObject` if and only if it lies in a family constructed by `LieFamily`.

```
gap> m:= [ [ 1, 0 ], [ 0, 1 ] ];;
gap> lo:= LieObject( m );
LieObject( [ [ 1, 0 ], [ 0, 1 ] ] )
gap> IsLieObject( m );
false
gap> IsLieObject( lo );
true
```

3 ▶ `LieFamily( `*Fam*` )` A

is a family F in bijection with the family *Fam*, but with the Lie bracket as infix multiplication. That is, for x, y in *Fam*, the product of the images in F will be the image of $x * y - y * x$.

The standard type of objects in a Lie family F is F`!.packedType`.

The bijection from *Fam* to F is given by `Embedding( `*Fam*`, `F` )`; this bijection respects addition and additive inverses.

4 ▶ `UnderlyingFamily( `*Fam*` )` A

If *Fam* is a Lie family then `UnderlyingFamily( `*Fam*` )` is a family F such that *Fam* `= LieFamily( `F` )`.

61.2 Constructing Lie algebras

In this section we describe functions that create Lie algebras. Creating and working with subalgebras goes exactly in the same way as for general algebras; so for that we refer to Chapter 60.

1 ▶ `LieAlgebraByStructureConstants( `R`, `*sctable*` )` F
▶ `LieAlgebraByStructureConstants( `R`, `*sctable*`, `*name*` )` F
▶ `LieAlgebraByStructureConstants( `R`, `*sctable*`, `*name1*`, `*name2*`, ... )` F

`LieAlgebraByStructureConstants` does the same as `AlgebraByStructureConstants`, except that the result is assumed to be a Lie algebra. Note that the function does not check whether *sctable* satisfies the Jacobi identity. (So if one creates a Lie algebra this way with a table that does not satisfy the Jacobi identity, errors may occur later on.)

```
gap> T:= EmptySCTable( 2, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1/2, 1 ] );
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 2 over Rationals>
```

2 ▶ `LieAlgebra( `L` )` F
▶ `LieAlgebra( `F`, `*gens*` )` F
▶ `LieAlgebra( `F`, `*gens*`, `*zero*` )` F
▶ `LieAlgebra( `F`, `*gens*`, "basis" )` F
▶ `LieAlgebra( `F`, `*gens*`, `*zero*`, "basis" )` F

For an associative algebra L, `LieAlgebra( `L` )` is the Lie algebra isomorphic to L as a vector space but with the Lie bracket as product.

`LieAlgebra( `F`, `*gens*` )` is the Lie algebra over the division ring F, generated **as Lie algebra** by the Lie objects corresponding to the vectors in the list *gens*.

Note that the algebra returned by `LieAlgebra` does not contain the vectors in *gens*. The elements in *gens* are wrapped up as Lie objects (see 61.1). This allows one to create Lie algebras from ring elements with respect to the Lie bracket as product. But of course the product in the Lie algebra is the usual `*`.

If there are three arguments, a division ring F and a list *gens* and an element *zero*, then `LieAlgebra( `F`, `*gens*`, `*zero*` )` is the corresponding F-Lie algebra with zero element the Lie object corresponding to *zero*.

If the last argument is the string `"basis"` then the vectors in *gens* are known to form a basis of the algebra (as an F-vector space).

Note that even if each element in *gens* is already a Lie element, i.e., is of the form `LieElement( elm )` for an object *elm*, the elements of the result lie in the Lie family of the family that contains *gens* as a subset.

```
gap> A:= FullMatrixAlgebra( GF( 7 ), 4 );;
gap> L:= LieAlgebra( A );
<Lie algebra of dimension 16 over GF(7)>
gap> mats:= [ [[ 1, 0 ], [ 0, -1 ]], [[ 0, 1 ], [ 0, 0 ]], [[ 0, 0 ], [ 1, 0]] ];;
gap> L:= LieAlgebra( Rationals, mats );
<Lie algebra over Rationals, with 3 generators>
```

3 ▶ `FreeLieAlgebra( R, rank )` F
▶ `FreeLieAlgebra( R, rank, name )` F
▶ `FreeLieAlgebra( R, name1, name2, ... )` F

Returns a free Lie algebra of rank *rank* over the ring R. `FreeLieAlgebra( R, name1, name2,...)` returns a free Lie algebra over R with generators named *name1*, *name2*, and so on. The elements of a free Lie algebra are written on the Hall-Lyndon basis.

```
gap> L:= FreeLieAlgebra( Rationals, "x", "y", "z" );
<Lie algebra over Rationals, with 3 generators>
gap> g:= GeneratorsOfAlgebra( L );; x:= g[1];; y:=g[2];; z:= g[3];;
gap> z*(y*(x*(z*y)));
(-1)*((x*(y*z))*(y*z))+(-1)*((x*((y*z)*z))*y)+(-1)*(((x*z)*(y*z))*y)
```

4 ▶ `FullMatrixLieAlgebra( R, n )` F
▶ `MatrixLieAlgebra( R, n )` F
▶ `MatLieAlgebra( R, n )` F

is the full matrix Lie algebra $R^{n\times n}$, for a ring R and a nonnegative integer n.

```
gap> FullMatrixLieAlgebra( GF(9), 10 );
<Lie algebra over GF(3^2), with 19 generators>
```

5 ▶ `RightDerivations( B )` A
▶ `LeftDerivations( B )` A
▶ `Derivations( B )` A

These functions all return the matrix Lie algebra of derivations of the algebra A with basis B.

`RightDerivations( B )` returns the algebra of derivations represented by their right action on the algebra A. This means that with respect to the basis B of A, the derivation D is described by the matrix $[d_{i,j}]$ which means that D maps the i-th basis element b_i to $\sum_{j=1}^{n} d_{ij} b_j$.

`LeftDerivations( B )` returns the Lie algebra of derivations represented by theor left action on the algebra A. So the matrices contained in the algebra output by `LeftDerivations( B )` are the transposes of the matrices contained in the output of `RightDerivations( B )`.

`Derivations` is just a synonym for `RightDerivations`.

```
gap> A:= OctaveAlgebra( Rationals );
<algebra of dimension 8 over Rationals>
gap> L:= Derivations( Basis( A ) );
<Lie algebra of dimension 14 over Rationals>
```

6 ▶ `SimpleLieAlgebra( type, n, F )` F

This function constructs the simple Lie algebra of type *type* and of rank n over the field F.

type must be one of A, B, C, D, E, F, G, H, K, S, W. For the types A to G, n must be a positive integer. The last four types only exist over fields of characteristic $p > 0$. If the type is H, then n must be a list of positive integers of even length. If the type is K, then n must be a list of positive integers of odd length. For the other types, S and W, n must be a list of positive integers of any length. In some cases the Lie algebra returned by this function is not simple. Examples are the Lie algebras of type A_n over a field of characteristic $p > 0$ where p divides $n + 1$, and the Lie algebras of type K_n where n is a list of length 1.

If *type* is one of A, B, C, D, E, F, G, and F is a field of characteristic zero, then the basis of the returned Lie algebra is a Chevalley basis.

```
gap> SimpleLieAlgebra( "E", 6, Rationals );
<Lie algebra of dimension 78 over Rationals>
gap> SimpleLieAlgebra( "A", 6, GF(5) );
<Lie algebra of dimension 48 over GF(5)>
gap> SimpleLieAlgebra( "W", [1,2], GF(5) );
<Lie algebra of dimension 250 over GF(5)>
gap> SimpleLieAlgebra( "H", [1,2], GF(5) );
<Lie algebra of dimension 123 over GF(5)>
```

61.3 Distinguished Subalgebras

Here we describe functions that calculate well-known subalgebras and ideals of a Lie algebra (such as the centre, the centralizer of a subalgebra, etc.).

1 ▶ `LieCentre( L )` A
▶ `LieCenter( L )` A

The **Lie** centre of the Lie algebra L is the kernel of the adjoint mapping, that is, the set $\{a \in L; \forall x \in L : ax = 0\}$.

In characteristic 2 this may differ from the usual centre (that is the set of all $a \in L$ such that $ax = xa$ for all $x \in L$). Therefore, this operation is named `LieCentre` and not `Centre`.

```
gap> L:= FullMatrixLieAlgebra( GF(3), 3 );
<Lie algebra over GF(3), with 5 generators>
gap> LieCentre( L );
<two-sided ideal in <Lie algebra of dimension 9 over GF(3)>, (dimension 1)>
```

2 ▶ `LieCentralizer( L, S )` O

is the annihilator of S in the Lie algebra L, that is, the set $\{a \in L; \forall s \in S : a * s = 0\}$. Here S may be a subspace or a subalgebra of L.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> b:= BasisVectors( Basis( L ) );;
gap> LieCentralizer( L, Subalgebra( L, [ b[1], b[2] ] ) );
<Lie algebra of dimension 1 over Rationals>
```

3 ▶ `LieNormalizer( L, U )` O

is the normalizer of the subspace U in the Lie algebra L, that is, the set $N_L(U) = \{x \in L; [x, U] \subset U\}$.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> b:= BasisVectors( Basis( L ) );;
gap> LieNormalizer( L, Subalgebra( L, [ b[1], b[2] ] ) );
<Lie algebra of dimension 8 over Rationals>
```

4 ▶ `LieDerivedSubalgebra( L )` A

is the (Lie) derived subalgebra of the Lie algebra L.

```
gap>  L:= FullMatrixLieAlgebra( GF( 3 ), 3 );
<Lie algebra over GF(3), with 5 generators>
gap> LieDerivedSubalgebra( L );
<Lie algebra of dimension 8 over GF(3)>
```

5 ▶ `LieNilRadical( L )` A

This function calculates the (Lie) nil radical of the Lie algebra L.

In the following two examples we temporarily increase the line length limit from its default value 80 to 81 in order to make the long output expressions fit each into one line.

```
gap> mats:= [ [[1,0],[0,0]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:= LieAlgebra( Rationals, mats );;
gap> SizeScreen([ 81, ]);;
gap> LieNilRadical( L );
<two-sided ideal in <Lie algebra of dimension 3 over Rationals>, (dimension 2)>
gap> SizeScreen([ 80, ]);;
```

6 ▶ `LieSolvableRadical( L )` A

Returns the (Lie) solvable radical of the Lie algebra L.

```
gap> L:= FullMatrixLieAlgebra( Rationals, 3 );;
gap> SizeScreen([ 81, ]);;
gap> LieSolvableRadical( L );
<two-sided ideal in <Lie algebra of dimension 9 over Rationals>, (dimension 1)>
gap> SizeScreen([ 80, ]);;
```

7 ▶ `CartanSubalgebra( L )` A

A Cartan subalgebra of a Lie algebra L is defined as a nilpotent subalgebra of L equal to its own Lie normalizer in L.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> CartanSubalgebra( L );
<Lie algebra of dimension 2 over Rationals>
```

61.4 Series of Ideals

1 ▶ `LieDerivedSeries( L )` A

is the (Lie) derived series of the Lie algebra L.

```
gap> mats:= [ [[1,0],[0,0]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:= LieAlgebra( Rationals, mats );;
gap> LieDerivedSeries( L );
[ <Lie algebra of dimension 3 over Rationals>,
  <Lie algebra of dimension 1 over Rationals>,
  <Lie algebra of dimension 0 over Rationals> ]
```

2 ▶ `LieLowerCentralSeries( L )` A

is the (Lie) lower central series of the Lie algebra L.

```
gap> mats:= [ [[ 1, 0 ], [ 0, 0 ]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:=LieAlgebra( Rationals, mats );;
gap> LieLowerCentralSeries( L );
[ <Lie algebra of dimension 3 over Rationals>,
  <Lie algebra of dimension 1 over Rationals> ]
```

3 ▶ `LieUpperCentralSeries( L )` A

is the (Lie) upper central series of the Lie algebra L.

```
gap> mats:= [ [[ 1, 0 ], [ 0, 0 ]], [[0,1],[0,0]], [[0,0],[0,1]] ];;
gap> L:=LieAlgebra( Rationals, mats );;
gap> LieUpperCentralSeries( L );
[ <two-sided ideal in <Lie algebra of dimension 3 over Rationals>,
      (dimension 1)>, <Lie algebra over Rationals, with 0 generators> ]
```

61.5 Properties of a Lie Algebra

1 ▶ `IsLieAbelian( L )` P

is `true` if L is a Lie algebra such that each product of elements in L is zero, and `false` otherwise.

```
gap>  T:= EmptySCTable( 5, 0, "antisymmetric" );;
gap>  L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 5 over Rationals>
gap> IsLieAbelian( L );
true
```

2 ▶ `IsLieNilpotent( L )` P

A Lie algebra L is defined to be (Lie) *nilpotent* when its (Lie) lower central series reaches the trivial subalgebra.

```
gap> T:= EmptySCTable( 5, 0, "antisymmetric" );;
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 5 over Rationals>
gap> IsLieNilpotent( L );
true
```

3 ▶ `IsLieSolvable( L )` P

A Lie algebra L is defined to be (Lie) *solvable* when its (Lie) derived series reaches the trivial subalgebra.

```
gap> T:= EmptySCTable( 5, 0, "antisymmetric" );;
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 5 over Rationals>
gap> IsLieSolvable( L );
true
```

61.6 Direct Sum Decompositions

In this section we describe two functions that calculate a direct sum decomposition of a Lie algebra; the so-called Levi decomposition and the decomposition into a direct sum of ideals.

1 ▶ `LeviMalcevDecomposition( L )` A

A Levi-Malcev subalgebra of the algebra L is a semisimple subalgebra complementary to the radical of L. This function returns a list with two components. The first component is a Levi-Malcev subalgebra, the second the radical. This function is implemented for associative and Lie algebras.

```
gap> L:= FullMatrixLieAlgebra( Rationals, 5 );;
gap> LeviMalcevDecomposition( L );
[ <Lie algebra of dimension 24 over Rationals>,
  <two-sided ideal in <Lie algebra of dimension 25 over Rationals>,
      (dimension 1)> ]
```

2 ▶ `DirectSumDecomposition( L )` A

This function calculates a list of ideals of the algebra L such that L is equal to their direct sum. Currently this is only implemented for semisimple associative algebras, and Lie algebras (semisimple or not).

```
gap> L:= FullMatrixLieAlgebra( Rationals, 5 );;
gap> DirectSumDecomposition( L );
[ <two-sided ideal in
      <two-sided ideal in <Lie algebra of dimension 25 over Rationals>,
          (dimension 1)>, (dimension 1)>,
  <two-sided ideal in <two-sided ideal in
          <Lie algebra of dimension 25 over Rationals>, (dimension 24)>,
      (dimension 24)> ]
```

61.7 Semisimple Lie Algebras and Root Systems

This section contains some functions for dealing with semisimple Lie algebras and their root systems.

1 ▶ `SemiSimpleType( L )` A

Let L be a semisimple Lie algebra, i.e., a direct sum of simple Lie algebras. Then `SemiSimpleType` returns the type of L, i.e., a string containing the types of the simple summands of L.

```
gap> L:= SimpleLieAlgebra( "E", 8, Rationals );;
gap> b:= BasisVectors( Basis( L ) );;
gap> K:= LieCentralizer( L, Subalgebra( L, [ b[61]+b[79]+b[101]+b[102] ] ) );
<Lie algebra of dimension 102 over Rationals>
gap> lev:= LeviMalcevDecomposition(K);;
gap> SemiSimpleType( lev[1] );
"B3 A1"
```

2 ▶ `ChevalleyBasis( L )` A

Here L must be a semisimple Lie algebra with a split Cartan subalgebra. Then `ChevalleyBasis( L )` returns a list consisting of three sublists. Together these sublists form a Chevalley basis of L. The first list contains the positive root vectors, the second list contains the negative root vectors, and the third list the Cartan elements of the Chevalley basis.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> ChevalleyBasis( L );
[ [ v.1, v.2, v.3, v.4, v.5, v.6 ], [ v.7, v.8, v.9, v.10, v.11, v.12 ],
  [ v.13, v.14 ] ]
```

3 ▶ `IsRootSystem( obj )` C

Category of root systems.

4 ▶ `IsRootSystemFromLieAlgebra( obj )` C

Category of root systems that come from (semisimple) Lie algebras. They often have special attributes such as `UnderlyingLieAlgebra`, `PositiveRootVectors`, `NegativeRootVectors`, `CanonicalGenerators`.

5 ▶ `RootSystem( L )` A

`RootSystem` calculates the root system of the semisimple Lie algebra L with a split Cartan subalgebra.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );
<Lie algebra of dimension 14 over Rationals>
gap> R:= RootSystem( L );
<root system of rank 2>
gap> IsRootSystem( R );
true
gap> IsRootSystemFromLieAlgebra( R );
true
```

6 ▶ `UnderlyingLieAlgebra( R )` A

Here R is a root system coming from a semisimple Lie algebra L. This function returns L.

7 ▶ `PositiveRoots( R )` A

The list of positive roots of the root system R.

8 ▶ `NegativeRoots( R )` A

The list of negative roots of the root system R.

9 ▶ `PositiveRootVectors( R )` A

A list of positive root vectors of the root system R that comes from a Lie algebra L. This is a list in bijection with the list `PositiveRoots( L )`. The root vector is a non-zero element of the root space (in L) of the corresponding root.

10 ▶ `NegativeRootVectors( R )` A

A list of negative root vectors of the root system R that comes from a Lie algebra L. This is a list in bijection with the list `NegativeRoots( L )`. The root vector is a non-zero element of the root space (in L) of the corresponding root.

11 ▶ `SimpleSystem( R )` A

A list of simple roots of the root system R.

12 ▶ `CartanMatrix( R )` A

The Cartan matrix of the root system R, relative to the simple roots in `SimpleSystem( R )`.

13 ▶ `BilinearFormMat( R )` A

The matrix of the bilinear form of the root system R. If we denote this matrix by B, then we have $B(i,j) = (\alpha_i, \alpha_j)$, where the α_i are the simple roots of R.

14 ▶ `CanonicalGenerators( R )` A

Here R must be a root system coming from a semisimple Lie algebra L. This function returns $3l$ generators of L, $x_1, \ldots, x_l, y_1, \ldots, y_l, h_1, \ldots, h_l$, where x_i lies in the root space corresponding to the i-th simple root of the root system of L, y_i lies in the root space corresponding to $-$ the i-th simple root, and the h_i are elements of the Cartan subalgebra. These elements satisfy the relations $h_i * h_j = 0$, $x_i * y_j = \delta_{ij} h_i$, $h_j * x_i = c_{ij} x_i$, $h_j * y_i = -c_{ij} y_i$, where c_{ij} is the entry of the Cartan matrix on position ij.

Also if a is a root of the root system R (so a is a list of numbers), then we have the relation $h_i * x = a[i]x$, where x is a root vector corresponding to a.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> R:= RootSystem( L );;
gap> UnderlyingLieAlgebra( R );
<Lie algebra of dimension 14 over Rationals>
gap> PositiveRoots( R );
[ [ 2, -1 ], [ -3, 2 ], [ -1, 1 ], [ 1, 0 ], [ 3, -1 ], [ 0, 1 ] ]
gap> x:= PositiveRootVectors( R );
[ v.1, v.2, v.3, v.4, v.5, v.6 ]
gap> g:=CanonicalGenerators( R );
[ [ v.1, v.2 ], [ v.7, v.8 ], [ v.13, v.14 ] ]
gap> g[3][1]*x[1];
(2)*v.1
gap> g[3][2]*x[1];
(-1)*v.1
gap> # i.e., x[1] is the root vector belonging to the root [ 2, -1 ]
gap> BilinearFormMat( R );
[ [ 1/12, -1/8 ], [ -1/8, 1/4 ] ]
```

The next few sections deal with the Weyl group of a root system. A Weyl group is represented by its action on the weight lattice. A **weight** is by definition a linear function $\lambda : H \to F$ (where F is the ground field), such that the values $\lambda(h_i)$ are all integers (where the h_i are the Cartan elements of the `CanonicalGenerators`). On the other hand each weight is determined by these values. Therefore we represent a weight by a vector of integers; the i-th entry of this vector is the value $\lambda(h_i)$. Now the elements of the Weyl group are represented by matrices, and if `g` is an element of a Weyl group and `w` a weight, then `w*g` gives the result of applying `g` to `w`. Another way of applying the i-th simple reflection to a weight is by using the function `ApplySimpleReflection` (see below).

A Weyl group is generated by the simple reflections. So `GeneratorsOfGroup( W )` for a Weyl group `W` gives a list of matrices and the i-th entry of this list is the simple reflection corresponding to the i-th simple root of the corresponding root system.

15 ▶ `IsWeylGroup( G )` P

A Weyl group is a group generated by reflections, with the attribute `SparseCartanMatrix` set.

16 ▶ `SparseCartanMatrix( W )` A

This is a sparse form of the Cartan matrix of the corresponding root system. If we denote the Cartan matrix by `C`, then the sparse Cartan matrix of W is a list (of length equal to the length of the Cartan matrix), where the `i`-th entry is a list consisting of elements `[ j, C[i][j] ]`, where `j` is such that `C[i][j]` is non-zero.

17 ▶ `WeylGroup( R )` A

The Weyl group of the root system R. It is generated by the simple reflections. A simple reflection is represented by a matrix, and the result of letting a simple reflection `m` act on a weight `w` is obtained by `w*m`.

```
gap> L:= SimpleLieAlgebra( "F", 4, Rationals );;
gap> R:= RootSystem( L );;
gap> W:= WeylGroup( R );
<matrix group with 4 generators>
gap> IsWeylGroup( W );
true
gap> SparseCartanMatrix( W );
[ [ [ 1, 2 ], [ 3, -1 ] ], [ [ 2, 2 ], [ 4, -1 ] ],
  [ [ 1, -1 ], [ 3, 2 ], [ 4, -1 ] ], [ [ 2, -1 ], [ 3, -2 ], [ 4, 2 ] ] ]
gap> g:= GeneratorsOfGroup( W );;
gap> [ 1, 1, 1, 1 ]*g[2];
[ 1, -1, 1, 2 ]
```

18 ▶ `ApplySimpleReflection( SC, i, wt )` O

Here SC is the sparse Cartan matrix of a Weyl group. This function applies the i-th simple reflection to the weight wt, thus changing wt.

```
gap> L:= SimpleLieAlgebra( "F", 4, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> C:= SparseCartanMatrix( W );;
gap> w:= [ 1, 1, 1, 1 ];;
gap> ApplySimpleReflection( C, 2, w );
gap> w;
[ 1, -1, 1, 2 ]
```

19 ▶ `LongestWeylWordPerm( W )` A

Let g_0 be the longest element in the Weyl group W, and let $\{\alpha_1, \ldots, \alpha_l\}$ be a simple system of the corresponding root system. Then g_0 maps α_i to $-\alpha_{\sigma(i)}$, where σ is a permutation of $(1, \ldots, l)$. This function returns that permutation.

```
gap> L:= SimpleLieAlgebra( "E", 6, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> LongestWeylWordPerm( W );
(1,6)(3,5)
```

20 ▶ `ConjugateDominantWeight( W, wt )` O
▶ `ConjugateDominantWeightWithWord( W, wt )` O

Here W is a Weyl group and wt a weight (i.e., a list of integers). This function returns the unique dominant weight conjugate to wt under W.

`ConjugateDominantWegihtWithWord( W, wt )` returns a list of two elements. The first of these is the dominant weight conjugate do wt. The second element is a list of indices of simple reflections that have to be applied to wt in order to get the dominant weight conjugate to it.

```
gap> L:= SimpleLieAlgebra( "E", 6, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> C:= SparseCartanMatrix( W );;
gap> w:= [ 1, -1, 2, -2, 3, -3 ];;
gap> ConjugateDominantWeight( W, w );
[ 2, 1, 0, 0, 0, 0 ]
gap> c:= ConjugateDominantWeightWithWord( W, w );
[ [ 2, 1, 0, 0, 0, 0 ], [ 2, 4, 2, 3, 6, 5, 4, 2, 3, 1 ] ]
gap> for i in [1..Length(c[2])] do
> ApplySimpleReflection( C, c[2][i], w );
> od;
gap> w;
[ 2, 1, 0, 0, 0, 0 ]
```

21 ▶ `WeylOrbitIterator( W, wt )` O

Returns an iterator for the orbit of the weight wt under the action of the Weyl group W.

```
gap> L:= SimpleLieAlgebra( "E", 6, Rationals );;
gap> W:= WeylGroup( RootSystem( L ) );;
gap> orb:= WeylOrbitIterator( W, [ 1, 1, 1, 1, 1, 1 ] );
<iterator>
gap> NextIterator( orb );
[ 1, 1, 1, 1, 1, 1 ]
gap> NextIterator( orb );
[ -1, -1, -1, -1, -1, -1 ]
gap> orb:= WeylOrbitIterator( W, [ 1, 1, 1, 1, 1, 1 ] );
<iterator>
gap> k:= 0;
0
gap> while not IsDoneIterator( orb ) do
> w:= NextIterator( orb ); k:= k+1;
> od;
gap> k;  # this is the size of the Weyl group of E6
51840
```

61.8 Restricted Lie algebras

A Lie algebra L over a field of characteristic $p > 0$ is called restricted if there is a map $x \mapsto x^p$ from L into L (called a p-map) such that $\mathrm{ad}x^p = (\mathrm{ad}x)^p$, $(\alpha x)^p = \alpha^p x^p$ and $(x + y)^p = x^p + y^p + \sum_{i=1}^{p-1} s_i(x, y)$, where $s_i : L \times L \to L$ are certain Lie polynomials in two variables. Using these relations we can calculate y^p for all $y \in L$, once we know x^p for x in a basis of L. Therefore a p-map is represented in GAP by a list containing the images of the basis vectors of a basis B of L. For this reason this list is an attribute of the basis B.

1 ▶ `IsRestrictedLieAlgebra( L )` P

Test whether L is restricted.

```
gap> L:= SimpleLieAlgebra( "W", [2], GF(5));
<Lie algebra of dimension 25 over GF(5)>
gap> IsRestrictedLieAlgebra( L );
false
gap> L:= SimpleLieAlgebra( "W", [1], GF(5));
<Lie algebra of dimension 5 over GF(5)>
```

```
gap> IsRestrictedLieAlgebra( L );
true
```

2 ▶ `PthPowerImages( B )` A

Here `B` is a basis of a restricted Lie algebra. This function returns the list of the images of the basis vectors of `B` under the p-map.

```
gap> L:= SimpleLieAlgebra( "W", [1], GF(11) );
<Lie algebra of dimension 11 over GF(11)>
gap> B:= Basis( L );
CanonicalBasis( <Lie algebra of dimension 11 over GF(11)> )
gap> PthPowerImages( B );
[ 0*v.1, v.2, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1 ]
```

3 ▶ `PthPowerImage( B, x )` O

B is a basis of a restricted Lie algebra L. This function calculates for an element x of L the image of x under the p-map.

```
gap> L:= SimpleLieAlgebra( "W", [1], GF(11) );;
gap> B:= Basis( L );;
gap> x:= B[1]+B[11];
v.1+v.11
gap> PthPowerImage( B, x );
v.1+v.11
```

4 ▶ `JenningsLieAlgebra( G )` A

Let G be a nontrivial p-group, and let $G = G_1 \supset G_2 \supset \cdots \supset G_m = 1$ be its Jennings series (see 37.17.14). Then the quotients G_i/G_{i+1} are elementary abelian p-groups, i.e., they can be viewed as vector spaces over `GF`(p). Now the Jennings-Lie algebra L of G is the direct sum of those vector spaces. The Lie bracket on L is induced by the commutator in G. Furthermore, the map $g \mapsto g^p$ in G induces a p-map in L making L into a restricted Lie algebra. In the canonical basis of L this p-map is added as an attribute. A Lie algebra created by `JenningsLieAlgebra` is naturally graded. The attribute `Grading` is set.

5 ▶ `PCentralLieAlgebra( G )` A

Here G is a nontrivial p-group. `PCentralLieAlgebra( G )` does the same as `JenningsLieAlgebra( G )` except that the p-central series is used instead of the Jennings series (see 37.17.13). This function also returns a graded Lie algebra. However, it is not necessarily restricted.

```
gap> G:= SmallGroup( 3^6, 123 );
<pc group of size 729 with 6 generators>
gap> L:= JenningsLieAlgebra( G );
<Lie algebra of dimension 6 over GF(3)>
gap> HasPthPowerImages( Basis( L ) );
true
gap> PthPowerImages( Basis( L ) );
[ v.6, 0*v.1, 0*v.1, 0*v.1, 0*v.1, 0*v.1 ]
gap> g:= Grading( L );
rec( min_degree := 1, max_degree := 3, source := Integers,
  hom_components := function( d ) ... end )
gap> List( [1,2,3], g.hom_components );
[ <vector space over GF(3), with 3 generators>,
  <vector space over GF(3), with 2 generators>,
  <vector space over GF(3), with 1 generators> ]
```

61.9 The Adjoint Representation

In this section we show functions for calculating with the adjoint representation of a Lie algebra (and the corresponding trace form, called the Killing form) (see also 60.8.5 and 60.8.6).

1 ▶ `AdjointMatrix( B, x )` O

is the matrix of the adjoint representation of the element x w.r.t. the basis B. The adjoint map is the left multiplication by x. The i-th column of the resulting matrix represents the image of the the i-th basis vector of B under left multiplication by x.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> AdjointMatrix( Basis( L ), Basis( L )[1] );
[ [ 0, 0, -2 ], [ 0, 0, 0 ], [ 0, 1, 0 ] ]
```

2 ▶ `AdjointAssociativeAlgebra( L, K )` A

is the associative matrix algebra (with 1) generated by the matrices of the adjoint representation of the subalgebra K on the Lie algebra L.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> AdjointAssociativeAlgebra( L, L );
<algebra of dimension 9 over Rationals>
gap> AdjointAssociativeAlgebra( L, CartanSubalgebra( L ) );
<algebra of dimension 3 over Rationals>
```

3 ▶ `KillingMatrix( B )` A

is the matrix of the Killing form κ with respect to the basis B, i.e., the matrix $(\kappa(b_i, b_j))$ where $b_1, b_2 \ldots$ are the basis vectors of B.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> KillingMatrix( Basis( L ) );
[ [ 0, 4, 0 ], [ 4, 0, 0 ], [ 0, 0, 8 ] ]
```

4 ▶ `KappaPerp( L, U )` O

is the orthogonal complement of the subspace U of the Lie algebra L with respect to the Killing form κ, that is, the set $U^\perp = \{x \in L; \kappa(x, y) = 0 \text{ for all } y \in L\}$.

$U^\perp$ is a subspace of L, and if U is an ideal of L then $U^\perp$ is a subalgebra of L.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> b:= BasisVectors( Basis( L ) );;
gap> V:= VectorSpace( Rationals, [b[1],b[2]] );;
gap> KappaPerp( L, V );
<vector space of dimension 1 over Rationals>
```

5 ▶ `IsNilpotentElement( L, x )` O

x is nilpotent in L if its adjoint matrix is a nilpotent matrix.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> IsNilpotentElement( L, Basis( L )[1] );
true
```

6 ▶ `NonNilpotentElement( L )` A

A non-nilpotent element of a Lie algebra L is an element x such that adx is not nilpotent. If L is not nilpotent, then by Engel's theorem non nilpotent elements exist in L. In this case this function returns a non nilpotent element of L, otherwise (if L is nilpotent) `fail` is returned.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> NonNilpotentElement( L );
v.13
gap> IsNilpotentElement( L, last );
false
```

7 ▶ `FindSl2( L, x )` O

This function tries to find a subalgebra S of the Lie algebra L with S isomorphic to sl_2 and such that the nilpotent element x of L is contained in S. If such an algebra exists then it is returned, otherwise `fail` is returned.

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> b:= BasisVectors( Basis( L ) );;
gap> IsNilpotentElement( L, b[1] );
true
gap> FindSl2( L, b[1] );
<Lie algebra of dimension 3 over Rationals>
```

61.10 Universal Enveloping Algebras

1 ▶ `UniversalEnvelopingAlgebra( L )` A
▶ `UniversalEnvelopingAlgebra( L, B )` O

Returns the universal enveloping algebra of the Lie algebra L. The elements of this algebra are written on a Poincare-Birkhoff-Witt basis.

In the second form B must be a basis of L. If this second argument is given, then an isomorphic copy of the universal enveloping algebra is returned, generated by the images (in the universal enveloping algebra) of the elements of B.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> UL:= UniversalEnvelopingAlgebra( L );
<algebra-with-one of dimension infinity over Rationals>
gap> g:= GeneratorsOfAlgebraWithOne( UL );
[ [(1)*x.1], [(1)*x.2], [(1)*x.3] ]
gap> g[3]^2*g[2]^2*g[1]^2;
[(-4)*x.1*x.2*x.3^3+(1)*x.1^2*x.2^2*x.3^2+(2)*x.3^3+(2)*x.3^4]
```

61.11 Finitely Presented Lie Algebras

Finitely presented Lie algebras can be constructed from free Lie algebras by using the `/` constructor, i.e., `FL/[r1...rk]` is the quotient of the free Lie algebra `FL` by the ideal generated by the elements `r1...rk` of `FL`. If the finitely presented Lie algebra `K` happens to be finite dimensional then an isomorphic structure constants Lie algebra can be constructed by `NiceAlgebraMonomorphism(K)`, which returns a surjective homomorphism. The structure constants Lie algebra can then be accessed by calling `Range` for this map. Also limited computations with elements of the finitely presented Lie algebra are possible.

```
gap> L:= FreeLieAlgebra( Rationals, "s", "t" );
<Lie algebra over Rationals, with 2 generators>
gap> gL:= GeneratorsOfAlgebra( L );; s:= gL[1];; t:= gL[2];;
gap> K:= L/[ s*(s*t), t*(t*(s*t)), s*(t*(s*t))-t*(s*t) ];
<Lie algebra over Rationals, with 2 generators>
gap> h:= NiceAlgebraMonomorphism( K );
[ [(1)*s], [(1)*t] ] -> [ v.1, v.2 ]
gap> U:= Range( h );
<Lie algebra of dimension 3 over Rationals>
gap> IsLieNilpotent( U );
true
gap> gK:= GeneratorsOfAlgebra( K );
[ [(1)*s], [(1)*t] ]
gap> gK[1]*(gK[2]*gK[1]) = Zero( K );
true
```

1 ▶ `FpLieAlgebraByCartanMatrix( C )` F

Here C must be a Cartan matrix. The function returns the finitely-presented Lie algebra over the field of rational numbers defined by this Cartan matrix. By Serre's theorem, this Lie algebra is a semisimple Lie algebra, and its root system has Cartan matrix C.

```
gap> C:= [ [ 2, -1 ], [ -3, 2 ] ];;
gap> K:= FpLieAlgebraByCartanMatrix( C );
<Lie algebra over Rationals, with 6 generators>
gap> h:= NiceAlgebraMonomorphism( K );
[ [(1)*x1], [(1)*x2], [(1)*x3], [(1)*x4], [(1)*x5], [(1)*x6] ] ->
[ v.1, v.2, v.3, v.4, v.5, v.6 ]
gap> SemiSimpleType( Range( h ) );
"G2"
```

2 ▶ `NilpotentQuotientOfFpLieAlgebra(` *FpL*, *max* `)` F
▶ `NilpotentQuotientOfFpLieAlgebra(` *FpL*, *max*, *weights* `)` F

Here *FpL* is a finitely presented Lie algebra. Let K be the quotient of *FpL* by the $max{+}1$-th term of its lower central series. This function calculates a surjective homomorphism of *FpL* onto K. When called with the third argument *weights*, the k-th generator of *FpL* gets assigned the k-th element of the list *weights*. In that case a quotient is calculated of *FpL* by the ideal generated by all elements of weight $max{+}1$. If the list *weights* only consists of 1's then the two calls are equivalent. The default value of *weights* is a list (of length equal to the number of generators of *FpL*) consisting of 1's.

If the relators of *FpL* are homogeneous, then the resulting algebra is naturally graded.

```
gap> L:= FreeLieAlgebra( Rationals, "x", "y" );;
gap> g:= GeneratorsOfAlgebra(L);; x:= g[1]; y:= g[2];
(1)*x
(1)*y
gap> rr:=[((y*x)*x)*x-6*(y*x)*y, 3*((((y*x)*x)*x)*x)*x-20*(((y*x)*x)*x)*y ];
[ (-1)*(x*(x*(x*y)))+(6)*((x*y)*y),
  (-3)*(x*(x*(x*(x*(x*y)))))+(20)*(x*(x*((x*y)*y)))+(-20)*((x*(x*y))*(x*y)) ]
gap> K:= L/rr;
<Lie algebra over Rationals, with 2 generators>
gap> h:=NilpotentQuotientOfFpLieAlgebra(K, 50, [1,2] );
[ [(1)*x], [(1)*y] ] -> [ v.1, v.2 ]
```

```
gap> L:= Range( h );
<Lie algebra of dimension 50 over Rationals>
gap> Grading( L );
rec( min_degree := 1, max_degree := 50, source := Integers,
  hom_components := function( d ) ... end )
```

61.12 Modules over Lie Algebras and Their Cohomology

Representations of Lie algebras are dealt with in the same way as representations of ordinary algebras (see 60.10). In this section we mainly deal with modules over general Lie algebras and their cohomology. The next section is devoted to modules over semisimple Lie algebras.

1 ► `FaithfulModule( A )` A

returns a faithful finite-dimensional left-module over the algebra A. This is only implemented for associative algebras, and for Lie algebras of characteristic 0. (It may also work for certain Lie algebras of characteristic $p > 0$.)

```
gap> T:= EmptySCTable( 3, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 3 ]);
gap> L:= LieAlgebraByStructureConstants( Rationals, T );
<Lie algebra of dimension 3 over Rationals>
gap> V:= FaithfulModule( L );
<left-module over <Lie algebra of dimension 3 over Rationals>>
gap> vv:= BasisVectors( Basis( V ) );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ]
gap> x:= Basis( L )[3];
v.3
gap> List( vv, v -> x^v );
[ [ 0, 0, 0 ], [ 1, 0, 0 ], [ 0, 0, 0 ] ]
```

An s-cochain of a module V over a Lie algebra L, is an s-linear map

$$c : L \times \cdots \times L \to V \ (s \text{ factors } L)$$

that is skew-symmetric (meaning that if any of the arguments are interchanged, c changes to $-c$).

Let $\{x_1, \ldots, x_n\}$ be a basis of L. Then any s-cochain is determined by the values $c(x_{i_1}, \ldots, x_{i_s})$, where $1 \leq i_1 < i_2 < \cdots < i_s \leq \dim L$. Now this value again is a linear combination of basis elements of V: $c(x_{i_1}, \ldots, x_{i_s}) = \sum \lambda^k_{i_1,\ldots,i_s} v_k$. Denote the dimension of V by r. Then we represent an s-cocycle by a list of r lists. The j-th of those lists consists of entries of the form

$$[[i_1, i_2, \ldots, i_s], \lambda^j_{i_1,\ldots,i_s}]$$

where the coefficient on the second position is non-zero. (We only store those entries for which this coefficient is non-zero.) It follows that every s-tuple $(i_1, \ldots, i_s)$ gives rise to r basis elements.

So the zero cochain is represented by a list of the form $[[], [], \ldots, []]$. Furthermore, if V is, e.g., 4-dimensional, then the 2-cochain represented by

```
[ [ [ [1,2], 2] ], [ ], [ [ [1,2], 1/2 ] ], [ ] ]
```

maps the pair (x_1, x_2) to $2v_1 + 1/2v_3$ (where v_1 is the first basis element of V, and v_3 the third), and all other pairs to zero.

By definition, 0-cochains are constant maps $c(x) = v_c \in V$ for all $x \in L$. So 0-cochains have a different representation: they are just represented by the list $[v_c]$.

Cochains are constructed using the function `Cochain` (see 61.12.3), if c is a cochain, then its corresponding list is returned by `ExtRepOfObj( c )`.

2 ▶ `IsCochain( obj )` C
▶ `IsCochainCollection( obj )` C

Categories of cochains and of collections of cochains.

3 ▶ `Cochain( V, s, obj )` O

Constructs a s-cochain given by the data in *obj*, with respect to the Lie algebra module V. If s is non-zero, then *obj* must be a list.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );
<3-dimensional left-module over <Lie algebra of dimension 3 over Rationals>>
gap> c1:= Cochain( V, 2, [ [ [ [ 1, 3 ], -1 ] ], [ ], [ [ [ 2, 3 ], 1/2 ] ] ]);
<2-cochain>
gap> ExtRepOfObj( c1 );
[ [ [ [ 1, 3 ], -1 ] ], [  ], [ [ [ 2, 3 ], 1/2 ] ] ]
gap> c2:= Cochain( V, 0, Basis( V )[1] );
<0-cochain>
gap> ExtRepOfObj( c2 );
v.1
gap> IsCochain( c2 );
true
```

4 ▶ `CochainSpace( V, s )` O

Returns the space of all s-cochains with respect to V.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );;
gap> C:=CochainSpace( V, 2 );
<vector space of dimension 9 over Rationals>
gap> BasisVectors( Basis( C ) );
[ <2-cochain>, <2-cochain>, <2-cochain>, <2-cochain>, <2-cochain>,
  <2-cochain>, <2-cochain>, <2-cochain>, <2-cochain> ]
gap> ExtRepOfObj( last[1] );
[ [ [ [ 1, 2 ], 1 ] ], [  ], [  ] ]
```

5 ▶ `ValueCochain( c, y1, y2, ..., ys )` F

Here c is an s-cochain. This function returns the value of c when applied to the s elements *y1* to *ys* (that lie in the Lie algebra acting on the module corresponding to c). It is also possible to call this function with two arguments: first c and then the list containing *y1*, ..., *ys*.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );;
gap> C:= CochainSpace( V, 2 );;
gap> c:= Basis( C )[1];
<2-cochain>
gap>  ValueCochain( c, Basis(L)[2], Basis(L)[1] );
(-1)*v.1
```

6 ▶ `LieCoboundaryOperator( c )` V

This is a function that takes an s-cochain, and returns an $s+1$-cochain. The coboundary operator is applied.

```
gap> L:= SimpleLieAlgebra( "A", 1, Rationals );;
gap> V:= AdjointModule( L );;
gap> C:= CochainSpace( V, 2 );;
gap> c:= Basis( C )[1];;
gap> c1:= LieCoboundaryOperator( c );
<3-cochain>
gap> c2:= LieCoboundaryOperator( c1 );
<4-cochain>
```

7 ► `Cocycles( V, s )` O

is the space of all s-cocycles with respect to the Lie algebra module V. That is the kernel of the coboundary operator when restricted to the space of s-cochains.

8 ► `Coboundaries( V, s )` O

is the space of all s-coboundaries with respect to the Lie algebra module V. That is the image of the coboundary operator, when applied to the space of *s-1*-cochains. By definition the space of all 0-coboundaries is zero.

```
gap> T:= EmptySCTable( 3, 0, "antisymmetric" );;
gap> SetEntrySCTable( T, 1, 2, [ 1, 3 ] );
gap> L:= LieAlgebraByStructureConstants( Rationals, T );;
gap> V:= FaithfulModule( L );
<left-module over <Lie algebra of dimension 3 over Rationals>>
gap> Cocycles( V, 2 );
<vector space of dimension 7 over Rationals>
gap> Coboundaries( V, 2 );
<vector space over Rationals, with 9 generators>
gap> Dimension( last );
5
```

61.13 Modules over Semisimple Lie Algebras

This section contains functions for calculating information on representations of semisimple Lie algebras. First we have some functions for calculating some combinatorial data (set of dominant weights, the dominant character, the decomposition of a tensor product, the dimension of a highest-weight module). Then there is a function for creating an admissible lattice in the universal enveloping algebra of a semisimple Lie algebra. Finally we have a function for constructing a highest-weight module over a semisimple Lie algebra.

1 ► `DominantWeights( R, maxw )` O

Returns a list consisting of two lists. The first of these contains the dominant weights (written on the basis of fundamental weights) of the irreducible highest-weight module over the Lie algebra with root system R. The i-th element of the second list is the level of the i-th dominant weight. (Where level is defined as follows. For a weight μ we write $\mu = \lambda - \sum_i k_i\alpha_i$, where the α_i are the simple roots, and λ the highest weight. Then the level of μ is $\sum_i k_i$.

2 ► `DominantCharacter( L, maxw )` O
► `DominantCharacter( R, maxw )` O

For a highest weight *maxw* and a semisimple Lie algebra L, this returns the dominant weights of the highest-weight module over L, with highest weight *maxw*. The output is a list of two lists, the first list contains the dominant weights; the second list contains their multiplicities.

The first argument can also be a root system, in which case the dominant character of the highest-weight module over the corresponding semisimple Lie algebra is returned.

3 ► `DecomposeTensorProduct( L, w1, w2 )` O

Here L is a semisimple Lie algebra and *w1*, *w2* are dominant weights. Let V_i be the irreducible highest-weight module over L with highest weight w_i for $i = 1, 2$. Let $W = V_1 \otimes V_2$. Then in general W is a reducible L-module. Now this function returns a list of two lists. The first of these is the list of highest weights of the irreducible modules occurring in the decomposition of W as a direct sum of irreducible modules. The second list contains the multiplicities of these weights (i.e., the number of copies of the irreducible module with the corresponding highest weight that occur in W). The algorithm uses Klimyk's formula (see [Kli68] or [Kli66] for the original Russian version).

4 ► `DimensionOfHighestWeightModule( L, w )` O

Here L is a semisimple Lie algebra, and w a dominant weight. This function returns the dimension of the highest-weight module over L with highest weight w. The algorithm uses Weyl's dimension formula.

```
gap> L:= SimpleLieAlgebra( "F", 4, Rationals );;
gap> R:= RootSystem( L );;
gap> DominantWeights( R, [ 1, 1, 0, 0 ] );
[ [ [ 1, 1, 0, 0 ], [ 2, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ],
      [ 1, 0, 0, 0 ], [ 0, 0, 0, 0 ] ], [ 0, 3, 4, 8, 11, 19 ] ]
gap> DominantCharacter( L, [ 1, 1, 0, 0 ] );
[ [ [ 1, 1, 0, 0 ], [ 2, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 0, 0 ],
      [ 1, 0, 0, 0 ], [ 0, 0, 0, 0 ] ], [ 1, 1, 4, 6, 14, 21 ] ]
gap> DecomposeTensorProduct( L, [ 1, 0, 0, 0 ], [ 0, 0, 1, 0 ] );
[ [ [ 1, 0, 1, 0 ], [ 1, 0, 0, 0 ], [ 0, 0, 0, 1 ], [ 0, 1, 0, 0 ],
      [ 2, 0, 0, 0 ], [ 0, 0, 1, 0 ], [ 1, 1, 0, 0 ] ],
  [ 1, 1, 1, 1, 1, 1, 1 ] ]
gap> DimensionOfHighestWeightModule( L, [ 1, 2, 3, 4 ] );
79316832731136
```

Let L be a semisimple Lie algebra over a field of characteristic 0, and let R be its root system. For a positive root α we let x_α and y_α be positive and negative root vectors respectively, both from a fixed Chevalley basis of L. Furthermore, $h_1, \ldots, h_l$ are the Cartan elements from the same Chevalley basis. Also we set

$$x_\alpha^{(n)} = \frac{x_\alpha^n}{n!}, \qquad y_\alpha^{(n)} = \frac{y_\alpha^n}{n!}.$$

Furthermore, let $\alpha_1, \ldots, \alpha_s$ denote the positive roots of R. For multi-indices $N = (n_1, \ldots, n_s)$, $M = (m_1, \ldots, m_s)$ and $K = (k_1, \ldots, k_s)$ (where $n_i, m_i, k_i \geq 0$) set

$$\begin{aligned} x^N &= x_{\alpha_1}^{(n_1)} \cdots x_{\alpha_s}^{(n_s)}, \\ y^M &= y_{\alpha_1}^{(m_1)} \cdots y_{\alpha_s}^{(m_s)}, \\ h^K &= \tbinom{h_1}{k_1} \cdots \tbinom{h_l}{k_l} \end{aligned}$$

Then by a theorem of Kostant, the $x_\alpha^{(n)}$ and $y_\alpha^{(n)}$ generate a subring of the universal enveloping algebra $U(L)$ spanned (as a free Z-module) by the elements

$$y^M h^K x^N$$

(see, e.g., [Hum72] or [Hum78], Section 26) So by the Poincare-Birkhoff-Witt theorem this subring is a lattice in $U(L)$. Furthermore, this lattice is invariant under the $x_\alpha^{(n)}$ and $y_\alpha^{(n)}$. Therefore, it is called an admissible lattice in $U(L)$.

The next functions enable us to construct the generators of such an admissible lattice.

5 ▶ `IsUEALatticeElement( obj )` C
▶ `IsUEALatticeElementCollection( obj )` C
▶ `IsUEALatticeElementFamily( fam )` C

is the category of elements of an admissible lattice in the universal enveloping algebra of a semisimple Lie algebra `L`.

6 ▶ `LatticeGeneratorsInUEA( L )` A

Here L must be a semisimple Lie algebra of characteristic 0. This function returns a list of generators of an admissible lattice in the universal enveloping algebra of L, relative to the Chevalley basis contained in `ChevalleyBasis( L )`. First are listed the negative root vectors (denoted by $y_1, \ldots, y_s$), then the positive root vectors (denoted by $x_1, \ldots, x_s$). At the end of the list there are the Cartan elements. They are printed as `( hi/1 )`, which means

$$\binom{h_i}{1}.$$

In general the printed form `( hi/ k )` means

$$\binom{h_i}{k}.$$

Also $y_i^{(m)}$ is printed as `yi^(m)`, which means that entering `yi^m` at the GAP prompt results in the output `m!*yi^(m)`.

Products of lattice generators are collected using the following order: first come the $y_i^{(m_i)}$ (in the same order as the positive roots), then the $\binom{h_i}{k_i}$, and then the $x_i^{(n_i)}$ (in the same order as the positive roots).

7 ▶ `ObjByExtRep( F, descr )` O

creates an object in the family F which has the external representation *descr*.

An UEALattice element is represented by a list of the form

[m1, c1, m2, c2,.....]

where the c1,c2 etc. are coefficients, and the m1, m2 etc. monomials. A monomial is a list of the form [ind1, e1, ind2, e2, ...] where ind1, ind2 are indices, and e1, e2 etc. are exponents. Let N be the number of positive roots of the underlying Lie algebra. The indices lie between 1 and $\dim L$. If an index lies between 1 and N, then it represents a negative root vector (corresponding to the root `NegativeRoots( R )[ind]`, where R is the root system of L). This leads to a factor `yind1^(e1)` in the printed form of the monomial (which equals z^e1/e1!, where z is a basis element of L). If an index lies between N+1 and 2N, then it represents a positive root vector. Finally, if ind lies between 2N+1 and 2N+rank, then it represents an element of the Cartan subalgebra. This is printed as (h_1/ e_1), meaning h_1 choose e_1 (h_1,...,h_rank are the canonical Cartan generators).

The zero element is represented by the empty list, the identity element by the list [[ ], 1].

```
gap> L:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> g:=LatticeGeneratorsInUEA( L );
[ y1, y2, y3, y4, y5, y6, x1, x2, x3, x4, x5, x6, ( h13/1 ), ( h14/1 ) ]
gap> IsUEALatticeElement( g[1] );
true
gap> g[1]^3;
6*y1^(3)
gap> q:= g[7]*g[1]^2;
-2*y1+2*y1*( h13/1 )+2*y1^(2)*x1
gap> ExtRepOfObj( q );
[ [ 1, 1 ], -2, [ 1, 1, 13, 1 ], 2, [ 1, 2, 7, 1 ], 2 ]
```

8 ► IsWeightRepElement(*obj*) C
► IsWeightRepElementCollection(*obj*) C
► IsWeightRepElementFamily(*fam*) C

Is a category of vectors, that is used to construct elements of highest-weight modules (by HighestWeightModule).

WeightRepElements are represented by a list of the form [v1, c1, v2, c2, ...], where the v*i* are basis vectors, and the c*i* coefficients. Furthermore a basis vector v is a weight vector. It is represented by a list of form [*k*, *mon*, *wt*], where *k* is an integer (the basis vectors are numbered from 1 to dim *V*, where *V* is the highest weight module), *mon* is an UEALatticeElement (which means that the result of applying *mon* to a highest weight vector is v) and *wt* is the weight of *v*. A WeightRepElement is printed as *mon**v0, where v0 denotes a fixed highest weight vector.

If *v* is a WeightRepElement, then ExtRepOfObj(*v*) returns the corresponding list, and if *list* is such a list and *fam* a WeightRepElementFamily, then ObjByExtRep(*list*, *fam*) returns the corresponding WeightRepElement.

9 ► HighestWeightModule(*L*, *wt*) F

returns the highest weight module with highest weight *wt* of the semisimple Lie algebra *L* of characteristic 0.

Note that the elements of such a module lie in the category IsLeftAlgebraModuleElement (and in particular they do not lie in the category IsWeightRepElement). However, if v is an element of such a module, then ExtRepOfObj(v) is a WeightRepElement.

Note that for the following examples of this chapter we increase the line length limit from its default value 80 to 81 in order to make some long output expressions fit into the lines.

```
gap> SizeScreen([ 81, ]);;
gap> K1:= SimpleLieAlgebra( "G", 2, Rationals );;
gap> K2:= SimpleLieAlgebra( "B", 2, Rationals );;
gap> L:= DirectSumOfAlgebras( K1, K2 );
<Lie algebra of dimension 24 over Rationals>
gap> V:= HighestWeightModule( L, [ 0, 1, 1, 1 ] );
<224-dimensional left-module over <Lie algebra of dimension 24 over Rationals>>
gap> vv:= GeneratorsOfLeftModule( V );;
gap> vv[100];
y5*y7*y10*v0
gap> e:= ExtRepOfObj( vv[100] );
y5*y7*y10*v0
gap> ExtRepOfObj( e );
[ [ 100, y5*y7*y10, [ -3, 2, -1, 1 ] ], 1 ]
gap> Basis(L)[17]^vv[100];
-1*y5*y7*y8*v0-1*y5*y9*v0
```

61.14 Tensor Products and Exterior and Symmetric Powers

1 ► TensorProductOfAlgebraModules(*list*) O
► TensorProductOfAlgebraModules(*V*, *W*) O

Here the elements of *list* must be algebra modules. The tensor product is returned as an algebra module.

```
gap> L:= SimpleLieAlgebra("G",2,Rationals);;
gap> V:= HighestWeightModule( L, [ 1, 0 ] );;
gap> W:= TensorProductOfAlgebraModules( [ V, V, V ] );
<343-dimensional left-module over <Lie algebra of dimension 14 over Rationals>>
gap> w:= Basis(W)[1];
1*(1*v0<x>1*v0<x>1*v0)
gap> Basis(L)[1]^w;
<0-tensor>
gap> Basis(L)[7]^w;
1*(1*v0<x>1*v0<x>y1*v0)+1*(1*v0<x>y1*v0<x>1*v0)+1*(y1*v0<x>1*v0<x>1*v0)
```

2 ► `ExteriorPowerOfAlgebraModule(` *V*, *k* `)` O

Here *V* must be an algebra module, defined over a Lie algebra. This function returns the *k*-th exterior power of *V* as an algebra module.

```
gap> L:= SimpleLieAlgebra("G",2,Rationals);;
gap> V:= HighestWeightModule( L, [ 1, 0 ] );;
gap> W:= ExteriorPowerOfAlgebraModule( V, 3 );
<35-dimensional left-module over <Lie algebra of dimension 14 over Rationals>>
gap> w:= Basis(W)[1];
1*(1*v0/\y1*v0/\y3*v0)
gap> Basis(L)[10]^w;
1*(1*v0/\y1*v0/\y6*v0)+1*(1*v0/\y3*v0/\y5*v0)+1*(y1*v0/\y3*v0/\y4*v0)
```

3 ► `SymmetricPowerOfAlgebraModule(` *V*, *k* `)` O

Here *V* must be an algebra module. This function returns the *k*-th symmetric power of *V* (as an algebra module).

```
gap> L:= SimpleLieAlgebra("G",2,Rationals);;
gap> V:= HighestWeightModule( L, [ 1, 0 ] );;
gap> W:= SymmetricPowerOfAlgebraModule( V, 3 );
<84-dimensional left-module over <Lie algebra of dimension 14 over Rationals>>
gap> w:= Basis(W)[1];
1*(1*v0.1*v0.1*v0)
gap> Basis(L)[2]^w;
<0-symmetric element>
gap> Basis(L)[7]^w;
3*(1*v0.1*v0.y1*v0)
```

4 ► `DirectSumOfAlgebraModules(` *list* `)` O
► `DirectSumOfAlgebraModules(` *V*, *W* `)` O

Here *list* must be a list of algebra modules. This function returns the direct sum of the elements in the list (as an algebra module). The modules must be defined over the same algebras.

In the second form is short for `DirectSumOfAlgebraModules( [` *V*, *W* `] )`

```
gap> L:= SimpleLieAlgebra( "C", 3, Rationals );;
gap> V:= HighestWeightModule( L, [ 1, 1, 0 ] );
<64-dimensional left-module over <Lie algebra of dimension 21 over Rationals>>
gap> W:= HighestWeightModule( L, [ 0, 0, 2 ] );
<84-dimensional left-module over <Lie algebra of dimension 21 over Rationals>>
gap> U:= DirectSumOfAlgebraModules( V, W );
<148-dimensional left-module over <Lie algebra of dimension 21 over Rationals>>
gap> SizeScreen([ 80, ]);;
```

62 Finitely Presented Algebras

Currently the GAP library contains only few functions dealing with general finitely presented algebras, so this file is merely a placeholder.

The special case of finitely presented **Lie** algebras is described in 61.11, and there is also a GAP package `fplsa` for computing structure constants of **f**initely **p**resented **L**ie (**s**uper)**a**lgebras.

63 Magma Rings

Given a magma M then the **free magma ring** (or **magma ring** for short) RM of M over a ring-with-one R is the set of finite sums $\sum_{i \in I} r_i m_i$ with $r_i \in R$, and $m_i \in M$. With the obvious addition and R-action from the left, RM is a free R-module with R-basis M, and with the usual convolution product, RM is a ring.

Typical examples of free magma rings are

- (multivariate) polynomial rings (see 64.14), where the magma is a free abelian monoid generated by the indeterminates,
- group rings (see 63.1.5), where the magma is a group,
- Laurent polynomial rings, which are group rings of the free abelian groups generated by the indeterminates,
- free algebras and free associative algebras, with or without one, where the magma is a free magma or a free semigroup, or a free magma-with-one or a free monoid, respectively.

Note that a free Lie algebra is **not** a magma ring, because of the additional relations given by the Jacobi identity; see 63.4 for a generalization of magma rings that covers such structures.

The coefficient ring R and the magma M cannot be regarded as subsets of RM, hence the natural **embeddings** of R and M into RM must be handled via explicit embedding maps (see 63.3). Note that in a magma ring, the addition of elements is in general different from an addition that may be defined already for the elements of the magma; for example, the addition in the group ring of a matrix group does in general **not** coincide with the addition of matrices. Consider the following example.

```
gap> a:= Algebra( GF(2), [ [ [ Z(2) ] ] ] );;  Size( a );
2
gap> rm:= FreeMagmaRing( GF(2), a );;
gap> emb:= Embedding( a, rm );;
gap> z:= Zero( a );;  o:= One( a );;
gap> imz:= z ^ emb;  IsZero( imz );
(Z(2)^0)*[ [ 0*Z(2) ] ]
false
gap> im1:= ( z + o ) ^ emb;
(Z(2)^0)*[ [ Z(2)^0 ] ]
gap> im2:= z ^ emb + o ^ emb;
(Z(2)^0)*[ [ 0*Z(2) ] ]+(Z(2)^0)*[ [ Z(2)^0 ] ]
gap> im1 = im2;
false
```

63.1 Free Magma Rings

1 ▶ `FreeMagmaRing( R, M )` F

is a free magma ring over the ring R, free on the magma M.

2 ▶ `GroupRing( R, G )` F

is the group ring of the group G, over the ring R.

3 ▶ `IsFreeMagmaRing( D )` C

A domain lies in the category `IsFreeMagmaRing` if it has been constructed as a free magma ring. In particular, if D lies in this category then the operations `LeftActingDomain` (see 55.1.11) and `UnderlyingMagma` (see 63.1.6) are applicable to D, and yield the ring R and the magma M such that D is the magma ring RM.

So being a magma ring in GAP includes the knowledge of the ring and the magma. Note that a magma ring RM may abstractly be generated as a magma ring by a magma different from the underlying magma M. For example, the group ring of the dihedral group of order 8 over the field with 3 elements is also spanned by a quaternion group of order 8 over the same field.

```
gap> d8:= DihedralGroup( 8 );
<pc group of size 8 with 3 generators>
gap> rm:= FreeMagmaRing( GF(3), d8 );
<algebra-with-one over GF(3), with 3 generators>
gap> emb:= Embedding( d8, rm );;
gap> gens:= List( GeneratorsOfGroup( d8 ), x -> x^emb );;
gap> x1:= gens[1] + gens[2];;
gap> x2:= ( gens[1] - gens[2] ) * gens[3];;
gap> x3:= gens[1] * gens[2] * ( One( rm ) - gens[3] );;
gap> g1:= x1 - x2 + x3;;
gap> g2:= x1 + x2;;
gap> q8:= Group( g1, g2 );;
gap> Size( q8 );
8
gap> ForAny( [ d8, q8 ], IsAbelian );
false
gap> List( [ d8, q8 ], g -> Number( AsList( g ), x -> Order( x ) = 2 ) );
[ 5, 1 ]
gap> Dimension( Subspace( rm, q8 ) );
8
```

4 ▶ `IsFreeMagmaRingWithOne( obj )` C

5 ▶ `IsGroupRing( obj )` P

A **group ring** is a magma ring where the underlying magma is a group.

6 ▶ `UnderlyingMagma( RM )` A

7 ▶ `AugmentationIdeal( RG )` A

is the augmentation ideal of the group ring RG, i.e., the kernel of the trivial representation of RG.

63.2 Elements of Free Magma Rings

1 ▶ `IsElementOfFreeMagmaRing( obj )` C
▶ `IsElementOfFreeMagmaRingCollection( obj )` C

2 ▶ `IsElementOfFreeMagmaRingFamily( Fam )` C

Elements of families in this category have trivial normalisation, i.e., efficient methods for `\=` and `\<`.

In order to treat elements of free magma rings uniformly, also without an external representation, the attributes `CoefficientsAndMagmaElements` (see 63.2.3) and `ZeroCoefficient` (see 63.2.4) were introduced that allow one to "take an element of an arbitrary magma ring into pieces".

Conversely, for constructing magma ring elements from coefficients and magma elements, `ElementOfMagmaRing` (see 63.2.5) can be used. (Of course one can also embed each magma element into the magma ring, see 63.3, and then form the linear combination, but many unnecessary intermediate elements are created this way.)

3 ▶ `CoefficientsAndMagmaElements( elm )` A

is a list that contains at the odd positions the magma elements, and at the even positions their coefficients in the element *elm*.

4 ▶ `ZeroCoefficient( elm )` A

For an element *elm* of a magma ring (modulo relations) RM, `ZeroCoefficient` returns the zero element of the coefficient ring R.

5 ▶ `ElementOfMagmaRing( Fam, zerocoeff, coeffs, mgmelms )` O

`ElementOfMagmaRing` returns the element $\sum_{i=1}^{n} c_i m_i'$, where $coeffs = [c_1, c_2, \ldots, c_n]$ is a list of coefficients, $mgmelms = [m_1, m_2, \ldots, m_n]$ is a list of magma elements, and m_i' is the image of m_i under an embedding of a magma containing m_i into a magma ring whose elements lie in the family *Fam*. *zerocoeff* must be the zero of the coefficient ring containing the c_i.

63.3 Natural Embeddings related to Magma Rings

Neither the coefficient ring R nor the magma M are regarded as subsets of the magma ring RM, so one has to use **embeddings** (see 31.1.10) explicitly whenever one needs for example the magma ring element corresponding to a given magma element. Here is an example.

```
gap> f:= Rationals;;  g:= SymmetricGroup( 3 );;
gap> fg:= FreeMagmaRing( f, g );
<algebra-with-one over Rationals, with 2 generators>
gap> Dimension( fg );
6
gap> gens:= GeneratorsOfAlgebraWithOne( fg );
[ (1)*(1,2,3), (1)*(1,2) ]
gap> ( 3*gens[1] - 2*gens[2] ) * ( gens[1] + gens[2] );
(-2)*()+(3)*(2,3)+(3)*(1,3,2)+(-2)*(1,3)
gap> One( fg );
(1)*()
gap> emb:= Embedding( g, fg );;
gap> elm:= (1,2,3)^emb;  elm in fg;
(1)*(1,2,3)
true
gap> new:= elm + One( fg );
(1)*()+(1)*(1,2,3)
```

```
gap> new^2;
(1)*()+(2)*(1,2,3)+(1)*(1,3,2)
gap> emb2:= Embedding( f, fg );;
gap> elm:= One( f )^emb2;  elm in fg;
(1)*()
true
```

63.4 Magma Rings modulo Relations

A more general construction than that of free magma rings allows one to create rings that are not free R-modules on a given magma M but arise from the magma ring RM by factoring out certain identities. Examples for such structures are finitely presented (associative) algebras and free Lie algebras (see 61.2.3).

In GAP, the use of magma rings modulo relations is limited to situations where a normal form of the elements is known and where one wants to guarantee that all elements actually constructed are in normal form. (In particular, the computation of the normal form must be cheap.) This is because the methods for comparing elements in magma rings modulo relations via `\=` and `\<` just compare the involved coefficients and magma elements, and also the vector space functions regard those monomials as linearly independent over the coefficients ring that actually occur in the representation of an element of a magma ring modulo relations.

Thus only very special finitely presented algebras will be represented as magma rings modulo relations, in general finitely presented algebras are dealt with via the mechanism described in Chapter 62.

1 ▶ `IsElementOfMagmaRingModuloRelations(` *obj* `)` C
▶ `IsElementOfMagmaRingModuloRelationsCollection(` *obj* `)` C

This category is used, e. g., for elements of free Lie algebras.

2 ▶ `IsElementOfMagmaRingModuloRelationsFamily(` *Fam* `)` C

3 ▶ `NormalizedElementOfMagmaRingModuloRelations(` *F*, *descr* `)` O

Let F be a family of magma ring elements modulo relations, and *descr* the description of an element in a magma ring modulo relations. `NormalizedElementOfMagmaRingModuloRelations` returns a description of the same element, but normalized w.r.t. the relations. So two elements are equal if and only if the result of `NormalizedElementOfMagmaRingModuloRelations` is equal for their internal data, that is, `CoefficientsAndMagmaElements` will return the same for the corresponding two elements.

`NormalizedElementOfMagmaRingModuloRelations` is allowed to return *descr* itself, it need not make a copy. This is the case for example in the case of free magma rings.

4 ▶ `IsMagmaRingModuloRelations(` *obj* `)` C

A GAP object lies in the category `IsMagmaRingModuloRelations` if it has been constructed as a magma ring modulo relations. Each element of such a ring has a unique normal form, so `CoefficientsAndMagmaElements` is well-defined for it.

This category is not inherited to factor structures, which are in general best described as finitely presented algebras, see Chapter 62.

63.5 Magma Rings modulo the Span of a Zero Element

1 ▶ `IsElementOfMagmaRingModuloSpanOfZeroFamily( Fam )` C

We need this for the normalization method, which takes a family as first argument.

2 ▶ `IsMagmaRingModuloSpanOfZero( RM )` C

3 ▶ `MagmaRingModuloSpanOfZero( R, M, z )` F

Let R be a ring, M a magma, and z an element of M with the property that $z * m = z$ for all $m \in M$. The element z could be called a "zero element" of M, but note that in general z cannot be obtained as `Zero( m )` for each $m \in M$, so this situation does not match the definition of `Zero` (see 30.10.3).

`MagmaRingModuloSpanOfZero` returns the magma ring RM modulo the relation given by the identification of z with zero. This is an example of a magma ring modulo relations, see 63.4.

63.6 Technical Details about the Implementation of Magma Rings

The **family** containing elements in the magma ring RM in fact contains all elements with coefficients in the family of elements of R and magma elements in the family of elements of M. So arithmetic operations with coefficients outside R or with magma elements outside M might create elements outside RM.

It should be mentioned that each call of `FreeMagmaRing` creates a new family of elements, so for example the elements of two group rings of permutation groups over the same ring lie in different families and therefore are regarded as different.

```
gap> g:= SymmetricGroup( 3 );;
gap> h:= AlternatingGroup( 3 );;
gap> IsSubset( g, h );
true
gap> f:= GF(2);;
gap> fg:= GroupRing( f, g );
<algebra-with-one over GF(2), with 2 generators>
gap> fh:= GroupRing( f, h );
<algebra-with-one over GF(2), with 1 generators>
gap> IsSubset( fg, fh );
false
gap> o1:= One( fh );  o2:= One( fg );  o1 = o2;
(Z(2)^0)*()
(Z(2)^0)*()
false
gap> emb:= Embedding( g, fg );;
gap> im:= Image( emb, h );
<group of size 3 with 1 generators>
gap> IsSubset( fg, im );
true
```

There is **no** generic **external representation** for elements in an arbitrary free magma ring. For example, polynomials are elements of a free magma ring, and they have an external representation relying on the special form of the underlying monomials. On the other hand, elements in a group ring of a permutation group do not admit such an external representation.

For convenience, magma rings constructed with `FreeAlgebra`, `FreeAssociativeAlgebra`, `FreeAlgebraWithOne`, and `FreeAssociativeAlgebraWithOne` support an external representation of their elements, which is defined as a list of length 2, the first entry being the zero coefficient, the second being a list with the

external representations of the magma elements at the odd positions and the corresponding coefficients at the even positions.

As the above examples show, there are several possible representations of magma ring elements, the representations used for polynomials (see 64) as well as the default representation `IsMagmaRingObjDefaultRep` of magma ring elements. The latter simply stores the zero coefficient and a list containing the coefficients of the element at the even positions and the corresponding magma elements at the odd positions, where the succession is compatible with the ordering of magma elements via .

64 Polynomials and Rational Functions

Let R be a commutative ring-with-one. We call a free associative algebra A over R a **polynomial ring** over R. The free generators of A are called **indeterminates**, they are usually denoted by $x_1, x_2, \ldots$. The number of indeterminates is called the **rank** of A. The elements of A are called **polynomials**. Products of indeterminates are called **monomials**, every polynomial can be expressed as a finite sum of products of monomials with ring elements in a form like $r_{1,0}x_1 + r_{1,1}x_1x_2 + r_{0,1}x_2 + \cdots$ with $r_{i,j} \in R$.

A polynomial ring of rank 1 is called an **univariate** polynomial ring, its elements are **univariate polynomials**.

Polynomial rings of smaller rank naturally embed in rings of higher rank; if S is a subring of R then a polynomial ring over S naturally embeds in a polynomial ring over R of the same rank. Note however that GAP does not consider R as a subset of a polynomial ring over R; for example the zero of R (0) and the zero of the polynomial ring ($0x^0$) are different objects.

Internally, indeterminates are represented by positive integers, but it is possible to give names to them to have them printed in a nicer way. Beware, however that there is not necessarily any relation between the way an indeterminate is called and the way it is printed. See section 64.1 for details.

If R is an integral domain, the polynomial ring A over R is an integral domain as well and one can therefore form its quotient field Q. This field is called a **field of rational functions**. Again A embeds naturally into Q and GAP will perform this embedding implicitly. (In fact it implements the ring of rational functions over R.) To avoid problems with leading coefficients, however, R must be a unique factorization domain.

64.1 Indeterminates

GAP implements a polynomial ring with countably many indeterminates. These indeterminates can be referred to by positive integers. If only a number *num* of indeterminates is required they default to `[1..`*num*`]`.

It is possible to assign names to indeterminates. These names only provide a means for printing the indeterminates in a nice way, but have not necessary any relations to variable names. Indeterminates that have not been assigned a name will be printed as "`x_`*nr*".

It is possible to assign the **same** name to **different** indeterminates (though it is probably not a good idea to do so). Asking **twice** for an indeterminate with the name *nam* will produce **two different** indeterminates!

When asking for indeterminates with certain names, GAP usually will take the first indeterminates that are not yet named, name these accordingly and return them. Thus when asking for named indeterminates, no relation between names and indeterminate numbers can be guaranteed. The attribute `IndeterminateNumberOfLaurentPolynomial(`*indet*`)` will return the number of the indeterminate *indet*.

1 ▶ `Indeterminate(` *R*`,` `[`*nr*`]` `)` O
▶ `Indeterminate(` *R*`,` `[`*avoid*`]` `)` O
▶ `Indeterminate(` *R*`,` *name*`[,` *avoid*`]` `)` O
▶ `Indeterminate(` *fam*`,` *nr* `)` O

returns indeterminate number *nr* over the ring *R*. If *nr* is not given it defaults to 1. If the number is not specified a list *avoid* of indeterminates may be given. The function will return an indeterminate that is

guaranteed to be different from all the indeterminates in *avoid*. The third usage returns an indeterminate called *name* (also avoiding the indeterminates in *avoid* if given).

```
gap> a:=Indeterminate(GF(3));
x_1
gap> x:=Indeterminate(GF(3),"x");
x
gap> Indeterminate(GF(3),"x")=x;
false
gap> y:=Indeterminate(GF(3),"y");z:=Indeterminate(GF(3),"X");
y
X
gap> Indeterminate(GF(3),3);
y
gap> Indeterminate(GF(3),[y,z]);
x
```

2 ▶ `IndeterminateNumberOfUnivariateRationalFunction(` *rfun* `)` A

returns the number of the indeterminate in which the univariate rational function *rfun* is expressed. (This also provides a way to obtain the number of a given indeterminate.)

A constant rational function might not possess an indeterminate number. In this case `IndeterminateNumberOfUnivariateRationalFunction` will default to a value of 1. Therefore two univariate polynomials may be considered to be in the same univariate polynomial ring if their indeterminates have the same number or one if of them is constant. (see also 64.1.5 and 64.20.7).

3 ▶ `IndeterminateOfUnivariateRationalFunction(` *rfun* `)` A

returns the indeterminate in which the univariate rational function *rfun* is expressed. (cf. 64.1.2.)

```
gap> IndeterminateNumberOfUnivariateRationalFunction(z);
4
gap> IndeterminateOfUnivariateRationalFunction(z^5+z);
X
```

4 ▶ `IndeterminateName(` *fam*, *nr* `)` O
▶ `HasIndeterminateName(` *fam*, *nr* `)` O
▶ `SetIndeterminateName(` *fam*, *nr*, *name* `)` O

`SetIndeterminateName` assigns the name *name* to indeterminate *nr* in the rational functions family *fam*. It issues an error if the indeterminate was already named.

`IndeterminateName` returns the name of the *nr*-th indeterminate (and returns `fail` if no name has been assigned).

`HasIndeterminateName` tests whether indeterminate *nr* has already been assigned a name

```
gap> IndeterminateName(FamilyObj(x),3);
"y"
gap> HasIndeterminateName(FamilyObj(x),5);
false
gap> SetIndeterminateName(FamilyObj(x),10,"bla");
gap> Indeterminate(GF(3),10);
bla
```

As a convenience there is a special method installed for `SetName` that will assign a name to an indeterminate.

```
gap> a:=Indeterminate(GF(3),5);
x_5
gap> SetName(a,"ah");
gap> a^5+a;
ah^5+ah
```

5 ▶ `CIUnivPols(` *upol*`,` *upol* `)` F

This function (whose name stands for "CommonIndeterminateOfUnivariatePolynomials") takes two univariate polynomials as arguments. If both polynomials are given in the same indeterminate number *indnum* (in this case they are "compatible" as univariate polynomials) it returns *indnum*. In all other cases it returns `fail`. `CIUnivPols` also accepts if either polynomial is constant but formally expressed in another indeterminate, in this situation the indeterminate of the other polynomial is selected.

64.2 Operations for Rational Functions

The rational functions form a field, therefore all arithmetic operations are applicable to rational functions.

1 ▶ *f* `+` *g*
▶ *f* `-` *g*
▶ *f* `*` *g*
▶ *f* `/` *g*

```
gap> x:=Indeterminate(Rationals,1);;y:=Indeterminate(Rationals,2);;
gap> f:=3+x*y+x^5;;g:=5+x^2*y+x*y^2;;
gap> a:=g/f;
(x_1^2*x_2+x_1*x_2^2+5)/(x_1^5+x_1*x_2+3)
```

Note that the quotient f/g of two polynomials might be represented as a rational function again. If g is known to divide f the call `Quotient(`f`,`g`)` (see 54.1.10) should be used instead.

2 ▶ *f* `mod` *g*

For two Laurent polynomials f and g, f `mod` g is the Euclidean remainder (see 54.6.4) of f modulo g.

At the moment GAP does not contain a proper multivariate Gcd algorithm. Therefore it cannot be guaranteed that rational functions will always be represented as a quotient of coprime polynomials. In certain unfortunate situations this might lead to a degree explosion.

All polynomials as well as all the univariate polynomials in the same indeterminate form subrings of this field. If two rational functions are known to be in the same subring, the result will be expressed as element in this subring.

64.3 Comparison of Rational Functions

1 ▶ *f* `=` *g*

Two rational functions f and g are equal if the product `Numerator(`f`)`$*$`Denominator(`g`)` equals `Numerator(`g`)`$*$ `Denominator(`f`)`.

```
gap> x:=Indeterminate(Rationals,"x");;y:=Indeterminate(Rationals,"y");;
gap> f:=3+x*y+x^5;;g:=5+x^2*y+x*y^2;;
gap> a:=g/f;
(x^2*y+x*y^2+5)/(x^5+x*y+3)
gap> b:=(g*f)/(f^2);
(x^7*y+x^6*y^2+5*x^5+x^3*y^2+x^2*y^3+3*x^2*y+3*x*y^2+5*x*y+15)/(x^10+2*x^6*y+6\
*x^5+x^2*y^2+6*x*y+9)
gap> a=b;
true
```

2 ► f < g

The ordering of rational functions is defined in several steps. Monomials (products of indeterminates) are sorted first by degree, then lexicographically (with $x_1 > x_2$) (see 64.16.8). Products of monomials with ring elements ("terms") are compared first by their monomials and then by their coefficients.

```
gap> x>y;
true
gap> x^2*y<x*y^2;
false
gap> x*y<x^2*y;
true
gap> x^2*y < 5* y*x^2;
true
```

Polynomials are compared by comparing the largest terms in turn until they differ.

```
gap> x+y<y;
false
gap> x<x+1;
true
```

Rational functions are compared by comparing the polynomial `Numerator`$(f) *$ `Denominator`(g) with the polynomial `Numerator`$(g) *$ `Denominator`(f). (As the ordering of monomials used by GAP is invariant under multiplication this is independent of common factors in numerator and denominator.)

```
gap> f/g<g/f;
false
gap> f/g<(g*g)/(f*g);
false
```

For univariate polynomials this reduces to an ordering first by total degree and then lexicographically on the coefficients.

64.4 Properties and Attributes of Rational Functions

1 ► `IsPolynomialFunction(` *obj* `)` C
► `IsRationalFunction(` *obj* `)` C

A polynomial function is an element of a polynomial ring (not necessarily an UFD).

A rational function is an element of the quotient field of a polynomial ring over an UFD. It is represented as a quotient of two polynomials, its numerator (see 64.4.2) and its denominator (see 64.4.3)

2 ► `NumeratorOfRationalFunction(` *ratfun* `)` A

returns the nominator of the rational function *ratfun*.

As no proper multivariate gcd has been implemented yet, numerators and denominators are not guaranteed to be reduced!

3 ► `DenominatorOfRationalFunction(` *ratfun* `)` A

returns the denominator of the rational function *ratfun*.

As no proper multivariate gcd has been implemented yet, numerators and denominators are not guaranteed to be reduced!

```
gap> x:=Indeterminate(Rationals,1);;y:=Indeterminate(Rationals,2);;
gap> DenominatorOfRationalFunction((x*y+x^2)/y);
y
gap> NumeratorOfRationalFunction((x*y+x^2)/y);
x^2+x*y
```

4 ► `IsPolynomial(` *ratfun* `)` P

A polynomial is a rational functions whose denominator is one. (If the coefficients family forms a field this is equivalent to the denominator being constant.)

If the base family is not a field, it may be impossible to represent the quotient of a polynomial by a ring element as a polynomial again, but it will have to be represented as a rational function.

```
gap> IsPolynomial((x*y+x^2*y^3)/y);
true
gap> IsPolynomial((x*y+x^2)/y);
false
```

5 ► `AsPolynomial(` *poly* `)` A

If *poly* is a rational function that is a polynomial this attribute returns an equal rational function p such that p is equal to its numerator and the denominator of p is one.

```
gap> AsPolynomial((x*y+x^2*y^3)/y);
x^2*y^2+x
```

6 ► `IsUnivariateRationalFunction(` *ratfun* `)` P

A rational function is univariate if its numerator and its denominator are both polynomials in the same one indeterminate. The attribute `IndeterminateNumberOfUnivariateRationalFunction` can be used to obtain the number of this common indeterminate.

7 ► `CoefficientsOfUnivariateRationalFunction(` *rfun* `)` A

if *rfun* is a univariate rational function, this attribute returns a list [*ncof*,*dcof*,*val*] where *ncof* and *dcof* are coefficient lists of univariate polynomials n and d and a valuation *val* such that $rfun = x^{val} \cdot n/d$ where x is the variable with the number given by 64.1.2. Numerator and Denominator are guaranteed to be cancelled.

8 ► `IsUnivariatePolynomial(` *ratfun* `)` P

A univariate polynomial is a polynomial in only one indeterminate.

9 ► `CoefficientsOfUnivariatePolynomial(` *pol* `)` A

`CoefficientsOfUnivariatePolynomial` returns the coefficient list of the polynomial *pol*, sorted in ascending order.

10 ► `IsLaurentPolynomial(` *ratfun* `)` P

A Laurent polynomial is a univariate rational function whose denominator is a monomial. Therefore every univariate polynomial is a Laurent polynomial.

The attribute `CoefficientsOfLaurentPolynomial` (see 64.12.2) gives a compact representation as Laurent polynomial.

11 ▶ `IsConstantRationalFunction( ratfun )` P

A constant rational function is a function whose numerator and denominator are polynomials of degree 0.

All these tests are applicable to **every** rational function. Depending on the internal representation of the rational function, however some of these tests (in particular, univariateness) might be expensive in some cases.

For reasons of performance within algorithms it can be useful to use other attributes, which give a slightly more technical representation. See section 64.19 for details.

12 ▶ `IsPrimitivePolynomial( F, pol )` O

For a univariate polynomial *pol* of degree d in the indeterminate X, with coefficients in a finite field F with q elements, say, `IsPrimitivePolynomial` returns `true` if

1. *pol* divides $X^{q^d-1} - 1$, and
2. for each prime divisor p of $q^d - 1$, *pol* does not divide $X^{(q^d-1)/p} - 1$,

and `false` otherwise.

13 ▶ `SplittingField( f )` A

returns the smallest field which contains the coefficients of f and the roots of f.

64.5 Univariate Polynomials

Some of the operations are actually defined on the larger domain of Laurent polynomials (see 64.12). For this section you can simply ignore the word "Laurent" if it occurs in a description.

1 ▶ `UnivariatePolynomial( ring, cofs[, ind] )` O

constructs an univariate polynomial over the ring *ring* in the indeterminate *ind* with the coefficients given by *coefs*.

2 ▶ `UnivariatePolynomialByCoefficients( fam, cofs, ind )` O

constructs an univariate polynomial over the coefficients family *fam* and in the indeterminate *ind* with the coefficients given by *coefs*. This function should be used in algorithms to create polynomials as it avoids overhead associated with `UnivariatePolynomial`.

3 ▶ `DegreeOfLaurentPolynomial( pol )` A

The degree of a univariate (Laurent) polynomial *pol* is the largest exponent n of a monomial x^n of *pol*.

```
gap> p:=UnivariatePolynomial(Rationals,[1,2,3,4],1);
4*x^3+3*x^2+2*x+1
gap> UnivariatePolynomialByCoefficients(FamilyObj(1),[9,2,3,4],73);
4*x_73^3+3*x_73^2+2*x_73+9
gap> CoefficientsOfUnivariatePolynomial(p);
[ 1, 2, 3, 4 ]
gap> DegreeOfLaurentPolynomial(p);
3
gap> IndeterminateNumberOfLaurentPolynomial(p);
1
gap> IndeterminateOfLaurentPolynomial(p);
x
```

We remark that some functions for multivariate polynomials (which will be defined in the following sections) permit a different syntax for univariate polynomials which drops the requirement to specify the indeterminate. Examples are `Value`, `Discriminant`, `Derivative`, `LeadingCoefficient` and `LeadingMonomial`:

```
gap> Value(p,Z(5));
Z(5)^2
gap> LeadingCoefficient(p);
4
gap> Derivative(p);
12*x^2+6*x+2
```

4 ▶ `RootsOfUPol(` *upol* `)` F
▶ `RootsOfUPol(` *field*`,` *upol* `)` F
▶ `RootsOfUPol( "split",` *upol* `)` F

This function returns a list of all roots of the univariate polynomial *upol* in its default domain. If *field* is given the roots over *field* are taken, if the first parameter is the string `"split"` the field is taken to be the splitting field of the polynomial.

```
gap> RootsOfUPol(50-45*x-6*x^2+x^3);
[ 10, 1, -5 ]
```

5 ▶ `UnivariatenessTestRationalFunction(` *f* `)` F

takes a rational function *f* and tests whether it is univariate or even a Laurent polynomial. It returns a list [*isunivariate*, *indet*, *islaurent*, *cofs*] where *indet* is the indeterminate number and *cofs* (if applicable) the coefficients lists. The list *cofs* is the `CoefficientsOfLaurentPolynomial` if *islaurent* is `true` and the `CoefficientsOfUnivariateRationalFunction` if *islaurent* is `false` and *isunivariate* `true`. As there is no proper multivariate gcd, it might return `fail` for *isunivariate*.

The info class for univariate polynomials is `InfoPoly`.

64.6 Polynomials as Univariate Polynomials in one Indeterminate

1 ▶ `DegreeIndeterminate(` *pol*, *ind* `)` O
▶ `DegreeIndeterminate(` *pol*, *inum* `)` O

returns the degree of the polynomial *pol* in the indeterminate *ind* (respectively indeterminate number *inum*).

```
gap> f:=x^5+3*x*y+9*y^7+4*y^5*x+3*y+2;
9*y^7+4*x*y^5+x^5+3*x*y+3*y+2
gap> DegreeIndeterminate(f,1);
5
gap> DegreeIndeterminate(f,y);
7
```

2 ▶ `PolynomialCoefficientsOfPolynomial(` *pol*, *ind* `)` O
▶ `PolynomialCoefficientsOfPolynomial(` *pol*, *inum* `)` O

`PolynomialCoefficientsOfPolynomial` returns the coefficient list (whose entries are polynomials not involving the indeterminate *ind*) describing the polynomial *pol* viewed as a polynomial in *ind*. Instead of *ind* also the indeterminate number *inum* can be given.

```
gap> PolynomialCoefficientsOfPolynomial(f,2);
[ x^5+2, 3*x+3, 0, 0, 0, 4*x, 0, 9 ]
```

3 ▶ `LeadingCoefficient(` *pol* `)` O

returns the leading coefficient (that is the coefficient of the leading monomial, see 64.6.4) of the polynomial *pol*.

4 ▶ `LeadingMonomial( pol )` F

returns the leading monomial (with respect to the ordering given by 64.16.14 of the polynomial *pol* as a list containing indeterminate numbers and exponents.

```
gap> LeadingCoefficient(f,1);
1
gap> LeadingCoefficient(f,2);
9
gap> LeadingMonomial(f);
[ 2, 7 ]
gap> LeadingCoefficient(f);
9
```

5 ▶ `Derivative( ufun )` O
▶ `Derivative( ratfun, ind )` O

returns the derivative $upoly'$ of the univariate rational function *ufun* by its indeterminant. The second version returns the derivative of *ratfun* by the indeterminate *ind* (respectively indeterminate number *ind*) when viewing *ratfun* as univariate in *ind*.

```
gap> Derivative(f,2);
63*y^6+20*x*y^4+3*x+3
```

6 ▶ `Discriminant( upol )` O
▶ `Discriminant( pol, ind )` O

returns the discriminant disc(*upoly*) of the univariate polynomial *upoly* by its indeterminant. The second version returns the discriminant of *pol* by the indeterminate *ind* (respectively indeterminate number *ind*).

```
gap> Discriminant(f,1);
20503125*y^28+262144*y^25+27337500*y^22+19208040*y^21+1474560*y^17+13668750*y^\
16+18225000*y^15+6075000*y^14+1105920*y^13+3037500*y^10+6489720*y^9+4050000*y^\
8+900000*y^7+62208*y^5+253125*y^4+675000*y^3+675000*y^2+300000*y+50000
```

7 ▶ `Resultant( pol1, pol2, inum )` O
▶ `Resultant( pol1, pol2, ind )` O

computes the resultant of the polynomials *pol1* and *pol2* with respect to the indeterminate *ind* or indeterminate number *inum*. The resultant considers *pol1* and *pol2* as univariate in *ind* and returns an element of the corresponding base ring (which might be a polynomial ring).

```
gap> Resultant(x^4+y,y^4+x,1);
y^16+y
gap> Resultant(x^4+y,y^4+x,2);
x^16+x
```

64.7 Multivariate Polynomials

1 ▶ `Value( ratfun, indets, vals[, one] )` O
▶ `Value( upol, value[, one] )` O

The first variant takes a rational function *ratfun* and specializes the indeterminates given in *indets* to the values given in *vals*, replacing the i-th indeterminate $indets_i$ by $vals_i$. If this specialization results in a constant polynomial, an element of the coefficient ring is returned. If the specialization would specialize the denominator of *ratfun* to zero, an error is raised.

A variation is the evaluation at elements of another ring R, for which a multiplication with elements of the coefficient ring of *ratfun* are defined. In this situation the identity element of R may be given by a further argument *one* which will be used for x^0 for any specialized indeterminate x.

The second version takes an univariate rational function and specializes the value of its indeterminate to *val*. Again, an optional argument *one* may be given.

```
gap> Value(x*y+y+x^7,[x,y],[5,7]);
78167
```

Note that the default values for `one` can lead to different results than one would expect: For example for a matrix M, the values $M + M^0$ and $M + 1$ are **different**. As `Value` defaults to the one of the coefficient ring, when evaluating Matrices in polynomials always the correct `one` should be given!

2 ► `OnIndeterminates(` *poly*`,` *perm* `)` F

A permutation *perm* acts on the multivariate polynomial *poly* by permuting the indeterminates as it permutes points.

```
gap> OnIndeterminates(x^7*y+x*y^4,(1,17)(2,28));
x_17^7*x_28+x_17*x_28^4
gap> Stabilizer(Group((1,2,3,4),(1,2)),x*y,OnIndeterminates);
Group([ (1,2), (3,4) ])
```

64.8 Minimal Polynomials

1 ► `MinimalPolynomial(` *R*`,` *elm*`[ ,` *ind*`] )` O

returns the **minimal polynomial** of *elm* over the ring R, expressed in the indeterminate number *ind*. If *ind* is not given, it defaults to 1.

The minimal polynomial is the monic polynomial of smallest degree with coefficients in R that has value zero at *elm*.

```
gap> MinimalPolynomial(Rationals,[[2,0],[0,2]]);
x-2
```

64.9 Cyclotomic Polynomials

1 ► `CyclotomicPolynomial(` *F*`,` *n* `)` F

is the n-th cyclotomic polynomial over the ring F.

```
gap> CyclotomicPolynomial(Rationals,5);
x^4+x^3+x^2+x+1
```

64.10 Polynomial Factorization

At the moment GAP provides only methods to factorize univariate polynomials over finite fields (see Chapter 57) and over subfields of cyclotomic fields (see Chapter 58).

1 ► `Factors([`*R*`,]`*upoly*`[,`*opt*`])`

returns a list of the irreducible factors of the univariate polynomial *upoly* in the polynomial ring R. (That is factors over the `CoefficientsRing` of R.)

It is possible to pass a record *opt* as a third argument. This record can contain the following components:

`onlydegs`
: is a set of positive integers. The factorization assumes that all irreducible factors have a degree in this set.

`stopdegs`
: is a set of positive integers. The factorization will stop once a factor of degree in `stopdegs` has been found and will return the factorization found so far.

```
gap> f:= CyclotomicPolynomial( GF(2), 7 );
x_1^6+x_1^5+x_1^4+x_1^3+x_1^2+x_1+Z(2)^0
gap> Factors( f );
[ x_1^3+x_1+Z(2)^0, x_1^3+x_1^2+Z(2)^0 ]
gap> Factors( PolynomialRing( GF(8) ), f );
[ x_1+Z(2^3), x_1+Z(2^3)^2, x_1+Z(2^3)^3, x_1+Z(2^3)^4, x_1+Z(2^3)^5,
  x_1+Z(2^3)^6 ]
gap> f:= MinimalPolynomial( Rationals, E(4) );
x^2+1
gap> Factors( f );
[ x^2+1 ]
gap> Factors( PolynomialRing( Rationals ), f );
[ x^2+1 ]
gap> Factors( PolynomialRing( CF(4) ), f );
[ x+-E(4), x+E(4) ]
```

2 ▶ `FactorsSquarefree(` *pring*, *upol*, *opt* `)` O

returns a factorization of the squarefree, monic, univariate polynomial *upoly* in the polynomial ring *pring*; *opt* must be a (possibly empty) record of options. *upol* must not have zero as a root. This function is used by the factoring algorithms.

64.11 Polynomials over the Rationals

The following functions are only available to polynomials with rational coefficients:

1 ▶ `PrimitivePolynomial(` *f* `)` F

takes a polynomial f with rational coefficients and computes a new polynomial with integral coefficients, obtained by multiplying with the Lcm of the denominators of the coefficients and casting out the content (the Gcd of the coefficients). The operation returns a list [*newpol*,*coeff*] with rational *coeff* such that *coeff***newpol*=*f*.

2 ▶ `PolynomialModP(` *pol*, *p* `)` F

for a rational polynomial *pol* this function returns a polynomial over the field with p elements, obtained by reducing the coefficients modulo p.

3 ▶ `GaloisType(` *f*`[,` *cand*`]` `)` F

Let f be an irreducible polynomial with rational coefficients. This function returns the type of Gal(f) (considered as a transitive permutation group of the roots of f). It returns a number i if Gal(f) is permutation isomorphic to `TransitiveGroup(`n`,`i`)` where n is the degree of f.

Identification is performed by factoring appropriate Galois resolvents as proposed in [MS85]. This function is provided for rational polynomials of degree up to 15. However, in some cases the required calculations become unfeasibly large.

For a few polynomials of degree 14, a complete discrimination is not yet possible, as it would require computations, that are not feasible with current factoring methods.

This function requires the transitive groups library to be installed (see 48.6).

4 ▶ `ProbabilityShapes( f )` F

Let f be an irreducible polynomial with rational coefficients. This function returns a list of the most likely type(s) of Gal(f) (see `GaloisType` – 64.11.3), based on factorization modulo a set of primes. It is very fast, but the result is only probabilistic.

This function requires the transitive groups library to be installed (see 48.6).

```
gap> f:=x^9-9*x^7+27*x^5-39*x^3+36*x-8;;
gap> GaloisType(f);
25
gap> TransitiveGroup(9,25);
[1/2.S(3)^3]3
gap> ProbabilityShapes(f);
[ 25 ]
```

The following operations are used by GAP inside the factorization algorithm but might be of interest also in other contexts.

5 ▶ `BombieriNorm( pol )` F

computes weighted Norm [pol]_2 of *pol* which is a good measure for factor coeffietients (see [BTW93]).

6 ▶ `MinimizedBombieriNorm( f )` A

This function applies linear Tschirnhaus transformations ($x \mapsto x + i$) to the polynomial f, trying to get the Bombieri norm of f small. It returns a list `[`*new_polynomial*`,` *i_of_transformation*`]`.

7 ▶ `HenselBound( pol, [minpol, den] )` F

returns the Hensel bound of the polynomial *pol*. If the computation takes place over an algebraic extension, then the minimal polynomial *minpol* and denominator *den* must be given.

8 ▶ `OneFactorBound( pol )` F

returns the coefficient bound for a single factor of the rational polynomial *pol*.

64.12 Laurent Polynomials

A univariate polynomial can be written in the form $r_0 + r_1 x + \cdots + r_n x^n$ with $r_i \in R$. Formally, there is no reason to start with 0, if m is an integer, we can consider objects of the form $r_m x^m + r_{m+1} x^{m+1} + \cdots + r_n x^n$. We call these **Laurent polynomials**. Laurent polynomials also can be considered as quotients of a univariate polynomial by a power of the indeterminate. The addition and multiplication of univariate polynomials extends to Laurent polynomials (though it might be impossible to interpret a Laurent polynomial as a function) and many functions for univariate polynomials extend to Laurent polynomials (or extended versions for Laurent polynomials exist).

1 ▶ `LaurentPolynomialByCoefficients( fam, cofs, val [, ind] )` O

constructs a Laurent polynomial over the coefficients family *fam* and in the indeterminate *ind* (defaulting to 1) with the coefficients given by *coefs* and valuation *val*.

2 ▶ `CoefficientsOfLaurentPolynomial( laurent )` A

For a Laurent polynomial this function returns a pair `[`*cof*`,` *val*`]`, consisting of the coefficient list (in ascending order) *cof* and the valuation *val* of the Laurent polynomial *laurent*.

```
gap> p:=LaurentPolynomialByCoefficients(FamilyObj(1),
> [1,2,3,4,5],-2);
5*x^2+4*x+3+2*x^-1+x^-2
gap> NumeratorOfRationalFunction(p);DenominatorOfRationalFunction(p);
5*x^4+4*x^3+3*x^2+2*x+1
x^2
gap> CoefficientsOfLaurentPolynomial(p*p);
[ [ 1, 4, 10, 20, 35, 44, 46, 40, 25 ], -4 ]
```

3 ► `IndeterminateNumberOfLaurentPolynomial(` *pol* `)` F

Is a synonym for `IndeterminateNumberOfUnivariateRationalFunction` (see 64.1.2).

4 ► `QuotRemLaurpols(` *left*, *right*, *mode* `)` F

takes two Laurent polynomials *left* and *right* and computes their quotient. Depending on the integer variable *mode* it returns:

1. the quotient (there might be some remainder),
2. the remainder,
3. a list [*q*,*r*] of quotient and remainder,
4. the quotient if there is no remainder and `fail` otherwise.

64.13 Univariate Rational Functions

1 ► `UnivariateRationalFunctionByCoefficients(` *fam*, *ncof*, *dcof*, *val*`[,` *ind*`]` `)` O

constructs a univariate rational function over the coefficients family *fam* and in the indeterminate *ind* (defaulting to 1) with numerator and denominator coefficients given by *ncof* and *dcof* and valuation *val*.

64.14 Polynomial Rings

While polynomials depend only on the family of the coefficients, polynomial rings A are defined over a base ring R. A polynomial is an element of A if and only if all its coefficients are contained in R. Besides providing domains and an easy way to create polynomials, polynomial rings can affect the behavior of operations like factorization into irreducibles.

1 ► `PolynomialRing(` *ring*, *rank*, `[`*avoid*`]` `)` O
► `PolynomialRing(` *ring*, *names*, `[`*avoid*`]` `)` O
► `PolynomialRing(` *ring*, *indets* `)` O
► `PolynomialRing(` *ring*, *indetnums* `)` O

creates a polynomial ring over *ring*. If a positive integer *rank* is given, this creates the polynomial ring in *rank* indeterminates. These indeterminates will have the internal index numbers 1 to *rank*. The second usage takes a list *names* of strings and returns a polynomial ring in indeterminates labelled by *names*. These indeterminates have "new" internal index numbers as if they had been created by calls to `Indeterminate`. (If the argument *avoid* is given it contains indeterminates that should be avoided, in this case internal index numbers are incremented to skip these variables). In the third version, a list of indeterminates *indets* is given. This creates the polynomial ring in the indeterminates *indets*. Finally, the fourth version specifies indeterminates by their index number.

To get the indeterminates of a polynomial ring use `IndeterminatesOfPolynomialRing`. (Indeterminates created independently with `Indeterminate` will usually differ, though they might be given the same name and display identically – see section 64.1).

2 ► `IndeterminatesOfPolynomialRing(` *pring* `)` A

returns a list of the indeterminates of the polynomial ring *pring*

3 ► `CoefficientsRing(` *pring* `)` A

returns the ring of coefficients of the polynomial ring *pring*, that is the ring over which *pring* was defined.

```
gap> r:=PolynomialRing(GF(7));
PolynomialRing(..., [ x_1 ])
gap>  r:=PolynomialRing(GF(7),3);
PolynomialRing(..., [ x_1, x_2, x_3 ])
gap> IndeterminatesOfPolynomialRing(r);
[ x_1, x_2, x_3 ]
gap> r2:=PolynomialRing(GF(7),[5,7,12]);
PolynomialRing(..., [ x_5, x_7, x_12 ])
gap> CoefficientsRing(r);
GF(7)
gap> r:=PolynomialRing(GF(7),3);
PolynomialRing(..., [ x_1, x_2, x_3 ])
gap> r2:=PolynomialRing(GF(7),3,IndeterminatesOfPolynomialRing(r));
PolynomialRing(..., [ x_4, x_5, x_6 ])
gap> r:=PolynomialRing(GF(7),["x","y","z","z2"]);
PolynomialRing(..., [ x, y, z, z2 ])
```

If you need to work with a polynomial ring and its indeterminates the following two approaches will produce a ring that contains given variables (see section 64.1 for details about the internal numbering): Either, first create the ring and then get the indeterminates as `IndeterminatesOfPolynomialRing`.

```
gap> r := PolynomialRing(Rationals,["x","y"]);;
gap> indets := IndeterminatesOfPolynomialRing(r);;
gap> x := indets[1]; y := indets[2];
x
y
```

Alternatively, first create the indeterminates and then create the ring including these indeterminates.

```
gap> x:=X(Rationals,"x");;y:=X(Rationals,"y");;
gap> PolynomialRing(Rationals,[x,y]);;
```

As a convenient shortcut, intended mainly for interactive working, the indeterminates of a polynomial ring 'r' can be accessed as 'r.i', which corresponds exactly to `IndeterminatesOfPolynomialRing(r)[`i`]` or, if they have names, as 'r.*name*'. **Note** that the number i is **not** an indeterminate number, but simply an index into the indeterminates list of `r`;

```
gap> r := PolynomialRing(Rationals, ["x", "y"] );;
gap> r.1; r.2; r.x; r.y;
x
y
x
y
gap> IndeterminateNumberOfLaurentPolynomial(r.1);
7
```

As GAP objects polynomials can exist without a polynomial ring being defined and polynomials cannot be associated to a particular polynomial ring. (For example dividing a polynomial which is in a polynomial ring

over the integers by another integer will result in a polynomial over the rationals, not in a rational function over the integers.)

4 ▶ `IsPolynomialRing( pring )` C

is the category of polynomial rings

5 ▶ `IsFiniteFieldPolynomialRing( pring )` C

is the category of polynomial rings over a finite field (see Chapter 57).

6 ▶ `IsAbelianNumberFieldPolynomialRing( pring )` C

is the category of polynomial rings over a field of cyclotomics (see the chapters 18 and 58).

7 ▶ `IsRationalsPolynomialRing( pring )` C

is the category of polynomial rings over the rationals (see Chapter 16).

```
gap> IsPolynomialRing(r);
true
gap> IsFiniteFieldPolynomialRing(r);
false
gap> IsRationalsPolynomialRing(r);
true
```

64.15 Univariate Polynomial Rings

1 ▶ `UnivariatePolynomialRing( R [, nr] )` O
▶ `UnivariatePolynomialRing( R [, avoid] )` O
▶ `UnivariatePolynomialRing( R, name [, avoid] )` O

returns a univariate polynomial ring in the indeterminate *nr* over the base ring *R*. if *nr* is not given it defaults to 1. If the number is not specified a list *avoid* of indeterminates may be given. The function will return a ring in an indeterminate that is guaranteed to be different from all the indeterminates in *avoid*. The third usage returns a ring in an indeterminate called *name* (also avoiding the indeterminates in *avoid* if given).

2 ▶ `IsUnivariatePolynomialRing( pring )` C

is the category of polynomial rings with one indeterminate.

```
gap> r:=UnivariatePolynomialRing(Rationals,"x");
PolynomialRing(..., [ x ])
gap> r2:=PolynomialRing(Rationals,["q"]);
PolynomialRing(..., [ q ])
gap> IsUnivariatePolynomialRing(r);
true
gap> IsUnivariatePolynomialRing(r2);
true
```

64.16 Monomial Orderings

It is often desirable to consider the monomials within a polynomial to be arranged with respect to a certain ordering. Such an ordering is called a **monomial ordering** if it is total, invariant under multiplication with other monomials and admits no infinite descending chains. For details on monomial orderings see [CLO97].

In GAP, monomial orderings are represented by objects that provide a way to compare monomials (as polynomials as well as – for efficiency purposes within algorithms – in the internal representation as lists).

Normally the ordering chosen should be **admissible**, i.e. it must be compatible with products: If $a < b$ then $ca < cb$ for all monomials a, b and c.

1 ▶ `IsMonomialOrdering(` *obj* `)` C

A monomial ordering is an object representing a monomial ordering. Its attributes `MonomialComparisonFunction` and `MonomialExtrepComparisonFun` are actual comparison functions.

2 ▶ `LeadingMonomialOfPolynomial(` *pol*, *ord* `)` F

returns the leading monomial (with respect to the ordering *ord*) of the polynomial *pol*.

```
gap> x:=X(Rationals,"x");;y:=X(Rationals,"y");;z:=X(Rationals,"z");;
gap> lexord:=MonomialLexOrdering();grlexord:=MonomialGrlexOrdering();
MonomialLexOrdering()
MonomialGrlexOrdering()
gap> f:=2*x+3*y+4*z+5*x^2-6*z^2+7*y^3;
7*y^3+5*x^2-6*z^2+2*x+3*y+4*z
gap> LeadingMonomialOfPolynomial(f,lexord);
x^2
gap> LeadingMonomialOfPolynomial(f,grlexord);
y^3
```

3 ▶ `LeadingTermOfPolynomial(` *pol*, *ord* `)` F

returns the leading term (with respect to the ordering *ord*) of the polynomial *pol*, i.e. the product of leading coefficient and leading monomial.

4 ▶ `LeadingCoefficientOfPolynomial(` *pol*, *ord* `)` O

returns the leading coefficient (that is the coefficient of the leading monomial, see 64.16.2) of the polynomial *pol*.

```
gap> LeadingTermOfPolynomial(f,lexord);
5*x^2
gap> LeadingTermOfPolynomial(f,grlexord);
7*y^3
gap> LeadingCoefficientOfPolynomial(f,lexord);
5
```

Each monomial ordering provides two functions to compare monomials. These functions work as "is less than", i.e. they return `true` if and only if the left argument is smaller.

5 ▶ `MonomialComparisonFunction(` *O* `)` A

If *O* is an object representing a monomial ordering, this attribute returns a **function** that can be used to compare or sort monomials (and polynomials which will be compared by their monomials in decreasing order) in this order.

```
gap> MonomialComparisonFunction(lexord);
function( a, b ) ... end
gap> l:=[f,Derivative(f,x),Derivative(f,y),Derivative(f,z)];;
gap> Sort(l,MonomialComparisonFunction(lexord));l;
[ -12*z+4, 21*y^2+3, 10*x+2, 7*y^3+5*x^2-6*z^2+2*x+3*y+4*z ]
```

6 ▶ `MonomialExtrepComparisonFun( O )` A

If O is an object representing a monomial ordering, this attribute returns a **function** that can be used to compare or sort monomials **in their external representation** (as lists). This comparison variant is used inside algorithms that manipulate the external representation.

The following monomial orderings are predefined in GAP:

7 ▶ `MonomialLexOrdering( )` F
▶ `MonomialLexOrdering( vari )` F

This function creates a lexicographic ordering for monomials. Monomials are compared first by the exponents of the largest variable, then the exponents of the second larhgest variable and so on.

The variables are ordered accoring to their (internal) index, i.e. x_1 is larger than x_2 and so on. If *vari* is given, and is a list of variables or variable indices, instead this arrangement of variables (in descending order; i.e. the first variable is larger than the second) is used as the underlying order of variables.

```
gap> l:=List(Tuples([1..3],3),i->x^(i[1]-1)*y^(i[2]-1)*z^(i[3]-1));
[ 1, z, z^2, y, y*z, y*z^2, y^2, y^2*z, y^2*z^2, x, x*z, x*z^2, x*y, x*y*z,
  x*y*z^2, x*y^2, x*y^2*z, x*y^2*z^2, x^2, x^2*z, x^2*z^2, x^2*y, x^2*y*z,
  x^2*y*z^2, x^2*y^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialLexOrdering()));l;
[ 1, z, z^2, y, y*z, y*z^2, y^2, y^2*z, y^2*z^2, x, x*z, x*z^2, x*y, x*y*z,
  x*y*z^2, x*y^2, x*y^2*z, x*y^2*z^2, x^2, x^2*z, x^2*z^2, x^2*y, x^2*y*z,
  x^2*y*z^2, x^2*y^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialLexOrdering([y,z,x])));l;
[ 1, x, x^2, z, x*z, x^2*z, z^2, x*z^2, x^2*z^2, y, x*y, x^2*y, y*z, x*y*z,
  x^2*y*z, y*z^2, x*y*z^2, x^2*y*z^2, y^2, x*y^2, x^2*y^2, y^2*z, x*y^2*z,
  x^2*y^2*z, y^2*z^2, x*y^2*z^2, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialLexOrdering([z,x,y])));l;
[ 1, y, y^2, x, x*y, x*y^2, x^2, x^2*y, x^2*y^2, z, y*z, y^2*z, x*z, x*y*z,
  x*y^2*z, x^2*z, x^2*y*z, x^2*y^2*z, z^2, y*z^2, y^2*z^2, x*z^2, x*y*z^2,
  x*y^2*z^2, x^2*z^2, x^2*y*z^2, x^2*y^2*z^2 ]
```

8 ▶ `MonomialGrlexOrdering( )` F
▶ `MonomialGrlexOrdering( vari )` F

This function creates a degree/lexicographic ordering. In this oredring monomials are compared first by their total degree, then lexicographically (see `MonomialLexOrdering`).

The variables are ordered accoring to their (internal) index, i.e. x_1 is larger than x_2 and so on. If *vari* is given, and is a list of variables or variable indices, instead this arrangement of variables (in descending order; i.e. the first variable is larger than the second) is used as the underlying order of variables.

9 ▶ `MonomialGrevlexOrdering( )` F
▶ `MonomialGrevlexOrdering( vari )` F

This function creates a "grevlex" ordering. In this ordering monomials are compared first by total degree and then backwards lexicographically. (This is different than "grlex" ordering with variables reversed.)

The variables are ordered accoring to their (internal) index, i.e. x_1 is larger than x_2 and so on. If *vari* is given, and is a list of variables or variable indices, instead this arrangement of variables (in descending order; i.e. the first variable is larger than the second) is used as the underlying order of variables.

```
gap> Sort(l,MonomialComparisonFunction(MonomialGrlexOrdering()));l;
[ 1, z, y, x, z^2, y*z, y^2, x*z, x*y, x^2, y*z^2, y^2*z, x*z^2, x*y*z,
  x*y^2, x^2*z, x^2*y, y^2*z^2, x*y*z^2, x*y^2*z, x^2*z^2, x^2*y*z, x^2*y^2,
  x*y^2*z^2, x^2*y*z^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialGrevlexOrdering()));l;
[ 1, z, y, x, z^2, y*z, x*z, y^2, x*y, x^2, y*z^2, x*z^2, y^2*z, x*y*z,
  x^2*z, x*y^2, x^2*y, y^2*z^2, x*y*z^2, x^2*z^2, x*y^2*z, x^2*y*z, x^2*y^2,
  x*y^2*z^2, x^2*y*z^2, x^2*y^2*z, x^2*y^2*z^2 ]
gap> Sort(l,MonomialComparisonFunction(MonomialGrlexOrdering([z,y,x])));l;
[ 1, x, y, z, x^2, x*y, y^2, x*z, y*z, z^2, x^2*y, x*y^2, x^2*z, x*y*z,
  y^2*z, x*z^2, y*z^2, x^2*y^2, x^2*y*z, x*y^2*z, x^2*z^2, x*y*z^2, y^2*z^2,
  x^2*y^2*z, x^2*y*z^2, x*y^2*z^2, x^2*y^2*z^2 ]
```

10 ▶ `EliminationOrdering(` *elim* `)` F
▶ `EliminationOrdering(` *elim*`,` *rest* `)` F

This function creates an elimination ordering for eliminating the variables in *elim*. Two monomials are compared first by the exponent vectors for the variables listed in *elim* (a lexicographic comparison with respect to the ordering indicated in *elim*). If these submonomial are equal, the submonomials given by the other variables are compared by a graded lexicographic ordering (with respect to the variable order given in *rest*, if called with two parameters).

Both *elim* and *rest* may be a list of variables of a list of variable indices.

11 ▶ `PolynomialReduction(` *poly*`,` *gens*`,` *order* `)` F

reduces the polynomial *poly* by the ideal generated by the polynomials in *gens*, using the order *order* of monomials. Unless *gens* is a Gröbner basis the result is not guaranteed to be unique.

The operation returns a list of length two, the first entry is the remainder after the reduction. The second entry is a list of quotients corresponding to *gens*.

Note that the strategy used by `PolynomialReduction` differs from the standard textbook reduction algorithm, which is provided by `PolynomialDivisionAlgorithm`.

12 ▶ `PolynomialReducedRemainder(` *poly*`,` *gens*`,` *order* `)` F

thios operation does the same way as `PolynomialReduction` (see 64.16.11) but does not keep track of the actual quotients and returns only the remainder (it is therfore slightly faster).

13 ▶ `PolynomialDivisionAlgorithm(` *poly*`,` *gens*`,` *order* `)` F

This function implements the division algorithm for multivariate polynomials as given in theorem 3 in chapter 2 of [CLO97]. (It might be slower than `PolynomialReduction` but the remainders are guaranteed to agree with the textbook.)

The operation returns a list of length two, the first entry is the remainder after the reduction. The second entry is a list of quotients corresponding to *gens*.

```
gap> bas:=[x^3*y*z,x*y^2*z,z*y*z^3+x];;
gap> pol:=x^7*z*bas[1]+y^5*bas[3]+x*z;;
gap> PolynomialReduction(pol,bas,MonomialLexOrdering());
[ -y*z^5, [ x^7*z, 0, y^5+z ] ]
gap> PolynomialReducedRemainder(pol,bas,MonomialLexOrdering());
-y*z^5
gap> PolynomialDivisionAlgorithm(pol,bas,MonomialLexOrdering());
[ -y*z^5, [ x^7*z, 0, y^5+z ] ]
```

14 ▶ `MonomialExtGrlexLess( a, b )` F

implements comparison of monomial in their external representation by a "grlex" order with $x_1 > x_2$ (This is exactly the same as the ordering by `MonomialGrlexOrdering()`, see 64.16). The function takes two monomials a and b in expanded form and returns whether the first is smaller than the second. (This ordering is also used by GAP internally for representing polynomials as a linear combination of monomials.)

See section 64.20 for details on the expanded form of monomials.

64.17 Groebner Bases

A **Groebner Basis** of an ideal Ii, in a polynomial ring R, with respect to a monomial ordering, is a set of ideal generators G such that the ideal generated by the leading monomials of all polynomials in G is equal to the ideal generated by the leading monomials of all polynomials in I.

For more details on Groebner bases see [CLO97].

1 ▶ `GroebnerBasis( L, O )` O
▶ `GroebnerBasis( I, O )` O
▶ `GroebnerBasisNC( L, O )` O

Let O be a monomial ordering and L be a list of polynomials that generate an ideal I. This operation returns a Groebner basis of I with respect to the ordering O."

`GroebnerBasisNC` works like `GroebnerBasis` with the only distinction that the first argument has to be a list of polynomials and that no test is performed to check whether the ordering is defined for all occuring variables.

Note that GAP at the moment only includes a na"ıve implementation of Buchberger's algorithm (which is mainly intended as a teaching tool). It might not be sufficient for serious problems.

```
gap> l:=[x^2+y^2+z^2-1,x^2+z^2-y,x-y];;
gap> GroebnerBasis(l,MonomialLexOrdering());
[ x^2+y^2+z^2-1, x^2+z^2-y, x-y, -y^2-y+1, -z^2+2*y-1, 1/2*z^4+2*z^2-1/2 ]
gap> GroebnerBasis(l,MonomialLexOrdering([z,x,y]));
[ x^2+y^2+z^2-1, x^2+z^2-y, x-y, -y^2-y+1 ]
gap> GroebnerBasis(l,MonomialGrlexOrdering());
[ x^2+y^2+z^2-1, x^2+z^2-y, x-y, -y^2-y+1, -z^2+2*y-1 ]
```

2 ▶ `ReducedGroebnerBasis( L, O )` O
▶ `ReducedGroebnerBasis( I, O )` O

a Groebner basis B (see 64.17.1) is **reduced** if no monomial in a polynomial in B is divisible by the leading monomial of another polynomial in B. This operation computes a Groebner basis with respect to O and then reduces it.

```
gap> ReducedGroebnerBasis(l,MonomialGrlexOrdering());
[ x-y, z^2-2*y+1, y^2+y-1 ]
gap> ReducedGroebnerBasis(l,MonomialLexOrdering());
[ z^4+4*z^2-1, -1/2*z^2+y-1/2, -1/4*z^2+x-1/4 ]
gap> ReducedGroebnerBasis(l,MonomialLexOrdering([y,z,x]));
[ 1, x^2+x-1, z^2-x+1/2, -x+y ]
```

3 ► `StoredGroebnerBasis(` *I* `)` A

For an ideal I in a polynomial ring, this attribute holds a list $[B,O]$ where B is a Groebner basis for the monomial ordering O. this can be used to test membership or canonical coset representatives.

4 ► `InfoGroebner` V

This info class gives information about Groebner basis calculations.

64.18 Rational Function Families

All rational functions defined over a ring lie in the same family, the rational functions family over this ring.

In GAP therefore the family of a polynomial depends only on the family of the coefficients, all polynomials whose coefficients lie in the same family are "compatible".

1 ► `RationalFunctionsFamily(` *fam* `)` A

creates a family containing rational functions with coefficients in *fam*. All elements of the `RationalFunctionsFamily` are rational functions (see 64.4.1).

2 ► `IsPolynomialFunctionsFamily(` *obj* `)` C
► `IsRationalFunctionsFamily(` *obj* `)` C

`IsPolynomialFunctionsFamily` is the category of a family of polynomials. For families over an UFD, the category becomes `IsRationalFunctionsFamily` (as rational functions and quotients are only provided for families over an UFD.)

```
gap> fam:=RationalFunctionsFamily(FamilyObj(1));
NewFamily( "RationalFunctionsFamily(...)", [ 618, 620 ],
[ 82, 85, 89, 93, 97, 100, 103, 107, 111, 618, 620 ] )
```

3 ► `CoefficientsFamily(` *rffam* `)` A

If *rffam* has been created as `RationalFunctionsFamily(`*cfam*`)` this attribute holds the coefficients family *cfam*.

GAP does **not** embed the base ring in the polynomial ring. While multiplication and addition of base ring elements to rational functions return the expected results, polynomials and rational functions are not equal.

```
gap> 1=Indeterminate(Rationals)^0;
false
```

64.19 The Representations of Rational Functions

GAP uses four representations of rational functions: Rational functions given by numerator and denominator, polynomials, univariate rational functions (given by coefficient lists for numerator and denominator and valuation) and Laurent polynomials (given by coefficient list and valuation).

These representations do not necessarily reflect mathematical properties: While an object in the Laurent polynomials representation must be a Laurent polynomial it might turn out that a rational function given by numerator and denominator is actually a Laurent polynomial and the property tests in section 64.4 will find this out.

Each representation is associated one or several "defining attributes" that give an "external" representation (see 64.20) of the representation in the form of lists and are the defining information that tells a rational function what it is.

GAP also implements methods to compute these attributes for rational functions in **other** representations, provided it would be possible to express an **mathematically equal** rational function in the representation associated with the attribute. (That is one can always get a numerator/denominator representation of a polynomial while an arbitrary function of course can compute a polynomial representation only if it is a polynomial.)

Therefore these attributes can be thought of as "conceptual" representations that allow us – as far as possible – to consider an object as a rational function, a polynomial or a Laurent polynomial, regardless of the way it is represented in the computer.

Functions thus usually do not need to care about the representation of a rational function. Depending on its (known in the context or determined) properties, they can access the attribute representing the rational function in the desired way.

Consequentially, methods for rational functions are installed for properties and not for representations.

When **creating** new rational functions however they must be created in one of the three representations. In most cases this will be the representation for which the "conceptual" representation in which the calculation was done is the defining attribute.

Iterated operations (like forming the product over a list) therefore will tend to stay in the most suitable representation and the calculation of another conceptual representation (which may be comparatively expensive in certain circumstances) is not necessary.

64.20 The Defining Attributes of Rational Functions

In general, rational functions are given in terms of monomials. They are represented by lists, using numbers (see 64.1) for the indeterminates.

A monomial is a product of powers of indeterminates. A monomial is stored as a list (we call this the **expanded form** of the monomial) of the form `[`*inum*`,`*exp*`,`*inum*`,`*exp*`,...]` where each *inum* is the number of an indeterminate and *exp* the corresponding exponent. The list must be sorted according to the numbers of the indeterminates. Thus for example, if x, y and z are the first three indeterminates, the expanded form of the monomial $x^5z^8 = z^8x^5$ is `[1,5,3,8]`.

The representation of a polynomials is a list of the form `[`*mon*`,`*coeff*`,`*mon*`,`*coeff*`,...]` where *mon* is a monomial in expanded form (that is given as list) and *coeff* its coefficient. The monomials must be sorted according to the total degree/lexicographic order (This is the same as given by the "grlex" monomial ordering, see 64.16.8). We call this the **external representation** of a polynomial. (The reason for ordering is that addition of polynomials becomes linear in the number of monomials instead of quadratic; the reason for the particular ordering chose is that it is compatible with multiplication and thus gives acceptable performance for quotient calculations.)

1 ▶ `IsRationalFunctionDefaultRep(` *obj* `)` R

is the default representation of rational functions. A rational function in this representation is defined by the attributes `ExtRepNumeratorRatFun` and `ExtRepDenominatorRatFun` where `ExtRepNumeratorRatFun` and `ExtRepDenominatorRatFun` are both external representations of a polynomial.

2 ▶ `ExtRepNumeratorRatFun(` *ratfun* `)` A

returns the external representation of the numerator polynomial of the rational function *ratfun*. Numerator and Denominator are not guaranteed to be cancelled against each other.

3 ▶ `ExtRepDenominatorRatFun(` *ratfun* `)` A

returns the external representation of the denominator polynomial of the rational function *ratfun*. Numerator and Denominator are not guaranteed to be cancelled against each other.

4 ▶ `ZeroCoefficientRatFun(` *ratfun* `)` O

returns the zero of the coefficient ring. This might be needed to represent the zero polynomial for which the external representation of the numerator is the empty list.

5 ▶ `IsPolynomialDefaultRep(` *obj* `)` R

is the default representation of polynomials. A polynomial in this representation is defined by the components and `ExtRepNumeratorRatFun` where `ExtRepNumeratorRatFun` is the external representation of the polynomial.

6 ▶ `ExtRepPolynomialRatFun(` *polynomial* `)` A

returns the external representation of a polynomial. The difference to `ExtRepNumeratorRatFun` is that rational functions might know to be a polynomial but can still have a non-vanishing denominator. In this case `ExtRepPolynomialRatFun` has to call a quotient routine.

7 ▶ `IsLaurentPolynomialDefaultRep(` *obj* `)` R

This representation is used for Laurent polynomials and univariate polynomials. It represents a Laurent polynomial via the attributes `CoefficientsOfLaurentPolynomial` (see 64.12.2) and `IndeterminateNumberOfLaurentPolynomial` (see 64.12.3).

The attributes that give a representation of a a rational function as a Laurent polynomial are `CoefficientsOfLaurentPolynomial` (see 64.12.2) and `IndeterminateNumberOfUnivariateRationalFunction` (see 64.1.2).

Algorithms should use only the attributes `ExtRepNumeratorRatFun`, `ExtRepDenominatorRatFun`, `ExtRepPolynomialRatFun`, `CoefficientsOfLaurentPolynomial` and – if the univariate function is not constant – `IndeterminateNumberOfUnivariateRationalFunction` as the low-level interface to work with a polynomial. They should not refer to the actual representation used.

64.21 Creation of Rational Functions

The operations `LaurentPolynomialByCoefficients` (see 64.12.1), `PolynomialByExtRep` and `RationalFunctionByExtRep` are used to construct objects in the three basic representations for rational functions.

1 ▶ `RationalFunctionByExtRep(` *rfam*`,` *num*`,` *den* `)` F

constructs a rational function (in the representation `IsRationalFunctionDefaultRep`) in the rational function family *rfam*, the rational function itself is given by the external representations *num* and *den* for numerator and denominator. No test for validity of the arguments is performed and no cancellation takes place.

2 ▶ `PolynomialByExtRep( rfam, extrep )` F

constructs a polynomial (in the representation `IsPolynomialDefaultRep`) in the rational function family *rfam*, the polynomial itself is given by the external representation *extrep*. No test for validity of the arguments is performed.

```
gap> fam:=RationalFunctionsFamily(FamilyObj(1));;
gap> p:=PolynomialByExtRep(fam,[[1,2],1,[2,1,15,7],3]);
3*y*x_15^7+x^2
gap> q:=p/(p+1);
(3*y*x_15^7+x^2)/(3*y*x_15^7+x^2+1)
gap> ExtRepNumeratorRatFun(q);
[ [ 1, 2 ], 1, [ 2, 1, 15, 7 ], 3 ]
gap> ExtRepDenominatorRatFun(q);
[ [  ], 1, [ 1, 2 ], 1, [ 2, 1, 15, 7 ], 3 ]
```

3 ▶ `LaurentPolynomialByExtRep( fam, cofs, val , ind )` F

creates a Laurent polynomial in the family *fam* with [*cofs*,*val*] as value of `CoefficientsOfLaurentPolynomial`. No coefficient shifting is performed. This is the lowest level function to create a Laurent polynomial but will rely on the coefficients being shifted properly and will not perform any tests. Unless this is guaranteed for the parameters, `LaurentPolynomialByCoefficients` (see 64.12.1) should be used.

64.22 Arithmetic for External Representations of Polynomials

The following operations are used internally to perform the arithmetic for polynomials in their "external" representation (see 64.20) as lists.

1 ▶ `ZippedSum( z1, z2, czero, funcs )` O

computes the sum of two external representations of polynomials *z1* and *z2*. *czero* is the appropriate coefficient zero and *funcs* a list [*monomial less*, *coefficient sum*] containing a monomial comparison and a coefficient addition function. This list can be found in the component *fam*`!.zippedSum` of the rational functions family.

Note that *coefficient sum* must be a proper "summation" function, not a function computing differences.

2 ▶ `ZippedProduct( z1, z2, czero, funcs )` O

computes the product of two external representations of polynomials *z1* and *z2*. *czero* is the appropriate coefficient zero and *funcs* a list [*monomial_prod*, *monomial_less*, *coefficient_sum*, *coefficient_prod*] containing functions to multiply and compare monomials, to add and to multiply coefficients. This list can be found in the component *fam*`!.zippedProduct` of the rational functions family.

3 ▶ `QuotientPolynomialsExtRep( fam, a, b )` F

Let *a* and *b* the external representations of two polynomials in the rational functions family *fam*. This function computes the external representation of the quotient of both polynomials, it returns `fail` if *b* does not divide *a*.

Functions to perform arithmetic with the coefficient lists of Laurent polynomials are described in section 23.3.

64.23 Cancellation Tests for Rational Functions

GAP does not contain a multivariate GCD algorithm. The following operations are used internally to try to keep the denominators as small as possible

1 ▶ `RationalFunctionByExtRepWithCancellation(` *rfam*`,` *num*`,` *den* `)` F

constructs a rational function as `RationalFunctionByExtRep` does but tries to cancel out common factors of numerator and denominator, calling `TryGcdCancelExtRepPolynomials`.

2 ▶ `TryGcdCancelExtRepPolynomials(` *fam*`,` *a*`,` *b* `)` F

Let f and g be two polynomials given by the ext reps a and b. This function tries to cancel common factors between a and b and returns a list $[ac,bc]$ of cancelled numerator and denominator ext rep. As there is no proper multivariate GCD cancellation is not guaranteed to be optimal.

3 ▶ `HeuristicCancelPolynomials(` *fam*`,` *ext1*`,` *ext2* `)` O

is called by `TryGcdCancelExtRepPol` to perform the actual work. It will return either `fail` or a new list $[num,den]$ of cancelled numerator and denominator. The cancellation performed is not necessarily optimal.

65 Algebraic extensions of fields

If we adjoin a root α of an irreducible polynomial $f \in K[x]$ to the field K we get an **algebraic extension** $K(\alpha)$, which is again a field. We call K the **base field** of $K(\alpha)$.

By Kronecker's construction, we may identify $K(\alpha)$ with the factor ring $K[x]/(f)$, an identification that also provides a method for computing in these extension fields.

It is important to note that different extensions of the same field are entirely different (and its elements lie in different families), even if mathematically one could be embedded in the other one.

Currently GAP only allows extension fields of fields K, when K itself is not an extension field.

65.1 Creation of Algebraic Extensions

1 ▶ `AlgebraicExtension( K, f )` O

constructs an extension L of the field K by one root of the irreducible polynomial f, using Kronecker's construction. L is a field whose `LeftActingDomain` is K. The polynomial p is the `DefiningPolynomial` of L and the attribute `RootOfDefiningPolynomial` of L holds a root of f in L (see 56.2.8).

```
gap> x:=Indeterminate(Rationals,"x");;
gap> p:=x^4+3*x^2+1;;
gap> e:=AlgebraicExtension(Rationals,p);
<field in characteristic 0>
gap> IsField(e);
true
gap> a:=RootOfDefiningPolynomial(e);
(a)
```

2 ▶ `IsAlgebraicExtension( obj )` C

is the category of algebraic extensions of fields.

```
gap> IsAlgebraicExtension(e);
true
gap> IsAlgebraicExtension(Rationals);
false
```

65.2 Elements in Algebraic Extensions

According to Kronecker's construction, the elements of an algebraic extension considered to be polynomials in the primitive element. The elements of the base field are represented as polynomials of degree 0. GAP therefore displays elements of an algebraic extension as polynomials in an indeterminate "a", which is a root of the defining polynomial of the extension. Polynomials of degree 0 are displayed with a leading exclamation mark to indicate that they are different from elements of the base field.

The usual field operations are applicable to algebraic elements.

```
gap> a^3/(a^2+a+1);
(-1/2*a+1/2*a^2-1/2*a^3)
gap> a*(1/a);
!1
```

The external representation of algebraic extension elements are the polynomial coefficients in the primitive element `a`, the operations `ExtRepOfObj` and `ObjByExtRep` can be used for conversion.

```
gap> ExtRepOfObj(One(a));
[ 1, 0, 0, 0 ]
gap> ExtRepOfObj(a^3+2*a-9);
[ -9, 2, 0, 1 ]
gap> ObjByExtRep(FamilyObj(a),[3,19,-27,433]);
(3+19*a-27*a^2+433*a^3)
```

GAP does **not** embed the base field in its algebraic extensions and therefore lists which contain elements of the base field and of the extension are not homogeneous and thus cannot be used as polynomial coefficients or to form matrices. The remedy is to multiply the list(s) with the `One` of the extension which will embed all entries in the extension.

```
gap> m:=[[1,a],[0,1]];
[ [ 1, (a) ], [ 0, 1 ] ]
gap> IsMatrix(m);
false
gap> m:=m*One(e);
[ [ !1, (a) ], [ !0, !1 ] ]
gap> IsMatrix(m);
true
gap> m^2;
[ [ !1, (2*a) ], [ !0, !1 ] ]
```

1 ▶ `IsAlgebraicElement(` *obj* `)` C

is the category for elements of an algebraic extension.

66 p-adic Numbers (preliminary)

In this chapter p is always a (fixed) prime.

The p-adic numbers Q_p are the completion of the rational numbers with respect to the valuation $\nu_p(p^v \frac{a}{b}) = v$ if p divides neither a nor b. They form a field of characteristic 0 which nevertheless shows some behaviour of the finite field with p elements.

A p-adic numbers can be approximated by a "p-adic expansion" which is similar to the decimal expansion used for the reals (but written from left to right). So for example if $p = 2$, the numbers 1,2,3,4,$\frac{1}{2}$ and $\frac{4}{5}$ are represented as $1(2)$, $0 \cdot 1(2)$, $1 \cdot 1(2)$, $0 \cdot 01(2)$, $10(2)$ and $0 \cdot 0101(2)$. Approximation means to ignore powers of p, so for example with only 2 digits accuracy $\frac{4}{5}$ would be approximated as $0 \cdot 01(2)$. The important difference to the decimal approximation is that p-adic approximation is a ring homomorphism on the subrings of p-adic numbers whose valuation is bounded from below.

In GAP, p-adic numbers are represented by approximations. A family of (approximated) p-adic numbers consists of p-adic numbers with a certain precision and arithmetic with these numbers is done with this precision.

66.1 Pure p-adic Numbers

Pure p-adic numbers are the p-adic numbers described so far.

1 ▶ `PurePadicNumberFamily(` *p*`,` *precision* `)` O

returns the family of pure p-adic numbers over the prime p with *precision* "digits".

2 ▶ `PadicNumber(`*fam*`,`*rat*`)`

returns the element of the p-adic number family *fam* that is used to represent the rational number *rat*.

p-adic numbers allow the usual operations for fields.

```
gap> fam:=PurePadicNumberFamily(2,3);;
gap> a:=PadicNumber(fam,4/5);
0.0101(2)
gap> 3*a;
0.0111(2)
gap> a/2;
0.101(2)
gap> a*10;
0.001(2)
```

3 ▶ `Valuation(` *obj* `)` O

The Valuation is the p-part of the p-adic number.

4 ▶ `ShiftedPadicNumber(` *padic*`,` *int* `)` O

ShiftedPadicNumber takes a p-adic number *padic* and an integer *shift* and returns the p-adic number c, that is *padic*`*` `p^`*shift*. The *shift* is just added to the p-part.

5 ► `IsPurePadicNumber(` *obj* `)` C

6 ► `IsPurePadicNumberFamily(` *fam* `)` C

66.2 Extensions of the p-adic Numbers

The usual Kronecker construction with an irreducible polynomial can be used to construct extensions of the p-adic numbers. Let L be such an extension. Then there is a subfield $K < L$ such that K is an unramified extension of the p-adic numbers and L/K is purely ramified. (For an explanation of "ramification" see for example [Neu92], section II.7 or another book on algebraic number theory. Essentially, an extension L of the p-adic numbers generated by a rational polynomial f is unramified if f remains squarefree modulo p and is completely ramified if modulo p the polynomial f is a power of a linear factor while remaining irreducible over the p-adic numbers.) The representation of extensions of p-adic numbers in GAP uses this subfield.

1 ► `PadicExtensionNumberFamily(` *p*, *precision*, *unram*, *ram* `)` F

An extended p-adic field L is given by two polynomials h and g with coeff.-lists *unram* (for the unramified part) and *ram* (for the ramified part). Then L is isomorphic to $Q_p[x,y]/(h(x),g(y))$.

This function takes the prime number p and the two coefficient lists *unram* and *ram* for the two polynomials. The polynomial given by the coefficients in *unram* must be a cyclotomic polynomial and the polynomial given by *ram* an Eisenstein-polynomial (or 1+x). **This is not checked by GAP.**

Every number out of L is represented as a coeff.-list for the basis $\{1, x, x^2, \ldots, y, xy, x^2y, \ldots\}$ of L. The integer *precision* is the number of "digits" that all the coefficients have.

A general comment: the polynomials with which `PadicExtensionNumberFamily` is called define an extension of Q_p. It must be ensured that both polynomials are really irreducible over Q_p! For example x^2+x+1 is **not** irreducible over Q_p. Therefore the "extension" PadicExtensionNumberFamily(3, 4, [1,1,1], [1,1]) contains non-invertible "pseudo-p-adic numbers". Conversely, if an "extension" contains noninvertible elements one of the polynomials was not irreducible.

2 ► `PadicNumber(` *fam*, *rat* `)` O
► `PadicNumber(` *purefam*, *list* `)` O
► `PadicNumber(` *extfam*, *list* `)` O

create a p-adic number in the p-adic numbers family *fam*. The first usage returns the p-adic number corresponding to the rational *rat*.

The second usage takes a pure p-adic numbers family *purefam* and a list *list* of length 2 and returns the number `p^`*list*`[1] * `*list*`[2]`. It must be guaranteed that no entry of list[2] is divisible by the prime p. (Otherwise precision will get lost.)

The third usage creates a number in the family *extfam* of a p-adic extension. The second entry must be a list L of length 2 such that *list*[2] is the list of coeff. for the basis $\{1, \ldots, x^{f-1} \cdot y^{e-1}\}$ of the extended p-adic field and *list*[1] is a common p-part of all the coeff.

p-adic numbers allow the usual field operations.

```
gap> efam:=PadicExtensionNumberFamily(3, 5, [1,1,1], [1,1]);;
gap> PadicNumber(efam,7/9);
padic(120(3),0(3))
```

A word of warning: Depending on the actual representation of quotients, precision may seem to "vanish". For example in PadicExtensionNumberFamily(3, 5, [1,1,1], [1,1]) the number (1.2000, 0.1210)(3) can be represented as `[ 0, [ 1.2000, 0.1210 ] ]` or as `[-1, [ 12.000, 1.2100 ] ]` (here the coefficients have to be multiplied by p^{-1}).

So there may be a number (1.2, 2.2)(3) which seems to have only two digits of precision instead of the declared 5. But internally the number is stored as `[-3, [ 0.0012, 0.0022 ] ]` and so has in fact maximum precision.

3 ► `IsPadicExtensionNumber(` *obj* `)` C

4 ► `IsPadicExtensionNumberFamily(` *fam* `)` C

67 The MeatAxe

The MeatAxe [Par84] is a tool for the examination of submodules of a group algebra. It is a basic tool for the examination of group actions on finite-dimensional modules.

GAP uses the improved MeatAxe of Derek Holt and Sarah Rees, and also incorporates further improvements of Ivanyos and Lux.

67.1 MeatAxe Modules

1 ► `GModuleByMats(` *gens*`,` *field* `)`
► `GModuleByMats(` *emptygens*`,` *dim*`,` *field* `)`

creates a MeatAxe module over *field* from a list of invertible matrices *gens* which reflect a group's action. If the list of generators is empty, the dimension must be given as second argument.

MeatAxe routines are on a level with Gaussian elimination. Therefore they do not deal with GAP modules but essentially with lists of matrices. For the MeatAxe, a module is a record with components

`generators`
: A list of matrices which represent a group operation on a finite dimensional row vector space.

`dimension`
: The dimension of the vector space (this is the common length of the row vectors (see 59.8.6)).

`field`
: The field over which the vector space is defined.

Once a module has been created its entries may not be changed. A MeatAxe may create a new component *NameOfMeatAxe* in which it can store private information. By a MeatAxe "submodule" or "factor module" we denote actually the **induced action** on the submodule, respectively factor module. Therefore the submodules or factor modules are again MeatAxe modules. The arrangement of `generators` is guaranteed to be the same for the induced modules, but to obtain the complete relation to the original module, the bases used are needed as well.

67.2 Module Constructions

1 ► `TensorProductGModule (`*m1*`,`*m2*`)` F

TensorProductGModule calculates the tensor product of the modules *m1* and *m2*. They are assumed to be modules over the same algebra so, in particular, they should have the same number of generators.

2 ► `WedgeGModule (`*module*`)`

WedgeGModule calculates the wedge product of a G-module. That is the action on antisymmetric tensors.

67.3 Selecting a Different MeatAxe

All MeatAxe routines are accessed via the global variable `MTX`, which is a record whose components hold the various functions. It is possible to have several implementations of a MeatAxe available. Each MeatAxe represents its routines in an own global variable and assigning `MTX` to this variable selects the corresponding MeatAxe.

67.4 Accessing a Module

Even though a MeatAxe module is a record, its components should never be accessed outside of MeatAxe functions. Instead the following operations should be used:

1 ▶ `MTX.Generators(` *module* `)`

returns a list of matrix generators of *module*.

2 ▶ `MTX.Dimension(` *module* `)`

returns the dimension in which the matrices act.

3 ▶ `MTX.Field(` *module* `)`

returns the field over which *module* is defined.

67.5 Irreducibility Tests

1 ▶ `MTX.IsIrreducible(` *module* `)` AST

tests whether the module *module* is irreducible (i.e. contains no proper submodules.)

2 ▶ `MTX.IsAbsolutelyIrreducible(` *module* `)` AST

A module is absolutely irreducible if it remains irreducible over the algebraic closure of the field. (Formally: If the tensor product $L \otimes_K M$ is irreducible where M is the module defined over K and L is the algebraic closure of K.)

3 ▶ `MTX.DegreeSplittingField(` *module* `)`

returns the degree of the splitting field as extension of the prime field.

67.6 Finding Submodules

1 ▶ `MTX.SubmoduleGModule(` *module*`,` *subspace* `)` F
▶ `MTX.SubGModule(` *module*`,` *subspace* `)` F

subspace should be a subspace of (or a vector in) the underlying vector space of *module* i.e. the full row space of the same dimension and over the same field as *module*. A normalized basis of the submodule of *module* generated by *subspace* is returned.

2 ▶ `MTX.ProperSubmoduleBasis(` *module* `)` F

returns the basis of a proper submodule of *module* and `fail` if no proper submodule exists.

3 ▶ `MTX.BasesSubmodules(` *module* `)` F

returns a list containing a basis for every submodule.

4 ▶ `MTX.BasesMinimalSubmodules(` *module* `)` F

returns a list of bases of all minimal submodules.

5 ► `MTX.BasesMaximalSubmodules(` *module* `)` F

returns a list of bases of all maximal submodules.

6 ► `MTX.BasisRadical(` *module* `)` F

returns a basis of the radical of *module*.

7 ► `MTX.BasisSocle(` *module* `)` F

returns a basis of the socle of *module*.

8 ► `MTX.BasesMinimalSupermodules(` *module*`,` *sub* `)` F

returns a list of bases of all minimal supermodules of the submodule given by the basis *sub*.

9 ► `MTX.BasesCompositionSeries(` *module* `)` F

returns a list of bases of submodules in a composition series in ascending order.

10 ► `MTX.CompositionFactors(` *module* `)` F

returns a list of composition factors of *module* in ascending order.

11 ► `MTX.CollectedFactors(` *module* `)` F

returns a list giving all irreducible composition factors with their frequencies.

67.7 Induced Actions

1 ► `MTX.NormedBasisAndBaseChange(`*sub*`)`

returns a list `[`*bas*`,change]` where *bas* is a normed basis (i.e. in echelon form with pivots normed to 1) for *sub* and change is the base change from *bas* to *sub* (the basis vectors of *bas* expressed in coefficients for *sub*)

2 ► `MTX.InducedActionSubmodule(` *module*`,` *sub* `)` F
► `MTX.InducedActionSubmoduleNB(` *module*`,` *sub* `)` F

creates a new module corresponding to the action of *module* on *sub*. In the `NB` version the basis *sub* must be normed. (That is it must be in echelon form with pivots normed to 1. See `MTX.NormedBasisAndBaseChange`)

3 ► `MTX.InducedActionFactorModule(` *module*`,` *sub*`[,` *compl*`]` `)` F

creates a new module corresponding to the action of *module* on the factor of *sub*. If *compl* is given, it has to be a basis of a (vector space-)complement of *sub*. The action then will correspond to *compl*.

The basis *sub* has to be given in normed form. (That is it must be in echelon form with pivots normed to 1. See `MTX.NormedBasisAndBaseChange`)

4 ► `MTX.InducedActionMatrix(`*mat*`,`*sub*`)`
► `MTX.InducedActionMatrixNB(`*mat*`,`*sub*`)`
► `MTX.InducedActionFactorMatrix(` *mat*`,` *sub*`[,` *compl*`]` `)` F

work the same way as the above functions for modules, but take as input only a single matrix.

5 ► `MTX.InducedAction(` *module*`,` *sub*`[,` *type*`]` `)` F

Computes induced actions on submodules or factormodules and also returns the corresponding bases. The action taken is binary encoded in *type*: 1 stands for subspace action, 2 for factor action and 4 for action of the full module on a subspace adapted basis. The routine returns the computed results in a list in sequence (*sub*,*quot*,*both*,*basis*) where *basis* is a basis for the whole space, extending *sub*. (Actions which are not computed are omitted, so the returned list may be shorter.) If no *type* is given, it is assumed to be 7. The basis given in *sub* must be normed!

All these routines return `fail` if *sub* is not a proper subspace.

67.8 Module Homomorphisms

1 ▶ `MTX.IsEquivalent(` *module1*`,` *module2* `)` F

tests two irreducible modules for equivalence.

2 ▶ `MTX.Isomorphism(` *module1*`,` *module2* `)` F

returns an isomorphism from *module1* to *module2* (if one exists) and `fail` otherwise. It requires that one of the modules is known to be irreducible. It implicitly assumes that the same group is acting, otherwise the results are unpredictable. The isomorphism is given by a matrix M, whose rows give the images of the standard basis vectors of module2 in the standard basis of module1. That is, conjugation of the generators of *module2* with M yields the generators of *module1*.

3 ▶ `MTX.Homomorphism(` *module1*`,` *module2*`,` *mat* `)` F

mat should be a *dim1* $\times$ *dim2* matrix defining a homomorphism from *module1* to *module2*. This function verifies that *mat* really does define a module homomorphism, and then returns the corresponding homomorphism between the underlying row spaces of the modules. This can be used for computing kernels, images and pre-images.

4 ▶ `MTX.Homomorphisms(` *module1*`,` *module2* `)` F

returns a basis of all homomorphisms from the irreducible module *module1* to *module2*.

5 ▶ `MTX.Distinguish(` *cf*`,` *nr* `)` F

Let *cf* be the output of `MTX.CollectedFactors`. This routine tries to find a group algebra element that has nullity zero on all composition factors except number *nr*.

67.9 Invariant Forms

The functions in this section can only be applied to an absolutely irreducible MeatAxe module *module*.

1 ▶ `MTX.InvariantBilinearForm(`*module*`)` F

returns an invariant bilinear form, which may be symmetric or anti-symmetric, of *module*, or `fail` if no such form exists.

2 ▶ `MTX.InvariantSesquilinearForm(`*module*`)` F

returns an invariant hermitian (= self-adjoint) sesquilinear form of *module*, which must be defined over a finite field whose order is a square, or `fail` if no such form exists.

3 ▶ `MTX.InvariantQuadraticForm(`*module*`)` F

returns an invariant quadratic form of *module*, or `fail` if no such form exists. If the characteristic of the field over which *module* is defined is not 2, then the invariant bilinear form (if any) divided by two will be returned. In characteristic 2, the form returned will be lower triangular.

4 ▶ `MTX.BasisInOrbit(`*module*`)` F

returns a basis of the underlying vector space of *module* which is contained in an orbit of the action of the generators of module on that space. This is used by `MTX.InvariantQuadraticForm` in characteristic 2.

5 ▶ `MTX.OrthogonalSign(`*module*`)` F

for an even dimensional module, returns 1 or -1, according as `MTX.InvariantQuadraticForm(`*module*`)` is of + or - type. For an odd dimensional module, returns 0. For a module with no invariant quadratic form, returns `fail`. This calculation uses an algorithm due to Jon Thackray.

67.10 The Smash MeatAxe

The standard MeatAxe provided in the GAP library is is based on the MeatAxe in the GAP 3 package Smash, originally written by Derek Holt and Sarah Rees [HR94]. It is accessible via the variable `SMTX` to which `MTX` is assigned by default. For the sake of completeness the remaining sections document more technical functions of this MeatAxe.

1 ► `SMTX.RandomIrreducibleSubGModule(` *module* `)` F

returns the module action on a random irreducible submodule.

2 ► `SMTX.GoodElementGModule(` *module* `)` F

finds an element with minimal possible nullspace dimension if *module* is known to be irreducible.

3 ► `SMTX.SortHomGModule(` *module1* `,` *module2* `,` *homs* `)` F

Function to sort the output of `Homomorphisms`.

4 ► `SMTX.MinimalSubGModules(` *module1* `,` *module2* `[,` *max*`] )`

returns (at most *max*) bases of submodules of *module2* which are isomorphic to the irreducible module *module1*.

5 ► `SMTX.Setter(` *string* `)`

returns a setter function for the component `smashMeataxe.(string)`.

6 ► `SMTX.Getter(` *string* `)`

returns a getter function for the component `smashMeataxe.(string)`.

7 ► `SMTX.IrreducibilityTest(` *module* `)`

Tests for irreducibility and sets a subbasis if reducible. It neither sets an irreducibility flag, nor tests it. Thus the routine also can simply be called to obtain a random submodule.

8 ► `SMTX.AbsoluteIrreducibilityTest(` *module* `)`

Tests for absolute irreducibility and sets splitting field degree. It neither sets an absolute irreducibility flag, nor tests it.

9 ► `SMTX.MinimalSubGModule(` *module*`,` *cf*`,` *nr* `)`

returns the basis of a minimal submodule of *module* containing the indicated composition factor. It assumes `Distinguish` has been called already.

10 ► `SMTX.MatrixSum(` *matrices1*`,` *matrices2* `)`

creates the direct sum of two matrix lists.

11 ► `SMTX.CompleteBasis(` *module*`,` *pbasis* `)`

extends the partial basis *pbasis* to a basis of the full space by action of *module*. It returns whether it succeeded.

67.11 Smash MeatAxe Flags

The following getter routines access internal flags. For each routine, the appropriate setter's name is prefixed with `Set`.

1 ▶ `SMTX.Subbasis`

Basis of a submodule.

2 ▶ `SMTX.AlgEl`

list `[newgens,coefflist]` giving an algebra element used for chopping.

3 ▶ `SMTX.AlgElMat`

matrix of `SMTX.AlgEl`.

4 ▶ `SMTX.AlgElCharPol`

minimal polynomial of `SMTX.AlgEl`.

5 ▶ `SMTX.AlgElCharPolFac`

uses factor of `SMTX.AlgEl`.

6 ▶ `SMTX.AlgElNullspaceVec`

nullspace of the matrix evaluated under this factor.

7 ▶ `SMTX.AlgElNullspaceDimension`

dimension of the nullspace.

8 ▶ `SMTX.CentMat`

9 ▶ `SMTX.CentMatMinPoly`

68 Tables of Marks

The concept of a **table of marks** was introduced by W. Burnside in his book "Theory of Groups of Finite Order", see [Bur55]. Therefore a table of marks is sometimes called a **Burnside matrix**.

The table of marks of a finite group G is a matrix whose rows and columns are labelled by the conjugacy classes of subgroups of G and where for two subgroups A and B the (A, B)–entry is the number of fixed points of B in the transitive action of G on the cosets of A in G. So the table of marks characterizes the set of all permutation representations of G.

Moreover, the table of marks gives a compact description of the subgroup lattice of G, since from the numbers of fixed points the numbers of conjugates of a subgroup B contained in a subgroup A can be derived.

A table of marks of a given group G can be constructed from the subgroup lattice of G (see 68.3). For several groups, the table of marks can be restored from the GAP library of tables of marks (see 68.14).

Given the table of marks of G, one can display it (see 68.4) and derive information about G and its Burnside ring from it (see 68.7, 68.8, 68.9). Moreover, tables of marks in GAP provide an easy access to the classes of subgroups of their underlying groups (see 68.11).

68.1 More about Tables of Marks

Let G be a finite group with n conjugacy classes of subgroups $C_1, C_2, \ldots, C_n$ and representatives $H_i \in C_i$, $1 \leq i \leq n$. The **table of marks** of G is defined to be the $n \times n$ matrix $M = (m_{ij})$ where the **mark** m_{ij} is the number of fixed points of the subgroup H_j in the action of G on the right cosets of H_i in G.

Since H_j can only have fixed points if it is contained in a point stablizer the matrix M is lower triangular if the classes C_i are sorted according to the condition that if H_i is contained in a conjugate of H_j then $i \leq j$.

Moreover, the diagonal entries m_{ii} are nonzero since m_{ii} equals the index of H_i in its normalizer in G. Hence M is invertible. Since any transitive action of G is equivalent to an action on the cosets of a subgroup of G, one sees that the table of marks completely characterizes the set of all permutation representations of G.

The marks m_{ij} have further meanings. If H_1 is the trivial subgroup of G then each mark m_{i1} in the first column of M is equal to the index of H_i in G since the trivial subgroup fixes all cosets of H_i. If $H_n = G$ then each m_{nj} in the last row of M is equal to 1 since there is only one coset of G in G. In general, m_{ij} equals the number of conjugates of H_i containing H_j, multiplied by the index of H_i in its normalizer in G. Moreover, the number c_{ij} of conjugates of H_j which are contained in H_i can be derived from the marks m_{ij} via the formula

$$c_{ij} = \frac{m_{ij}\, m_{j1}}{m_{i1}\, m_{jj}}.$$

Both the marks m_{ij} and the numbers of subgroups c_{ij} are needed for the functions described in this chapter.

A brief survey of properties of tables of marks and a description of algorithms for the interactive construction of tables of marks using GAP can be found in [Pfe97].

68.2 Table of Marks Objects in GAP

A table of marks of a group G in GAP is represented by an immutable (see 12.6) object *tom* in the category `IsTableOfMarks` (see 68.6.2), with defining attributes `SubsTom` (see 68.7.1) and `MarksTom` (see 68.7.1). These two attributes encode the matrix of marks in a compressed form. The `SubsTom` value of *tom* is a list where for each conjugacy class of subgroups the class numbers of its subgroups are stored. These are exactly the positions in the corresponding row of the matrix of marks which have nonzero entries. The marks themselves are stored via the `MarksTom` value of *tom*, which is a list that contains for each entry in `SubsTom(` *tom* `)` the corresponding nonzero value of the table of marks.

It is possible to create table of marks objects that do not store a group, moreover one can create a table of marks object from a matrix of marks (see 68.3.1). So it may happen that a table of marks object in GAP is in fact **not** the table of marks of a group. To some extent, the consistency of a table of marks object can be checked (see 68.9), but GAP knows no general way to prove or disprove that a given matrix of nonnegative integers is the matrix of marks for a group. Many functions for tables of marks work well without access to the group –this is one of the arguments why tables of marks are so useful–, but for example normalizers (see 68.9.4) and derived subgroups (see 68.9.2) of subgroups are in general not uniquely determined by the matrix of marks.

GAP tables of marks are assumed to be in lower triangular form, that is, if a subgroup from the conjugacy class corresponding to the i-th row is contained in a subgroup from the class corresponding to the j-th row j then $i \leq j$.

The `MarksTom` information can be computed from the values of the attributes `NrSubsTom`, `LengthsTom`, `OrdersTom`, and `SubsTom` (see 68.7.2, 68.7.3, 68.7.2). `NrSubsTom` stores a list containing for each entry in the `SubsTom` value the corresponding number of conjugates that are contained in a subgroup, `LengthsTom` a list containing for each conjugacy class of subgroups its length, and `OrdersTom` a list containing for each class of subgroups their order. So the `MarksTom` value of *tom* may be missing provided that the values of `NrSubsTom`, `LengthsTom`, and `OrdersTom` are stored in *tom*.

Additional information about a table of marks is needed by some functions. The class numbers of normalizers in G and the number of the derived subgroup of G can be stored via appropriate attributes (see 68.9.4, 68.9.2).

If *tom* stores its group G and a bijection from the rows and columns of the matrix of marks of *tom* to the classes of subgroups of G then clearly normalizers, derived subgroup etc. can be computed from this information. But in general a table of marks need not have access to G, for example *tom* might have been constructed from a generic table of marks (see 68.13), or as table of marks of a factor group from a given table of marks (see 68.9.11). Access to the group G is provided by the attribute `UnderlyingGroup` (see 68.7.7) if this value is set. Access to the relevant information about conjugacy classes of subgroups of G –compatible with the ordering of rows and columns of the marks in *tom*– is signalled by the filter `IsTableOfMarksWithGens` (see 68.11).

Several examples in this chapter require the GAP Library of Tables of Marks to be available. If it is not yet loaded then we load it now.

```
gap> LoadPackage( "tomlib" );
true
```

68.3 Constructing Tables of Marks

1 ▶ `TableOfMarks(` *G* `)` A
▶ `TableOfMarks(` *string* `)` A
▶ `TableOfMarks(` *matrix* `)` A

In the first form, G must be a finite group, and `TableOfMarks` constructs the table of marks of G. This computation requires the knowledge of the complete subgroup lattice of G (see 37.20.1). If the lattice

is not yet stored then it will be constructed. This may take a while if G is large. The result has the `IsTableOfMarksWithGens` value `true` (see 68.11).

In the second form, *string* must be a string, and `TableOfMarks` gets the table of marks with name *string* from the GAP library (see 68.14). If no table of marks with this name is contained in the library then `fail` is returned.

In the third form, *matrix* must be a matrix or a list of rows describing a lower triangular matrix where the part above the diagonal is omitted. For such an argument *matrix*, `TableOfMarks` returns a table of marks object (see 68.2) for which *matrix* is the matrix of marks. Note that not every matrix (containing only nonnegative integers and having lower triangular shape) describes a table of marks of a group. Necessary conditions are checked with `IsInternallyConsistent` (see 68.9), and `fail` is returned if *matrix* is proved not to describe a matrix of marks; but if `TableOfMarks` returns a table of marks object created from a matrix then it may still happen that this object does not describe the table of marks of a group.

For an overview of operations for table of marks objects, see the introduction to the Chapter 68.

```
gap> tom:= TableOfMarks( AlternatingGroup( 5 ) );
TableOfMarks( Alt( [ 1 .. 5 ] ) )
gap> TableOfMarks( "J5" );
fail
gap> a5:= TableOfMarks( "A5" );
TableOfMarks( "A5" )
gap> mat:=
> [ [ 60, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0 ],
>   [ 20, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0 ],
>   [ 12, 0, 0, 0, 2, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0 ],
>   [ 6, 2, 0, 0, 1, 0, 1, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 0, 1, 0 ],
>   [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ];;
gap> TableOfMarks( mat );
TableOfMarks( <9 classes> )
```

The following `TableOfMarks` methods for a group are installed.

- If the group is known to be cyclic then `TableOfMarks` constructs the table of marks essentially without the group, instead the knowledge about the structure of cyclic groups is used.
- If the lattice of subgroups is already stored in the group then `TableOfMarks` computes the table of marks from the lattice (see 68.3.2).
- If the group is known to be solvable then `TableOfMarks` takes the lattice of subgroups (see 37.20.1) of the group –which means that the lattice is computed if it is not yet stored– and then computes the table of marks from it. This method is also accessible via the global function `TableOfMarksByLattice` (see 68.3.2).
- If the group doesn't know its lattice of subgroups or its conjugacy classes of subgroups then the table of marks and the conjugacy classes of subgroups are computed at the same time by the cyclic extension method. Only the table of marks is stored because the conjugacy classes of subgroups or the lattice of subgroups can be easily read off (see 68.3.3).

Conversely, the lattice of subgroups of a group with known table of marks can be computed using the table of marks, via the function `LatticeSubgroupsByTom`. This is also installed as a method for `LatticeSubgroups`.

2 ▶ `TableOfMarksByLattice( `G` )` F

`TableOfMarksByLattice` computes the table of marks of the group G from the lattice of subgroups of G. This lattice is computed via `LatticeSubgroups` (see 37.20.1) if it is not yet stored in G. The function `TableOfMarksByLattice` is installed as a method for `TableOfMarks` for solvable groups and groups with

stored subgroup lattice, and is available as a global variable only in order to provide explicit access to this method.

3 ► `LatticeSubgroupsByTom( G )` F

`LatticeSubgroupsByTom` computes the lattice of subgroups of G from the table of marks of G, using `RepresentativeTom` (see 68.11.4).

68.4 Printing Tables of Marks

The default `ViewObj` (see 6.2.3) method for tables of marks prints the string `"TableOfMarks"`, followed by –if known– the identifier (see 68.7.9) or the group of the table of marks enclosed in brackets; if neither group nor identifier are known then just the number of conjugacy classes of subgroups is printed instead.

The default `PrintObj` (see 6.2.3) method for tables of marks does the same as `ViewObj`, except that the group is is `Print`-ed instead of `View`-ed.

The default `Display` (see 6.2.4) method for a table of marks *tom* produces a formatted output of the marks in *tom*. Each line of output begins with the number of the corresponding class of subgroups. This number is repeated if the output spreads over several pages. The number of columns printed at one time depends on the actual line length, which can be accessed and changed by the function `SizeScreen` (see 6.11.1).

The optional second argument *arec* of `Display` can be used to change the default style for displaying a character as shown above. *arec* must be a record, its relevant components are the following.

`classes`
: a list of class numbers to select only the rows and columns of the matrix that correspond to this list for printing,

`form`
: one of the strings `"subgroups"`, `"supergroups"`; in the former case, at position (i, j) of the matrix the number of conjugates of H_j contained in H_i is printed, and in the latter case, at position (i, j) the number of conjugates of H_i which contain H_j is printed.

```
gap> tom:= TableOfMarks( "A5" );;
gap> Display( tom );
1:  60
2:  30 2
3:  20 . 2
4:  15 3 . 3
5:  12 . . . 2
6:  10 2 1 . . 1
7:   6 2 . . 1 . 1
8:   5 1 2 1 . . . 1
9:   1 1 1 1 1 1 1 1 1

gap> Display( tom, rec( classes:= [ 1, 2, 3, 4, 8 ] ) );
1:  60
2:  30 2
3:  20 . 2
4:  15 3 . 3
8:   5 1 2 1 1
```

```
gap> Display( tom, rec( form:= "subgroups" ) );
1:  1
2:  1  1
3:  1  .  1
4:  1  3  . 1
5:  1  .  . . 1
6:  1  3  1 . .  1
7:  1  5  . . 1  . 1
8:  1  3  4 1 .  . . 1
9:  1 15 10 5 6 10 6 5 1

gap> Display( tom, rec( form:= "supergroups" ) );
1:   1
2:  15 1
3:  10 . 1
4:   5 1 . 1
5:   6 . . . 1
6:  10 2 1 . . 1
7:   6 2 . . 1 . 1
8:   5 1 2 1 . . . 1
9:   1 1 1 1 1 1 1 1 1
```

68.5 Sorting Tables of Marks

1 ▶ `SortedTom( `*tom*`, `*perm*` )` O

`SortedTom` returns a table of marks where the rows and columns of the table of marks *tom* are reordered according to the permutation *perm*.

Note that in each table of marks in GAP, the matrix of marks is assumed to have lower triangular shape (see 68.2). If the permutation *perm* does **not** have this property then the functions for tables of marks might return wrong results when applied to the output of `SortedTom`.

The returned table of marks has only those attribute values stored that are known for *tom* and listed in `TableOfMarksComponents` (see 68.6.4).

```
gap> tom:= TableOfMarksCyclic( 6 );;  Display( tom );
1:  6
2:  3 3
3:  2 . 2
4:  1 1 1 1

gap> sorted:= SortedTom( tom, (2,3) );;  Display( sorted );
1:  6
2:  2 2
3:  3 . 3
4:  1 1 1 1

gap> wrong:= SortedTom( tom, (1,2) );;  Display( wrong );
1:  3
2:  . 6
3:  . 2 2
4:  1 1 1 1
```

2 ▶ `PermutationTom( tom )` A

For the table of marks *tom* of the group G stored as `UnderlyingGroup` value of *tom* (see 68.7.7), `PermutationTom` is a permutation π such that the i-th conjugacy class of subgroups of G belongs to the i^π-th column and row of marks in *tom*.

This attribute value is bound only if *tom* was obtained from another table of marks by permuting with `SortedTom` (see 68.5.1), and there is no default method to compute its value.

The attribute is necessary because the original and the sorted table of marks have the same identifier and the same group, and information computed from the group may depend on the ordering of marks, for example the fusion from the ordinary character table of G into *tom*.

```
gap> MarksTom( tom )[2];
[ 3, 3 ]
gap> MarksTom( sorted )[2];
[ 2, 2 ]
gap> HasPermutationTom( sorted );
true
gap> PermutationTom( sorted );
(2,3)
```

68.6 Technical Details about Tables of Marks

1 ▶ `InfoTom` V

is the info class for computations concerning tables of marks.

2 ▶ `IsTableOfMarks( obj )` C

Each table of marks belongs to this category.

3 ▶ `TableOfMarksFamily` V

Each table of marks belongs to this family.

4 ▶ `TableOfMarksComponents` V

The list `TableOfMarksComponents` is used when a table of marks object is created from a record via `ConvertToTableOfMarks` (see 68.6.5). `TableOfMarksComponents` contains at position $2i-1$ a name of an attribute and at position $2i$ the corresponding attribute getter function.

5 ▶ `ConvertToTableOfMarks( record )` F

`ConvertToTableOfMarks` converts a record with components from `TableOfMarksComponents` into a table of marks object with the corresponding attributes.

```
gap> record:= rec( MarksTom:= [ [ 4 ], [ 2, 2 ], [ 1, 1, 1 ] ],
>  SubsTom:= [ [ 1 ], [ 1, 2 ], [ 1, 2, 3 ] ] );;
gap> ConvertToTableOfMarks( record );;
gap> record;
TableOfMarks( <3 classes> )
```

68.7 Attributes of Tables of Marks

1 ▶ `MarksTom( tom )` A
▶ `SubsTom( tom )` A

The matrix of marks (see 68.1) of the table of marks *tom* is stored in a compressed form where zeros are omitted, using the attributes `MarksTom` and `SubsTom`. If M is the square matrix of marks of *tom* (see 68.7.10) then the `SubsTom` value of *tom* is a list that contains at position i the list of all positions of nonzero entries of the i-th row of M, and the `MarksTom` value of *tom* is a list that contains at position i the list of the corresponding marks.

`MarksTom` and `SubsTom` are defining attributes of tables of marks (see 68.2). There is no default method for computing the `SubsTom` value, and the default `MarksTom` method needs the values of `NrSubsTom` and `OrdersTom` (see 68.7.2, 68.7.2).

```
gap> a5:= TableOfMarks( "A5" );
TableOfMarks( "A5" )
gap> MarksTom( a5 );
[ [ 60 ], [ 30, 2 ], [ 20, 2 ], [ 15, 3, 3 ], [ 12, 2 ], [ 10, 2, 1, 1 ],
  [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ], [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
gap> SubsTom( a5 );
[ [ 1 ], [ 1, 2 ], [ 1, 3 ], [ 1, 2, 4 ], [ 1, 5 ], [ 1, 2, 3, 6 ],
  [ 1, 2, 5, 7 ], [ 1, 2, 3, 4, 8 ], [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ] ]
```

2 ▶ `NrSubsTom( tom )` A
▶ `OrdersTom( tom )` A

Instead of storing the marks (see 68.7.1) of the table of marks *tom* one can use a matrix which contains at position (i, j) the number of subgroups of conjugacy class j that are contained in one member of the conjugacy class i. These values are stored in the `NrSubsTom` value in the same way as the marks in the `MarksTom` value.

`OrdersTom` returns a list that contains at position i the order of a representative of the i-th conjugacy class of subgroups of *tom*.

One can compute the `NrSubsTom` and `OrdersTom` values from the `MarksTom` value of *tom* and vice versa.

```
gap> NrSubsTom( a5 );
[ [ 1 ], [ 1, 1 ], [ 1, 1 ], [ 1, 3, 1 ], [ 1, 1 ], [ 1, 3, 1, 1 ],
  [ 1, 5, 1, 1 ], [ 1, 3, 4, 1, 1 ], [ 1, 15, 10, 5, 6, 10, 6, 5, 1 ] ]
gap> OrdersTom( a5 );
[ 1, 2, 3, 4, 5, 6, 10, 12, 60 ]
```

3 ▶ `LengthsTom( tom )` A

For a table of marks *tom*, `LengthsTom` returns a list of the lengths of the conjugacy classes of subgroups.

```
gap> LengthsTom( a5 );
[ 1, 15, 10, 5, 6, 10, 6, 5, 1 ]
```

4 ▶ `ClassTypesTom( tom )` A

`ClassTypesTom` distinguishes isomorphism types of the classes of subgroups of the table of marks *tom* as far as this is possible from the `SubsTom` and `MarksTom` values of *tom*.

Two subgroups are clearly not isomorphic if they have different orders. Moreover, isomorphic subgroups must contain the same number of subgroups of each type.

Each type is represented by a positive integer. `ClassTypesTom` returns the list which contains for each class of subgroups its corresponding type.

```
gap> a6:= TableOfMarks( "A6" );;
gap> ClassTypesTom( a6 );
[ 1, 2, 3, 3, 4, 5, 6, 6, 7, 7, 8, 9, 10, 11, 11, 12, 13, 13, 14, 15, 15, 16 ]
```

5 ▶ `ClassNamesTom( tom )` A

`ClassNamesTom` constructs generic names for the conjugacy classes of subgroups of the table of marks *tom*. In general, the generic name of a class of non–cyclic subgroups consists of three parts and has the form `"(`*o*`)_`*t*`l"`, where *o* indicates the order of the subgroup, *t* is a number that distinguishes different types of subgroups of the same order, and *l* is a letter that distinguishes classes of subgroups of the same type and order. The type of a subgroup is determined by the numbers of its subgroups of other types (see 68.7.4). This is slightly weaker than isomorphism.

The letter is omitted if there is only one class of subgroups of that order and type, and the type is omitted if there is only one class of that order. Moreover, the braces around the type are omitted if the type number has only one digit.

For classes of cyclic subgroups, the parentheses round the order and the type are omitted. Hence the most general form of their generic names is `"`*o*`,`*l*`"`. Again, the letter is omitted if there is only one class of cyclic subgroups of that order.

```
gap> ClassNamesTom( a6 );
[ "1", "2", "3a", "3b", "5", "4", "(4)_2a", "(4)_2b", "(6)a", "(6)b", "(8)",
  "(9)", "(10)", "(12)a", "(12)b", "(18)", "(24)a", "(24)b", "(36)", "(60)a",
  "(60)b", "(360)" ]
```

6 ▶ `FusionsTom( tom )` AM

For a table of marks *tom*, `FusionsTom` is a list of fusions into other tables of marks. Each fusion is a list of length two, the first entry being the `Identifier` (see 68.7.9) value of the image table, the second entry being the list of images of the class positions of *tom* in the image table.

This attribute is mainly used for tables of marks in the GAP library (see 68.14).

```
gap> fus:= FusionsTom( a6 );;
gap> fus[1];
[ "L3(4)", [ 1, 2, 3, 3, 14, 5, 9, 7, 15, 15, 24, 26, 27, 32, 33, 50, 57, 55,
      63, 73, 77, 90 ] ]
```

7 ▶ `UnderlyingGroup( tom )` A

`UnderlyingGroup` is used to access an underlying group that is stored on the table of marks *tom*. There is no default method to compute an underlying group if it is not stored.

```
gap> UnderlyingGroup( a6 );
Group([ (1,2)(3,4), (1,2,4,5)(3,6) ])
```

8 ▶ `IdempotentsTom( tom )` A
▶ `IdempotentsTomInfo( tom )` A

`IdempotentsTom` encodes the idempotents of the integral Burnside ring described by the table of marks *tom*. The return value is a list *l* of positive integers such that each row vector describing a primitive idempotent has value 1 at all positions with the same entry in *l*, and 0 at all other positions.

According to A. Dress [Dre69] (see also [Pfe97]), these idempotents correspond to the classes of perfect subgroups, and each such idempotent is the characteristic function of all those subgroups that arise by cyclic extension from the corresponding perfect subgroup (see 68.9.7).

`IdempotentsTomInfo` returns a record with components `fixpointvectors` and `primidems`, both bound to lists. The i-th entry of the `fixpointvectors` list is the $0-1$-vector describing the i-th primitive idempotent, and the i-th entry of `primidems` is the decomposition of this idempotent in the rows of *tom*.

```
gap> IdempotentsTom( a5 );
[ 1, 1, 1, 1, 1, 1, 1, 1, 9 ]
gap> IdempotentsTomInfo( a5 );
rec(
  primidems := [ [ 1, -2, -1, 0, 0, 1, 1, 1 ], [ -1, 2, 1, 0, 0, -1, -1, -1,
          1 ] ],
  fixpointvectors := [ [ 1, 1, 1, 1, 1, 1, 1, 1, 0 ], [ 0, 0, 0, 0, 0, 0, 0,
          0, 1 ] ] )
```

9 ▶ `Identifier( tom )` A

The identifier of a table of marks *tom* is a string. It is used for printing the table of marks (see 68.4) and in fusions between tables of marks (see 68.7.6).

If *tom* is a table of marks from the GAP library of tables of marks (see 68.14) then it has an identifier, and if *tom* was constructed from a group with `Name` value (see 12.8.2) then this name is chosen as `Identifier` value. There is no default method to compute an identifier in all other cases.

```
gap> Identifier( a5 );
"A5"
```

10 ▶ `MatTom( tom )` A

`MatTom` returns the square matrix of marks (see 68.1) of the table of marks *tom* which is stored in a compressed form using the attributes `MarksTom` and `SubsTom` (see 68.7.1). This may need substantially more space than the values of `MarksTom` and `SubsTom`.

```
gap> MatTom( a5 );
[ [ 60, 0, 0, 0, 0, 0, 0, 0, 0 ], [ 30, 2, 0, 0, 0, 0, 0, 0, 0 ],
  [ 20, 0, 2, 0, 0, 0, 0, 0, 0 ], [ 15, 3, 0, 3, 0, 0, 0, 0, 0 ],
  [ 12, 0, 0, 0, 2, 0, 0, 0, 0 ], [ 10, 2, 1, 0, 0, 1, 0, 0, 0 ],
  [ 6, 2, 0, 0, 1, 0, 1, 0, 0 ], [ 5, 1, 2, 1, 0, 0, 0, 1, 0 ],
  [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

11 ▶ `MoebiusTom( tom )` A

`MoebiusTom` computes the Möbius values both of the subgroup lattice of the group G with table of marks *tom* and of the poset of conjugacy classes of subgroups of G. It returns a record where the component `mu` contains the Möbius values of the subgroup lattice, and the component `nu` contains the Möbius values of the poset.

Moreover, according to an observation of Isaacs et al. (see [HIÖ89], [Pah93]), the values on the subgroup lattice often can be derived from those of the poset of conjugacy classes. These "expected values" are returned in the component `ex`, and the list of numbers of those subgroups where the expected value does not coincide with the actual value are returned in the component `hyp`. For the computation of these values, the position of the derived subgroup of G is needed (see 68.9.2). If it is not uniquely determined then the result does not have the components `ex` and `hyp`.

```
gap> MoebiusTom( a5 );
rec( mu := [ -60, 4, 2,,, -1, -1, -1, 1 ], nu := [ -1, 2, 1,,, -1, -1, -1, 1 ]
    , ex := [ -60, 4, 2,,, -1, -1, -1, 1 ], hyp := [  ] )
gap> tom:= TableOfMarks( "M12" );;
gap> moebius:= MoebiusTom( tom );;
gap> moebius.hyp;
[ 1, 2, 4, 16, 39, 45, 105 ]
gap> moebius.mu[1];  moebius.ex[1];
95040
190080
```

12 ▶ `WeightsTom( tom )` A

`WeightsTom` extracts the **weights** from the table of marks *tom*, i.e., the diagonal entries of the matrix of marks (see 68.7.1), indicating the index of a subgroup in its normalizer.

```
gap> wt:= WeightsTom( a5 );
[ 60, 2, 2, 3, 2, 1, 1, 1, 1 ]
```

This information may be used to obtain the numbers of conjugate supergroups from the marks.

```
gap> marks:= MarksTom( a5 );;
gap> List( [ 1 .. 9 ], x -> marks[x] / wt[x] );
[ [ 1 ], [ 15, 1 ], [ 10, 1 ], [ 5, 1, 1 ], [ 6, 1 ], [ 10, 2, 1, 1 ],
  [ 6, 2, 1, 1 ], [ 5, 1, 2, 1, 1 ], [ 1, 1, 1, 1, 1, 1, 1, 1, 1 ] ]
```

68.8 Properties of Tables of Marks

For a table of marks *tom* of a group G, the following properties have the same meaning as the corresponding properties for G. Additionally, if a positive integer *sub* is given as the second argument then the value of the corresponding property for the *sub*-th class of subgroups of *tom* is returned.

1 ▶ `IsAbelianTom( tom[, sub] )`
▶ `IsCyclicTom( tom[, sub] )`
▶ `IsNilpotentTom( tom[, sub] )`
▶ `IsPerfectTom( tom[, sub] )`
▶ `IsSolvableTom( tom[, sub] )`

```
gap> tom:= TableOfMarks( "A5" );;
gap> IsAbelianTom( tom );  IsPerfectTom( tom );
false
true
gap> IsAbelianTom( tom, 3 );  IsNilpotentTom( tom, 7 );
true
false
gap> IsPerfectTom( tom, 7 );  IsSolvableTom( tom, 7 );
false
true
gap> for i in [ 1 .. 6 ] do
> Print( i, ": ", IsCyclicTom(a5, i), "  " );
> od;  Print( "\n" );
1: true  2: true  3: true  4: false  5: true  6: false
```

68.9 Other Operations for Tables of Marks

1 ▶ `IsInternallyConsistent( tom )` O

For a table of marks *tom*, `IsInternallyConsistent` decomposes all tensor products of rows of *tom*. It returns `true` if all decomposition numbers are nonnegative integers, and `false` otherwise. This provides a strong consistency check for a table of marks.

2 ▶ `DerivedSubgroupTom( tom, sub )` O
▶ `DerivedSubgroupsTom( tom )` F

For a table of marks *tom* and a positive integer *sub*, `DerivedSubgroupTom` returns either a positive integer i or a list l of positive integers. In the former case, the result means that the derived subgroups of the subgroups in the *sub*-th class of *tom* lie in the i-th class. In the latter case, the class of the derived subgroups could not be uniquely determined, and the position of the class of derived subgroups is an entry of l.

Values computed with `DerivedSubgroupTom` are stored using the attribute `DerivedSubgroupsTomPossible` (see 68.9.3).

`DerivedSubgroupsTom` is just the list of `DerivedSubgroupTom` values for all values of *sub*.

3 ▶ `DerivedSubgroupsTomPossible( tom )` AM
▶ `DerivedSubgroupsTomUnique( tom )` A

Let *tom* be a table of marks. The value of the attribute `DerivedSubgroupsTomPossible` is a list in which the value at position i –if bound– is a positive integer or a list; the meaning of the entry is the same as in `DerivedSubgroupTom` (see 68.9.2).

If the value of the attribute `DerivedSubgroupsTomUnique` is known for *tom* then it is a list of positive integers, the value at position i being the position of the class of derived subgroups of the i-th class of subgroups in *tom*. The derived subgroups are in general not uniquely determined by the table of marks if no `UnderlyingGroup` value is stored, so there is no default method for `DerivedSubgroupsTomUnique`. But in some cases the derived subgroups are explicitly set when the table of marks is constructed. The `DerivedSubgroupsTomUnique` value is automatically set when the last missing unique value is entered in the `DerivedSubgroupsTomPossible` list by `DerivedSubgroupTom`.

```
gap> a5:= TableOfMarks( "A5" );
TableOfMarks( "A5" )
gap> DerivedSubgroupTom( a5, 2 );
1
gap> DerivedSubgroupsTom( a5 );
[ 1, 1, 1, 1, 1, 3, 5, 4, 9 ]
```

4 ▶ `NormalizerTom( tom, sub )` O
▶ `NormalizersTom( tom )` A

Let *tom* be the table of marks of a group G, say. `NormalizerTom` tries to find the conjugacy class of the normalizer N in G of a subgroup U in the *sub*-th class of *tom*. The return value is either the list of class numbers of those subgroups that have the right size and contain the subgroup and all subgroups that clearly contain it as a normal subgroup, or the class number of the normalizer if it is uniquely determined by these conditions. If *tom* knows the subgroup lattice of G (see 68.11.3) then all normalizers are uniquely determined. `NormalizerTom` should never return an empty list.

`NormalizersTom` returns the list of positions of the classes of normalizers of subgroups in *tom*. In addition to the criteria for a single class of subgroup used by `NormalizerTom`, the approximations of normalizers for several classes are used and thus `NormalizersTom` may return better approximations than `NormalizerTom`.

```
gap> NormalizerTom( a5, 4 );
8
gap> NormalizersTom( a5 );
[ 9, 4, 6, 8, 7, 6, 7, 8, 9 ]
```

The example shows that a subgroup with class number 4 in A_5 (which is a Kleinian four group) is normalized by a subgroup in class 8. This class contains the subgroups of A_5 which are isomorphic to A_4.

5 ▶ `ContainedTom( tom, sub1, sub2 )` O

`ContainedTom` returns the number of subgroups in class *sub1* of the table of marks *tom* that are contained in one fixed member of the class *sub2*.

6 ▶ `ContainingTom( tom, sub1, sub2 )` O

`ContainingTom` returns the number of subgroups in class *sub2* of the table of marks *tom* that contain one fixed member of the class *sub1*.

```
gap> ContainedTom( a5, 3, 5 );  ContainedTom( a5, 3, 8 );
0
4
gap> ContainingTom( a5, 3, 5 );  ContainingTom( a5, 3, 8 );
0
2
```

7 ▶ `CyclicExtensionsTom( tom )` A
▶ `CyclicExtensionsTom( tom, p )` O
▶ `CyclicExtensionsTom( tom, list )` O

According to A. Dress [Dre69], two columns of the table of marks *tom* are equal modulo the prime p if and only if the corresponding subgroups are connected by a chain of normal extensions of order p.

In the second form, `CyclicExtensionsTom` returns the classes of this equivalence relation. In the third form, *list* must be a list of primes, and the return value is the list of classes of the relation obtained by considering chains of normal extensions of prime order where all primes are in *list*. In the first form, the result is the same as in the third form, with second argument the set of prime divisors of the size of the group of *tom*.

(This information is not used by `NormalizerTom` (see 68.9.4) although it might give additional restrictions in the search of normalizers.)

```
gap> CyclicExtensionsTom( a5, 2 );
[ [ 1, 2, 4 ], [ 3, 6 ], [ 5, 7 ], [ 8 ], [ 9 ] ]
```

8 ▶ `DecomposedFixedPointVector( tom, fix )` O

Let *tom* be the table of marks of the group G, say, and let *fix* be a vector of fixed point numbers w.r.t. an action of G, i.e., a vector which contains for each class of subgroups the number of fixed points under the given action. `DecomposedFixedPointVector` returns the decomposition of *fix* into rows of the table of marks. This decomposition corresponds to a decomposition of the action into transitive constituents. Trailing zeros in *fix* may be omitted.

```
gap> DecomposedFixedPointVector( a5, [ 16, 4, 1, 0, 1, 1, 1 ] );
[ 0, 0, 0, 0, 0, 1, 1 ]
```

The vector *fix* may be any vector of integers. The resulting decomposition, however, will not be integral, in general.

```
gap> DecomposedFixedPointVector( a5, [ 0, 0, 0, 0, 1, 1 ] );
[ 2/5, -1, -1/2, 0, 1/2, 1 ]
```

9 ▶ `EulerianFunctionByTom( tom, n[, sub] )` O

In the first form `EulerianFunctionByTom` computes the Eulerian function (see 37.16.3) of the underlying group G of the table of marks *tom*, that is, the number of n-tuples of elements in G that generate G. In the second form `EulerianFunctionByTom` computes the Eulerian function of each subgroup in the *sub*-th class of subgroups of *tom*.

For a group G whose table of marks is known, `EulerianFunctionByTom` is installed as a method for `EulerianFunction` (see 37.16.3).

```
gap> EulerianFunctionByTom( a5, 2 );
2280
gap> EulerianFunctionByTom( a5, 3 );
200160
gap> EulerianFunctionByTom( a5, 2, 3 );
8
```

10 ▶ `IntersectionsTom( tom, sub1, sub2 )` O

The intersections of the groups in the *sub1*-th conjugacy class of subgroups of the table of marks *tom* with the groups in the *sub2*-th conjugacy classes of subgroups of *tom* are determined up to conjugacy by the decomposition of the tensor product of their rows of marks. `IntersectionsTom` returns a list l that describes this decomposition. The i-th entry in l is the multiplicity of groups in the i-th conjugacy class as an intersection.

```
gap> IntersectionsTom( a5, 8, 8 );
[ 0, 0, 1, 0, 0, 0, 0, 1 ]
```

Any two subgroups of class number 8 (A_4) of A_5 are either equal and their intersection has again class number 8, or their intersection has class number 3, and is a cyclic subgroup of order 3.

11 ▶ `FactorGroupTom( tom, n )` O

For a table of marks *tom* of the group G, say, and the normal subgroup N of G corresponding to the n-th class of subgroups of *tom*, `FactorGroupTom` returns the table of marks of the factor group G/N.

```
gap> s4:= TableOfMarks( SymmetricGroup( 4 ) );
TableOfMarks( Sym( [ 1 .. 4 ] ) )
gap> LengthsTom( s4 );
[ 1, 3, 6, 4, 1, 3, 3, 4, 3, 1, 1 ]
gap> OrdersTom( s4 );
[ 1, 2, 2, 3, 4, 4, 4, 6, 8, 12, 24 ]
gap> s3:= FactorGroupTom( s4, 5 );
TableOfMarks( Group([ f1, f2 ]) )
gap> Display( s3 );
1:  6
2:  3 1
3:  2 . 2
4:  1 1 1 1
```

12 ▶ `MaximalSubgroupsTom( tom )` A
▶ `MaximalSubgroupsTom( tom, sub )` O

In the first form `MaximalSubgroupsTom` returns a list of length two, the first entry being the list of positions of the classes of maximal subgroups of the whole group of the table of marks *tom*, the second entry being

the list of class lengths of these groups. In the second form the same information for the *sub*-th class of subgroups is returned.

13 ▶ `MinimalSupergroupsTom( tom, sub )` O

For a table of marks *tom*, `MinimalSupergroupsTom` returns a list of length two, the first entry being the list of positions of the classes containing the minimal supergroups of the groups in the *sub*-th class of subgroups of *tom*, the second entry being the list of class lengths of these groups.

```
gap> MaximalSubgroupsTom( s4 );
[ [ 10, 9, 8 ], [ 1, 3, 4 ] ]
gap> MaximalSubgroupsTom( s4, 10 );
[ [ 5, 4 ], [ 1, 4 ] ]
gap> MinimalSupergroupsTom( s4, 5 );
[ [ 9, 10 ], [ 3, 1 ] ]
```

68.10 Standard Generators of Groups

An s-tuple of **standard generators** of a given group G is a vector $(g_1, g_2, \ldots, g_s)$ of elements $g_i \in G$ satisfying certain conditions (depending on the isomorphism type of G) such that

1. $\langle g_1, g_2, \ldots, g_s \rangle = G$ and
2. the vector is unique up to automorphisms of G, i.e., for two vectors $(g_1, g_2, \ldots, g_s)$ and $(h_1, h_2, \ldots, h_s)$ of standard generators, the map $g_i \mapsto h_i$ extends to an automorphism of G.

For details about standard generators, see [Wil96].

1 ▶ `StandardGeneratorsInfo( G )` A

When called with the group G, `StandardGeneratorsInfo` returns a list of records with at least one of the components `script` and `description`. Each such record defines **standard generators** of groups isomorphic to G, the i-th record is referred to as the i-th set of standard generators for such groups. The value of `script` is a dense list of lists, each encoding a command that has one of the following forms.

A **definition** $[i, n, k]$ or $[i, n]$
: means to search for an element of order n, and to take its k-th power as candidate for the i-th standard generator (the default for k is 1),

a **relation** $[i_1, k_1, i_2, k_2, \ldots, i_m, k_m, n]$ with $m > 1$
: means a check whether the element $g_{i_1}^{k_1} g_{i_2}^{k_2} \cdots g_{i_m}^{k_m}$ has order n; if g_j occurs then of course the j-th generator must have been defined before,

a **relation** $[[i_1, i_2, \ldots, i_m], slp, n]$
: means a check whether the result of the straight line program *slp* (see 35.8) applied to the candidates $g_{i_1}, g_{i_2}, \ldots, g_{i_m}$ has order n, where the candidates g_j for the j-th standard generators must have been defined before,

a **condition** $[[i_1, k_1, i_2, k_2, \ldots, i_m, k_m], f, v]$
: means a check whether the GAP function in the global list `StandardGeneratorsFunctions` (see 68.10.3) that is followed by the list f of strings returns the value v when it is called with G and $g_{i_1}^{k_1} g_{i_2}^{k_2} \cdots g_{i_m}^{k_m}$.

Optional components of the returned records are

`generators`
: a string of names of the standard generators,

`description`
: a string describing the `script` information in human readable form, in terms of the `generators` value,

`classnames`
: a list of strings, the i-th entry being the name of the conjugacy class containing the i-th standard generator, according to the ATLAS character table of the group (see 69.8.10), and

`ATLAS`
: a boolean; `true` means that the standard generators coincide with those defined in Rob Wilson's ATLAS of Group Representations (see [Wil]), and `false` means that this property is not guaranteed.

There is no default method for an arbitrary isomorphism type, since in general the definition of standard generators is not obvious.

The function `StandardGeneratorsOfGroup` (see 68.10.5) can be used to find standard generators of a given group isomorphic to G.

The `generators` and `description` values, if not known, can be computed by `HumanReadableDefinition` (see 68.10.2).

```
gap> StandardGeneratorsInfo( TableOfMarks( "L3(3)" ) );
[ rec( generators := "a, b",
      description := "|a|=2, |b|=3, |C(b)|=9, |ab|=13, |ababb|=4",
      script := [ [ 1, 2 ], [ 2, 3 ], [ [ 2, 1 ], [ "|C(",, ")|" ], 9 ],
          [ 1, 1, 2, 1, 13 ], [ 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 4 ] ],
      ATLAS := true ) ]
```

2 ▶ `HumanReadableDefinition( `*info*` )` F
▶ `ScriptFromString( `*string*` )` F

Let *info* be a record that is valid as value of `StandardGeneratorsInfo` (see 68.10.1). `HumanReadableDefinition` returns a string that describes the definition of standard generators given by the `script` component of *info* in human readable form. The names of the generators are taken from the `generators` component (default names `"a"`, `"b"` etc. are computed if necessary), and the result is stored in the `description` component.

`ScriptFromString` does the converse of `HumanReadableDefinition`, i.e., it takes a string *string* as returned by `HumanReadableDefinition`, and returns a corresponding `script` list.

If "condition" lines occur in the script (see 68.10.1) then the functions that occur must be contained in `StandardGeneratorsFunctions` (see 68.10.3).

```
gap> scr:= ScriptFromString( "|a|=2, |b|=3, |C(b)|=9, |ab|=13, |ababb|=4" );
[ [ 1, 2 ], [ 2, 3 ], [ [ 2, 1 ], [ "|C(",, ")|" ], 9 ], [ 1, 1, 2, 1, 13 ],
  [ 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 4 ] ]
gap> info:= rec( script:= scr );
rec( script := [ [ 1, 2 ], [ 2, 3 ], [ [ 2, 1 ], [ "|C(",, ")|" ], 9 ],
      [ 1, 1, 2, 1, 13 ], [ 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 4 ] ] )
gap> HumanReadableDefinition( info );
"|a|=2, |b|=3, |C(b)|=9, |ab|=13, |ababb|=4"
gap> info;
rec( script := [ [ 1, 2 ], [ 2, 3 ], [ [ 2, 1 ], [ "|C(",, ")|" ], 9 ],
      [ 1, 1, 2, 1, 13 ], [ 1, 1, 2, 1, 1, 1, 2, 1, 2, 1, 4 ] ],
  generators := "a, b",
  description := "|a|=2, |b|=3, |C(b)|=9, |ab|=13, |ababb|=4" )
```

3 ▶ `StandardGeneratorsFunctions` V

`StandardGeneratorsFunctions` is a list of even length. At position $2i-1$, a function of two arguments is stored, which are expected to be a group and a group element. At position $2i$ a list of strings is stored such

that first inserting a generator name in all holes and then forming the concatenation yields a string that describes the function at the previous position; this string must contain the generator enclosed in round brackets `(` and `)`.

This list is used by the functions `StandardGeneratorsInfo` (see 68.10.1), `HumanReadableDefinition`, and `ScriptFromString` (see 68.10.2). Note that the lists at even positions must be pairwise different.

```
gap> StandardGeneratorsFunctions{ [ 1, 2 ] };
[ function( G, g ) ... end, [ "|C(",, ")|" ] ]
```

4 ▶ `IsStandardGeneratorsOfGroup(` *info*`,` *G*`,` *gens* `)` F

Let *info* be a record that is valid as value of `StandardGeneratorsInfo` (see 68.10.1), *G* a group, and *gens* a list of generators for *G*. In this case, `IsStandardGeneratorsOfGroup` returns `true` if *gens* satisfies the conditions of the `script` component of *info*, and `false` otherwise.

Note that the result `true` means that *gens* is a list of standard generators for *G* only if *G* has the isomorphism type for which *info* describes standard generators.

5 ▶ `StandardGeneratorsOfGroup(` *info*`,` *G*`[,` *randfunc*`] )` F

Let *info* be a record that is valid as value of `StandardGeneratorsInfo` (see 68.10.1), and *G* a group of the isomorphism type for which *info* describes standard generators. In this case, `StandardGeneratorsOfGroup` returns a list of standard generators (see Section 68.10) of *G*.

The optional argument *randfunc* must be a function that returns an element of *G* when called with *G*; the default is `PseudoRandom`.

In each call to `StandardGeneratorsOfGroup`, the `script` component of *info* is scanned line by line. *randfunc* is used to find an element of the prescribed order whenever a definition line is met, and for the relation and condition lines in the `script` list, the current generator candidates are checked; if a condition is not fulfilled, all candidates are thrown away, and the procedure starts again with the first line. When the conditions are fulfilled after processing the last line of the `script` list, the standard generators are returned.

Note that if *G* has the wrong isomorphism type then `StandardGeneratorsOfGroup` returns a list of elements in *G* that satisfy the conditions of the `script` component of *info* if such elements exist, and does not terminate otherwise. In the former case, obviously the returned elements need not be standard generators of *G*.

```
gap> a5:= AlternatingGroup( 5 );
Alt( [ 1 .. 5 ] )
gap> info:= StandardGeneratorsInfo( TableOfMarks( "A5" ) )[1];
rec( generators := "a, b", description := "|a|=2, |b|=3, |ab|=5",
  script := [ [ 1, 2 ], [ 2, 3 ], [ 1, 1, 2, 1, 5 ] ], ATLAS := true )
gap> IsStandardGeneratorsOfGroup( info, a5, [ (1,3)(2,4), (3,4,5) ] );
true
gap> IsStandardGeneratorsOfGroup( info, a5, [ (1,3)(2,4), (1,2,3) ] );
false
gap> s5:= SymmetricGroup( 5 );;
gap> RepresentativeAction( s5, [ (1,3)(2,4), (3,4,5) ],
>        StandardGeneratorsOfGroup( info, a5 ), OnPairs ) <> fail;
true
```

68.11 Accessing Subgroups via Tables of Marks

Let *tom* be the table of marks of the group G, and assume that *tom* has access to G via the `UnderlyingGroup` value (see 68.7.7). Then it makes sense to use *tom* and its ordering of conjugacy classes of subgroups of G for storing information for constructing representatives of these classes. The group G is in general not sufficient for this, *tom* needs more information; this is available if and only if the `IsTableOfMarksWithGens` value of *tom* is `true` (see 68.11.3). In this case, `RepresentativeTom` (see 68.11.4) can be used to get a subgroup of the i-th class, for all i.

GAP provides two different possibilities to store generators of the representatives of classes of subgroups. The first is implemented by the attribute `GeneratorsSubgroupsTom` (see 68.11.1), which uses explicit generators. The second, more general, possibility is implemented by the attributes `StraightLineProgramsTom` (see 68.11.2) and `StandardGeneratorsInfo` (see 68.11.5). The `StraightLineProgramsTom` value encodes the generators as straight line programs (see 35.8) that evaluate to the generators in question when applied to standard generators of G. This means that on the one hand, standard generators of G must be known in order to use `StraightLineProgramsTom`. On the other hand, the straight line programs allow one to compute easily generators not only of a subgroup U of G but also generators of the image of U in any representation of G, provided that one knows standard generators of the image of G under this representation (see 68.11.4 for details and an example).

1 ▶ `GeneratorsSubgroupsTom(` *tom* `)` A

Let *tom* be a table of marks with `IsTableOfMarksWithGens` value `true`. Then `GeneratorsSubgroupsTom` returns a list of length two, the first entry being a list l of elements of the group stored as `UnderlyingGroup` value of *tom*, the second entry being a list that contains at position i a list of positions in l of generators of a representative of a subgroup in class i.

The `GeneratorsSubgroupsTom` value is known for all tables of marks that have been computed with `TableOfMarks` (see 68.3.1) from a group, and there is a method to compute the value for a table of marks that admits `RepresentativeTom` (see 68.11.4).

2 ▶ `StraightLineProgramsTom(` *tom* `)` A

For a table of marks *tom* with `IsTableOfMarksWithGens` value `true`, `StraightLineProgramsTom` returns a list that contains at position i either a list of straight line programs or a straight line program (see 35.8), encoding the generators of a representative of the i-th conjugacy class of subgroups of `UnderlyingGroup(` *tom* `)`; in the former case, each straight line program returns a generator, in the latter case, the program returns the list of generators.

There is no default method to compute the `StraightLineProgramsTom` value of a table of marks if they are not yet stored. The value is known for all tables of marks that belong to the GAP library of tables of marks (see 68.14).

3 ▶ `IsTableOfMarksWithGens(` *tom* `)` F

This filter shall express the union of the filters `IsTableOfMarks and HasStraightLineProgramsTom` and `IsTableOfMarks and HasGeneratorsSubgroupsTom`. If a table of marks *tom* has this filter set then *tom* can be asked to compute information that is in general not uniquely determined by a table of marks, for example the positions of derived subgroups or normalizers of subgroups (see 68.9.2, 68.9.4).

```
gap> a5:= TableOfMarks( "A5" );;  IsTableOfMarksWithGens( a5 );
true
gap> HasGeneratorsSubgroupsTom( a5 );  HasStraightLineProgramsTom( a5 );
false
true
gap> alt5:= TableOfMarks( AlternatingGroup( 5 ) );;
gap> IsTableOfMarksWithGens( alt5 );
```

```
true
gap> HasGeneratorsSubgroupsTom( alt5 );  HasStraightLineProgramsTom( alt5 );
true
false
gap> progs:= StraightLineProgramsTom( a5 );;
gap> OrdersTom( a5 );
[ 1, 2, 3, 4, 5, 6, 10, 12, 60 ]
gap> IsCyclicTom( a5, 4 );
false
gap> Length( progs[4] );
2
gap> progs[4][1];
<straight line program>
gap> Display( progs[4][1] );  # first generator of an el. ab group of order 4
# input:
r:= [ g1, g2 ];
# program:
r[3]:= r[2]*r[1];
r[4]:= r[3]*r[2]^-1*r[1]*r[3]*r[2]^-1*r[1]*r[2];
# return value:
r[4]
gap> x:= [ [ Z(2)^0, 0*Z(2) ], [ Z(2^2), Z(2)^0 ] ];;
gap> y:= [ [ Z(2^2), Z(2)^0 ], [ 0*Z(2), Z(2^2)^2 ] ];;
gap> res1:= ResultOfStraightLineProgram( progs[4][1], [ x, y ] );
[ [ Z(2)^0, 0*Z(2) ], [ Z(2^2)^2, Z(2)^0 ] ]
gap> res2:= ResultOfStraightLineProgram( progs[4][2], [ x, y ] );
[ [ Z(2)^0, 0*Z(2) ], [ Z(2^2), Z(2)^0 ] ]
gap> w:= y*x;;
gap> res1 = w*y^-1*x*w*y^-1*x*y;
true
gap> subgrp:= Group( res1, res2 );;  Size( subgrp );  IsCyclic( subgrp );
4
false
```

4 ▶ `RepresentativeTom( tom, sub )` O
▶ `RepresentativeTomByGenerators( tom, sub, gens )` O
▶ `RepresentativeTomByGeneratorsNC( tom, sub, gens )` O

Let *tom* be a table of marks with `IsTableOfMarksWithGens` value `true` (see 68.11.3), and *sub* a positive integer. `RepresentativeTom` returns a representative of the *sub*-th conjugacy class of subgroups of *tom*.

`RepresentativeTomByGenerators` and `RepresentativeTomByGeneratorsNC` return a representative of the *sub*-th conjugacy class of subgroups of *tom*, as a subgroup of the group generated by *gens*. This means that the standard generators of *tom* are replaced by *gens*.

`RepresentativeTomByGenerators` checks whether mapping the standard generators of *tom* to *gens* extends to a group isomorphism, and returns `fail` if not. `RepresentativeTomByGeneratorsNC` omits all checks. So `RepresentativeTomByGenerators` is thought mainly for debugging purposes; note that when several representatives are constructed, it is cheaper to construct (and check) the isomorphism once, and to map the groups returned by `RepresentativeTom` under this isomorphism. The idea behind `RepresentativeTomByGeneratorsNC`, however, is to avoid the overhead of using isomorphisms when *gens* are known to be standard generators.

```
gap> RepresentativeTom( a5, 4 );
Group([ (2,3)(4,5), (2,4)(3,5) ])
```

5 ► `StandardGeneratorsInfo( tom )` A

For a table of marks *tom*, a stored value of `StandardGeneratorsInfo` equals the value of this attribute for the underlying group (see 68.7.7) of *tom*, cf. Section 68.10.

In this case, the `GeneratorsOfGroup` value of the underlying group G of *tom* is assumed to be in fact a list of standard generators for G; So one should be careful when setting the `StandardGeneratorsInfo` value by hand.

There is no default method to compute the `StandardGeneratorsInfo` value of a table of marks if it is not yet stored.

```
gap> std:= StandardGeneratorsInfo( a5 );
[ rec( generators := "a, b", description := "|a|=2, |b|=3, |ab|=5",
      script := [ [ 1, 2 ], [ 2, 3 ], [ 1, 1, 2, 1, 5 ] ], ATLAS := true ) ]
gap> # Now find standard generators of an isomorphic group.
gap> g:= SL(2,4);;
gap> repeat
>   x:= PseudoRandom( g );
> until Order( x ) = 2;
gap> repeat
>   y:= PseudoRandom( g );
> until Order( y ) = 3 and Order( x*y ) = 5;
gap> # Compute a representative w.r.t. these generators.
gap> RepresentativeTomByGenerators( a5, 4, [ x, y ] );
Group([ [ [ Z(2)^0, Z(2^2)^2 ], [ 0*Z(2), Z(2)^0 ] ],
  [ [ Z(2)^0, Z(2^2) ], [ 0*Z(2), Z(2)^0 ] ] ])
gap> # Show that the new generators are really good.
gap> grp:= UnderlyingGroup( a5 );;
gap> iso:= GroupGeneralMappingByImages( grp, g,
>              GeneratorsOfGroup( grp ), [ x, y ] );;
gap> IsGroupHomomorphism( iso );
true
gap> IsBijective( iso );
true
```

68.12 The Interface between Tables of Marks and Character Tables

The following examples require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

```
gap> LoadPackage( "ctbllib" );
true
```

1 ► `FusionCharTableTom( tbl, tom )` O
► `PossibleFusionsCharTableTom( tbl, tom[, options] )` O

Let *tbl* be the ordinary character table of the group G, say, and *tom* the table of marks of G. `FusionCharTableTom` determines the fusion of the classes of elements from *tbl* to the classes of cyclic subgroups on *tom*, that is, a list that contains at position i the position of the class of cyclic subgroups in *tom* that are generated by elements in the i-th conjugacy class of elements in *tbl*.

Three cases are handled differently.

1. The fusion is explicitly stored on *tbl*. Then nothing has to be done. This happens only if both *tbl* and *tom* are tables from the GAP library (see 68.14 and the manual of the GAP Character Table Library).
2. The `UnderlyingGroup` values of *tbl* and *tom* are known and equal. Then the group is used to compute the fusion.
3. There is neither fusion nor group information available. In this case only necessary conditions can be checked, and if they are not sufficient to detemine the fusion uniquely then `fail` is returned by `FusionCharTableTom`.

`PossibleFusionsCharTableTom` computes the list of possible fusions from *tbl* to *tom*, according to the criteria that have been checked. So if `FusionCharTableTom` returns a unique fusion then the list returned by `PossibleFusionsCharTableTom` for the same arguments contains exactly this fusion, and if `FusionCharTableTom` returns `fail` then the length of this list is different from 1.

The optional argument *options* must be a record that may have the following components.

`fusionmap`
: a parametrized map which is an approximation of the desired map,

`quick`
: a Boolean; if `true` then as soon as only one possibility remains this possibility is returned immediately; the default value is `false`.

```
gap> a5c:= CharacterTable( "A5" );;
gap> fus:= FusionCharTableTom( a5c, a5 );
[ 1, 2, 3, 5, 5 ]
```

2 ► `PermCharsTom( `*fus*`, `*tom*` )` O
► `PermCharsTom( `*tbl*`, `*tom*` )` O

`PermCharsTom` returns the list of transitive permutation characters from the table of marks *tom*. In the first form, *fus* must be the fusion map from the ordinary character table of the group of *tom* to *tom* (see 68.12.1). In the second form, *tbl* must be the character table of the group of which *tom* is the table of marks. If the fusion map is not uniquely determined (see 68.12.1) then `fail` is returned.

If the fusion map *fus* is given as first argument then each transitive permutation character is represented by its values list. If the character table *tbl* is given then the permutation characters are class function objects (see Chapter 70).

```
gap> PermCharsTom( a5c, a5 );
[ Character( CharacterTable( "A5" ), [ 60, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 30, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 20, 0, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 15, 3, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 12, 0, 0, 2, 2 ] ),
  Character( CharacterTable( "A5" ), [ 10, 2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 6, 2, 0, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ) ]
gap> PermCharsTom( fus, a5 )[1];
[ 60, 0, 0, 0, 0 ]
```

68.13 Generic Construction of Tables of Marks

The following three operations construct a table of marks only from the data given, i.e., without underlying group.

1 ▶ `TableOfMarksCyclic( n )` O

`TableOfMarksCyclic` returns the table of marks of the cyclic group of order n.

A cyclic group of order n has as its subgroups for each divisor d of n a cyclic subgroup of order d.

2 ▶ `TableOfMarksDihedral( n )` O

`TableOfMarksDihedral` returns the table of marks of the dihedral group of order m.

For each divisor d of m, a dihedral group of order $m = 2n$ contains subgroups of order d according to the following rule. If d is odd and divides n then there is only one cyclic subgroup of order d. If d is even and divides n then there are a cyclic subgroup of order d and two classes of dihedral subgroups of order d (which are cyclic, too, in the case $d = 2$, see the example below). Otherwise (i.e., if d does not divide n) there is just one class of dihedral subgroups of order d.

3 ▶ `TableOfMarksFrobenius( p, q )` O

`TableOfMarksFrobenius` computes the table of marks of a Frobenius group of order pq, where p is a prime and q divides $p - 1$.

```
gap> Display( TableOfMarksCyclic( 6 ) );
1:  6
2:  3 3
3:  2 . 2
4:  1 1 1 1

gap> Display( TableOfMarksDihedral( 12 ) );
 1:  12
 2:   6 6
 3:   6 . 2
 4:   6 . . 2
 5:   4 . . . 4
 6:   3 3 1 1 . 1
 7:   2 2 . . 2 . 2
 8:   2 . 2 . 2 . . 2
 9:   2 . . 2 2 . . . 2
10:   1 1 1 1 1 1 1 1 1 1

gap> Display( TableOfMarksFrobenius( 5, 4 ) );
1:  20
2:  10 2
3:   5 1 1
4:   4 . . 4
5:   2 2 . 2 2
6:   1 1 1 1 1 1
```

68.14 The Library of Tables of Marks

The GAP package `TomLib` provides access to several hundred tables of marks of almost simple groups and their maximal subgroups. If this package is installed then the tables from this database can be accessed via `TableOfMarks` with argument a string (see 68.3.1). If also the GAP Character Table Library is installed and contains the ordinary character table of the group for which one wants to fetch the table of marks then one can also call `TableOfMarks` with argument the character table.

A list of all names of tables of marks in the database can be obtained via `AllLibTomNames`.

```
gap> names:= AllLibTomNames();;
gap> "A5" in names;
true
```

69 Character Tables

This chapter describes operations for **character tables of finite groups**.

Operations for **characters** (or, more general, **class functions**) are described in Chapter 70.

For a description of the GAP Library of Character Tables, see the separate manual for the GAP package `ctbllib`.

Several examples in this chapter require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

```
gap> LoadPackage( "ctbllib" );
true
```

69.1 Some Remarks about Character Theory in GAP

It seems to be necessary to state some basic facts –and maybe warnings– at the beginning of the character theory package. This holds for people who are familiar with character theory because there is no global reference on computational character theory, although there are many papers on this topic, such as [NPP84] or [LP91]. It holds, however, also for people who are familiar with GAP because the general concept of domains (see Chapter 12.4) plays no important role here –we will justify this later in this section.

Intuitively, **characters** (or more generally, **class functions**) of a finite group G can be thought of as certain mappings defined on G, with values in the complex number field; the set of all characters of G forms a semiring, with both addition and multiplication defined pointwise, which is naturally embedded into the ring of **generalized** (or **virtual**) **characters** in the natural way. A $\mathbb{Z}$-basis of this ring, and also a vector space basis of the complex vector space of class functions of G, is given by the irreducible characters of G.

At this stage one could ask where there is a problem, since all these algebraic structures are supported by GAP. But in practice, these structures are of minor importance, compared to individual characters and the **character tables** themselves (which are not domains in the sense of GAP).

For computations with characters of a finite group G with n conjugacy classes, say, we fix an ordering of the classes, and then identify each class with its position according to this ordering. Each character of G can be represented by a list of length n in which the character value for elements of the i-th class is stored at the i-th position. Note that we need not know the conjugacy classes of G physically, even our knowledge of G may be implicit in the sense that, e.g., we know how many classes of involutions G has, and which length these classes have, but we never have seen an element of G, or a presentation or representation of G. This allows us to work with the character tables of very large groups, e.g., of the so-called monster, where GAP has (currently) no chance to deal with the group.

As a consequence, also other information involving characters is given implicitly. For example, we can talk about the kernel of a character not as a group but as a list of classes (more exactly: a list of their positions according to the chosen ordering of classes) forming this kernel; we can deduce the group order, the contained cyclic subgroups and so on, but we do not get the group itself.

So typical calculations with characters involve loops over lists of character values. For example, the scalar product of two characters χ, ψ of G given by

$$[\chi, \psi] = \frac{1}{|G|} \sum_{g \in G} \chi(g)\psi(g^{-1})$$

can be written as

```
Sum( [ 1 .. n ], i -> SizesConjugacyClasses( t )[i] * chi[i]
                          * ComplexConjugate( psi[i] ) );
```

where `t` is the character table of G, and `chi`, `psi` are the lists of values of χ, ψ, respectively.

It is one of the advantages of character theory that after one has translated a problem concerning groups into a problem concerning only characters, the necessary calculations are mostly simple. For example, one can often prove that a group is a Galois group over the rationals using calculations with structure constants that can be computed from the character table, and information about (the character tables of) maximal subgroups. When one deals with such questions, the translation back to groups is just an interpretation by the user, it does not take place in GAP.

GAP uses character **tables** to store information such as class lengths, element orders, the irreducible characters of G etc. in a consistent way; in the example above, we have seen that `SizesConjugacyClasses( t )` is the list of class lengths of the character table `t`. Note that the values of these attributes rely on the chosen ordering of conjugacy classes, a character table is not determined by something similar to generators of groups or rings in GAP where knowledge could in principle be recovered from the generators but is stored mainly for the sake of efficiency.

Note that the character table of a group G in GAP must **not** be mixed up with the list of complex irreducible characters of G. The irreducible characters are stored in a character table via the attribute `Irr` (see 69.8.2).

Two further important instances of information that depends on the ordering of conjugacy classes are **power maps** and **fusion maps**. Both are represented as lists of integers in GAP. The k-th power map maps each class to the class of k-th powers of its elements, the corresponding list contains at each position the position of the image. A class fusion map between the classes of a subgroup H of G and the classes of G maps each class c of H to that class of G that contains c, the corresponding list contains again the positions of image classes; if we know only the character tables of H and G but not the groups themselves, this means with respect to a fixed embedding of H into G. More about power maps and fusion maps can be found in Chapter 71.

So class functions, power maps, and fusion maps are represented by lists in GAP. If they are plain lists then they are regarded as class functions etc. of an appropriate character table when they are passed to GAP functions that expect class functions etc. For example, a list with all entries equal to 1 is regarded as the trivial character if it is passed to a function that expects a character. Note that this approach requires the character table as an argument for such a function.

One can construct class function objects that store their underlying character table and other attribute values (see Chapter 70). This allows one to omit the character table argument in many functions, and it allows one to use infix operations for tensoring or inducing class functions.

69.2 History of Character Theory Stuff in GAP

GAP provides functions for dealing with group characters since the version GAP 3.1, which was released in March 1992. The reason for adding this branch of mathematics to the topics of GAP was (apart from the usefulness of character theoretic computations in general) the insight that GAP provides an ideal environment for developing the algorithms needed. In particular, it had been decided at Lehrstuhl D für Mathematik that the CAS system (a standalone Fortran program together with a database of character tables, see [NPP84])

should not be developed further and the functionality of CAS should be made available in GAP. The background was that extending CAS (by new Fortran code) had turned out to be much less flexible than writing analogous GAP library code.

For integrating the existing character theory algorithms, GAP's memory management and long integer arithmetic were useful as well as the list handling –it is an important feature of character theoretic methods that questions about groups are translated into manipulations of lists; on the other hand, the datatype of cyclotomics (see Chapter 18.1.2) was added to the GAP kernel because of the character theory algorithms. For developing further code, also other areas of GAP's library became interesting, such as permutation groups, finite fields, and polynomials.

The development of character theory code for GAP has been supported by several DFG grants, in particular the project "Representation Theory of Finite Groups and Finite Dimensional Algebras" (until 1991), and the Schwerpunkt "Algorithmische Zahlentheorie und Algebra" (from 1991 until 1997). Besides that, several Diploma theses at Lehrstuhl D were concerned with the development and/or implementation of algorithms dealing with characters in GAP.

The major contributions can be listed as follows.

- The arithmetic for the cyclotomics data type, following [Zum89], was first implemented by Marco van Meegen; an alternative approach was studied in the diploma thesis of Michael Scherner (see [Sch92]) but was not efficient enough; later Martin Schönert replaced the implementation by a better one.
- The basic routines for characters and character tables were written by Thomas Breuer and Götz Pfeiffer.
- The lattice related functions, such as `LLL`, `OrthogonalEmbeddings`, and `DnLattice`, were implemented by Ansgar Kaup (see [Kau92]).
- Functions for computing possible class fusions, possible power maps, and table automorphisms were written by Thomas Breuer (see [Bre91]).
- Functions for computing possible permutation characters were written by Thomas Breuer (see [Bre91]) and Götz Pfeiffer (see [Pfe91]).
- Functions for computing character tables from groups were written by Alexander Hulpke (Dixon-Schneider algorithm, see [Hul93]) and Hans Ulrich Besche (Baum algorithm and Conlon algorithm, see [Bes92]).
- Functions for dealing with Clifford matrices were written by Ute Schiffer (see [Sch94]).
- Functions for monomiality questions were written by Thomas Breuer and Erzsébet Horváth.

Since then, the code has been maintained and extended further by Alexander Hulpke (code related to his implementation of the Dixon-Schneider algorithm) and Thomas Breuer.

Currently GAP does not provide special functionality for computing Brauer character tables, but there is an interface to the MOC system (see [HJLP]), and the GAP Character Table Library contains many known Brauer character tables.

69.3 Creating Character Tables

There are in general five different ways to get a character table in GAP. You can

1. compute the table from a group,
2. read a file that contains the table data,
3. construct the table using generic formulae,
4. derive it from known character tables, or
5. combine partial information about conjugacy classes, power maps of the group in question, and about (character tables of) some subgroups and supergroups.

In 1., the computation of the irreducible characters is the hardest part; the different algorithms available for this are described in 69.12. Possibility 2. is used for the character tables in the GAP Character Table Library, see the manual of this library. Generic character tables –as addressed by 3.– are described in "ctbllib:generic character tables" in the manual of the GAP Character Table Library. Several occurrences of 4. are described in 69.18. The last of the above possibilities **@is currently not supported and will be described in a chapter of its own when it becomes available@**.

The operation `CharacterTable` (see 69.3.1) can be used for the cases 1.–3.

1 ▶ `CharacterTable(` *G* `)` O
▶ `CharacterTable(` *G*`,` *p* `)` O
▶ `CharacterTable(` *ordtbl*`,` *p* `)` O
▶ `CharacterTable(` *name*`[,` *param*`]` `)` O

Called with a group G, `CharacterTable` calls the attribute `OrdinaryCharacterTable` (see 69.8.4). Called with first argument a group G or an ordinary character table *ordtbl*, and second argument a prime p, `CharacterTable` calls the operation `BrauerTable` (see 69.3.2). Called with a string *name* and perhaps optional parameters *param*, `CharacterTable` delegates to `CharacterTableFromLibrary`, which tries to access the GAP Character Table Library (see the manual of this library for an overview of admissible strings *name*).

Probably the most interesting information about the character table is its list of irreducibles, which can be accessed as the value of the attribute `Irr` (see 69.8.2). If the argument of `CharacterTable` is a string *name* then the irreducibles are just read from the library file, therefore the returned table stores them already. However, if `CharacterTable` is called with a group G or with an ordinary character table *ordtbl*, the irreducible characters are **not** computed by `CharacterTable`. They are only computed when the `Irr` value is accessed for the first time, for example when `Display` is called for the table (see 69.11). This means for example that `CharacterTable` returns its result very quickly, and the first call of `Display` for this table may take some time because the irreducible characters must be computed at that time before they can be displayed together with other information stored on the character table. The value of the filter `HasIrr` indicates whether the irreducible characters have been computed already.

The reason why `CharacterTable` does not compute the irreducible characters is that there are situations where one only needs the "table head", that is, the information about class lengths, power maps etc., but not the irreducibles. For example, if one wants to inspect permutation characters of a group then all one has to do is to induce the trivial characters of subgroups one is interested in; for that, only class lengths and the class fusion are needed. Or if one wants to compute the Molien series (see 70.12.1) for a given complex matrix group, the irreducible characters of this group are in general of no interest.

For details about different algorithms to compute the irreducible characters, see 69.12.

If the group G is given as an argument, `CharacterTable` accesses the conjugacy classes of G and therefore causes that these classes are computed if they were not yet stored (see 69.6).

2 ▶ `BrauerTable(` *ordtbl*`,` *p* `)` O
▶ `BrauerTable(` *G*`,` *p* `)` O
▶ `BrauerTableOp(` *ordtbl*`,` *p* `)` O
▶ `ComputedBrauerTables(` *ordtbl* `)` AM

Called with an ordinary character table *ordtbl* or a group G, `BrauerTable` returns its p-modular character table if GAP can compute this table, and `fail` otherwise. The p-modular table can be computed for p-solvable groups (using the Fong-Swan Theorem) and in the case that *ordtbl* is a table from the GAP character table library for which also the p-modular table is contained in the table library.

The default method for a group and a prime delegates to `BrauerTable` for the ordinary character table of this group. The default method for *ordtbl* uses the attribute `ComputedBrauerTables` for storing the computed Brauer table at position p, and calls the operation `BrauerTableOp` for computing values that are not yet known.

So if one wants to install a new method for computing Brauer tables then it is sufficient to install it for `BrauerTableOp`.

The `\mod` operator for a character table and a prime (see 69.7) delegates to `BrauerTable`.

```
gap> g:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> tbl:= CharacterTable( g );;  HasIrr( tbl );
false
gap> tblmod2:= CharacterTable( tbl, 2 );
BrauerTable( Sym( [ 1 .. 4 ] ), 2 )
gap> tblmod2 = CharacterTable( tbl, 2 );
true
gap> tblmod2 = BrauerTable( tbl, 2 );
true
gap> tblmod2 = BrauerTable( g, 2 );
true
gap> CharacterTable( "A5" );
CharacterTable( "A5" )
gap> CharacterTable( "Symmetric", 4 );
CharacterTable( "Sym(4)" )
gap> ComputedBrauerTables( tbl );
[ , BrauerTable( Sym( [ 1 .. 4 ] ), 2 ) ]
```

3 ► `SupportedCharacterTableInfo` V

`SupportedCharacterTableInfo` is a list that contains at position $3i-2$ an attribute getter function, at position $3i-1$ the name of this attribute, and at position $3i$ a list containing one or two of the strings `"class"`, `"character"`, depending on whether the attribute value relies on the ordering of classes or characters. This allows one to set exactly the components with these names in the record that is later converted to the new table, in order to use the values as attribute values. So the record components that shall **not** be regarded as attribute values can be ignored. Also other attributes of the old table are ignored.

`SupportedCharacterTableInfo` is used when (ordinary or Brauer) character table objects are created from records, using `ConvertToCharacterTable` (see 69.3.4).

New attributes and properties can be notified to `SupportedCharacterTableInfo` by creating them with `DeclareAttributeSuppCT` and `DeclarePropertySuppCT` instead of `DeclareAttribute` and `DeclareProperty`.

4 ► `ConvertToCharacterTable(` *record* `)` F
► `ConvertToCharacterTableNC(` *record* `)` F

Let *record* be a record. `ConvertToCharacterTable` converts *record* into a component object (see 3.9 in "Programming in GAP") representing a character table. The values of those components of *record* whose names occur in `SupportedCharacterTableInfo` (see 69.3.3) correspond to attribute values of the returned character table. All other components of the record simply become components of the character table object.

If inconsistencies in *record* are detected, `fail` is returned. *record* must have the component `UnderlyingCharacteristic` bound (see 69.8.9), since this decides about whether the returned character table lies in `IsOrdinaryTable` or in `IsBrauerTable` (see 69.4.1, 69.4.1).

`ConvertToCharacterTableNC` does the same except that all checks of *record* are omitted.

An example of a conversion from a record to a character table object can be found in Section 69.11.2.

69.4 Character Table Categories

1 ▶ `IsNearlyCharacterTable( obj )` C
▶ `IsCharacterTable( obj )` C
▶ `IsOrdinaryTable( obj )` C
▶ `IsBrauerTable( obj )` C
▶ `IsCharacterTableInProgress( obj )` C

Every "character table like object" in GAP lies in the category `IsNearlyCharacterTable`. There are four important subcategories, namely the **ordinary** tables in `IsOrdinaryTable`, the **Brauer** tables in `IsBrauerTable`, the union of these two in `IsCharacterTable`, and the **incomplete ordinary** tables in `IsCharacterTableInProgress`.

We want to distinguish ordinary and Brauer tables because a Brauer table may delegate tasks to the ordinary table of the same group, for example the computation of power maps. A Brauer table is constructed from an ordinary table and stores this table upon construction (see 69.8.4).

Furthermore, `IsOrdinaryTable` and `IsBrauerTable` denote character tables that provide enough information to compute all power maps and irreducible characters (and in the case of Brauer tables to get the ordinary table), for example because the underlying group (see 69.6.1) is known or because the table is a library table (see the manual of the GAP Character Table Library). We want to distinguish these tables from partially known ordinary tables that cannot be asked for all power maps or all irreducible characters.

The character table objects in `IsCharacterTable` are always immutable (see 12.6). This means mainly that the ordering of conjugacy classes used for the various attributes of the character table cannot be changed; see 69.19 for how to compute a character table with a different ordering of classes.

The GAP objects in `IsCharacterTableInProgress` represent incomplete ordinary character tables. This means that not all irreducible characters, not all power maps are known, and perhaps even the number of classes and the centralizer orders are known. Such tables occur when the character table of a group G is constructed using character tables of related groups and information about G but for example without explicitly computing the conjugacy classes of G. An object in `IsCharacterTableInProgress` is first of all **mutable**, so **nothing is stored automatically** on such a table, since otherwise one has no control of side-effects when a hypothesis is changed. Operations for such tables may return more general values than for other tables, for example class functions may contain unknowns (see Chapter 19) or lists of possible values in certain positions, the same may happen also for power maps and class fusions (see 71.3). **@Incomplete tables in this sense are currently not supported and will be described in a chapter of their own when they become available.@** Note that the term "incomplete table" shall express that GAP cannot compute certain values such as irreducible characters or power maps. A table with access to its group is therefore always complete, also if its irreducible characters are not yet stored.

```
gap> g:= SymmetricGroup( 4 );;
gap> tbl:= CharacterTable( g );  modtbl:= tbl mod 2;
CharacterTable( Sym( [ 1 .. 4 ] ) )
BrauerTable( Sym( [ 1 .. 4 ] ), 2 )
gap> IsCharacterTable( tbl );  IsCharacterTable( modtbl );
true
true
gap> IsBrauerTable( modtbl );  IsBrauerTable( tbl );
true
false
gap> IsOrdinaryTable( tbl );  IsOrdinaryTable( modtbl );
true
false
gap> IsCharacterTable( g );  IsCharacterTable( Irr( g ) );
false
```

```
false
```

2 ▶ `InfoCharacterTable` V

is the info class (see 7.4) for computations with character tables.

3 ▶ `NearlyCharacterTablesFamily` V

Every character table like object lies in this family (see 13.1).

69.5 Conventions for Character Tables

The following few conventions should be noted.

- The class of the **identity element** is expected to be the first one; thus the degree of a character is the character value at position 1.
- The **trivial character** of a character table need not be the first in the list of irreducibles.
- Most functions that take a character table as an argument and work with characters expect these characters as an argument, too. For some functions, the list of irreducible characters serves as the default, i.e, the value of the attribute `Irr` (see 69.8.2); in these cases, the `Irr` value is automatically computed if it was not yet known.
- For a stored class fusion, the image table is denoted by its `Identifier` value (see 69.8.11); each library table has a unique identifier by which it can be accessed (see "ctbllib:access to library character tables" in the manual for the GAP Character Table Library), tables constructed from groups get an identifier that is unique in the current GAP session.

69.6 The Interface between Character Tables and Groups

For a character table with underlying group (see 69.6.1), the interface between table and group consists of three attribute values, namely the **group**, the **conjugacy classes** stored in the table (see `ConjugacyClasses` below) and the **identification** of the conjugacy classes of table and group (see `IdentificationOfConjugacyClasses` below).

Character tables constructed from groups know these values upon construction, and for character tables constructed without groups, these values are usually not known and cannot be computed from the table.

However, given a group G and a character table of a group isomorphic to G (for example a character table from the GAP table library), one can tell GAP to use the given table as the character table of G (see 69.6.4).

Tasks may be delegated from a group to its character table or vice versa only if these three attribute values are stored in the character table.

1 ▶ `UnderlyingGroup(` *ordtbl* `)` A

For an ordinary character table *ordtbl* of a finite group, the group can be stored as value of `UnderlyingGroup`.

Brauer tables do not store the underlying group, they access it via the ordinary table (see 69.8.4).

2 ▶ `ConjugacyClasses(` *tbl* `)` A

For a character table *tbl* with known underlying group G, the `ConjugacyClasses` value of *tbl* is a list of conjugacy classes of G. All those lists stored in the table that are related to the ordering of conjugacy classes (such as sizes of centralizers and conjugacy classes, orders of representatives, power maps, and all class functions) refer to the ordering of this list.

This ordering need **not** coincide with the ordering of conjugacy classes as stored in the underlying group of the table (see 69.19). One reason for this is that otherwise we would not be allowed to use a library table as the character table of a group for which the conjugacy classes are stored already. (Another, less important

reason is that we can use the same group as underlying group of character tables that differ only w.r.t. the ordering of classes.)

The class of the identity element must be the first class (see 69.5).

If *tbl* was constructed from G then the conjugacy classes have been stored at the same time when G was stored. If G and *tbl* were connected later than in the construction of *tbl*, the recommended way to do this is via `ConnectGroupAndCharacterTable` (see 69.6.4). So there is no method for `ConjugacyClasses` that computes the value for *tbl* if it is not yet stored.

Brauer tables do not store the (p-regular) conjugacy classes, they access them via the ordinary table (see 69.8.4) if necessary.

3 ▶ `IdentificationOfConjugacyClasses( tbl )` A

For an ordinary character table *tbl* with known underlying group G, `IdentificationOfConjugacyClasses` returns a list of positive integers that contains at position i the position of the i-th conjugacy class of *tbl* in the list `ConjugacyClasses`(G).

```
gap> g:= SymmetricGroup( 4 );;
gap> repres:= [ (1,2), (1,2,3), (1,2,3,4), (1,2)(3,4), () ];;
gap> ccl:= List( repres, x -> ConjugacyClass( g, x ) );;
gap> SetConjugacyClasses( g, ccl );
gap> tbl:= CharacterTable( g );;   # the table stores already the values
gap> HasConjugacyClasses( tbl );  HasUnderlyingGroup( tbl );
true
true
gap> UnderlyingGroup( tbl ) = g;
true
gap> HasIdentificationOfConjugacyClasses( tbl );
true
gap> IdentificationOfConjugacyClasses( tbl );
[ 5, 1, 2, 3, 4 ]
```

4 ▶ `ConnectGroupAndCharacterTable( G, tbl[, arec] )` F
▶ `ConnectGroupAndCharacterTable( G, tbl, bijection )` F

Let G be a group and *tbl* a character table of (a group isomorphic to) G, such that G does not store its `OrdinaryCharacterTable` value and *tbl* does not store its `UnderlyingGroup` value. `ConnectGroupAndCharacterTable` calls `CompatibleConjugacyClasses`, trying to identify the classes of G with the columns of *tbl*.

If this identification is unique up to automorphisms of *tbl* (see 69.8.8) then *tbl* is stored as `CharacterTable` value of G, in *tbl* the values of `UnderlyingGroup`, `ConjugacyClasses`, and `IdentificationOfConjugacyClasses` are set, and `true` is returned.

Otherwise, i.e., if GAP cannot identify the classes of G up to automorphisms of G, `false` is returned.

If a record *arec* is present as third argument, its meaning is the same as for `CompatibleConjugacyClasses` (see 69.6.5).

If a list *bijection* is entered as third argument, it is used as value of `IdentificationOfConjugacyClasses`, relative to `ConjugacyClasses( G )`, without further checking, and `true` is returned.

5 ▶ `CompatibleConjugacyClasses( G, ccl, tbl[, arec] )` O
▶ `CompatibleConjugacyClasses( tbl[, arec] )` O

In the first form, *ccl* must be a list of the conjugacy classes of the group G, and *tbl* the ordinary character table of G. Then `CompatibleConjugacyClasses` returns a list l of positive integers that describes an identification of the columns of *tbl* with the conjugacy classes *ccl* in the sense that $l[i]$ is the position in *ccl* of the class

corresponding to the i-th column of *tbl*, if this identification is unique up to automorphisms of *tbl* (see 69.8.8); if GAP cannot identify the classes, `fail` is returned.

In the second form, *tbl* must be an ordinary character table, and `CompatibleConjugacyClasses` checks whether the columns of *tbl* can be identified with the conjugacy classes of a group isomorphic to that for which *tbl* is the character table; the return value is a list of all those sets of class positions for which the columns of *tbl* cannot be distinguished with the invariants used, up to automorphisms of *tbl*. So the identification is unique if and only if the returned list is empty.

The usual approach is that one first calls `CompatibleConjugacyClasses` in the second form for checking quickly whether the first form will be successful, and only if this is the case the more time consuming calculations with both group and character table are done.

The following invariants are used.

1. element orders (see 69.8.5),
2. class lengths (see 69.8.7),
3. power maps (see 71.1.1, 71.1.1),
4. symmetries of the table (see 69.8.8).

If the optional argument *arec* is present then it must be a record whose components describe additional information for the class identification. The following components are supported.

`natchar`
: if G is a permutation group or matrix group then the value of this component is regarded as the list of values of the natural character (see 70.7.2) of G, w.r.t. the ordering of classes in *tbl*,

`bijection`
: a list describing a partial bijection; the i-th entry, if bound, is the position of the i-th conjugacy class of *tbl* in the list *ccl*.

```
gap> g:= AlternatingGroup( 5 );
Alt( [ 1 .. 5 ] )
gap> tbl:= CharacterTable( "A5" );
CharacterTable( "A5" )
gap> HasUnderlyingGroup( tbl );  HasOrdinaryCharacterTable( g );
false
false
gap> CompatibleConjugacyClasses( tbl );   # unique identification
[  ]
gap> ConnectGroupAndCharacterTable( g, tbl );
true
gap> HasConjugacyClasses( tbl );  HasUnderlyingGroup( tbl );
true
true
gap> IdentificationOfConjugacyClasses( tbl );
[ 1, 2, 3, 4, 5 ]
gap> # Here is an example where the identification is not unique.
gap> CompatibleConjugacyClasses( CharacterTable( "J2" ) );
[ [ 17, 18 ], [ 9, 10 ] ]
```

69.7 Operators for Character Tables

The following infix operators are defined for character tables.

tbl1 `*` *tbl2*
: the direct product of two character tables (see 69.18.1),

tbl `/` *list*
: the table of the factor group modulo the normal subgroup spanned by the classes in the list *list* (see 69.18.3),

tbl `mod` *p*
: the p-modular Brauer character table corresponding to the ordinary character table *tbl* (see 69.3.1),

tbl`.`*name*
: the position of the class with name *name* in *tbl* (see 69.8.10).

69.8 Attributes and Properties of Character Tables

Several **attributes for groups** are valid also for character tables. These are on one hand those that have the same meaning for both group and character table, and whose values can be read off or computed, respectively, from the character table, such as `Size`, `IsAbelian`, or `IsSolvable`. On the other hand, there are attributes whose meaning for character tables is different from the meaning for groups, such as `ConjugacyClasses`.

1 ▶ `CharacterDegrees(` *G* `)` A
▶ `CharacterDegrees(` *G*`,` *p* `)` O
▶ `CharacterDegrees(` *tbl* `)` A

In the first two forms, `CharacterDegrees` returns a collected list of the degrees of the absolutely irreducible characters of the group G; the optional second argument p must be either zero or a prime integer denoting the characteristic, the default value is zero. In the third form, *tbl* must be an (ordinary or Brauer) character table, and `CharacterDegrees` returns a collected list of the degrees of the absolutely irreducible characters of *tbl*.

(The default method for the call with only argument a group is to call the operation with second argument `0`.)

For solvable groups, the default method is based on [Con90b].

```
gap> CharacterDegrees( SymmetricGroup( 4 ) );
[ [ 1, 2 ], [ 2, 1 ], [ 3, 2 ] ]
gap> CharacterDegrees( SymmetricGroup( 4 ), 2 );
[ [ 1, 1 ], [ 2, 1 ] ]
gap> CharacterDegrees( CharacterTable( "A5" ) );
[ [ 1, 1 ], [ 3, 2 ], [ 4, 1 ], [ 5, 1 ] ]
gap> CharacterDegrees( CharacterTable( "A5" ) mod 2 );
[ [ 1, 1 ], [ 2, 2 ], [ 4, 1 ] ]
```

2 ▶ `Irr(` *G* `)` A
▶ `Irr(` *G*`,` *p* `)` O
▶ `Irr(` *tbl* `)` A

Called with a group G, `Irr` returns the irreducible characters of the ordinary character table of G. Called with a group G and a prime integer p, `Irr` returns the irreducible characters of the p-modular Brauer table of G. Called with an (ordinary or Brauer) character table *tbl*, `Irr` returns the list of all complex absolutely irreducible characters of *tbl*.

For a character table *tbl* with underlying group, `Irr` may delegate to the group. For a group G, `Irr` may delegate to its character table only if the irreducibles are already stored there.

If a group has both ordinary irreducible characters in `Irr` and irreducible representations in `IrreducibleRepresentations` these are **not** guaranteed to be ordered in the same way.

(If G is p-solvable (see 37.15.18) then the p-modular irreducible characters can be computed by the Fong-Swan Theorem; in all other cases, there may be no method.)

Note that the ordering of columns in the `Irr` matrix of the group G refers to the ordering of conjugacy classes in `CharacterTable( G )`, which may differ from the ordering of conjugacy classes in G (see 69.6). As an extreme example, for a character table obtained from sorting the classes of `CharacterTable( G )`, the ordering of columns in the `Irr` matrix respects the sorting of classes (see 69.19), so the irreducibles of such a table will in general not coincide with the irreducibles stored as `Irr( G )` although also the sorted table stores the group G.

In the following example we temporarily increase the line length limit from its default value 80 to 85 in order to get a nicer output format.

```
gap> Irr( SymmetricGroup( 4 ) );
[ Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, -1, 1, 1, -1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 3, -1, -1, 0, 1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 2, 0, 2, -1, 0 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 3, 1, -1, 0, -1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1, 1 ] ) ]
gap> Irr( SymmetricGroup( 4 ), 2 );
[ Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 1, 1 ] ),
  Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 2, -1 ] ) ]
gap> SizeScreen([ 85, ]);;
gap> Irr( CharacterTable( "A5" ) );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] ),
  Character( CharacterTable( "A5" ), [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ]
gap> SizeScreen([ 80, ]);;
gap> Irr( CharacterTable( "A5" ) mod 2 );
[ Character( BrauerTable( "A5", 2 ), [ 1, 1, 1, 1 ] ),
  Character( BrauerTable( "A5", 2 ), [ 2, -1, E(5)+E(5)^4, E(5)^2+E(5)^3 ] ),
  Character( BrauerTable( "A5", 2 ), [ 2, -1, E(5)^2+E(5)^3, E(5)+E(5)^4 ] ),
  Character( BrauerTable( "A5", 2 ), [ 4, 1, -1, -1 ] ) ]
```

3 ▶ `LinearCharacters( G )` A
▶ `LinearCharacters( G, p )` O
▶ `LinearCharacters( tbl )` A

`LinearCharacters` returns the linear (i.e., degree 1) characters in the `Irr` (see 69.8.2) list of the group G or the character table *tbl*, respectively. In the second form, `LinearCharacters` returns the p-modular linear characters of the group G.

For a character table *tbl* with underlying group, `LinearCharacters` may delegate to the group. For a group G, `LinearCharacters` may delegate to its character table only if the irreducibles are already stored there.

The ordering of linear characters in *tbl* need not coincide with the ordering of linear characters in the irreducibles of *tbl* (see 69.8.2).

```
gap> LinearCharacters( SymmetricGroup( 4 ) );
[ Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, -1, 1, 1, -1 ] ) ]
```

4 ▶ `OrdinaryCharacterTable( G )` A

▶ `OrdinaryCharacterTable( modtbl )` A

`OrdinaryCharacterTable` returns the ordinary character table of the group G or the Brauer character table *modtbl*, respectively.

Since Brauer character tables are constructed from ordinary tables, the attribute value for *modtbl* is already stored (cf. 69.4).

```
gap> OrdinaryCharacterTable( SymmetricGroup( 4 ) );
CharacterTable( Sym( [ 1 .. 4 ] ) )
gap> tbl:= CharacterTable( "A5" );;  modtbl:= tbl mod 2;
BrauerTable( "A5", 2 )
gap> OrdinaryCharacterTable( modtbl ) = tbl;
true
```

The following operations for groups are applicable to character tables and mean the same for a character table as for the group; see the chapter about groups for the definition.

```
AbelianInvariants
       CommutatorLength
       Exponent
       IsAbelian
       IsCyclic
       IsFinite
       IsMonomial
       IsNilpotent
       IsPerfect
       IsSimple
       IsSolvable
       IsSporadicSimple
       IsSupersolvable
       NrConjugacyClasses
       Size
```

These operations are mainly useful for selecting character tables with certain properties, also for character tables without access to a group.

```
gap> tables:= [ CharacterTable( CyclicGroup( 3 ) ),
>               CharacterTable( SymmetricGroup( 4 ) ),
>               CharacterTable( AlternatingGroup( 5 ) ) ];;
gap> List( tables, AbelianInvariants );
[ [ 3 ], [ 2 ], [  ] ]
gap> List( tables, CommutatorLength );
[ 1, 1, 1 ]
gap> List( tables, Exponent );
[ 3, 12, 30 ]
gap> List( tables, IsAbelian );
[ true, false, false ]
gap> List( tables, IsCyclic );
[ true, false, false ]
```

```
gap> List( tables, IsFinite );
[ true, true, true ]
gap> List( tables, IsMonomial );
[ true, true, false ]
gap> List( tables, IsNilpotent );
[ true, false, false ]
gap> List( tables, IsPerfect );
[ false, false, true ]
gap> List( tables, IsSimple );
[ true, false, true ]
gap> List( tables, IsSolvable );
[ true, true, false ]
gap> List( tables, IsSupersolvable );
[ true, false, false ]
gap> List( tables, NrConjugacyClasses );
[ 3, 5, 5 ]
gap> List( tables, Size );
[ 3, 24, 60 ]
```

The following three **attributes for character tables** would make sense also for groups but are in fact **not** used for groups. This is because the values depend on the ordering of conjugacy classes stored as value of `ConjugacyClasses`, and this value may differ for a group and its character table (see 69.6). Note that for character tables, the consistency of attribute values must be guaranteed, whereas for groups, there is no need to impose such a consistency rule.

5 ▶ `OrdersClassRepresentatives( tbl )` A

is a list of orders of representatives of conjugacy classes of the character table *tbl*, in the same ordering as the conjugacy classes of *tbl*.

6 ▶ `SizesCentralizers( tbl )` A

is a list that stores at position i the size of the centralizer of any element in the i-th conjugacy class of the character table *tbl*.

7 ▶ `SizesConjugacyClasses( tbl )` A

is a list that stores at position i the size of the i-th conjugacy class of the character table *tbl*.

```
gap> tbl:= CharacterTable( "A5" );;
gap> OrdersClassRepresentatives( tbl );
[ 1, 2, 3, 5, 5 ]
gap> SizesCentralizers( tbl );
[ 60, 4, 3, 5, 5 ]
gap> SizesConjugacyClasses( tbl );
[ 1, 15, 20, 12, 12 ]
```

The following attributes apply only to character tables, not to groups.

8 ▶ `AutomorphismsOfTable( tbl )` A

is the permutation group of all column permutations of the character table *tbl* that leave the set of irreducibles and each power map of *tbl* invariant (see also 69.20.2).

```
gap> tbl:= CharacterTable( "Dihedral", 8 );;
gap> AutomorphismsOfTable( tbl );
Group([ (4,5) ])
gap> OrdersClassRepresentatives( tbl );
[ 1, 4, 2, 2, 2 ]
gap> SizesConjugacyClasses( tbl );
[ 1, 2, 1, 2, 2 ]
```

9 ▶ `UnderlyingCharacteristic( tbl )` A
▶ `UnderlyingCharacteristic( psi )` A

For an ordinary character table *tbl*, the result is `0`, for a p-modular Brauer table *tbl*, it is p. The underlying characteristic of a class function *psi* is equal to that of its underlying character table.

The underlying characteristic must be stored when the table is constructed, there is no method to compute it.

We cannot use the attribute `Characteristic` (see 30.10.1) to denote this, since of course each Brauer character is an element of characteristic zero in the sense of GAP (see Chapter 70).

```
gap> tbl:= CharacterTable( "A5" );;
gap> UnderlyingCharacteristic( tbl );
0
gap> UnderlyingCharacteristic( tbl mod 17 );
17
```

10 ▶ `ClassNames( tbl )` A
▶ `ClassNames( tbl, "ATLAS" )` O
▶ `CharacterNames( tbl )` A

`ClassNames` and `CharacterNames` return lists of strings, one for each conjugacy class or irreducible character, respectively, of the character table *tbl*. These names are used when *tbl* is displayed.

The default method for `ClassNames` computes class names consisting of the order of an element in the class and at least one distinguishing letter.

The default method for `CharacterNames` returns the list `[ "X.1", "X.2", ... ]`, whose length is the number of irreducible characters of *tbl*.

The position of the class with name *name* in *tbl* can be accessed as *tbl*`.`*name*.

When `ClassNames` is called with two arguments, the second being the string `"ATLAS"`, the class names returned obey the convention used in Chapter 7, Section 5 of the ATLAS of Finite Groups [CCN+85].

```
gap> tbl:= CharacterTable( "A5" );;
gap> ClassNames( tbl );
[ "1a", "2a", "3a", "5a", "5b" ]
gap> tbl.2a;
2
```

11 ▶ `Identifier( tbl )` A

is a string that identifies the character table *tbl* in the current GAP session. It is used mainly for class fusions into *tbl* that are stored on other character tables. For character tables without group, the identifier is also used to print the table; this is the case for library tables, but also for tables that are constructed as direct products, factors etc. involving tables that may or may not store their groups.

The default method for ordinary tables constructs strings of the form `"CT`n`"`, where n is a positive integer. `LARGEST_IDENTIFIER_NUMBER` is a list containing the largest integer n used in the current GAP session.

The default method for Brauer tables returns the concatenation of the identifier of the ordinary table, the string `"mod"`, and the (string of the) underlying characteristic.

```
gap> Identifier( CharacterTable( "A5" ) );
"A5"
gap> tbl:= CharacterTable( Group( () ) );;
gap> Identifier( tbl );  Identifier( tbl mod 2 );
"CT8"
"CT8mod2"
```

12 ▶ `InfoText(` *tbl* `)` A

is a mutable string with information about the character table *tbl*. There is no default method to create an info text.

This attribute is used mainly for library tables (see the manual of the GAP Character Table Library). Usual parts of the information are the origin of the table, tests it has passed (`1.o.r.` for the test of orthogonality, `pow[`*p*`]` for the construction of the *p*-th power map, `DEC` for the decomposition of ordinary into Brauer characters, `TENS` for the decomposition of tensor products of irreducibles), and choices made without loss of generality.

```
gap> Print( InfoText( CharacterTable( "A5" ) ), "\n" );
origin: ATLAS of finite groups, tests: 1.o.r., pow[2,3,5]
```

13 ▶ `InverseClasses(` *tbl* `)` A

For a character table *tbl*, `InverseClasses` returns the list mapping each conjugacy class to its inverse class. This list can be regarded as (-1)-st power map of *tbl* (see 71.1.1).

14 ▶ `RealClasses(` *tbl* `)` A

For a character table *tbl*, `RealClasses` returns the strictly sorted list of positions of classes in *tbl* that consist of real elements.

An element x is **real** iff it is conjugate to its inverse $x^{-1} = x^{o(x)-1}$.

```
gap> InverseClasses( CharacterTable( "A5" ) );
[ 1, 2, 3, 4, 5 ]
gap> InverseClasses( CharacterTable( "Cyclic", 3 ) );
[ 1, 3, 2 ]
gap> RealClasses( CharacterTable( "A5" ) );
[ 1, 2, 3, 4, 5 ]
gap> RealClasses( CharacterTable( "Cyclic", 3 ) );
[ 1 ]
```

15 ▶ `ClassOrbit(` *tbl*, *cc* `)` O

is the list of positions of those conjugacy classes of the character table *tbl* that are Galois conjugate to the *cc*-th class. That is, exactly the classes at positions given by the list returned by `ClassOrbit` contain generators of the cyclic group generated by an element in the *cc*-th class.

This information is computed from the power maps of *tbl*.

16 ▶ `ClassRoots(` *tbl* `)` A

For a character table *tbl*, `ClassRoots` returns a list containing at position i the list of positions of the classes of all nontrivial p-th roots, where p runs over the prime divisors of `Size(` *tbl* `)`.

This information is computed from the power maps of *tbl*.

```
gap> ClassOrbit( CharacterTable( "A5" ), 4 );
[ 4, 5 ]
gap> ClassRoots( CharacterTable( "A5" ) );
[ [ 2, 3, 4, 5 ], [  ], [  ], [  ], [  ] ]
gap> ClassRoots( CharacterTable( "Cyclic", 6 ) );
[ [ 3, 4, 5 ], [  ], [ 2 ], [ 2, 6 ], [ 6 ], [  ] ]
```

The following attributes for a character table *tbl* correspond to attributes for the group G of *tbl*. But instead of a normal subgroup (or a list of normal subgroups) of G, they return a strictly sorted list of positive integers (or a list of such lists) which are the positions –relative to `ConjugacyClasses( tbl )`– of those classes forming the normal subgroup in question.

17 ▶ `ClassPositionsOfNormalSubgroups( ordtbl )` A
▶ `ClassPositionsOfMaximalNormalSubgroups( ordtbl )` A

correspond to `NormalSubgroups` and `MaximalNormalSubgroups` for the group of the ordinary character table *ordtbl* (see 37.19.7, 37.19.8).

The entries of the result lists are sorted according to increasing length. (So this total order respects the partial order of normal subgroups given by inclusion.)

18 ▶ `ClassPositionsOfAgemo( ordtbl, p )` O

corresponds to `Agemo` (see 37.14.2) for the group of the ordinary character table *ordtbl*.

19 ▶ `ClassPositionsOfCentre( ordtbl )` A

corresponds to `Centre` (see 33.4.5) for the group of the ordinary character table *ordtbl*.

20 ▶ `ClassPositionsOfDirectProductDecompositions( tbl )` A
▶ `ClassPositionsOfDirectProductDecompositions( tbl, nclasses )` O

Let *tbl* be the ordinary character table of the group G, say. Called with the only argument *tbl*, `ClassPositionsOfDirectProductDecompositions` returns the list of all those pairs $[l_1, l_2]$ where l_1 and l_2 are lists of class positions of normal subgroups N_1, N_2 of G such that G is their direct product and $|N_1| \leq |N_2|$ holds. Called with second argument a list *nclasses* of class positions of a normal subgroup N of G, `ClassPositionsOfDirectProductDecompositions` returns the list of pairs describing the decomposition of N as a direct product of two normal subgroups of G.

21 ▶ `ClassPositionsOfDerivedSubgroup( ordtbl )` A

corresponds to `DerivedSubgroup` (see 37.12.3) for the group of the ordinary character table *ordtbl*.

22 ▶ `ClassPositionsOfElementaryAbelianSeries( ordtbl )` A

corresponds to `ElementaryAbelianSeries` (see 37.17.9) for the group of the ordinary character table *ordtbl*.

23 ▶ `ClassPositionsOfFittingSubgroup( ordtbl )` A

corresponds to `FittingSubgroup` (see 37.12.5) for the group of the ordinary character table *ordtbl*.

24 ▶ `ClassPositionsOfLowerCentralSeries( tbl )` A

corresponds to `LowerCentralSeries` (see 37.17.11) for the group of the ordinary character table *ordtbl*.

25 ▶ `ClassPositionsOfUpperCentralSeries( ordtbl )` A

corresponds to `UpperCentralSeries` (see 37.17.12) for the group of the ordinary character table *ordtbl*.

26 ▶ `ClassPositionsOfSupersolvableResiduum( ordtbl )` A

corresponds to `SupersolvableResiduum` (see 37.12.11) for the group of the ordinary character table *ordtbl*.

27 ▶ `ClassPositionsOfNormalClosure( ordtbl, classes )` O

is the sorted list of the positions of all conjugacy classes of the ordinary character table *ordtbl* that form the normal closure (see 37.11.4) of the conjugacy classes at positions in the list *classes*.

```
gap> tbla5:= CharacterTable( "A5" );;
gap> tbls4:= CharacterTable( "Symmetric", 4 );;
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> ClassPositionsOfNormalSubgroups( tbls4 );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1 .. 5 ] ]
gap> ClassPositionsOfAgemo( tbls4, 2 );
[ 1, 3, 4 ]
gap> ClassPositionsOfCentre( tbld8 );
[ 1, 3 ]
gap> ClassPositionsOfDerivedSubgroup( tbld8 );
[ 1, 3 ]
gap> ClassPositionsOfElementaryAbelianSeries( tbls4 );
[ [ 1 .. 5 ], [ 1, 3, 4 ], [ 1, 3 ], [ 1 ] ]
gap> ClassPositionsOfElementaryAbelianSeries( tbla5 );
fail
gap> ClassPositionsOfFittingSubgroup( tbls4 );
[ 1, 3 ]
gap> ClassPositionsOfLowerCentralSeries( tbls4 );
[ [ 1 .. 5 ], [ 1, 3, 4 ] ]
gap> ClassPositionsOfLowerCentralSeries( tbld8 );
[ [ 1 .. 5 ], [ 1, 3 ], [ 1 ] ]
gap> ClassPositionsOfUpperCentralSeries( tbls4 );
[ [ 1 ] ]
gap> ClassPositionsOfUpperCentralSeries( tbld8 );
[ [ 1, 3 ], [ 1, 2, 3, 4, 5 ] ]
gap> ClassPositionsOfSupersolvableResiduum( tbls4 );
[ 1, 3 ]
gap> ClassPositionsOfNormalClosure( tbls4, [ 1, 4 ] );
[ 1, 3, 4 ]
```

69.9 Operations Concerning Blocks

1 ▶ `PrimeBlocks( ordtbl, p )` O
▶ `PrimeBlocksOp( ordtbl, p )` O
▶ `ComputedPrimeBlockss( tbl )` A

For an ordinary character table *ordtbl* and a prime integer p, `PrimeBlocks` returns a record with the following components.

`block`
: a list, the value j at position i means that the i-th irreducible character of *ordtbl* lies in the j-th p-block of *ordtbl*,

`defect`
: a list containing at position i the defect of the i-th block,

`height`
: a list containing at position i the height of the i-th irreducible character of *ordtbl* in its block,

`relevant`
: a list of class positions such that only the restriction to these classes need be checked for deciding whether two characters lie in the same block, and

`centralcharacter`
: a list containing at position i a list whose values at the positions stored in the component `relevant` are the values of a central character in the i-th block.

The components `relevant` and `centralcharacters` are used by `SameBlock` (see 69.9.2).

If `InfoCharacterTable` has level at least 2, the defects of the blocks and the heights of the characters are printed.

The default method uses the attribute `ComputedPrimeBlockss` for storing the computed value at position p, and calls the operation `PrimeBlocksOp` for computing values that are not yet known.

Two ordinary irreducible characters χ, ψ of a group G are said to lie in the same p-**block** if the images of their central characters ω_χ, ω_ψ (see 70.8.17) under the ring homomorphism $*: R \to R/M$ are equal, where R denotes the ring of algebraic integers in the complex number field, and M is a maximal ideal in R with $pR \subseteq M$. (The distribution to p-blocks is in fact independent of the choice of M, see [Isa76].)

For $|G| = p^a m$ where p does not divide m, the **defect** of a block is the integer d such that p^{a-d} is the largest power of p that divides the degrees of all characters in the block.

The **height** of a character χ in the block is defined as the largest exponent h for which p^h divides $\chi(1)/p^{a-d}$.

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> pbl:= PrimeBlocks( tbl, 2 );
rec( block := [ 1, 1, 1, 1, 1, 2 ], defect := [ 3, 0 ],
  height := [ 0, 0, 0, 1, 0, 0 ], relevant := [ 3, 5 ],
  centralcharacter := [ [ ,, 56,, 24 ], [ ,, -7,, 3 ] ] )
```

2 ▶ `SameBlock( `*p*`, `*omega1*`, `*omega2*`, `*relevant*` )` F

Let p be a prime integer, *omega1* and *omega2* be two central characters (or their values lists) of a character table, and *relevant* be a list of positions as is stored in the component `relevantclasses` of a record returned by `PrimeBlocks` (see 69.9.1).

`SameBlock` returns `true` if *omega1* and *omega2* are equal modulo any maximal ideal in the ring of complex algebraic integers containing the ideal spanned by p, and `false` otherwise.

```
gap> omega:= List( Irr( tbl ), CentralCharacter );;
gap> SameBlock( 2, omega[1], omega[2], pbl.relevant );
true
gap> SameBlock( 2, omega[1], omega[6], pbl.relevant );
false
```

3 ▶ `BlocksInfo( `*modtbl*` )` A

For a Brauer character table *modtbl*, the value of `BlocksInfo` is a list of (mutable) records, the i-th entry containing information about the i-th block. Each record has the following components.

`defect`
: the defect of the block,

`ordchars`
: the list of positions of the ordinary characters that belong to the block, relative to `Irr( OrdinaryCharacterTable( `*modtbl*` ) )`,

`modchars`
: the list of positions of the Brauer characters that belong to the block, relative to `IBr( `*modtbl*` )`.

Optional components are

`basicset`
: a list of positions of ordinary characters in the block whose restriction to *modtbl* is maximally linearly independent, relative to `Irr( OrdinaryCharacterTable( `*modtbl*` ) )`,

`decmat`
: the decomposition matrix of the block, it is stored automatically when `DecompositionMatrix` is called for the block (see 69.9.4),

`decinv`
: inverse of the decomposition matrix of the block, restricted to the ordinary characters described by `basicset`,

`brauertree`
: a list that describes the Brauer tree of the block, in the case that the block is of defect 1.

```
gap> BlocksInfo( CharacterTable( "L3(2)" ) mod 2 );
[ rec( defect := 3, ordchars := [ 1, 2, 3, 4, 5 ], modchars := [ 1, 2, 3 ],
      decinv := [ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ] ],
      basicset := [ 1, 2, 3 ] ),
  rec( defect := 0, ordchars := [ 6 ], modchars := [ 4 ], decinv := [ [ 1 ] ],
      basicset := [ 6 ] ) ]
```

4 ▶ `DecompositionMatrix( `*modtbl*` )` A
▶ `DecompositionMatrix( `*modtbl*`, `*blocknr*` )` O

Let *modtbl* be a Brauer character table.

In the first version `DecompositionMatrix` returns the decomposition matrix of *modtbl*, where the rows and columns are indexed by the irreducible characters of the ordinary character table of *modtbl* and the irreducible characters of *modtbl*, respectively,

In the second version `DecompositionMatrix` returns the decomposition matrix of the block of *modtbl* with number *blocknr*; the matrix is stored as value of the `decmat` component of the *blocknr*-th entry of the `BlocksInfo` list (see 69.9.3) of *modtbl*.

An ordinary irreducible character is in block i if and only if all characters before the first character of the same block lie in $i-1$ different blocks. An irreducible Brauer character is in block i if it has nonzero scalar product with an ordinary irreducible character in block i.

`DecompositionMatrix` is based on the more general function `Decomposition` (see 25.4.1).

```
gap> modtbl:= CharacterTable( "L3(2)" ) mod 2;
BrauerTable( "L3(2)", 2 )
gap> DecompositionMatrix( modtbl );
[ [ 1, 0, 0, 0 ], [ 0, 1, 0, 0 ], [ 0, 0, 1, 0 ], [ 0, 1, 1, 0 ],
  [ 1, 1, 1, 0 ], [ 0, 0, 0, 1 ] ]
gap> DecompositionMatrix( modtbl, 1 );
[ [ 1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, 1 ], [ 0, 1, 1 ], [ 1, 1, 1 ] ]
gap> DecompositionMatrix( modtbl, 2 );
[ [ 1 ] ]
```

5 ▶ `LaTeXStringDecompositionMatrix( `*modtbl*`[, `*blocknr*`][, `*options*`] )` F

is a string that contains LaTeX code to print a decomposition matrix (see 69.9.4) nicely.

The optional argument *options*, if present, must be a record with components `phi`, `chi` (strings used in each label for columns and rows), `collabels`, `rowlabels` (subscripts for the labels). The defaults for `phi` and

`chi` are `"\tt Y"` and `"\tt X"`, the defaults for `collabels` and `rowlabels` are the lists of positions of the Brauer characters and ordinary characters in the respective lists of irreducibles in the character tables.

The optional components `nrows` and `ncols` denote the maximal number of rows and columns per array; if they are present then each portion of `nrows` rows and `ncols` columns forms an array of its own which is enclosed in `\[`, `\]`.

If the component `decmat` is bound in *options* then it must be the decomposition matrix in question, in this case the matrix is not computed from the information in *modtbl*.

For those character tables from the GAP table library that belong to the ATLAS of Finite Groups [CCN+85], `AtlasLabelsOfIrreducibles` constructs character labels that are compatible with those used in the ATLAS (see "ctbllib:atlas tables" and "ctbllib:atlaslabelsofirreducibles" in the manual of the GAP Character Table Library).

```
gap> modtbl:= CharacterTable( "L3(2)" ) mod 2;;
gap> Print( LaTeXStringDecompositionMatrix( modtbl, 1 ) );
\[
\begin{array}{r|rrr} \hline
 & {\tt Y}_{1}
 & {\tt Y}_{2}
 & {\tt Y}_{3}
 \rule[-7pt]{0pt}{20pt} \\ \hline
{\tt X}_{1} & 1 & . & . \rule[0pt]{0pt}{13pt} \\
{\tt X}_{2} & . & 1 & . \\
{\tt X}_{3} & . & . & 1 \\
{\tt X}_{4} & . & 1 & 1 \\
{\tt X}_{5} & 1 & 1 & 1 \rule[-7pt]{0pt}{5pt} \\
\hline
\end{array}
\]
gap> options:= rec( phi:= "\\varphi", chi:= "\\chi" );;
gap> Print( LaTeXStringDecompositionMatrix( modtbl, 1, options ) );
\[
\begin{array}{r|rrr} \hline
 & \varphi_{1}
 & \varphi_{2}
 & \varphi_{3}
 \rule[-7pt]{0pt}{20pt} \\ \hline
\chi_{1} & 1 & . & . \rule[0pt]{0pt}{13pt} \\
\chi_{2} & . & 1 & . \\
\chi_{3} & . & . & 1 \\
\chi_{4} & . & 1 & 1 \\
\chi_{5} & 1 & 1 & 1 \rule[-7pt]{0pt}{5pt} \\
\hline
\end{array}
\]
```

69.10 Other Operations for Character Tables

In the following, we list operations for character tables that are not attributes.

1 ▶ `IsInternallyConsistent( tbl )` O

For an **ordinary** character table *tbl*, `IsInternallyConsistent` checks the consistency of the following attribute values (if stored).

– `Size`, `SizesCentralizers`, and `SizesConjugacyClasses`.

– `SizesCentralizers` and `OrdersClassRepresentatives`.

– `ComputedPowerMaps` and `OrdersClassRepresentatives`.

– `SizesCentralizers` and `Irr`.

– `Irr` (first orthogonality relation).

For a **Brauer** table *tbl*, `IsInternallyConsistent` checks the consistency of the following attribute values (if stored).

– `Size`, `SizesCentralizers`, and `SizesConjugacyClasses`.

– `SizesCentralizers` and `OrdersClassRepresentatives`.

– `ComputedPowerMaps` and `OrdersClassRepresentatives`.

– `Irr` (closure under complex conjugation and Frobenius map).

If no inconsistency occurs, `true` is returned, otherwise each inconsistency is printed to the screen if the level of `InfoWarning` is at least 1 (see 7.4), and `false` is returned at the end.

2 ► `IsPSolvableCharacterTable(` *tbl*`,` *p* `)` O
► `IsPSolvableCharacterTableOp(` *tbl*`,` *p* `)` O
► `ComputedIsPSolvableCharacterTables(` *tbl* `)` A

`IsPSolvableCharacterTable` for the ordinary character table *tbl* corresponds to `IsPSolvable` for the group of *tbl* (see 37.15.18). *p* must be either a prime integer or `0`.

The default method uses the attribute `ComputedIsPSolvableCharacterTables` for storing the computed value at position *p*, and calls the operation `IsPSolvableCharacterTableOp` for computing values that are not yet known.

```
gap> tbl:= CharacterTable( "Sz(8)" );;
gap> IsPSolvableCharacterTable( tbl, 2 );
false
gap> IsPSolvableCharacterTable( tbl, 3 );
true
```

3 ► `IsClassFusionOfNormalSubgroup(` *subtbl*`,` *fus*`,` *tbl* `)` F

For two ordinary character tables *tbl* and *subtbl* of a group G and its subgroup U, say, and a list *fus* of positive integers that describes the class fusion of U into G, `IsClassFusionOfNormalSubgroup` returns `true` if U is a normal subgroup of G, and `false` otherwise.

```
gap> tblc2:= CharacterTable( "Cyclic", 2 );;
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> fus:= PossibleClassFusions( tblc2, tbld8 );
[ [ 1, 3 ], [ 1, 4 ], [ 1, 5 ] ]
gap> List( fus, map -> IsClassFusionOfNormalSubgroup( tblc2, map, tbld8 ) );
[ true, false, false ]
```

4 ► `Indicator(` *tbl*`,` *n* `)` O
► `Indicator(` *tbl*`[,` *characters*`],` *n* `)` O
► `Indicator(` *modtbl*`, 2 )` O
► `IndicatorOp(` *tbl*`,` *characters*`,` *n* `)` O
► `ComputedIndicators(` *tbl* `)` A

If *tbl* is an ordinary character table then `Indicator` returns the list of n-th Frobenius-Schur indicators of the characters in the list *characters*; the default of *characters* is `Irr(` *tbl* `)`.

The n-th Frobenius-Schur indicator $\nu_n(\chi)$ of an ordinary character χ of the group G is given by $\nu_n(\chi) = \frac{1}{|G|}\sum_{g\in G}\chi(g^n)$.

If *tbl* is a Brauer table in characteristic $\neq 2$ and $n = 2$ then `Indicator` returns the second indicator.

The default method uses the attribute `ComputedIndicators` for storing the computed value at position n, and calls the operation `IndicatorOp` for computing values that are not yet known.

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> Indicator( tbl, 2 );
[ 1, 0, 0, 1, 1, 1 ]
```

5 ▶ `NrPolyhedralSubgroups( tbl, c1, c2, c3 )` F

returns the number and isomorphism type of polyhedral subgroups of the group with ordinary character table *tbl* which are generated by an element g of class *c1* and an element h of class *c2* with the property that the product gh lies in class *c3*.

According to p. 233 in [NPP84], the number of polyhedral subgroups of isomorphism type V_4, D_{2n}, A_4, S_4, and A_5 can be derived from the class multiplication coefficient (see 69.10.6) and the number of Galois conjugates of a class (see 69.8.15).

The classes *c1*, *c2* and *c3* in the parameter list must be ordered according to the order of the elements in these classes.

```
gap> NrPolyhedralSubgroups( tbl, 2, 2, 4 );
rec( number := 21, type := "D8" )
```

6 ▶ `ClassMultiplicationCoefficient( tbl, i, j, k )` O

returns the class multiplication coefficient of the classes i, j, and k of the group G with ordinary character table *tbl*.

The class multiplication coefficient $c_{i,j,k}$ of the classes i, j, k equals the number of pairs (x, y) of elements $x, y \in G$ such that x lies in class i, y lies in class j, and their product xy is a fixed element of class k.

In the center of the group algebra of G, these numbers are found as coefficients of the decomposition of the product of two class sums K_i and K_j into class sums,

$$K_iK_j = \sum_k c_{ijk}K_k.$$

Given the character table of a finite group G, whose classes are $C_1, \ldots, C_r$ with representatives $g_i \in C_i$, the class multiplication coefficient c_{ijk} can be computed by the following formula.

$$c_{ijk} = \frac{|C_i||C_j|}{|G|}\sum_{\chi\in Irr(G)}\frac{\chi(g_i)\chi(g_j)\overline{\chi(g_k)}}{\chi(1)}.$$

On the other hand the knowledge of the class multiplication coefficients admits the computation of the irreducible characters of G. (see 69.12.1).

7 ▶ `ClassStructureCharTable( tbl, classes )` F

returns the so-called class structure of the classes in the list *classes*, for the character table *tbl* of the group G. The length of *classes* must be at least 2.

Let $C = (C_1, C_2, \ldots, C_n)$ denote the n-tuple of conjugacy classes of G that are indexed by *classes*. The class structure $n(C)$ equals the number of n-tuples $(g_1, g_2, \ldots, g_n)$ of elements $g_i \in C_i$ with $g_1 g_2 \cdots g_n = 1$. Note the difference to the definition of the class multiplication coefficients in `ClassMultiplicationCoefficient` (see 69.10.6).

$n(C_1, C_2, \ldots, C_n)$ is computed using the formula

$$n(C_1, C_2, \ldots, C_n) = \frac{|C_1||C_2| \cdots |C_n|}{|G|} \sum_{\chi \in Irr(G)} \frac{\chi(g_1)\chi(g_2) \cdots \chi(g_n)}{\chi(1)^{n-2}}.$$

8 ▶ `MatClassMultCoeffsCharTable( `*tbl*`, `*i*` )` F

For an ordinary character table *tbl* and a class position i, `MatClassMultCoeffsCharTable` returns the matrix $[a_{ijk}]_{j,k}$ of structure constants (see 69.10.6).

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> ClassMultiplicationCoefficient( tbl, 2, 2, 4 );
4
gap> ClassStructureCharTable( tbl, [ 2, 2, 4 ] );
168
gap> ClassStructureCharTable( tbl, [ 2, 2, 2, 4 ] );
1848
gap> MatClassMultCoeffsCharTable( tbl, 2 );
[ [ 0, 1, 0, 0, 0, 0 ], [ 21, 4, 3, 4, 0, 0 ], [ 0, 8, 6, 8, 7, 7 ],
  [ 0, 8, 6, 1, 7, 7 ], [ 0, 0, 3, 4, 0, 7 ], [ 0, 0, 3, 4, 7, 0 ] ]
```

69.11 Printing Character Tables

The default `ViewObj` (see 6.2.3) method for ordinary character tables prints the string `"CharacterTable"`, followed by the identifier (see 69.8.11) or, if known, the group of the character table enclosed in brackets. `ViewObj` for Brauer tables does the same, except that the first string is replaced by `"BrauerTable"`, and that the characteristic is also shown.

The default `PrintObj` (see 6.2.3) method for character tables does the same as `ViewObj`, except that the group is is `Print`-ed instead of `View`-ed.

The default `Display` (see 6.2.4) method for a character table *tbl* prepares the data contained in *tbl* for a pretty columnwise output. The number of columns printed at one time depends on the actual line length, which can be accessed and changed by the function `SizeScreen` (see 6.11.1).

`Display` shows certain characters (by default all irreducible characters) of *tbl*, together with the orders of the centralizers in factorized form and the available power maps (see 71.1.1). Each displayed character is given a name `X.`n.

The first lines of the output describe the order of the centralizer of an element of the class factorized into its prime divisors.

The next line gives the name of each class. If no class names are stored on *tbl*, `ClassNames` is called (see 69.8.10).

Preceded by a name `P`n, the next lines show the nth power maps of *tbl* in terms of the former shown class names.

Every ambiguous or unknown (see Chapter 19) value of the table is displayed as a question mark `?`.

Irrational character values are not printed explicitly because the lengths of their printed representation might disturb the layout. Instead of that every irrational value is indicated by a name, which is a string of at least one capital letter.

Once a name for an irrational value is found, it is used all over the printed table. Moreover the complex conjugate (see 18.5.2, 18.5.1) and the star of an irrationality (see 18.5.3) are represented by that very name preceded by a `/` and a `*`, respectively.

The printed character table is then followed by a legend, a list identifying the occurring symbols with their actual values. Occasionally this identification is supplemented by a quadratic representation of the irrationality together with the corresponding ATLAS-notation (see [CCN+85]).

The optional second argument *arec* of `Display` can be used to change the default style (mentioned above) for displaying a character. *arec* must be a record, its relevant components are the following.

`chars`
: an integer or a list of integers to select a sublist of the irreducible characters of *tbl*, or a list of characters of *tbl* (in this case the letter `"X"` is replaced by `"Y"`),

`classes`
: an integer or a list of integers to select a sublist of the classes of *tbl*,

`centralizers`
: suppresses the printing of the orders of the centralizers if `false`,

`powermap`
: an integer or a list of integers to select a subset of the available power maps, or `false` to suppress the printing of power maps,

`letter`
: a single capital letter (e. g. `"P"` for permutation characters) to replace `"X"`,

`indicator`
: `true` enables the printing of the second Frobenius Schur indicator, a list of integers enables the printing of the corresponding indicators (see 69.10.4),

`StringEntry`
: a function that takes either a character value or a character value and the return value of `StringEntryData` (see below), and returns the string that is actually displayed; it is called for all character values to be displayed, and also for the displayed indicator values (see above); the default `StringEntry` function is `CharacterTableDisplayStringEntryDefault`,

`StringEntryData`
: a unary function that is called once with argument *tbl* before the character values are displayed; it returns an object that is used as second argument of the function `StringEntry`; the default `StringEntryData` function is `CharacterTableDisplayStringEntryDataDefault`,

`PrintLegend`
: a function that is called with the result of the `StringEntryData` call after the character table has been displayed; the default `PrintLegend` function is `CharacterTableDisplayPrintLegendDefault`.

If the value of `DisplayOptions` (see 69.11.1) is stored on *tbl*, it is used as default value for *arec* in the one argument call of `Display`.

1 ▶ `DisplayOptions(` *tbl* `)` A

There is no default method to compute a value, one can set a value with `SetDisplayOptions`.

```
gap> tbl:= CharacterTable( "A5" );;
gap> Display( tbl );
A5

     2  2  2  .  .  .
     3  1  .  1  .  .
     5  1  .  .  1  1
```

```
         1a 2a 3a 5a 5b
      2P 1a 1a 3a 5b 5a
      3P 1a 2a 1a 5b 5a
      5P 1a 2a 3a 1a 1a

X.1     1  1  1  1  1
X.2     3 -1  .  A *A
X.3     3 -1  . *A  A
X.4     4  .  1 -1 -1
X.5     5  1 -1  .  .

A = -E(5)-E(5)^4
  = (1-ER(5))/2 = -b5
gap> options:= rec( chars:= 4, classes:= [ tbl.3a .. tbl.5a ],
>                   centralizers:= false, indicator:= true,
>                   powermap:= [ 2 ] );;
gap> Indicator( tbl, 2 );;
gap> Display( tbl, options );
A5

          3a 5a
       2P 3a 5b
          2
X.4    +   1 -1
gap> SetDisplayOptions( tbl, options );  Display( tbl );
A5

          3a 5a
       2P 3a 5b
          2
X.4    +   1 -1
```

2 ▶ `PrintCharacterTable(` *tbl*, *varname* `)` F

Let *tbl* be a nearly character table, and *varname* a string. `PrintCharacterTable` prints those values of the supported attributes (see 69.3.3) that are known for *tbl*;

The output of `PrintCharacterTable` is GAP readable; actually reading it into GAP will bind the variable with name *varname* to a character table that coincides with *tbl* for all printed components.

This is used mainly for saving character tables to files. A more human readable form is produced by `Display`.

```
gap> PrintCharacterTable( CharacterTable( "Cyclic", 2 ), "tbl" );
tbl:= function()
local tbl;
tbl:=rec();
tbl.Irr:=
[ [ 1, 1 ], [ 1, -1 ] ];
tbl.NrConjugacyClasses:=
2;
tbl.Size:=
2;
tbl.OrdersClassRepresentatives:=
[ 1, 2 ];
```

```
tbl.SizesCentralizers:=
[ 2, 2 ];
tbl.UnderlyingCharacteristic:=
0;
tbl.ClassParameters:=
[ [ 1, 0 ], [ 1, 1 ] ];
tbl.CharacterParameters:=
[ [ 1, 0 ], [ 1, 1 ] ];
tbl.Identifier:=
"C2";
tbl.InfoText:=
"computed using generic character table for cyclic groups";
tbl.ComputedPowerMaps:=
[ , [ 1, 1 ] ];
ConvertToLibraryCharacterTableNC(tbl);
return tbl;
end;
tbl:= tbl();
```

69.12 Computing the Irreducible Characters of a Group

Several algorithms are available for computing the irreducible characters of a finite group G. The default method for arbitrary finite groups is to use the Dixon-Schneider algorithm (see 69.12.1). For supersolvable groups, Conlon's algorithm can be used (see 69.12.2). For abelian-by-supersolvable groups, the Baum-Clausen algorithm for computing the irreducible representations (see 69.12.4) can be used to compute the irreducible characters (see 69.12.3).

These functions are installed in methods for `Irr` (see 69.8.2), but explicitly calling one of them will **not** set the `Irr` value of G.

1 ▶ `IrrDixonSchneider( G )` A

computes the irreducible characters of the finite group G, using the Dixon-Schneider method (see 69.14). It calls `DixonInit` and `DixonSplit`, and finally returns the list returned by `DixontinI` (see 69.15, 69.16, 69.17).

2 ▶ `IrrConlon( G )` A

For a finite solvable group G, `IrrConlon` returns a list of certain irreducible characters of G, among those all irreducibles that have the supersolvable residuum of G in their kernels; so if G is supersolvable, all irreducible characters of G are returned. An error is signalled if G is not solvable.

The characters are computed using Conlon's algorithm (see [Con90a] and [Con90b]). For each irreducible character in the returned list, the monomiality information (see 72.3.1) is stored.

3 ▶ `IrrBaumClausen( G )` A

`IrrBaumClausen` returns the absolutely irreducible ordinary characters of the factor group of the finite solvable group G by the derived subgroup of its supersolvable residuum.

The characters are computed using the algorithm by Baum and Clausen (see [BC94]). An error is signalled if G is not solvable.

In the following example we temporarily increase the line length limit from its default value 80 to 87 in order to get a nicer output format.

```
gap> g:= SL(2,3);;
gap> SizeScreen([ 87, ]);;
gap> irr1:= IrrDixonSchneider( g );
[ Character( CharacterTable( SL(2,3) ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3)^2, E(3), E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3), E(3)^2, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 2, -2, 1, 1, -1, -1, 0 ] ),
  Character( CharacterTable( SL(2,3) ), [ 2, -2, E(3), E(3)^2, -E(3)^2, -E(3), 0 ] ),
  Character( CharacterTable( SL(2,3) ), [ 2, -2, E(3)^2, E(3), -E(3), -E(3)^2, 0 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] ) ]
gap> irr2:= IrrConlon( g );
[ Character( CharacterTable( SL(2,3) ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3), E(3)^2, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3)^2, E(3), E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] ) ]
gap> irr3:= IrrBaumClausen( g );
[ Character( CharacterTable( SL(2,3) ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3), E(3)^2, E(3)^2, E(3), 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 1, 1, E(3)^2, E(3), E(3), E(3)^2, 1 ] ),
  Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] ) ]
gap> SizeScreen([ 80, ]);;
gap> chi:= irr2[4];;  HasTestMonomial( chi );
true
```

4 ▶ `IrreducibleRepresentations( G )` A
▶ `IrreducibleRepresentations( G, F )` O

Called with a finite group G and a field F, `IrreducibleRepresentations` returns a list of representatives of the irreducible matrix representations of G over F, up to equivalence.

If G is the only argument then `IrreducibleRepresentations` returns a list of representatives of the absolutely irreducible complex representations of G, up to equivalence.

At the moment, methods are available for the following cases: If G is abelian by supersolvable the method of [BC94] is used.

Otherwise, if F and G are both finite, the regular module of G is split by MeatAxe methods which can make this an expensive operation.

Finally, if F is not given (i.e. it defaults to the cyclotomic numbers) and G is a finite group, the method of [Dix93] (see 69.12.5) is used.

For other cases no methods are implemented yet.

See also `IrreducibleModules`, which provides efficient methods for solvable groups.

```
gap> g:= AlternatingGroup( 4 );;
gap> repr:= IrreducibleRepresentations( g );
[ Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ 1 ] ], [ [ 1 ] ], [ [ 1 ] ] ],
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ E(3) ] ], [ [ 1 ] ], [ [ 1 ] ] ],
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ E(3)^2 ] ], [ [ 1 ] ], [ [ 1 ] ] ],
  Pcgs([ (2,4,3), (1,3)(2,4), (1,2)(3,4) ]) ->
    [ [ [ 0, 0, 1 ], [ 1, 0, 0 ], [ 0, 1, 0 ] ],
      [ [ -1, 0, 0 ], [ 0, 1, 0 ], [ 0, 0, -1 ] ],
```

```
      [ [ 1, 0, 0 ], [ 0, -1, 0 ], [ 0, 0, -1 ] ] ] ]
gap> ForAll( repr, IsGroupHomomorphism );
true
gap> Length( repr );
4
gap> gens:= GeneratorsOfGroup( g );
[ (1,2,3), (2,3,4) ]
gap> List( gens, x -> x^repr[1] );
[ [ [ 1 ] ], [ [ 1 ] ] ]
gap>  List( gens, x -> x^repr[4] );
[ [ [ 0, 0, -1 ], [ 1, 0, 0 ], [ 0, -1, 0 ] ],
  [ [ 0, 1, 0 ], [ 0, 0, 1 ], [ 1, 0, 0 ] ] ]
```

5 ► `IrreducibleRepresentationsDixon( G )` F
► `IrreducibleRepresentationsDixon( G, chi )` F
► `IrreducibleRepresentationsDixon( G, chilist )` F

computes (reresentatives of) all irreducible complex representations for the finite group G, using the method of [Dix93], which computes the character table and computes the representation as constituent of an induced monomial representation of a subgroup.

This method can be quite expensive for larger groups, for example it might involve calculation of the subgroup lattice of G.

If given, *chi* must be a character, in this case only a representation for *chi* is returned.

If given, *chilist* must be a list of characters, in this case only representations for characters in *chilist* are computed.

Note that this method might fail if for an irreducible representation there is no subgroup in which its reduction has a linear constituent with multiplicity one.

```
gap> a5:= AlternatingGroup( 5 );
Alt( [ 1 .. 5 ] )
gap> char:= First( Irr( a5 ), x -> x[1] = 4 );
Character( CharacterTable( Alt( [ 1 .. 5 ] ) ), [ 4, 0, 1, -1, -1 ] )
gap> hom:=IrreducibleRepresentationsDixon( a5, char );;
gap> Order( a5.1*a5.2 ) = Order( Image( hom, a5.1 )*Image( hom, a5.2 ) );
true
gap> reps:= List( ConjugacyClasses( a5 ), Representative );;
gap> List( reps, g -> TraceMat( Image( hom, g ) ) );
[ 4, 0, 1, -1, -1 ]
```

69.13 Representations given by modules

1 ► `IrreducibleModules( G, F, dim )` O

returns a list of length 2. The first entry is a generating system of G. The second entry is a list of all irreducible modules of G over the field F in dimension *dim*, given as MeatAxe modules (see 67.1.1).

2 ► `AbsoluteIrreducibleModules( G, F, dim )` O
► `AbsolutIrreducibleModules( G, F, dim )` O

returns a list of length 2. The first entry is a generating system of G. The second entry is a list of all absolute irreducible modules of G over the field F in dimension *dim*, given as MeatAxe modules (see 67.1.1).

3 ► `RegularModule( G, F )` O

returns a list of length 2. The first entry is a generating system of G. The second entry is the regular module of G over F, given as a MeatAxe module (see 67.1.1).

(Extensions by modules can be formed by the command `Extensions`, see 44.8.4.)

69.14 The Dixon-Schneider Algorithm

The GAP library implementation of the Dixon-Schneider algorithm first computes the linear characters, using the commutator factor group. If irreducible characters are missing afterwards, they are computed using the techniques described in [Dix67], [Sch90] and [Hul93].

Called with a group G, the function `CharacterTable` (see 69.3.1) returns a character table object that stores already information such as class lengths, but not the irreducible characters. The routines that compute the irreducibles may use the information that is already contained in this table object. In particular the ordering of classes in the computed characters coincides with the ordering of classes in the character table of G (see 69.6). Thus it is possible to combine computations using the group with character theoretic computations (see 69.15 for details), for example one can enter known characters. Note that the user is responsible for the correctness of the characters. (There is little use in providing the trivial character to the routine.)

The computation of irreducible characters from the group needs to identify the classes of group elements very often, so it can be helpful to store a class list of all group elements. Since this is obviously limited by the group order, it is controlled by the global function `IsDxLargeGroup` (see 69.15.8).

The routines compute in a prime field of size p, such that the exponent of the group divides $(p-1)$ and such that $2\sqrt{|G|} < p$. Currently prime fields of size smaller than 65 536 are handled more efficiently than larger prime fields, so the runtime of the character calculation depends on how large the chosen prime is.

The routine stores a Dixon record (see 69.15.1) in the group that helps routines that identify classes, for example `FusionConjugacyClasses`, to work much faster. Note that interrupting Dixon-Schneider calculations will prevent GAP from cleaning up the Dixon record; when the computation by `IrrDixonSchneider` is complete, the possibly large record is shrunk to an acceptable size.

69.15 Advanced Methods for Dixon-Schneider Calculations

The computation of irreducible characters of very large groups may take quite some time. On the other hand, for the expert only a few irreducible characters may be needed, since the other ones can be computed using character theoretic methods such as tensoring, induction, and restriction. Thus GAP provides also step-by-step routines for doing the calculations. These routines allow one to compute some characters and to stop before all are calculated. Note that there is no "safety net": The routines (being somehow internal) do no error checking, and assume the information given is correct.

When the info level of `InfoCharacterTable` if positive, information about the progress of splitting is printed. (The default value is zero.)

1 ▶ `DixonRecord(` *G* `)` AM

The `DixonRecord` of a group contains information used by the routines to compute the irreducible characters and related information via the Dixon-Schneider algorithm such as class arrangement and character spaces split obtained so far. Usually this record is passed as argument to all subfunctions to avoid a long argument list. It has a component `.conjugacyClasses` which contains the classes of G **ordered as the algorithm needs them**.

2 ▶ `DixonInit(` *G* `)` F

This function does all the initializations for the Dixon-Schneider algorithm. This includes calculation of conjugacy classes, power maps, linear characters and character morphisms. It returns a record (see 69.15.1, 69.16) that can be used when calculating the irreducible characters of G interactively.

3 ▶ `DixontinI(` *D* `)` F

This function ends a Dixon-Schneider calculation. It sorts the characters according to the degree and unbinds components in the Dixon record that are not of use any longer. It returns a list of irreducible characters.

4 ► `DixonSplit( D )` F

This function performs one splitting step in the Dixon-Schneider algorithm. It selects a class, computes the (partial) class sum matrix, uses it to split character spaces and stores all the irreducible characters obtained that way.

The class to use for splitting is chosen via `BestSplittingMatrix` and the options described for this function apply here.

`DixonSplit` returns `true` if a split was performed and `fail` otherwise.

5 ► `BestSplittingMatrix( D )` F

returns the number of the class sum matrix that is assumed to yield the best (cost/earning ration) split. This matrix then will be the next one computed and used.

The global option `maxclasslen` (defaulting to `infinity`) is recognized by `BestSplittingMatrix`: Only classes whose length is limited by the value of this option will be considered for splitting. If no usable class remains, `fail` is returned.

6 ► `DxIncludeIrreducibles( D, new[, newmod] )` F

This function takes a list of irreducible characters *new*, each given as a list of values (corresponding to the class arrangement in D), and adds these to a partial computed list of irreducibles as maintained by the Dixon record D. This permits one to add characters in interactive use obtained from other sources and to continue the Dixon-Schneider calculation afterwards. If the optional argument *newmod* is given, it must be a list of reduced characters, corresponding to *new*. (Otherwise the function has to reduce the characters itself.)

The function closes the new characters under the action of Galois automorphisms and tensor products with linear characters.

7 ► `SplitCharacters( D, list )` F

This routine decomposes the characters given in *list* according to the character spaces found up to this point. By applying this routine to tensor products etc., it may result in characters with smaller norm, even irreducible ones. Since the recalculation of characters is only possible if the degree is small enough, the splitting process is applied only to characters of sufficiently small degree.

8 ► `IsDxLargeGroup( G )` F

returns `true` if the order of the group G is smaller than the current value of the global variable `DXLARGEGROUPORDER`, and `false` otherwise. In Dixon-Schneider calculations, for small groups in the above sense a class map is stored, whereas for large groups, each occurring element is identified individually.

69.16 Components of a Dixon Record

The "Dixon record" D returned by `DixonInit` (see 69.15.2) stores all the information that is used by the Dixon-Schneider routines while computing the irreducible characters of a group. Some entries, however, may be useful to know about when using the algorithm interactively (see 69.17).

`group`:
: the group G of which the character table is to be computed,

`conjugacyClasses`:
: classes of G (all characters stored in the Dixon record correspond to this arrangement of classes),

`irreducibles`:
: the already known irreducible characters (given as lists of their values on the conjugacy classes),

`characterTable`:
: the `CharacterTable` value of G (whose characters are not yet known),

`ClassElement( D, el )`:
: a function that returns the number of the class of G that contains the element *el*.

69.17 An Example of Advanced Dixon-Schneider Calculations

First, we set the appropriate info level higher

```
gap> SetInfoLevel( InfoCharacterTable, 1 );
```

for printout of some internal results. We now define our group, which is isomorphic to $\mathrm{PSL}_4(3)$.

```
gap> g:= PrimitiveGroup(40,5);
PSL(4,3)
gap> Size(g);
6065280
gap> d:= DixonInit( g );;
#I  29 classes
#I  choosing prime 28081
gap> c:= d.characterTable;;
```

After the initialisation, one structure matrix is evaluated, yielding smaller spaces and several irreducible characters.

```
gap> DixonSplit( d );
#I  Matrix 4,Representative of Order 3,Centralizer: 5832
#I  Dimensions: [ 1, 12, 2, 2, 4, 2, 1, 1, 1, 1, 1 ]
#I  Two-dim space split
#I  Two-dim space split
#I  Two-dim space split
4
```

In this case spaces of the listed dimensions are a result of the splitting process. The three two dimensional spaces are split successfully by combinatoric means.

We obtain several irreducible characters by tensor products and notify them to the Dixon record.

```
gap> asp:= AntiSymmetricParts( c, d.irreducibles, 2 );;
gap> ro:= ReducedOrdinary( c, d.irreducibles, asp );;
gap> Length( ro.irreducibles );
3
gap> DxIncludeIrreducibles( d, ro.irreducibles );
```

The tensor products of the nonlinear characters among each other are reduced with the irreducible characters. The result is split according to the spaces found, which yields characters of smaller norms, but no new irreducibles.

```
gap> nlc:= Filtered( d.irreducibles, i -> i[1] > 1 );;
gap> t:= Tensored( nlc, nlc );;
gap> ro:= ReducedOrdinary( c, d.irreducibles, t );;  ro.irreducibles;
[  ]
gap> List( ro.remainders, i -> ScalarProduct( c, i, i) );
[ 2, 2, 4, 4, 4, 4, 13, 13, 18, 18, 19, 21, 21, 36, 36, 29, 34, 34, 42, 34,
  48, 54, 62, 68, 68, 78, 84, 84, 90, 88, 159, 169, 169, 172, 172, 266, 271,
  271, 268, 274, 274, 280, 328, 373, 373, 456, 532, 576, 679, 683, 683, 754,
  768, 768, 890, 912, 962, 1453, 1453, 1601, 1601, 1728, 1739, 1739, 1802,
  2058, 2379, 2414, 2543, 2744, 2744, 2920, 3078, 3078, 4275, 4275, 4494,
  4760, 5112, 5115, 5115, 5414, 6080, 6318, 7100, 7369, 7369, 7798, 8644,
  10392, 12373, 12922, 14122, 14122, 18948, 21886, 24641, 24641, 25056,
  38942, 44950, 78778 ]
gap> t:= SplitCharacters( d, ro.remainders );;
```

```
gap> List( t, i -> ScalarProduct( c, i, i ) );
[ 2, 2, 4, 2, 2, 4, 4, 6, 3, 5, 5, 9, 9, 4, 12, 13, 18, 18, 18, 26, 32, 32,
  16, 42, 36, 84, 84, 90, 88, 159, 169, 169, 172, 172, 266, 271, 271, 268,
  274, 274, 280, 328, 373, 373, 456, 532, 576, 679, 683, 683, 754, 768, 768,
  890, 912, 962, 1453, 1453, 1601, 1601, 1728, 1739, 1739, 1802, 2058, 2379,
  2414, 2543, 2744, 2744, 2920, 3078, 3078, 4275, 4275, 4494, 4760, 5112,
  5115, 5115, 5414, 6080, 6318, 7100, 7369, 7369, 7798, 8644, 10392, 12373,
  12922, 14122, 14122, 18948, 21886, 24641, 24641, 25056, 38942, 44950, 78778
 ]
```

Finally we calculate the characters induced from all cyclic subgroups and obtain the missing irreducibles by applying the LLL-algorithm to them.

```
gap> ic:= InducedCyclic( c, "all" );;
gap> ro:= ReducedOrdinary( c, d.irreducibles, ic );;
gap> Length( ro.irreducibles );
0
gap> l:= LLL( c, ro.remainders );;
gap> Length( l.irreducibles );
13
```

The LLL returns class function objects (see Chapter 70), and the Dixon record works with character values lists. So we convert them to a list of values before feeding them in the machinery of the Dixon-algorithm.

```
gap> l.irreducibles[1];
Character( CharacterTable( PSL(4,3) ),
[ 640, 0, 0, -8, 0, 0, E(13)^7+E(13)^8+E(13)^11, E(13)+E(13)^3+E(13)^9,
  E(13)^2+E(13)^5+E(13)^6, E(13)^4+E(13)^10+E(13)^12, 0, 0, 0, 0, 0, 0, 0, 0,
  0, -8, -8, 0, 0, 0, 0, 0, 1, 1, 1 ] )
gap> l:=List(l.irreducibles,ValuesOfClassFunction);;
gap> DxIncludeIrreducibles( d, l );
gap> Length( d.irreducibles );
29
gap> Length( d.classes );
29
```

It turns out we have found all irreducible characters. As the last step, we obtain the irreducible characters and tell them to the group. This makes them available also to the character table.

```
gap> irrs:= DixontinI( d );;
#I  Total:1 matrices,[ 4 ]
gap> SetIrr(g,irrs);
gap> Length(Irr(c));
29
gap> SetInfoLevel( InfoCharacterTable, 0 );
```

69.18 Constructing Character Tables from Others

The following operations take one or more character table arguments, and return a character table. This holds also for `BrauerTable` (see 69.3.2); note that the return value of `BrauerTable` will in general not know the irreducible Brauer characters, and GAP might be unable to compute these characters.

Note that whenever fusions between input and output tables occur in these operations, they are stored on the concerned tables, and the `NamesOfFusionSources` values are updated.

(The interactive construction of character tables using character theoretic methods and incomplete tables is not described here.) **@Currently it is not supported and will be described in a chapter of its own when it becomes available@**.

1 ► `CharacterTableDirectProduct( tbl1, tbl2 )` O

is the table of the direct product of the character tables *tbl1* and *tbl2*.

The matrix of irreducibles of this table is the Kronecker product (see 24.4.8) of the irreducibles of *tbl1* and *tbl2*.

Products of ordinary and Brauer character tables are supported.

In general, the result will not know an underlying group, so missing power maps (for prime divisors of the result) and irreducibles of the input tables may be computed in order to construct the table of the direct product.

The embeddings of the input tables into the direct product are stored, they can be fetched with `GetFusionMap` (see 71.2.3); if *tbl1* is equal to *tbl2* then the two embeddings are distinguished by their `specification` components `"1"` and `"2"`, respectively.

Analogously, the projections from the direct product onto the input tables are stored, and can be distinguished by the `specification` components.

The attribute `FactorsOfDirectProduct` (see 69.18.2) is set to the lists of arguments.

The `*` operator for two character tables (see 69.7) delegates to `CharacterTableDirectProduct`.

```
gap> c2:= CharacterTable( "Cyclic", 2 );;
gap> s3:= CharacterTable( "Symmetric", 3 );;
gap> Display( CharacterTableDirectProduct( c2, s3 ) );
C2xSym(3)

     2  2  2  1  2  2  1
     3  1  .  1  1  .  1

       1a 2a 3a 2b 2c 6a
    2P 1a 1a 3a 1a 1a 3a
    3P 1a 2a 1a 2b 2c 2b

X.1     1 -1  1  1 -1  1
X.2     2  . -1  2  . -1
X.3     1  1  1  1  1  1
X.4     1 -1  1 -1  1 -1
X.5     2  . -1 -2  .  1
X.6     1  1  1 -1 -1 -1
```

2 ► `FactorsOfDirectProduct( tbl )` A

For an ordinary character table that has been constructed via `CharacterTableDirectProduct` (see 69.18.1), the value of `FactorsOfDirectProduct` is the list of arguments in the `CharacterTableDirectProduct` call.

Note that there is no default method for **computing** the value of `FactorsOfDirectProduct`.

3 ► `CharacterTableFactorGroup( tbl, classes )` O

is the character table of the factor group of the ordinary character table *tbl* by the normal closure of the classes whose positions are contained in the list *classes*.

The operator for a character table and a list of class positions (see 69.7) delegates to `CharacterTableFactorGroup`.

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> ClassPositionsOfNormalSubgroups( s4 );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1 .. 5 ] ]
gap> f:= CharacterTableFactorGroup( s4, [ 3 ] );
CharacterTable( "Sym(4)/[ 1, 3 ]" )
gap> Display( f );
Sym(4)/[ 1, 3 ]

     2  1  1  .
     3  1  .  1

       1a 2a 3a
    2P 1a 1a 3a
    3P 1a 2a 1a

X.1     1 -1  1
X.2     2  . -1
X.3     1  1  1
```

4 ▶ `CharacterTableIsoclinic( tbl )` O
▶ `CharacterTableIsoclinic( tbl, classes )` O
▶ `CharacterTableIsoclinic( tbl, classes, centre )` O

If *tbl* is the character table of a group with structure $2.G.2$ with a central subgroup Z of order 2 and a normal subgroup N of index 2 that contains Z then `CharacterTableIsoclinic` returns the character table of the isoclinic group in the sense of the ATLAS of Finite Groups [CCN+85], Chapter 6, Section 7. If N is not uniquely determined then the positions of the classes forming N must be entered as list *classes*. If Z is not unique in N then the position of the class consisting of the involution in Z must be entered as *centre*.

```
gap> d8:= CharacterTable( "Dihedral", 8 );;
gap> nsg:= ClassPositionsOfNormalSubgroups( d8 );
[ [ 1 ], [ 1, 3 ], [ 1 .. 3 ], [ 1, 3, 4 ], [ 1, 3 .. 5 ], [ 1 .. 5 ] ]
gap> Display( CharacterTableIsoclinic( d8, nsg[3] ) );
Isoclinic(Dihedral(8))

     2  3  2  3  2  2

       1a 4a 2a 4b 4c
    2P 1a 2a 1a 2a 2a

X.1     1  1  1  1  1
X.2     1  1  1 -1 -1
X.3     1 -1  1  1 -1
X.4     1 -1  1 -1  1
X.5     2  . -2  .  .
```

5 ▶ `CharacterTableWreathSymmetric( tbl, n )` F

returns the character table of the wreath product of a group G with the full symmetric group on n points, where *tbl* is the character table of G.

The result has values for `ClassParameters` and `CharacterParameters` (see "ctbllib:classparameters" in the manual for the GAP Character Table Library) stored, the entries in these lists are sequences of partitions. Note that this parametrization prevents the principal character from being the first one in the list of irreducibles.

```
gap> c3:= CharacterTable( "Cyclic", 3 );;
gap> wr:= CharacterTableWreathSymmetric( c3, 2 );;
gap> Display( wr );
C3wrS2

     2  1   .   .   1  .   1  1   1   1
     3  2   2   2   2  2   2  1   1   1

        1a  3a  3b  3c 3d  3e 2a  6a  6b
    2P  1a  3b  3a  3e 3d  3c 1a  3c  3e
    3P  1a  1a  1a  1a 1a  1a 2a  2a  2a

X.1      1   1   1   1  1   1 -1  -1  -1
X.2      2   A  /A   B -1  /B  .   .   .
X.3      2  /A   A  /B -1   B  .   .   .
X.4      1 -/A  -A  -A  1 -/A -1  /A   A
X.5      2  -1  -1   2 -1   2  .   .   .
X.6      1  -A -/A -/A  1  -A -1   A  /A
X.7      1   1   1   1  1   1  1   1   1
X.8      1 -/A  -A  -A  1 -/A  1 -/A  -A
X.9      1  -A -/A -/A  1  -A  1  -A -/A

A = -E(3)^2
  = (1+ER(-3))/2 = 1+b3
B = 2*E(3)
  = -1+ER(-3) = 2b3
gap> CharacterParameters( wr )[1];
[ [ 1, 1 ], [  ], [  ] ]
```

69.19 Sorted Character Tables

1 ▶ `CharacterTableWithSortedCharacters( `*tbl*` )` O
▶ `CharacterTableWithSortedCharacters( `*tbl*`, `*perm*` )` O

is a character table that differs from *tbl* only by the succession of its irreducible characters. This affects the values of the attributes `Irr` (see 69.8.2) and `CharacterParameters` (see "ctbllib:characterparameters" in the manual for the GAP Character Table Library). Namely, these lists are permuted by the permutation *perm*.

If no second argument is given then a permutation is used that yields irreducible characters of increasing degree for the result. For the succession of characters in the result, see 69.19.2.

The result has all those attributes and properties of *tbl* that are stored in `SupportedCharacterTableInfo` and do not depend on the ordering of characters (see 69.3.3).

2 ▶ `SortedCharacters( `*tbl*`, `*chars*` )` O
▶ `SortedCharacters( `*tbl*`, `*chars*`, "norm" )` O
▶ `SortedCharacters( `*tbl*`, `*chars*`, "degree" )` O

is a list containing the characters *chars*, ordered as specified by the other arguments.

There are three possibilities to sort characters: They can be sorted according to ascending norms (parameter `"norm"`), to ascending degree (parameter `"degree"`), or both (no third parameter), i.e., characters with same norm are sorted according to ascending degree, and characters with smaller norm precede those with bigger norm.

Rational characters in the result precede other ones with same norm and/or same degree.

The trivial character, if contained in *chars*, will always be sorted to the first position.

3 ▶ `CharacterTableWithSortedClasses(` *tbl* `)` O
▶ `CharacterTableWithSortedClasses(` *tbl*`, "centralizers" )` O
▶ `CharacterTableWithSortedClasses(` *tbl*`, "representatives" )` O
▶ `CharacterTableWithSortedClasses(` *tbl*`,` *permutation* `)` O

is a character table obtained by permutation of the classes of *tbl*. If the second argument is the string `"centralizers"` then the classes of the result are sorted according to descending centralizer orders. If the second argument is the string `"representatives"` then the classes of the result are sorted according to ascending representative orders. If no second argument is given then the classes of the result are sorted according to ascending representative orders, and classes with equal representative orders are sorted according to descending centralizer orders.

If the second argument is a permutation *perm* then the classes of the result are sorted by application of this permutation.

The result has all those attributes and properties of *tbl* that are stored in `SupportedCharacterTableInfo` and do not depend on the ordering of classes (see 69.3.3).

4 ▶ `SortedCharacterTable(` *tbl*`,` *kernel* `)` F
▶ `SortedCharacterTable(` *tbl*`,` *normalseries* `)` F
▶ `SortedCharacterTable(` *tbl*`,` *facttbl*`,` *kernel* `)` F

is a character table obtained on permutation of the classes and the irreducibles characters of *tbl*.

The first form sorts the classes at positions contained in the list *kernel* to the beginning, and sorts all characters in `Irr(` *tbl* `)` such that the first characters are those that contain *kernel* in their kernel.

The second form does the same successively for all kernels k_i in the list $normalseries = [k_1, k_2, \ldots, k_n]$ where k_i must be a sublist of k_{i+1} for $1 \leq i \leq n-1$.

The third form computes the table F of the factor group of *tbl* modulo the normal subgroup formed by the classes whose positions are contained in the list *kernel*; F must be permutation equivalent to the table *facttbl*, in the sense of `TransformingPermutationsCharacterTables` (see 69.20.4), otherwise `fail` is returned. The classes of *tbl* are sorted such that the preimages of a class of F are consecutive, and that the succession of preimages is that of *facttbl*. `Irr(` *tbl* `)` is sorted as with `SortCharTable(` *tbl*`,` *kernel* `)`.

(**Note** that the transformation is only unique up to table automorphisms of F, and this need not be unique up to table automorphisms of *tbl*.)

All rearrangements of classes and characters are stable, i.e., the relative positions of classes and characters that are not distinguished by any relevant property is not changed.

The result has all those attributes and properties of *tbl* that are stored in `SupportedCharacterTableInfo` and do not depend on the ordering of classes and characters (see 69.3.3).

The `ClassPermutation` value of *tbl* is changed if necessary, see 69.5.

`SortedCharacterTable` uses `CharacterTableWithSortedClasses` and `CharacterTableWithSortedCharacters` (see 69.19.3, 69.19.1).

5 ▶ `ClassPermutation(` *tbl* `)` A

is a permutation π of classes of the character table *tbl*. If it is stored then class fusions into *tbl* that are stored on other tables must be followed by π in order to describe the correct fusion.

This attribute value is bound only if *tbl* was obtained from another table by permuting the classes, using `CharacterTableWithSortedClasses` or `SortedCharacterTable`, (see 69.19.3, 69.19.4).

It is necessary because the original table and the sorted table have the same identifier (and the same group if known), and hence the same fusions are valid for the two tables.

```
gap> tbl:= CharacterTable( "Symmetric", 4 );
CharacterTable( "Sym(4)" )
gap> Display( tbl );
Sym(4)

     2  3  2  3  .  2
     3  1  .  .  1  .

       1a 2a 2b 3a 4a
    2P 1a 1a 1a 3a 2b
    3P 1a 2a 2b 1a 4a

X.1     1 -1  1  1 -1
X.2     3 -1 -1  .  1
X.3     2  .  2 -1  .
X.4     3  1 -1  . -1
X.5     1  1  1  1  1

gap> srt1:= CharacterTableWithSortedCharacters( tbl );
CharacterTable( "Sym(4)" )
gap> List( Irr( srt1 ), Degree );
[ 1, 1, 2, 3, 3 ]
gap> srt2:= CharacterTableWithSortedClasses( tbl );
CharacterTable( "Sym(4)" )
gap> SizesCentralizers( tbl );
[ 24, 4, 8, 3, 4 ]
gap> SizesCentralizers( srt2 );
[ 24, 8, 4, 3, 4 ]
gap> nsg:= ClassPositionsOfNormalSubgroups( tbl );
[ [ 1 ], [ 1, 3 ], [ 1, 3, 4 ], [ 1 .. 5 ] ]
gap> srt3:= SortedCharacterTable( tbl, nsg );
CharacterTable( "Sym(4)" )
gap> nsg:= ClassPositionsOfNormalSubgroups( srt3 );
[ [ 1 ], [ 1, 2 ], [ 1 .. 3 ], [ 1 .. 5 ] ]
gap> Display( srt3 );
Sym(4)

     2  3  3  .  2  2
     3  1  .  1  .  .

       1a 2a 3a 2b 4a
    2P 1a 1a 3a 1a 2a
    3P 1a 2a 1a 2b 4a

X.1     1  1  1  1  1
X.2     1  1  1 -1 -1
X.3     2  2 -1  .  .
X.4     3 -1  . -1  1
X.5     3 -1  .  1 -1

gap> ClassPermutation( srt3 );
(2,4,3)
```

69.20 Automorphisms and Equivalence of Character Tables

1 ▶ `MatrixAutomorphisms(` *mat*`[,` *maps*`,` *subgroup*`] )` O

For a matrix *mat*, `MatrixAutomorphisms` returns the group of those permutations of the columns of *mat* that leave the set of rows of *mat* invariant.

If the arguments *maps* and *subgroup* are given, only the group of those permutations is constructed that additionally fix each list in the list *maps* under pointwise action `OnTuples`, and *subgroup* is a permutation group that is known to be a subgroup of this group of automorphisms.

Each entry in *maps* must be a list of same length as the rows of *mat*. For example, if *mat* is a list of irreducible characters of a group then the list of element orders of the conjugacy classes (see 69.8.5) may be an entry in *maps*.

2 ▶ `TableAutomorphisms(` *tbl*`,` *characters* `)` O
▶ `TableAutomorphisms(` *tbl*`,` *characters*`, "closed" )` O
▶ `TableAutomorphisms(` *tbl*`,` *characters*`,` *subgroup* `)` O

`TableAutomorphisms` returns the permutation group of those matrix automorphisms (see 69.20.1) of the list *characters* that leave the element orders (see 69.8.5) and all stored power maps (see 71.1.1) of the character table *tbl*.

If *characters* is closed under Galois conjugacy –this is always fulfilled for ordinary character tables– the string `"closed"` may be entered as the third argument. Alternatively, a known subgroup *subgroup* of the table automorphisms can be entered as the third argument.

The attribute `AutomorphismsOfTable` (see 69.8.8) can be used to compute and store the table automorphisms for the case that *characters* equals `Irr(` *tbl* `)`.

```
gap> tbld8:= CharacterTable( "Dihedral", 8 );;
gap> irrd8:= Irr( tbld8 );
[ Character( CharacterTable( "Dihedral(8)" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 1, 1, 1, -1, -1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 1, -1, 1, 1, -1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 1, -1, 1, -1, 1 ] ),
  Character( CharacterTable( "Dihedral(8)" ), [ 2, 0, -2, 0, 0 ] ) ]
gap> orders:= OrdersClassRepresentatives( tbld8 );
[ 1, 4, 2, 2, 2 ]
gap> MatrixAutomorphisms( irrd8 );
Group([ (4,5), (2,4) ])
gap> MatrixAutomorphisms( irrd8, [ orders ], Group( () ) );
Group([ (4,5) ])
gap> TableAutomorphisms( tbld8, irrd8 );
Group([ (4,5) ])
```

3 ▶ `TransformingPermutations(` *mat1*`,` *mat2* `)` O

Let *mat1* and *mat2* be matrices. `TransformingPermutations` tries to construct a permutation π that transforms the set of rows of the matrix *mat1* to the set of rows of the matrix *mat2* by permuting the columns.

If such a permutation exists, a record with components `columns`, `rows`, and `group` is returned, otherwise `fail`. For `TransformingPermutations`$(mat1, mat2) = r \neq$ `fail`, we have *mat2* `= Permuted( List(` *mat1*`, x -> Permuted( x,` r`.columns ) ),`r`.rows )`.

r`.group` is the group of matrix automorphisms of *mat2* (see 69.20.1). This group stabilizes the transformation in the sense that applying any of its elements to the columns of *mat2* preserves the set of rows of *mat2*.

4 ▶ `TransformingPermutationsCharacterTables( tbl1, tbl2 )` O

Let *tbl1* and *tbl2* be character tables. `TransformingPermutationsCharacterTables` tries to construct a permutation π that transforms the set of rows of the matrix `Irr( tbl1 )` to the set of rows of the matrix `Irr( tbl2 )` by permuting the columns (see 69.20.3), such that π transforms also the power maps and the element orders.

If such a permutation π exists then a record with the components `columns` (π), `rows` (the permutation of `Irr( tbl1 )` corresponding to π), and `group` (the permutation group of table automorphisms of *tbl2*, see 69.8.8) is returned. If no such permutation exists, `fail` is returned.

```
gap> tblq8:= CharacterTable( "Quaternionic", 8 );;
gap> irrq8:= Irr( tblq8 );
[ Character( CharacterTable( "Q8" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "Q8" ), [ 1, 1, 1, -1, -1 ] ),
  Character( CharacterTable( "Q8" ), [ 1, -1, 1, 1, -1 ] ),
  Character( CharacterTable( "Q8" ), [ 1, -1, 1, -1, 1 ] ),
  Character( CharacterTable( "Q8" ), [ 2, 0, -2, 0, 0 ] ) ]
gap> OrdersClassRepresentatives( tblq8 );
[ 1, 4, 2, 4, 4 ]
gap> TransformingPermutations( irrd8, irrq8 );
rec( columns := (), rows := (), group := Group([ (4,5), (2,4) ]) )
gap> TransformingPermutationsCharacterTables( tbld8, tblq8 );
fail
gap> tbld6:= CharacterTable( "Dihedral", 6 );;
gap> tbls3:= CharacterTable( "Symmetric", 3 );;
gap> TransformingPermutationsCharacterTables( tbld6, tbls3 );
rec( columns := (2,3), rows := (1,3,2), group := Group(()) )
```

5 ▶ `FamiliesOfRows( mat, maps )` F

distributes the rows of the matrix *mat* into families as follows. Two rows of *mat* belong to the same family if there is a permutation of columns that maps one row to the other row. Each entry in the list *maps* is regarded to form a family of length 1.

`FamiliesOfRows( mat, maps )` returns a record with components

`famreps`
: the list of representatives for each family,

`permutations`
: the list that contains at position `i` a list of permutations that map the members of the family with representative `famreps[i]` to that representative,

`families`
: the list that contains at position i the list of positions of members of the family of representative `famreps[`i`]`; (for the element *maps*`[i]` the only member of the family will get the number `Length(` *mat* `) + i`).

69.21 Storing Normal Subgroup Information

1 ► `NormalSubgroupClassesInfo( tbl )` AM

Let *tbl* be the ordinary character table of the group G. Many computations for group characters of G involve computations in normal subgroups or factor groups of G.

In some cases the character table *tbl* is sufficient; for example questions about a normal subgroup N of G can be answered if one knows the conjugacy classes that form N, e.g., the question whether a character of G restricts irreducibly to N. But other questions require the computation of N or even more information, like the character table of N.

In order to do these computations only once, one stores in the group a record with components to store normal subgroups, the corresponding lists of conjugacy classes, and (if necessary) the factor groups, namely

`nsg`:
: list of normal subgroups of G, may be incomplete,

`nsgclasses`:
: at position i, the list of positions of conjugacy classes of *tbl* forming the i-th entry of the `nsg` component,

`nsgfactors`:
: at position i, if bound, the factor group modulo the i-th entry of the `nsg` component.

`NormalSubgroupClasses`, `FactorGroupNormalSubgroupClasses`, and `ClassPositionsOfNormalSubgroup` each use these components, and they are the only functions to do so.

So if you need information about a normal subgroup for that you know the conjugacy classes, you should get it using `NormalSubgroupClasses`. If the normal subgroup was already used it is just returned, with all the knowledge it contains. Otherwise the normal subgroup is added to the lists, and will be available for the next call.

For example, if you are dealing with kernels of characters using the `KernelOfCharacter` function you make use of this feature because `KernelOfCharacter` calls `NormalSubgroupClasses`.

2 ► `ClassPositionsOfNormalSubgroup( tbl, N )` F

is the list of positions of conjugacy classes of the character table *tbl* that are contained in the normal subgroup N of the underlying group of *tbl*.

3 ► `NormalSubgroupClasses( tbl, classes )` F

returns the normal subgroup of the underlying group G of the ordinary character table *tbl* that consists of those conjugacy classes of *tbl* whose positions are in the list *classes*.

If `NormalSubgroupClassesInfo( tbl ).nsg` does not yet contain the required normal subgroup, and if `NormalSubgroupClassesInfo( tbl ).normalSubgroups` is bound then the result will be identical to the group in `NormalSubgroupClassesInfo( tbl ).normalSubgroups`.

4 ► `FactorGroupNormalSubgroupClasses( tbl, classes )` F

is the factor group of the underlying group G of the ordinary character table *tbl* modulo the normal subgroup of G that consists of those conjugacy classes of *tbl* whose positions are in the list *classes*.

```
gap> g:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> SetName( g, "S4" );
gap> tbl:= CharacterTable( g );
CharacterTable( S4 )
gap> irr:= Irr( g );
[ Character( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] ),
  Character( CharacterTable( S4 ), [ 3, -1, -1, 0, 1 ] ),
  Character( CharacterTable( S4 ), [ 2, 0, 2, -1, 0 ] ),
  Character( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] ),
  Character( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] ) ]
gap> kernel:= KernelOfCharacter( irr[3] );
Group([ (1,2)(3,4), (1,3)(2,4) ])
gap> HasNormalSubgroupClassesInfo( tbl );
true
gap> NormalSubgroupClassesInfo( tbl );
rec( nsg := [ Group([ (1,2)(3,4), (1,3)(2,4) ]) ], nsgclasses := [ [ 1, 3 ] ],
  nsgfactors := [  ] )
gap> ClassPositionsOfNormalSubgroup( tbl, kernel );
[ 1, 3 ]
gap> FactorGroupNormalSubgroupClasses( tbl, [ 1, 3 ] );
Group([ f1, f2 ])
gap> NormalSubgroupClassesInfo( tbl );
rec( nsg := [ Group([ (1,2)(3,4), (1,3)(2,4) ]) ], nsgclasses := [ [ 1, 3 ] ],
  nsgfactors := [ Group([ f1, f2 ]) ] )
```

70 Class Functions

This chapter describes operations for **class functions of finite groups**. For operations concerning **character tables**, see Chapter 69.

Several examples in this chapter require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

```
gap> LoadPackage( "ctbllib" );
true
```

1 ▶ `IsClassFunction( obj )` C

A **class function** (in characteristic p) of a finite group G is a map from the set of (p-regular) elements in G to the cyclotomics that is constant on conjugacy classes of G.

Each class function in GAP is represented by an **immutable list**, where at the i-th position the value on the i-th conjugacy class of the character table of G is stored. The ordering of the conjugacy classes is the one used in the underlying character table. Note that if the character table has access to its underlying group then the ordering of conjugacy classes in the group and in the character table may differ (see 69.6); class functions always refer to the ordering of classes in the character table.

Class function objects in GAP are not just plain lists, they store the character table of the group G as value of the attribute `UnderlyingCharacterTable` (see 70.2.1). The group G itself is accessible only via the character table and thus only if the character table stores its group, as value of the attribute `UnderlyingGroup`. The reason for this is that many computations with class functions are possible without using their groups, for example class functions of character tables in the GAP character table library do in general not have access to their underlying groups.

There are (at least) two reasons why class functions in GAP are **not** implemented as mappings. First, we want to distinguish class functions in different characteristics, for example to be able to define the Frobenius character of a given Brauer character; viewed as mappings, the trivial characters in all characteristics coprime to the order of G are equal. Second, the product of two class functions shall be again a class function, whereas the product of general mappings is defined as composition.

A further argument is that the typical operations for mappings such as `Image` (see 31.3.6) and `PreImage` (see 31.4.6) play no important role for class functions.

70.1 Why Class Functions?

In principle it is possible to represent group characters or more general class functions by the plain lists of their values, and in fact many operations for class functions work with plain lists of class function values. But this has two disadvantages.

First, it is then necessary to regard a values list explicitly as a class function of a particular character table, by supplying this character table as an argument. In practice this means that with this setup, the user has the task to put the objects into the right context. For example, forming the scalar product or the tensor product of two class functions or forming an induced class function or a conjugate class function then needs

three arguments in this case; this is particularly inconvenient in cases where infix operations cannot be used because of the additional argument, as for tensor products and induced class functions.

Second, when one says that "χ is a character of a group G" then this object χ carries a lot of information. χ has certain properties such as being irreducible or not. Several subgroups of G are related to χ, such as the kernel and the centre of χ. Other attributes of characters are the determinant and the central character. This knowledge cannot be stored in a plain list.

For dealing with a group together with its characters, and maybe also subgroups and their characters, it is desirable that GAP keeps track of the interpretation of characters. On the other hand, for using characters without accessing their groups, such as characters of tables from the GAP table library, dealing just with values lists is often sufficient. In particular, if one deals with incomplete character tables then it is often necessary to specify the arguments explicitly, for example one has to choose a fusion map or power map from a set of possibilities.

The main idea behind class function objects is that a class function object is equal to its values list in the sense of `\=`, so class function objects can be used wherever their values lists can be used, but there are operations for class function objects that do not work just with values lists. GAP library functions prefer to return class function objects rather than returning just values lists, for example `Irr` lists (see 69.8.2) consist of class function objects, and `TrivialCharacter` (see 70.7.1) returns a class function object.

Here is an **example** that shows both approaches. First we define some groups.

```
gap> S4:= SymmetricGroup( 4 );;  SetName( S4, "S4" );
gap> D8:= SylowSubgroup( S4, 2 );; SetName( D8, "D8" );
```

We do some computations using the functions described later in this Chapter, first with class function objects.

```
gap> irrS4:= Irr( S4 );;
gap> irrD8:= Irr( D8 );;
gap> chi:= irrD8[4];
Character( CharacterTable( D8 ), [ 1, 1, 1, -1, -1 ] )
gap> chi * chi;
Character( CharacterTable( D8 ), [ 1, 1, 1, 1, 1 ] )
gap> ind:= chi ^ S4;
Character( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] )
gap> List( irrS4, x -> ScalarProduct( x, ind ) );
[ 0, 0, 0, 1, 0 ]
gap> det:= Determinant( ind );
Character( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] )
gap> cent:= CentralCharacter( ind );
ClassFunction( CharacterTable( S4 ), [ 1, 2, -1, 0, -2 ] )
gap> rest:= Restricted( cent, D8 );
ClassFunction( CharacterTable( D8 ), [ 1, 2, -1, -1, -2 ] )
```

Now we repeat these calculations with plain lists of character values. Here we need the character tables in some places.

```
gap> tS4:= CharacterTable( S4 );;
gap> tD8:= CharacterTable( D8 );;
gap> chi:= ValuesOfClassFunction( irrD8[4] );
[ 1, 1, 1, -1, -1 ]
gap> Tensored( [ chi ], [ chi ] )[1];
[ 1, 1, 1, 1, 1 ]
gap> ind:= InducedClassFunction( tD8, chi, tS4 );
```

```
ClassFunction( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] )
gap> List( Irr( tS4 ), x -> ScalarProduct( tS4, x, ind ) );
[ 0, 0, 0, 1, 0 ]
gap> det:= DeterminantOfCharacter( tS4, ind );
ClassFunction( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] )
gap> cent:= CentralCharacter( tS4, ind );
ClassFunction( CharacterTable( S4 ), [ 1, 2, -1, 0, -2 ] )
gap> rest:= Restricted( tS4, cent, tD8 );
ClassFunction( CharacterTable( D8 ), [ 1, 2, -1, -1, -2 ] )
```

If one deals with character tables from the GAP table library then one has no access to their groups, but often the tables provide enough information for computing induced or restricted class functions, symmetrizations etc., because the relevant class fusions and power maps are often stored on library tables. In these cases it is possible to use the tables instead of the groups as arguments. (If necessary information is not uniquely determined by the tables then an error is signalled.)

```
gap> s5 := CharacterTable( "A5.2" );; irrs5 := Irr( s5  );;
gap> m11:= CharacterTable( "M11"  );; irrm11:= Irr( m11 );;
gap> chi:= TrivialCharacter( s5 );
Character( CharacterTable( "A5.2" ), [ 1, 1, 1, 1, 1, 1, 1 ] )
gap> chi ^ m11;
Character( CharacterTable( "M11" ), [ 66, 10, 3, 2, 1, 1, 0, 0, 0, 0 ] )
gap> Determinant( irrs5[4] );
Character( CharacterTable( "A5.2" ), [ 1, 1, 1, 1, -1, -1, -1 ] )
```

Functions that compute **normal** subgroups related to characters have counterparts that return the list of class positions corresponding to these groups.

```
gap> ClassPositionsOfKernel( irrs5[2] );
[ 1, 2, 3, 4 ]
gap> ClassPositionsOfCentre( irrs5[2] );
[ 1, 2, 3, 4, 5, 6, 7 ]
```

Non-normal subgroups cannot be described this way, so for example inertia subgroups (see 70.8.13) can in general not be computed from character tables without access to their groups.

70.2 Basic Operations for Class Functions

Basic operations for class functions are `UnderlyingCharacterTable` (see 70.2.1), `ValuesOfClassFunction` (see 70.2.2), and the basic operations for lists (see 21.2).

1 ▶ `UnderlyingCharacterTable( psi )` A

For a class function *psi* of the group G, say, the character table of G is stored as value of `UnderlyingCharacterTable`. The ordering of entries in the list *psi* (see 70.2.2) refers to the ordering of conjugacy classes in this character table.

If *psi* is an ordinary class function then the underlying character table is the ordinary character table of G (see 69.8.4), if *psi* is a class function in characteristic $p \neq 0$ then the underlying character table is the p-modular Brauer table of G (see 69.3.2). So the underlying characteristic of *psi* can be read off from the underlying character table.

2 ▶ `ValuesOfClassFunction( psi )` A

is the list of values of the class function *psi*, the i-th entry being the value on the i-th conjugacy class of the underlying character table (see 70.2.1).

```
gap> g:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> psi:= TrivialCharacter( g );
Character( CharacterTable( Sym( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1, 1 ] )
gap> UnderlyingCharacterTable( psi );
CharacterTable( Sym( [ 1 .. 4 ] ) )
gap> ValuesOfClassFunction( psi );
[ 1, 1, 1, 1, 1 ]
gap> IsList( psi );
true
gap> psi[1];
1
gap> Length( psi );
5
gap> IsBound( psi[6] );
false
gap> Concatenation( psi, [ 2, 3 ] );
[ 1, 1, 1, 1, 1, 2, 3 ]
```

70.3 Comparison of Class Functions

With respect to \= and \<, class functions behave equally to their lists of values (see 70.2.2). So two class functions are equal if and only if their lists of values are equal, no matter whether they are class functions of the same character table, of the same group but w.r.t. different class ordering, or of different groups.

```
gap> grps:= Filtered( AllSmallGroups( 8 ), g -> not IsAbelian( g ) );
[ <pc group of size 8 with 3 generators>,
  <pc group of size 8 with 3 generators> ]
gap> t1:= CharacterTable( grps[1] );  SetName( t1, "t1" );
CharacterTable( <pc group of size 8 with 3 generators> )
gap> t2:= CharacterTable( grps[2] );  SetName( t2, "t2" );
CharacterTable( <pc group of size 8 with 3 generators> )
gap> irr1:= Irr( grps[1] );
[ Character( t1, [ 1, 1, 1, 1, 1 ] ), Character( t1, [ 1, -1, -1, 1, 1 ] ),
  Character( t1, [ 1, -1, 1, 1, -1 ] ), Character( t1, [ 1, 1, -1, 1, -1 ] ),
  Character( t1, [ 2, 0, 0, -2, 0 ] ) ]
gap> irr2:= Irr( grps[2] );
[ Character( t2, [ 1, 1, 1, 1, 1 ] ), Character( t2, [ 1, -1, -1, 1, 1 ] ),
  Character( t2, [ 1, -1, 1, 1, -1 ] ), Character( t2, [ 1, 1, -1, 1, -1 ] ),
  Character( t2, [ 2, 0, 0, -2, 0 ] ) ]
gap> irr1 = irr2;
true
gap> IsSSortedList( irr1 );
false
gap> irr1[1] < irr1[2];
false
gap> irr1[2] < irr1[3];
true
```

70.4 Arithmetic Operations for Class Functions

Class functions are **row vectors** of cyclotomics. The **additive** behaviour of class functions is defined such that they are equal to the plain lists of class function values except that the results are represented again as class functions whenever this makes sense. The **multiplicative** behaviour, however, is different. This is motivated by the fact that the tensor product of class functions is a more interesting operation than the vector product of plain lists. (Another candidate for a multiplication of compatible class functions would have been the inner product, which is implemented via the function `ScalarProduct`, see 70.8.5.) In terms of filters, the arithmetic of class functions is based on the decision that they lie in `IsGeneralizedRowVector`, with additive nesting depth 1, but they do **not** lie in `IsMultiplicativeGeneralizedRowVector` (see 21.12).

More specifically, the scalar multiple of a class function with a cyclotomic is a class function, and the sum and the difference of two class functions of the same underlying character table (see 70.2.1) are again class functions of this table. The sum and the difference of a class function and a list that is **not** a class function are plain lists, as well as the sum and the difference of two class functions of different character tables.

```
gap> g:= SymmetricGroup( 4 );;  tbl:= CharacterTable( g );;
gap> SetName( tbl, "S4" );  irr:= Irr( g );
[ Character( S4, [ 1, -1, 1, 1, -1 ] ), Character( S4, [ 3, -1, -1, 0, 1 ] ),
  Character( S4, [ 2, 0, 2, -1, 0 ] ), Character( S4, [ 3, 1, -1, 0, -1 ] ),
  Character( S4, [ 1, 1, 1, 1, 1 ] ) ]
gap> 2 * irr[5];
Character( S4, [ 2, 2, 2, 2, 2 ] )
gap> irr[1] / 7;
ClassFunction( S4, [ 1/7, -1/7, 1/7, 1/7, -1/7 ] )
gap> lincomb:= irr[3] + irr[1] - irr[5];
VirtualCharacter( S4, [ 2, -2, 2, -1, -2 ] )
gap> lincomb:= lincomb + 2 * irr[5];
VirtualCharacter( S4, [ 4, 0, 4, 1, 0 ] )
gap> IsCharacter( lincomb );
true
gap> lincomb;
Character( S4, [ 4, 0, 4, 1, 0 ] )
gap> irr[5] + 2;
[ 3, 3, 3, 3, 3 ]
gap> irr[5] + [ 1, 2, 3, 4, 5 ];
[ 2, 3, 4, 5, 6 ]
gap> zero:= 0 * irr[1];
VirtualCharacter( S4, [ 0, 0, 0, 0, 0 ] )
gap> zero + Z(3);
[ Z(3), Z(3), Z(3), Z(3), Z(3) ]
gap> irr[5] + TrivialCharacter( DihedralGroup( 8 ) );
[ 2, 2, 2, 2, 2 ]
```

The product of two class functions of the same character table is the tensor product (pointwise product) of these class functions. Thus the set of all class functions of a fixed group forms a ring, and for any field F of cyclotomics, the F-span of a given set of class functions forms an algebra.

The product of two class functions of **different** tables and the product of a class function and a list that is **not** a class function are not defined, an error is signalled in these cases. Note that in this respect, class functions behave differently from their values lists, for which the product is defined as the standard scalar product.

```
gap> tens:= irr[3] * irr[4];
Character( S4, [ 6, 0, -2, 0, 0 ] )
gap> ValuesOfClassFunction( irr[3] ) * ValuesOfClassFunction( irr[4] );
4
```

Class functions without zero values are invertible, the **inverse** is defined pointwise. As a consequence, for example groups of linear characters can be formed.

```
gap> tens / irr[1];
Character( S4, [ 6, 0, -2, 0, 0 ] )
```

Other (somewhat strange) implications of the definition of arithmetic operations for class functions, together with the general rules of list arithmetic (see 21.11), apply to the case of products involving **lists** of class functions. No inverse of the list of irreducible characters as a matrix is defined; if one is interested in the inverse matrix then one can compute it from the matrix of class function values.

```
gap> Inverse( List( irr, ValuesOfClassFunction ) );
[ [ 1/24, 1/8, 1/12, 1/8, 1/24 ], [ -1/4, -1/4, 0, 1/4, 1/4 ],
  [ 1/8, -1/8, 1/4, -1/8, 1/8 ], [ 1/3, 0, -1/3, 0, 1/3 ],
  [ -1/4, 1/4, 0, -1/4, 1/4 ] ]
```

Also the product of a class function with a list of class functions is **not** a vector-matrix product but the list of pointwise products.

```
gap> irr[1] * irr{ [ 1 .. 3 ] };
[ Character( S4, [ 1, 1, 1, 1, 1 ] ), Character( S4, [ 3, 1, -1, 0, -1 ] ),
  Character( S4, [ 2, 0, 2, -1, 0 ] ) ]
```

And the product of two lists of class functions is **not** the matrix product but the sum of the pointwise products.

```
gap> irr * irr;
Character( S4, [ 24, 4, 8, 3, 4 ] )
```

The **powering** operator `\^` has several meanings for class functions. The power of a class function by a nonnegative integer is clearly the tensor power. The power of a class function by an element that normalizes the underlying group or by a Galois automorphism is the conjugate class function. (As a consequence, the application of the permutation induced by such an action cannot be denoted by `\^`; instead one can use `Permuted`, see 21.20.16.) The power of a class function by a group or a character table is the induced class function (see 70.9.3). The power of a group element by a class function is the class function value at (the conjugacy class containing) this element.

```
gap> irr[3] ^ 3;
Character( S4, [ 8, 0, 8, -1, 0 ] )
gap> lin:= LinearCharacters( DerivedSubgroup( g ) );
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3)^2, E(3) ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2 ] ) ]
gap> List( lin, chi -> chi ^ (1,2) );
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, 1, 1 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2 ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3)^2, E(3) ] ) ]
gap> Orbit( GaloisGroup( CF(3) ), lin[2] );
[ Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3)^2, E(3) ] ),
  Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2 ] ) ]
```

```
gap> lin[1]^g;
Character( S4, [ 2, 0, 2, 2, 0 ] )
gap> (1,2,3)^lin[2];
E(3)^2
```

The **characteristic** of class functions is zero, as for all list of cyclotomics. For class functions of a p-modular character table, such as Brauer characters, the prime p is given by the `UnderlyingCharacteristic` (see 69.8.9) value of the character table.

```
gap> Characteristic( irr[1] );
0
gap> irrmod2:= Irr( g, 2 );
[ Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 1, 1 ] ),
  Character( BrauerTable( Sym( [ 1 .. 4 ] ), 2 ), [ 2, -1 ] ) ]
gap> Characteristic( irrmod2[1] );
0
gap> UnderlyingCharacteristic( UnderlyingCharacterTable( irrmod2[1] ) );
2
```

The operations `ComplexConjugate`, `GaloisCyc`, and `Permuted` return a class function when they are called with a class function; The complex conjugate of a class function that is known to be a (virtual) character is again known to be a (virtual) character, and applying an arbitrary Galois automorphism to an ordinary (virtual) character yields a (virtual) character.

```
gap> ComplexConjugate( lin[2] );
Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2 ] )
gap> GaloisCyc( lin[2], 5 );
Character( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, 1, E(3), E(3)^2 ] )
gap> Permuted( lin[2], (2,3,4) );
ClassFunction( CharacterTable( Alt( [ 1 .. 4 ] ) ), [ 1, E(3), 1, E(3)^2 ] )
```

By definition of `Order` for arbitrary monoid elements, the determinantal order (see 70.8.18) of characters cannot be the return value of `Order` for characters. One can use `Order( Determinant(` *chi* `) )` to compute the determinantal order of the class function *chi*.

```
gap> det:= Determinant( irr[3] );
Character( S4, [ 1, -1, 1, 1, -1 ] )
gap> Order( det );
2
```

70.5 Printing Class Functions

The default `ViewObj` (see 6.2.3) methods for class functions print one of the strings `"ClassFunction"`, `"VirtualCharacter"`, `"Character"` (depending on whether the class function is known to be a character or virtual character, see 70.8.1, 70.8.2), followed by the `ViewObj` output for the underlying character table (see 69.11), and the list of values. The table is chosen (and not the group) in order to distinguish class functions of different underlying characteristic (see 69.8.9).

The default `PrintObj` (see 6.2.3) method for class functions does the same as `ViewObj`, except that the character table is is `Print`-ed instead of `View`-ed.

Note that if a class function is shown only with one of the strings `"ClassFunction"`, `"VirtualCharacter"`, it may still be that it is in fact a character; just this was not known at the time when the class function was printed.

In order to reduce the space that is needed to print a class function, it may be useful to give a name (see 12.8.2) to the underlying character table.

The default `Display` (see 6.2.4) method for a class function *chi* calls `Display` for its underlying character table (see 69.11), with *chi* as the only entry in the `chars` list of the options record.

```
gap> chi:= TrivialCharacter( CharacterTable( "A5" ) );
Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] )
gap> Display( chi );
A5

     2  2  2  .  .  .
     3  1  .  1  .  .
     5  1  .  .  1  1

       1a 2a 3a 5a 5b
    2P 1a 1a 3a 5b 5a
    3P 1a 2a 1a 5b 5a
    5P 1a 2a 3a 1a 1a

Y.1     1  1  1  1  1
```

70.6 Creating Class Functions from Values Lists

1 ▶ `ClassFunction(` *tbl*`,` *values* `)` O
▶ `ClassFunction(` *G*`,` *values* `)` O

In the first form, `ClassFunction` returns the class function of the character table *tbl* with values given by the list *values* of cyclotomics. In the second form, *G* must be a group, and the class function of its ordinary character table is returned.

Note that *tbl* determines the underying characteristic of the returned class function (see 69.8.9).

2 ▶ `VirtualCharacter(` *tbl*`,` *values* `)` O
▶ `VirtualCharacter(` *G*`,` *values* `)` O

`VirtualCharacter` returns the virtual character (see 70.8.2) of the character table *tbl* or the group *G*, respectively, with values given by the list *values*.

It is **not** checked whether the given values really describe a virtual character.

3 ▶ `Character(` *tbl*`,` *values* `)` O

`Character` returns the character (see 70.8.1) of the character table *tbl* or the group *G*, respectively, with values given by the list *values*.

It is **not** checked whether the given values really describe a character.

```
gap> g:= DihedralGroup( 8 );  tbl:= CharacterTable( g );
<pc group of size 8 with 3 generators>
CharacterTable( <pc group of size 8 with 3 generators> )
gap> SetName( tbl, "D8" );
gap> phi:= ClassFunction( g, [ 1, -1, 0, 2, -2 ] );
ClassFunction( D8, [ 1, -1, 0, 2, -2 ] )
gap> psi:= ClassFunction( tbl,
>              List( Irr( g ), chi -> ScalarProduct( chi, phi ) ) );
ClassFunction( D8, [ -3/8, 1/8, 9/8, 5/8, -1/4 ] )
gap> chi:= VirtualCharacter( g, [ 0, 0, 8, 0, 0 ] );
```

```
VirtualCharacter( D8, [ 0, 0, 8, 0, 0 ] )
gap> reg:= Character( tbl, [ 8, 0, 0, 0, 0 ] );
Character( D8, [ 8, 0, 0, 0, 0 ] )
```

4 ▶ `ClassFunctionSameType( tbl, chi, values )` F

Let *tbl* be a character table, *chi* a class function object (**not** necessarily a class function of *tbl*), and *values* a list of cyclotomics. `ClassFunctionSameType` returns the class function ψ of *tbl* with values list *values*, constructed with `ClassFunction` (see 70.6.1).

If *chi* is known to be a (virtual) character then ψ is also known to be a (virtual) character.

```
gap> h:= Centre( g );;
gap> centbl:= CharacterTable( h );;  SetName( centbl, "C2" );
gap> ClassFunctionSameType( centbl, phi, [ 1, 1 ] );
ClassFunction( C2, [ 1, 1 ] )
gap> ClassFunctionSameType( centbl, chi, [ 1, 1 ] );
VirtualCharacter( C2, [ 1, 1 ] )
gap> ClassFunctionSameType( centbl, reg, [ 1, 1 ] );
Character( C2, [ 1, 1 ] )
```

70.7 Creating Class Functions using Groups

1 ▶ `TrivialCharacter( tbl )` A
▶ `TrivialCharacter( G )` A

is the **trivial character** of the group G or its character table *tbl*, respectively. This is the class function with value equal to 1 for each class.

```
gap> TrivialCharacter( CharacterTable( "A5" ) );
Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] )
gap> TrivialCharacter( SymmetricGroup( 3 ) );
Character( CharacterTable( Sym( [ 1 .. 3 ] ) ), [ 1, 1, 1 ] )
```

2 ▶ `NaturalCharacter( G )` A
▶ `NaturalCharacter( hom )` A

If the argument is a permutation group G then `NaturalCharacter` returns the (ordinary) character of the natural permutation representation of G on the set of moved points (see 40.2.3), that is, the value on each class is the number of points among the moved points of G that are fixed by any permutation in that class.

If the argument is a matrix group G in characteristic zero then `NaturalCharacter` returns the (ordinary) character of the natural matrix representation of G, that is, the value on each class is the trace of any matrix in that class.

If the argument is a group homomorphism *hom* whose image is a permutation group or a matrix group then `NaturalCharacter` returns the restriction of the natural character of the image of *hom* to the preimage of *hom*.

```
gap> NaturalCharacter( SymmetricGroup( 3 ) );
Character( CharacterTable( Sym( [ 1 .. 3 ] ) ), [ 3, 1, 0 ] )
gap> NaturalCharacter( Group( [ [ 0, -1 ], [ 1, -1 ] ] ) );
Character( CharacterTable( Group([ [ [ 0, -1 ], [ 1, -1 ] ] ]) ),
[ 2, -1, -1 ] )
gap> d8:= DihedralGroup( 8 );;  hom:= IsomorphismPermGroup( d8 );;
gap> NaturalCharacter( hom );
Character( CharacterTable( <pc group of size 8 with 3 generators> ),
```

```
[ 8, 0, 0, 0, 0 ] )
```

3 ▶ `PermutationCharacter( G, D, opr )` O
▶ `PermutationCharacter( G, U )` O

Called with a group G, an action domain or proper set D, and an action function *opr* (see Chapter 39), `PermutationCharacter` returns the **permutation character** of the action of G on D via *opr*, that is, the value on each class is the number of points in D that are fixed by an element in this class under the action *opr*.

If the arguments are a group G and a subgroup U of G then `PermutationCharacter` returns the permutation character of the action of G on the right cosets of U via right multiplication.

To compute the permutation character of a **transitive permutation group** G on the cosets of a point stabilizer U, the attribute `NaturalCharacter( G )` can be used instead of `PermutationCharacter( G, U )`.

More facilities concerning permutation characters are the transitivity test (see Section 70.8) and several tools for computing possible permutation characters (see 70.13, 70.14).

```
gap> PermutationCharacter( GL(2,2), AsSSortedList( GF(2)^2 ), OnRight );
Character( CharacterTable( SL(2,2) ), [ 4, 2, 1 ] )
gap> s3:= SymmetricGroup( 3 );;  a3:= DerivedSubgroup( s3 );;
gap> PermutationCharacter( s3, a3 );
Character( CharacterTable( Sym( [ 1 .. 3 ] ) ), [ 2, 0, 2 ] )
```

70.8 Operations for Class Functions

In the description of the following operations, the optional first argument *tbl* is needed only if the argument *chi* is a plain list and not a class function object. In this case, *tbl* must always be the character table of which *chi* shall be regarded as a class function.

1 ▶ `IsCharacter( [tbl, ]chi )` P

An **ordinary character** of a group G is a class function of G whose values are the traces of a complex matrix representation of G.

A **Brauer character** of G in characteristic p is a class function of G whose values are the complex lifts of a matrix representation of G with image a finite field of characteristic p.

2 ▶ `IsVirtualCharacter( [tbl, ]chi )` P

A **virtual character** is a class function that can be written as the difference of two proper characters (see 70.8.1).

3 ▶ `IsIrreducibleCharacter( [tbl, ]chi )` P

A character is **irreducible** if it cannot be written as the sum of two characters. For ordinary characters this can be checked using the scalar product of class functions (see 70.8.5). For Brauer characters there is no generic method for checking irreducibility.

```
gap> S4:= SymmetricGroup( 4 );;  SetName( S4, "S4" );
gap> psi:= ClassFunction( S4, [ 1, 1, 1, -2, 1 ] );
ClassFunction( CharacterTable( S4 ), [ 1, 1, 1, -2, 1 ] )
gap> IsVirtualCharacter( psi );
true
gap> IsCharacter( psi );
false
gap> chi:= ClassFunction( S4, SizesCentralizers( CharacterTable( S4 ) ) );
ClassFunction( CharacterTable( S4 ), [ 24, 4, 8, 3, 4 ] )
gap> IsCharacter( chi );
true
gap> IsIrreducibleCharacter( chi );
false
gap> IsIrreducibleCharacter( TrivialCharacter( S4 ) );
true
```

4 ► `DegreeOfCharacter(` *chi* `)` A

is the value of the character *chi* on the identity element. This can also be obtained as *chi*`[1]`.

```
gap> List( Irr( S4 ), DegreeOfCharacter );
[ 1, 3, 2, 3, 1 ]
gap> nat:= NaturalCharacter( S4 );
Character( CharacterTable( S4 ), [ 4, 2, 0, 1, 0 ] )
gap> nat[1];
4
```

5 ► `ScalarProduct( [`*tbl*`, ]`*chi*`,` *psi* `)` O

For two class functions *chi* and *psi* of the same ordinary character table *tbl*, `ScalarProduct` returns the scalar product.

If G is the underlying group of *tbl* and the two class functions are χ and ψ then this is defined as $\frac{1}{|G|}\sum_{g\in G}\chi(g)\psi(g^{-1})$.

6 ► `MatScalarProducts( [`*tbl*`, ]`*list1*`,` *list2* `)` O
► `MatScalarProducts( [`*tbl*`, ]`*list* `)` O

The first form returns the matrix of scalar products (see above) of the class functions in the list *list1* with the class functions in the list *list2*. More precisely, the matrix contains in the i-th row the list of scalar products of *list2*$[i]$ with the entries of *list1*.

The second form returns a lower triangular matrix of scalar products, containing for $(j \leq i)$ in the i-th row in column j the value `ScalarProduct`$(tbl, list[j], list[i])$.

7 ► `Norm( [`*tbl*`, ]`*chi* `)` A

For an ordinary class function *chi* of the group G, say, we have $chi = \sum_{\chi\in Irr(G)} a_\chi\chi$, with complex coefficients a_χ. The **norm** of *chi* is defined as $\sum_{\chi\in Irr(G)} a_\chi\overline{a_\chi}$.

```
gap> tbl:= CharacterTable( "A5" );;
gap> ScalarProduct( TrivialCharacter( tbl ), Sum( Irr( tbl ) ) );
1
gap> ScalarProduct( tbl, [ 1, 1, 1, 1, 1 ], Sum( Irr( tbl ) ) );
1
gap> chars:= Irr( tbl ){ [ 2 .. 4 ] };;
gap> chars:= Set( Tensored( chars, chars ) );;
```

```
gap> MatScalarProducts( Irr( tbl ), chars );
[ [ 0, 0, 0, 1, 1 ], [ 1, 1, 0, 0, 1 ], [ 1, 0, 1, 0, 1 ], [ 0, 1, 0, 1, 1 ],
  [ 0, 0, 1, 1, 1 ], [ 1, 1, 1, 1, 1 ] ]
gap> MatScalarProducts( tbl, chars );
[ [ 2 ], [ 1, 3 ], [ 1, 2, 3 ], [ 2, 2, 1, 3 ], [ 2, 1, 2, 2, 3 ],
  [ 2, 3, 3, 3, 3, 5 ] ]
gap> List( chars, Norm );
[ 2, 3, 3, 3, 3, 5 ]
```

8 ► `ConstituentsOfCharacter( [`*tbl*`, ]`*chi* `)` A

is the set of irreducible characters that occur in the decomposition of the (virtual) character *chi* with nonzero coefficient.

```
gap> nat:= NaturalCharacter( S4 );
Character( CharacterTable( S4 ), [ 4, 2, 0, 1, 0 ] )
gap> ConstituentsOfCharacter( nat );
[ Character( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( S4 ), [ 3, 1, -1, 0, -1 ] ) ]
```

9 ► `KernelOfCharacter( [`*tbl*`, ]`*chi* `)` A

For a class function *chi* of the group G, say, `KernelOfCharacter` returns the normal subgroup of G that is formed by those conjugacy classes for which the value of *chi* equals the degree of *chi*. If the underlying character table of *chi* does not store the group G then an error is signalled. (See 70.8.10 for a way to handle the kernel implicitly, by listing the positions of conjugacy classes in the kernel.)

The returned group is the kernel of any representation of G that affords *chi*.

10 ► `ClassPositionsOfKernel(` *chi* `)` A

is the list of positions of those conjugacy classes that form the kernel of the character *chi*, that is, those positions with character value equal to the character degree.

```
gap> List( Irr( S4 ), KernelOfCharacter );
[ Group([ (), (1,2)(3,4), (1,2,3) ]), Group(()),
  Group([ (1,2)(3,4), (1,4)(2,3) ]), Group(()),
  Group([ (), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4) ]) ]
gap> List( Irr( S4 ), ClassPositionsOfKernel );
[ [ 1, 3, 4 ], [ 1 ], [ 1, 3 ], [ 1 ], [ 1, 2, 3, 4, 5 ] ]
```

11 ► `CentreOfCharacter( [`*tbl*`, ]`*chi* `)` A

For a character *chi* of the group G, say, `CentreOfCharacter` returns the **centre** of *chi*, that is, the normal subgroup of all those elements of G for which the quotient of the value of *chi* by the degree of *chi* is a root of unity.

If the underlying character table of *psi* does not store the group G then an error is signalled. (See 70.8.12 for a way to handle the centre implicitly, by listing the positions of conjugacy classes in the centre.)

12 ► `ClassPositionsOfCentre(` *chi* `)` A

is the list of positions of classes forming the centre of the character *chi* (see 70.8.11).

```
gap> List( Irr( S4 ), CentreOfCharacter );
[ Group([ (), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4) ]), Group(()),
  Group([ (1,2)(3,4), (1,4)(2,3) ]), Group(()),
  Group([ (), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4) ]) ]
gap> List( Irr( S4 ), ClassPositionsOfCentre );
[ [ 1, 2, 3, 4, 5 ], [ 1 ], [ 1, 3 ], [ 1 ], [ 1, 2, 3, 4, 5 ] ]
```

13 ▶ `InertiaSubgroup( [`*tbl*`, ]`*G*`,` *chi* `)` O

Let *chi* be a character of the group H, say, and *tbl* the character table of H; if the argument *tbl* is not given then the underlying character table of *chi* (see 70.2.1) is used instead. Furthermore, let G be a group that contains H as a normal subgroup.

`InertiaSubgroup` returns the stabilizer in G of *chi*, w.r.t. the action of G on the classes of H via conjugation. In other words, `InertiaSubgroup` returns the group of all those elements $g \in G$ that satisfy $chi^g = chi$.

```
gap> der:= DerivedSubgroup( S4 );
Group([ (1,3,2), (2,4,3) ])
gap> List( Irr( der ), chi -> InertiaSubgroup( S4, chi ) );
[ S4, Alt( [ 1 .. 4 ] ), Alt( [ 1 .. 4 ] ), S4 ]
```

14 ▶ `CycleStructureClass( [`*tbl*`, ]`*chi*`,` *class* `)` O

Let *permchar* be a permutation character, and *class* the position of a conjugacy class of the character table of *permchar*. `CycleStructureClass` returns a list describing the cycle structure of each element in class *class* in the underlying permutation representation, in the same format as the result of `CycleStructurePerm` (see 40.3.2).

```
gap> nat:= NaturalCharacter( S4 );
Character( CharacterTable( S4 ), [ 4, 2, 0, 1, 0 ] )
gap> List( [ 1 .. 5 ], i -> CycleStructureClass( nat, i ) );
[ [  ], [ 1 ], [ 2 ], [ , 1 ], [ ,, 1 ] ]
```

15 ▶ `IsTransitive( [`*tbl*`, ]`*chi* `)` P

For a permutation character *chi* of the group G that corresponds to an action on the G-set Ω (see 70.7.3), `IsTransitive` returns `true` if the action of G on Ω is transitive, and `false` otherwise.

16 ▶ `Transitivity( [`*tbl*`, ]`*chi* `)` A

For a permutation character *chi* of the group G that corresponds to an action on the G-set Ω (see 70.7.3), `Transitivity` returns the maximal nonnegative integer k such that the action of G on Ω is k-transitive.

```
gap> IsTransitive( nat );  Transitivity( nat );
true
4
gap> Transitivity( 2 * TrivialCharacter( S4 ) );
0
```

17 ▶ `CentralCharacter( [`*tbl*`, ]`*chi* `)` A

For a character *chi* of the group G, say, `CentralCharacter` returns the **central character** of *chi*.

The central character of χ is the class function ω_χ defined by $\omega_\chi(g) = |g^G| \cdot \chi(g)/\chi(1)$ for each $g \in G$.

18 ▶ `DeterminantOfCharacter( [`*tbl*`, ]`*chi* `)` A

`DeterminantOfCharacter` returns the **determinant character** of the character *chi*. This is defined to be the character obtained by taking the determinant of representing matrices of any representation affording *chi*; the determinant can be computed using `EigenvaluesChar` (see 70.8.19).

It is also possible to call `Determinant` instead of `DeterminantOfCharacter`.

Note that the determinant character is well-defined for virtual characters.

```
gap> CentralCharacter( TrivialCharacter( S4 ) );
ClassFunction( CharacterTable( S4 ), [ 1, 6, 3, 8, 6 ] )
gap> DeterminantOfCharacter( Irr( S4 )[3] );
Character( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] )
```

19 ▶ `EigenvaluesChar( [`*tbl*`, ]`*chi*`,` *class* `)` O

Let *chi* be a character of the group G, say. For an element $g \in G$ in the *class*-th conjugacy class, of order n, let M be a matrix of a representation affording *chi*.

`EigenvaluesChar(` *tbl*`,` *chi*`,` *class* `)` is the list of length n where at position k the multiplicity of $\mathtt{E}(n)^k = \exp(\frac{2\pi ik}{n})$ as an eigenvalue of M is stored.

We have *chi*`[` *class* `] = List( [ 1 ..` *n* `], k -> E(n)^k ) * EigenvaluesChar(` *tbl*`,` *chi*`,` *class* `)`.

It is also possible to call `Eigenvalues` instead of `EigenvaluesChar`.

```
gap> chi:= Irr( CharacterTable( "A5" ) )[2];
Character( CharacterTable( "A5" ), [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3
 ] )
gap> List( [ 1 .. 5 ], i -> Eigenvalues( chi, i ) );
[ [ 3 ], [ 2, 1 ], [ 1, 1, 1 ], [ 0, 1, 1, 0, 1 ], [ 1, 0, 0, 1, 1 ] ]
```

20 ▶ `Tensored(` *chars1*`,` *chars2* `)` O

Let *chars1* and *chars2* be lists of (values lists of) class functions of the same character table. `Tensored` returns the list of tensor products of all entries in *chars1* with all entries in *chars2*.

```
gap> irra5:= Irr( CharacterTable( "A5" ) );;
gap> chars1:= irra5{ [ 1 .. 3 ] };;  chars2:= irra5{ [ 2, 3 ] };;
gap> Tensored( chars1, chars2 );
[ Character( CharacterTable( "A5" ), [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3
     ] ), Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
  Character( CharacterTable( "A5" ),
    [ 9, 1, 0, -2*E(5)-E(5)^2-E(5)^3-2*E(5)^4, -E(5)-2*E(5)^2-2*E(5)^3-E(5)^4
     ] ), Character( CharacterTable( "A5" ), [ 9, 1, 0, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 9, 1, 0, -1, -1 ] ),
  Character( CharacterTable( "A5" ),
    [ 9, 1, 0, -E(5)-2*E(5)^2-2*E(5)^3-E(5)^4, -2*E(5)-E(5)^2-E(5)^3-2*E(5)^4
     ] ) ]
```

70.9 Restricted and Induced Class Functions

For restricting a class function of a group G to a subgroup H and for inducing a class function of H to G, the **class fusion** from H to G must be known (see 71.2).

If F is the factor group of G by the normal subgroup N then each class function of F can be naturally regarded as a class function of G, with N in its kernel. For a class function of F, the corresponding class function of G is called the **inflated** class function. Restriction and inflation are in principle the same, namely indirection of a class function by the appropriate fusion map, and thus no extra operation is needed for this process. But note that contrary to the case of a subgroup fusion, the factor fusion can in general not be computed from the groups G and F; either one needs the natural homomorphism or the factor fusion to

the character table of F must be stored on the table of G. This explains the different syntax for computing restricted and inflated class functions.

In the following, the meaning of the optional first argument *tbl* is the same as in Section 70.8.

1 ▶ `RestrictedClassFunction( [`*tbl*`, ]`*chi*`, `*H*` )` O
▶ `RestrictedClassFunction( [`*tbl*`, ]`*chi*`, `*hom*` )` O
▶ `RestrictedClassFunction( [`*tbl*`, ]`*chi*`, `*subtbl*` )` O

For a class function *chi* of the group G, say, and either a subgroup H of G or a homomorphism from H to G or the character table *subtbl* of this subgroup, `RestrictedClassFunction` returns the class function of H obtained by restricting *chi* to H.

In the situation that *chi* is a class function of a factor group F of H, the variant where *hom* is a homomorphism can be always used, the calls with argument H or *subtbl* work only if the factor fusion is stored on the character table.

2 ▶ `RestrictedClassFunctions( [`*tbl*`, ]`*chars*`, `*H*` )` O
▶ `RestrictedClassFunctions( [`*tbl*`, ]`*chars*`, `*hom*` )` O
▶ `RestrictedClassFunctions( [`*tbl*`, ]`*chars*`, `*subtbl*` )` O

`RestrictedClassFunctions` is similar to `RestrictedClassFunction` (see 70.9.1), the only difference is that it takes a list *chars* of class functions instead of one class function, and returns the list of restricted class functions.

```
gap> a5:= CharacterTable( "A5" );;  s5:= CharacterTable( "S5" );;
gap> RestrictedClassFunction( Irr( s5 )[2], a5 );
Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] )
gap> RestrictedClassFunctions( Irr( s5 ), a5 );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 6, -2, 0, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ]
gap> hom:= NaturalHomomorphismByNormalSubgroup( S4, der );;
gap> RestrictedClassFunctions( Irr( Image( hom ) ), hom );
[ Character( CharacterTable( S4 ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( S4 ), [ 1, -1, 1, 1, -1 ] ) ]
```

3 ▶ `InducedClassFunction( [`*tbl*`, ]`*chi*`, `*H*` )` O
▶ `InducedClassFunction( [`*tbl*`, ]`*chi*`, `*hom*` )` O
▶ `InducedClassFunction( [`*tbl*`, ]`*chi*`, `*suptbl*` )` O

For a class function *chi* of the group G, say, and either a supergroup H of G or a homomorphism from G to H or the character table *suptbl* of this supergroup, `InducedClassFunction` returns the class function of H obtained by inducing *chi* to H.

4 ▶ `InducedClassFunctions( [`*tbl*`, ]`*chars*`, `*H*` )` O
▶ `InducedClassFunctions( [`*tbl*`, ]`*chars*`, `*hom*` )` O
▶ `InducedClassFunctions( [`*tbl*`, ]`*chars*`, `*suptbl*` )` O

`InducedClassFunctions` is similar to `InducedClassFunction` (see 70.9.3), the only difference is that it takes a list *chars* of class functions instead of one class function, and returns the list of induced class functions.

```
gap> InducedClassFunctions( Irr( a5 ), s5 );
[ Character( CharacterTable( "A5.2" ), [ 2, 2, 2, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, -2, 0, 1, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, -2, 0, 1, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 8, 0, 2, -2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 10, 2, -2, 0, 0, 0, 0 ] ) ]
```

5 ► `InducedCyclic(` *tbl* `)` O
► `InducedCyclic(` *tbl*`, "all" )` O
► `InducedCyclic(` *tbl*`,` *classes* `)` O
► `InducedCyclic(` *tbl*`,` *classes*`, "all" )` O

`InducedCyclic` calculates characters induced up from cyclic subgroups of the ordinary character table *tbl* to *tbl*, and returns the strictly sorted list of the induced characters.

If `"all"` is specified then all irreducible characters of these subgroups are induced, otherwise only the permutation characters are calculated.

If a list *classes* is specified then only those cyclic subgroups generated by these classes are considered, otherwise all classes of *tbl* are considered.

```
gap> InducedCyclic( a5, "all" );
[ Character( CharacterTable( "A5" ), [ 12, 0, 0, 2, 2 ] ),
  Character( CharacterTable( "A5" ), [ 12, 0, 0, E(5)^2+E(5)^3, E(5)+E(5)^4
     ] ), Character( CharacterTable( "A5" ),
    [ 12, 0, 0, E(5)+E(5)^4, E(5)^2+E(5)^3 ] ),
  Character( CharacterTable( "A5" ), [ 20, 0, -1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 20, 0, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 30, -2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 30, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 60, 0, 0, 0, 0 ] ) ]
```

70.10 Reducing Virtual Characters

The following operations are intended for the situation that one is given a list of virtual characters of a character table and is interested in the irreducible characters of this table. The idea is to compute virtual characters of small norm from the given ones, hoping to get eventually virtual characters of norm 1.

1 ► `ReducedClassFunctions( [`*tbl*`, ]`*constituents*`,` *reducibles* `)` O
► `ReducedClassFunctions( [`*tbl*`, ]`*reducibles* `)` O

Let *reducibles* be a list of ordinary virtual characters of the group G, say. If *constituents* is given then it must also be a list of ordinary virtual characters of G, otherwise we have *constituents* equal to *reducibles* in the following.

`ReducedClassFunctions` returns a record with components `remainders` and `irreducibles`, both lists of virtual characters of G. These virtual characters are computed as follows.

Let `rems` be the set of nonzero class functions obtained by subtraction of

$$\sum_{\chi} \frac{[reducibles[i], \chi]}{[\chi, \chi]} \cdot \chi$$

from $reducibles[i]$, where the summation runs over *constituents* and $[\chi, \psi]$ denotes the scalar product of G-class functions. Let `irrs` be the list of irreducible characters in `rems`.

We project `rems` into the orthogonal space of `irrs` and all those irreducibles found this way until no new irreducibles arise. Then the `irreducibles` list is the set of all found irreducible characters, and the `remainders` list is the set of all nonzero remainders.

2 ► `ReducedCharacters( [`*tbl*`, ]`*constituents*`,` *reducibles* `)` O

`ReducedCharacters` is similar to `ReducedClassFunctions`, the only difference is that *constituents* and *reducibles* are assumed to be lists of characters. This means that only those scalar products must be formed where the degree of the character in *constituents* does not exceed the degree of the character in *reducibles*.

```
gap> tbl:= CharacterTable( "A5" );;
gap> chars:= Irr( tbl ){ [ 2 .. 4 ] };;
gap> chars:= Set( Tensored( chars, chars ) );;
gap> red:= ReducedClassFunctions( chars );
rec( remainders := [  ],
  irreducibles := [ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "A5" ),
        [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3 ] ),
      Character( CharacterTable( "A5" ),
        [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ),
      Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
      Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ] )
```

3 ► `IrreducibleDifferences(` *tbl*`,` *reducibles*`,` *reducibles2* `)` F
► `IrreducibleDifferences(` *tbl*`,` *reducibles*`,` *reducibles2*`,` *scprmat* `)` F
► `IrreducibleDifferences(` *tbl*`,` *reducibles*`, "triangle" )` F
► `IrreducibleDifferences(` *tbl*`,` *reducibles*`, "triangle",` *scprmat* `)` F

`IrreducibleDifferences` returns the list of irreducible characters which occur as difference of two elements of *reducibles* (if `"triangle"` is specified) or of an element of *reducibles* and an element of *reducibles2*.

If *scprmat* is not specified then it will be calculated, otherwise we must have *scprmat* `= MatScalarProducts(` *tbl*`,` *reducibles* `)` or *scprmat* `= MatScalarProducts(` *tbl*`,` *reducibles*`,` *reducibles2* `)`, respectively.

```
gap> IrreducibleDifferences( a5, chars, "triangle" );
[ Character( CharacterTable( "A5" ), [ 3, -1, 0, -E(5)-E(5)^4, -E(5)^2-E(5)^3
     ] ), Character( CharacterTable( "A5" ),
    [ 3, -1, 0, -E(5)^2-E(5)^3, -E(5)-E(5)^4 ] ) ]
```

4 ► `LLL(` *tbl*`,` *characters*`[,` *y*`][, "sort"][, "linearcomb"] )` F

`LLL` calls the LLL algorithm (see 25.5.1) in the case of lattices spanned by the virtual characters *characters* of the ordinary character table *tbl* (see 70.8.5). By finding shorter vectors in the lattice spanned by *characters*, i.e., virtual characters of smaller norm, in some cases `LLL` is able to find irreducible characters.

`LLL` returns a record with at least components `irreducibles` (the list of found irreducible characters), `remainders` (a list of reducible virtual characters), and `norms` (the list of norms of the vectors in `remainders`). `irreducibles` together with `remainders` form a basis of the $\mathbb{Z}$-lattice spanned by *characters*.

Note that the vectors in the `remainders` list are in general **not** orthogonal (see 70.10.1) to the irreducible characters in `irreducibles`.

Optional arguments of `LLL` are

y
: controls the sensitivity of the algorithm, see 25.5.1,

`"sort"`
: `LLL` sorts *characters* and the `remainders` component of the result according to the degrees,

`"linearcomb"`
: the returned record contains components `irreddecomp` and `reddecomp`, which are decomposition matrices of `irreducibles` and `remainders`, with respect to *characters*.

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> chars:= [ [ 8, 0, 0, -1, 0 ], [ 6, 0, 2, 0, 2 ],
>     [ 12, 0, -4, 0, 0 ], [ 6, 0, -2, 0, 0 ], [ 24, 0, 0, 0, 0 ],
>     [ 12, 0, 4, 0, 0 ], [ 6, 0, 2, 0, -2 ], [ 12, -2, 0, 0, 0 ],
>     [ 8, 0, 0, 2, 0 ], [ 12, 2, 0, 0, 0 ], [ 1, 1, 1, 1, 1 ] ];;
gap> LLL( s4, chars );
rec(
  irreducibles := [ Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0
        ] ), Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, 1, -1, 0, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ) ],
  remainders := [  ], norms := [  ] )
```

5 ► `Extract(` *tbl*`,` *reducibles*`,` *grammat*`[,` *missing* `] )` F

Let *tbl* be an ordinary character table, *reducibles* a list of characters of *tbl*, and *grammat* the matrix of scalar products of *reducibles* (see 70.8.6). `Extract` tries to find irreducible characters by drawing conclusions out of the scalar products, using combinatorial and backtrack means.

The optional argument *missing* is the maximal number of irreducible characters that occur as constituents of *reducibles*. Specification of *missing* may accelerate `Extract`.

`Extract` returns a record *ext* with components `solution` and `choice`, where the value of `solution` is a list $[M_1, \ldots, M_n]$ of decomposition matrices M_i (up to permutations of rows) with the property that $M_i^{tr} \cdot X$ is equal to the sublist at the positions *ext*`.choice[i]` of *reducibles*, for a matrix X of irreducible characters; the value of `choice` is a list of length n whose entries are lists of indices.

So the j-th column in each matrix M_i corresponds to *reducibles*$[j]$, and each row in M_i corresponds to an irreducible character. `Decreased` (see 70.10.7) can be used to examine the solution for computable irreducibles.

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> red:= [ [ 5, 1, 5, 2, 1 ], [ 2, 0, 2, 2, 0 ], [ 3, -1, 3, 0, -1 ],
>           [ 6, 0, -2, 0, 0 ], [ 4, 0, 0, 1, 2 ] ];;
gap> gram:= MatScalarProducts( s4, red, red );
[ [ 6, 3, 2, 0, 2 ], [ 3, 2, 1, 0, 1 ], [ 2, 1, 2, 0, 0 ], [ 0, 0, 0, 2, 1 ],
  [ 2, 1, 0, 1, 2 ] ]
gap> ext:= Extract( s4, red, gram, 5 );
rec(
  solution := [ [ [ 1, 1, 0, 0, 2 ], [ 1, 0, 1, 0, 1 ], [ 0, 1, 0, 1, 0 ], [
            0, 0, 1, 0, 1 ], [ 0, 0, 0, 1, 0 ] ] ],
  choice := [ [ 2, 5, 3, 4, 1 ] ] )
gap> dec:= Decreased( s4, red, ext.solution[1], ext.choice[1] );;
gap> Display( dec );
rec(
  irreducibles :=
   [ Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
```

```
        Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
        Character( CharacterTable( "Sym(4)" ), [ 3, 1, -1, 0, -1 ] ),
        Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ) ],
    remainders := [  ],
    matrix := [  ] )
```

6 ▶ `OrthogonalEmbeddingsSpecialDimension( `*tbl*`, `*reducibles*`, `*grammat*`, ["positive", ] `*dim*` )` F

`OrthogonalEmbeddingsSpecialDimension` is a variant of `OrthogonalEmbeddings` (see 25.6.1) for the situation that *tbl* is an ordinary character table, *reducibles* is a list of virtual characters of *tbl*, *grammat* is the matrix of scalar products (see 70.8.6), and *dim* is an upper bound for the number of irreducible characters of *tbl* that occur as constituents of *reducibles*; if the vectors in *reducibles* are known to be proper characters then the string `"positive"` may be entered as fourth argument. (See 25.6.1 for information why this may help.)

`OrthogonalEmbeddingsSpecialDimension` first uses `OrthogonalEmbeddings` (see 25.6.1) to compute all orthogonal embeddings of *grammat* into a standard lattice of dimension up to *dim*, and then calls `Decreased` (see 70.10.7) in order to find irreducible characters of *tbl*.

`OrthogonalEmbeddingsSpecialDimension` returns a record with components

`irreducibles`
: a list of found irreducibles, the intersection of all lists of irreducibles found by `Decreased`, for all possible embeddings, and

`remainders`
: a list of remaining reducible virtual characters.

```
gap> s6:= CharacterTable( "S6" );;
gap> red:= InducedCyclic( s6, "all" );;
gap> Add( red, TrivialCharacter( s6 ) );
gap> lll:= LLL( s6, red );;
gap> irred:= lll.irreducibles;
[ Character( CharacterTable( "A6.2_1" ), [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ] )
    , Character( CharacterTable( "A6.2_1" ), [ 9, 1, 0, 0, 1, -1, -3, -3, 1,
      0, 0 ] ), Character( CharacterTable( "A6.2_1" ),
    [ 16, 0, -2, -2, 0, 1, 0, 0, 0, 0, 0 ] ) ]
gap> Set( Flat( MatScalarProducts( s6, irred, lll.remainders ) ) );
[ 0 ]
gap> dim:= NrConjugacyClasses( s6 ) - Length( lll.irreducibles );
8
gap> rem:= lll.remainders;;  Length( rem );
8
gap> gram:= MatScalarProducts( s6, rem, rem );;  RankMat( gram );
8
gap> emb1:= OrthogonalEmbeddings( gram, 8 );
rec( vectors := [ [ -1, 0, 1, 0, 1, 0, 1, 0 ], [ 1, 0, 0, 1, 0, 1, 0, 0 ],
      [ 0, 1, 1, 0, 0, 0, 1, 1 ], [ 0, 1, 1, 0, 0, 0, 1, 0 ],
      [ 0, 1, 1, 0, 0, 0, 0, 0 ], [ 0, 1, 0, 0, 0, 0, 1, 0 ],
      [ 0, -1, 0, 0, 0, 0, 0, 1 ], [ 0, 1, 0, 0, 0, 0, 0, 0 ],
      [ 0, 0, 1, 0, 0, 0, 1, 1 ], [ 0, 0, 1, 0, 0, 0, 0, 1 ],
      [ 0, 0, 1, 0, 0, 0, 0, 0 ], [ 0, 0, 0, -1, 1, 0, 0, 0 ],
      [ 0, 0, 0, 0, 0, 1, 0, 0 ], [ 0, 0, 0, 0, 0, 0, 1, 1 ],
      [ 0, 0, 0, 0, 0, 0, 1, 0 ], [ 0, 0, 0, 0, 0, 0, 0, 1 ] ],
  norms := [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
```

```
solutions := [ [ 1, 2, 3, 7, 11, 12, 13, 15 ],
    [ 1, 2, 4, 8, 10, 12, 13, 14 ], [ 1, 2, 5, 6, 9, 12, 13, 16 ] ] )
```

In the following example we temporarily decrease the line length limit from its default value 80 to 62 in order to get a nicer output format.

```
gap> SizeScreen([ 62, ]);;
gap> emb2:= OrthogonalEmbeddingsSpecialDimension( s6, rem, gram, 8 );
rec(
  irreducibles := [ Character( CharacterTable( "A6.2_1" ),
        [ 5, 1, -1, 2, -1, 0, 1, -3, -1, 1, 0 ] ),
      Character( CharacterTable( "A6.2_1" ),
        [ 5, 1, 2, -1, -1, 0, -3, 1, -1, 0, 1 ] ),
      Character( CharacterTable( "A6.2_1" ),
        [ 10, -2, 1, 1, 0, 0, -2, 2, 0, 1, -1 ] ),
      Character( CharacterTable( "A6.2_1" ),
        [ 10, -2, 1, 1, 0, 0, 2, -2, 0, -1, 1 ] ) ],
  remainders :=
    [ VirtualCharacter( CharacterTable( "A6.2_1" ),
        [ 0, 0, 3, -3, 0, 0, 4, -4, 0, 1, -1 ] ),
      VirtualCharacter( CharacterTable( "A6.2_1" ),
        [ 6, 2, 3, 0, 0, 1, 2, -2, 0, -1, -2 ] ),
      VirtualCharacter( CharacterTable( "A6.2_1" ),
        [ 10, 2, 1, 1, 2, 0, 2, 2, -2, -1, -1 ] ),
      VirtualCharacter( CharacterTable( "A6.2_1" ),
        [ 14, 2, 2, -1, 0, -1, 6, 2, 0, 0, -1 ] ) ] )
gap> SizeScreen([ 80, ]);;
```

7► `Decreased(` *tbl*`,` *chars*`,` *decompmat*`[,` *choice*`] )` F

Let *tbl* be an ordinary character table, *chars* a list of virtual characters of *tbl*, and *decompmat* a decomposition matrix, that is, a matrix M with the property that $M^{tr} \cdot X = chars$ holds, where X is a list of irreducible characters of *tbl*. `Decreased` tries to compute the irreducibles in X or at least some of them.

Usually `Decreased` is applied to the output of `Extract` (see 70.10.5) or `OrthogonalEmbeddings` (see 25.6.1, 70.10.6); in the case of `Extract`, the choice component corresponding to the decomposition matrix must be entered as argument *choice* of `Decreased`.

`Decreased` returns `fail` if it can prove that no list X of irreducible characters corresponding to the arguments exists; otherwise `Decreased` returns a record with components

`irreducibles`
: the list of found irreducible characters,

`remainders`
: the remaining reducible characters, and

`matrix`
: the decomposition matrix of the characters in the `remainders` component.

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> x:= Irr( s4 );;
gap> red:= [ x[1]+x[2], -x[1]-x[3], -x[1]+x[3], -x[2]-x[4] ];;
gap> mat:= MatScalarProducts( s4, red, red );
[ [ 2, -1, -1, -1 ], [ -1, 2, 0, 0 ], [ -1, 0, 2, 0 ], [ -1, 0, 0, 2 ] ]
gap> emb:= OrthogonalEmbeddings( mat );
```

```
rec( vectors := [ [ -1, 1, 1, 0 ], [ -1, 1, 0, 1 ], [ 1, -1, 0, 0 ],
      [ -1, 0, 1, 1 ], [ -1, 0, 1, 0 ], [ -1, 0, 0, 1 ], [ 0, -1, 1, 0 ],
      [ 0, -1, 0, 1 ], [ 0, 1, 0, 0 ], [ 0, 0, -1, 1 ], [ 0, 0, 1, 0 ],
      [ 0, 0, 0, 1 ] ], norms := [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  solutions := [ [ 1, 6, 7, 12 ], [ 2, 5, 8, 11 ], [ 3, 4, 9, 10 ] ] )
gap> dec:= Decreased( s4, red, emb.vectors{ emb.solutions[1] } );;
gap> Display( dec );
rec(
  irreducibles :=
   [ Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, 1, -1, 0, -1 ] ) ],
  remainders := [  ],
  matrix := [  ] )
gap> Decreased( s4, red, emb.vectors{ emb.solutions[2] } );
fail
gap> Decreased( s4, red, emb.vectors{ emb.solutions[3] } );
fail
```

8 ▶ `DnLattice(` *tbl*, *grammat*, *reducibles* `)` F

Let *tbl* be an ordinary character table, and *reducibles* a list of virtual characters of *tbl*.

`DnLattice` searches for sublattices isomorphic to root lattices of type D_n, for $n \geq 4$, in the lattice that is generated by *reducibles*; each vector in *reducibles* must have norm 2, and the matrix of scalar products (see 70.8.6) of *reducibles* must be entered as argument *grammat*.

`DnLattice` is able to find irreducible characters if there is a lattice of type D_n with $n > 4$. In the case $n = 4$, `DnLattice` may fail to determine irreducibles.

`DnLattice` returns a record with components

`irreducibles`
: the list of found irreducible characters,

`remainders`
: the list of remaining reducible virtual characters, and

`gram`
: the Gram matrix of the vectors in `remainders`.

The `remainders` list is transformed in such a way that the `gram` matrix is a block diagonal matrix that exhibits the structure of the lattice generated by the vectors in `remainders`. So `DnLattice` might be useful even if it fails to find irreducible characters.

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> red:= [ [ 2, 0, 2, 2, 0 ], [ 4, 0, 0, 1, 2 ],
>            [ 5, -1, 1, -1, 1 ], [ -1, 1, 3, -1, -1 ] ];;
gap> gram:= MatScalarProducts( s4, red, red );
[ [ 2, 1, 0, 0 ], [ 1, 2, 1, -1 ], [ 0, 1, 2, 0 ], [ 0, -1, 0, 2 ] ]
gap> dn:= DnLattice( s4, gram, red );; Display( dn );
rec(
  gram := [  ],
  remainders := [  ],
  irreducibles :=
   [ Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
```

```
        Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
        Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
        Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ) ] )
```

9 ► `DnLatticeIterative( `*tbl*`, `*reducibles*` )` F

Let *tbl* be an ordinary character table, and *reducibles* either a list of virtual characters of *tbl* or a record with components `remainders` and `norms`, for example a record returned by `LLL` (see 70.10.4).

`DnLatticeIterative` was designed for iterative use of `DnLattice` (see 70.10.8). `DnLatticeIterative` selects the vectors of norm 2 among the given virtual character, calls `DnLattice` for them, reduces the virtual characters with found irreducibles, calls `DnLattice` again for the remaining virtual characters, and so on, until no new irreducibles are found.

`DnLatticeIterative` returns a record with the same components and meaning of components as `LLL` (see 70.10.4).

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> red:= [ [ 2, 0, 2, 2, 0 ], [ 4, 0, 0, 1, 2 ],
>           [ 5, -1, 1, -1, 1 ], [ -1, 1, 3, -1, -1 ] ];;
gap> dn:= DnLatticeIterative( s4, red );; Display( dn );
rec(
  irreducibles :=
   [ Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, -1, 0 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, -1, 1, 1, -1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 1, 1, 1, 1, 1 ] ),
      Character( CharacterTable( "Sym(4)" ), [ 3, -1, -1, 0, 1 ] ) ],
  remainders := [  ],
  norms := [  ] )
```

70.11 Symmetrizations of Class Functions

1 ► `Symmetrizations( [`*tbl*`, ]`*characters*`, `*n*` )` O
► `Symmetrizations( [`*tbl*`, ]`*characters*`, `*Sn*` )` O

`Symmetrizations` returns the list of symmetrizations of the characters *characters* of the ordinary character table *tbl* with the ordinary irreducible characters of the symmetric group of degree n; instead of the integer n, the table of the symmetric group can be entered as *Sn*.

The symmetrization $\chi^{[\lambda]}$ of the character χ of *tbl* with the character λ of the symmetric group S_n of degree n is defined by

$$\chi^{[\lambda]}(g) = \frac{1}{n!} \sum_{\rho \in S_n} \lambda(\rho) \prod_{k=1}^{n} \chi(g^k)^{a_k(\rho)},$$

where $a_k(\rho)$ is the number of cycles of length k in ρ.

For special kinds of symmetrizations, see 70.11.2, 70.11.3, 71.4.5 and 70.11.4, 70.11.5.

Note that the returned list may contain zero class functions, and duplicates are not deleted.

```
gap> tbl:= CharacterTable( "A5" );;
gap> Symmetrizations( Irr( tbl ){ [ 1 .. 3 ] }, 3 );
[ VirtualCharacter( CharacterTable( "A5" ), [ 0, 0, 0, 0, 0 ] ),
  VirtualCharacter( CharacterTable( "A5" ), [ 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 8, 0, -1, -E(5)-E(5)^4, -E(5)^2-E(5)^3
     ] ), Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 8, 0, -1, -E(5)^2-E(5)^3, -E(5)-E(5)^4
     ] ), Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ) ]
```

2 ▶ `SymmetricParts( `*tbl*`, `*characters*`, `*n*` )` F

is the list of symmetrizations of the characters *characters* of the character table *tbl* with the trivial character of the symmetric group of degree *n* (see 70.11.1).

```
gap> SymmetricParts( tbl, Irr( tbl ), 3 );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 20, 0, 2, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 35, 3, 2, 0, 0 ] ) ]
```

3 ▶ `AntiSymmetricParts( `*tbl*`, `*characters*`, `*n*` )` F

is the list of symmetrizations of the characters *characters* of the character table *tbl* with the alternating character of the symmetric group of degree *n* (see 70.11.1).

```
gap> AntiSymmetricParts( tbl, Irr( tbl ), 3 );
[ VirtualCharacter( CharacterTable( "A5" ), [ 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 10, -2, 1, 0, 0 ] ) ]
```

4 ▶ `OrthogonalComponents( `*tbl*`, `*chars*`, `*m*` )` F

If χ is a nonlinear character with indicator $+1$, a splitting of the tensor power χ^m is given by the so-called Murnaghan functions (see [Mur58]). These components in general have fewer irreducible constituents than the symmetrizations with the symmetric group of degree m (see 70.11.1).

`OrthogonalComponents` returns the Murnaghan components of the nonlinear characters of the character table *tbl* in the list *chars* up to the power m, where m is an integer between 2 and 6.

The Murnaghan functions are implemented as in [Fra82].

Note: If *chars* is a list of character objects (see 70.8.1) then also the result consists of class function objects. It is not checked whether all characters in *chars* do really have indicator $+1$; if there are characters with

indicator 0 or −1, the result might contain virtual characters (see also 70.11.5), therefore the entries of the result do in general not know that they are characters.

```
gap> tbl:= CharacterTable( "A8" );;  chi:= Irr( tbl )[2];
Character( CharacterTable( "A8" ), [ 7, -1, 3, 4, 1, -1, 1, 2, 0, -1, 0, 0,
  -1, -1 ] )
gap> OrthogonalComponents( tbl, [ chi ], 3 );
[ ClassFunction( CharacterTable( "A8" ), [ 21, -3, 1, 6, 0, 1, -1, 1, -2, 0,
      0, 0, 1, 1 ] ), ClassFunction( CharacterTable( "A8" ),
    [ 27, 3, 7, 9, 0, -1, 1, 2, 1, 0, -1, -1, -1, -1 ] ),
  ClassFunction( CharacterTable( "A8" ), [ 105, 1, 5, 15, -3, 1, -1, 0, -1,
      1, 0, 0, 0, 0 ] ), ClassFunction( CharacterTable( "A8" ),
    [ 35, 3, -5, 5, 2, -1, -1, 0, 1, 0, 0, 0, 0, 0 ] ),
  ClassFunction( CharacterTable( "A8" ), [ 77, -3, 13, 17, 2, 1, 1, 2, 1, 0,
      0, 0, 2, 2 ] ) ]
```

5 ▶ `SymplecticComponents( `*tbl*`, `*chars*`, `*m*` )` F

If χ is a (nonlinear) character with indicator −1, a splitting of the tensor power χ^m is given in terms of the so-called Murnaghan functions (see [Mur58]). These components in general have fewer irreducible constituents than the symmetrizations with the symmetric group of degree m (see 70.11.1).

`SymplecticComponents` returns the symplectic symmetrizations of the nonlinear characters of the character table *tbl* in the list *chars* up to the power m, where m is an integer between 2 and 5.

Note: If *chars* is a list of character objects (see 70.8.1) then also the result consists of class function objects. It is not checked whether all characters in *chars* do really have indicator −1; if there are characters with indicator 0 or +1, the result might contain virtual characters (see also 70.11.4), therefore the entries of the result do in general not know that they are characters.

```
gap> tbl:= CharacterTable( "U3(3)" );;  chi:= Irr( tbl )[2];
Character( CharacterTable( "U3(3)" ), [ 6, -2, -3, 0, -2, -2, 2, 1, -1, -1,
  0, 0, 1, 1 ] )
gap> SymplecticComponents( tbl, [ chi ], 3 );
[ ClassFunction( CharacterTable( "U3(3)" ), [ 14, -2, 5, -1, 2, 2, 2, 1, 0,
      0, 0, 0, -1, -1 ] ), ClassFunction( CharacterTable( "U3(3)" ),
    [ 21, 5, 3, 0, 1, 1, 1, -1, 0, 0, -1, -1, 1, 1 ] ),
  ClassFunction( CharacterTable( "U3(3)" ), [ 64, 0, -8, -2, 0, 0, 0, 0, 1,
      1, 0, 0, 0, 0 ] ), ClassFunction( CharacterTable( "U3(3)" ),
    [ 14, 6, -4, 2, -2, -2, 2, 0, 0, 0, 0, 0, -2, -2 ] ),
  ClassFunction( CharacterTable( "U3(3)" ), [ 56, -8, 2, 2, 0, 0, 0, -2, 0,
      0, 0, 0, 0, 0 ] ) ]
```

70.12 Molien Series

1 ▶ `MolienSeries( `*psi*` )` F
▶ `MolienSeries( `*psi*`, `*chi*` )` F
▶ `MolienSeries( `*tbl*`, `*psi*` )` F
▶ `MolienSeries( `*tbl*`, `*psi*`, `*chi*` )` F

The **Molien series** of the character ψ, relative to the character χ, is the rational function given by the series

$$M_{\psi,\chi}(z) = \sum_{d=0}^{\infty} [\chi, \psi^{[d]}] z^d$$

where $\psi^{[d]}$ denotes the symmetrization of ψ with the trivial character of the symmetric group S_d (see 70.11.2).

`MolienSeries` returns the Molien series of *psi*, relative to *chi*, where *psi* and *chi* must be characters of the same character table; this table must be entered as *tbl* if *chi* and *psi* are only lists of character values. The default for *chi* is the trivial character of *tbl*.

The return value of `MolienSeries` stores a value for the attribute `MolienSeriesInfo` (see 70.12.2). This admits the computation of coefficients of the series with `ValueMolienSeries` (see 70.12.3). Furthermore, this attribute gives access to numerator and denominator of the Molien series viewed as rational function, where the denominator is a product of polynomials of the form $(1-z^r)^k$; the Molien series is also displayed in this form. Note that such a representation is not unique, one can use `MolienSeriesWithGivenDenominator` (see 70.12.4) to obtain the series with a prescribed denominator.

For more information about Molien series, see [NPP84].

```
gap> t:= CharacterTable( AlternatingGroup( 5 ) );;
gap> psi:= First( Irr( t ), x -> Degree( x ) = 3 );;
gap> mol:= MolienSeries( psi );
( 1-z^2-z^3+z^6+z^7-z^9 ) / ( (1-z^5)*(1-z^3)*(1-z^2)^2 )
```

2 ▶ `MolienSeriesInfo(` *ratfun* `)` A

If the rational function *ratfun* was constructed by `MolienSeries` (see 70.12.1), a representation as quotient of polynomials is known such that the denominator is a product of terms of the form $(1-z^r)^k$. This information is encoded as value of `MolienSeriesInfo`. Additionally, there is a special `PrintObj` method for Molien series based on this.

`MolienSeriesInfo` returns a record that describes the rational function *ratfun* as a Molien series. The components of this record are

`numer`
: numerator of *ratfun* (in general a multiple of the numerator one gets by `NumeratorOfRationalFunction`),

`denom`
: denominator of *ratfun* (in general a multiple of the denominator one gets by `NumeratorOfRationalFunction`),

`ratfun`
: the rational function *ratfun* itself,

`numerstring`
: string corresponding to the polynomial `numer`, expressed in terms of `z`,

`denomstring`
: string corresponding to the polynomial `denom`, expressed in terms of `z`,

`denominfo`
: a list of the form $[[r_1, k_1], \ldots, [r_n, k_n]]$ such that `denom` is $\prod_{i=1}^{n}(1 - z^{r_i})^{k_i}$.

`summands`
: a list of records, each with the components `numer`, `r`, and `k`, describing the summand `numer`$/(1-z^r)^k$,

`size`
: the order of the underlying matrix group,

`degree`
: the degree of the underlying matrix representation.

```
gap> HasMolienSeriesInfo( mol );
true
gap> MolienSeriesInfo( mol );
rec( summands := [ rec( numer := [ -24, -12, -24 ], r := 5, k := 1 ),
      rec( numer := [ -20 ], r := 3, k := 1 ),
      rec( numer := [ -45/4, 75/4, -15/4, -15/4 ], r := 2, k := 2 ),
      rec( numer := [ -1 ], r := 1, k := 3 ),
      rec( numer := [ -15/4 ], r := 1, k := 1 ) ], size := 60, degree := 3,
  numer := -x_1^9+x_1^7+x_1^6-x_1^3-x_1^2+1,
  denom := x_1^12-2*x_1^10-x_1^9+x_1^8+x_1^7+x_1^5+x_1^4-x_1^3-2*x_1^2+1,
  denominfo := [ 5, 1, 3, 1, 2, 2 ], numerstring := "1-z^2-z^3+z^6+z^7-z^9",
  denomstring := "(1-z^5)*(1-z^3)*(1-z^2)^2",
  ratfun := ( 1-z^2-z^3+z^6+z^7-z^9 ) / ( (1-z^5)*(1-z^3)*(1-z^2)^2 ) )
```

3 ▶ `ValueMolienSeries( `*molser*`, `*i*` )` F

is the i-th coefficient of the Molien series *series* computed by `MolienSeries`.

```
gap> List( [ 0 .. 20 ], i -> ValueMolienSeries( mol, i ) );
[ 1, 0, 1, 0, 1, 0, 2, 0, 2, 0, 3, 0, 4, 0, 4, 1, 5, 1, 6, 1, 7 ]
```

4 ▶ `MolienSeriesWithGivenDenominator( `*molser*`, `*list*` )` F

is a Molien series equal to *molser* as rational function, but viewed as quotient with denominator $\prod_{i=1}^{n}(1-z^{r_i})$, where $list = [r_1, r_2, \ldots, r_n]$. If *molser* cannot be represented this way, `fail` is returned.

```
gap> MolienSeriesWithGivenDenominator( mol, [ 2, 6, 10 ] );
( 1+z^15 ) / ( (1-z^10)*(1-z^6)*(1-z^2) )
```

70.13 Possible Permutation Characters

For groups H and G with $H \leq G$, the induced character $(1_G)^H$ is called the **permutation character** of the operation of G on the right cosets of H. If only the character table of G is available and not the group G itself, one can try to get information about possible subgroups of G by inspection of those G-class functions that might be permutation characters, using that such a class function π must have at least the following properties. (For details, see [Isa76], Theorem 5.18.)

(a) π is a character of G,

(b) $\pi(g)$ is a nonnegative integer for all $g \in G$,

(c) $\pi(1)$ divides $|G|$,

(d) $\pi(g^n) \geq \pi(g)$ for $g \in G$ and integers n,

(e) $[\pi, 1_G] = 1$,

(f) the multiplicity of any rational irreducible G-character ψ as a constituent of π is at most $\psi(1)/[\psi, \psi]$,

(g) $\pi(g) = 0$ if the order of g does not divide $|G|/\pi(1)$,

(h) $\pi(1)|N_G(g)|$ divides $\pi(g)|G|$ for all $g \in G$,

(i) $\pi(g) \leq (|G| - \pi(1))/(|g^G||Gal_G(g)|)$ for all nonidentity $g \in G$, where $|Gal_G(g)|$ denotes the number of conjugacy classes of G that contain generators of the group $\langle g \rangle$,

(j) if p is a prime that divides $|G|/\pi(1)$ only once then $s/(p-1)$ divides $|G|/\pi(1)$ and is congruent to 1 modulo p, where s is the number of elements of order p in the (hypothetical) subgroup H for which $\pi = (1_H)^G$ holds. (Note that $s/(p-1)$ equals the number of Sylow p subgroups in H.)

Any G-class function with these properties is called a **possible permutation character** in GAP.

(Condition (d) is checked only for those power maps that are stored in the character table of G; clearly (d) holds for all integers if it holds for all prime divisors of the group order $|G|$.)

GAP provides some algorithms to compute possible permutation characters (see 70.14.1), and also provides functions to check a few more criteria whether a given character can be a transitive permutation character (see 70.14.2).

Some information about the subgroup U can be computed from the permutation character $(1_U)^G$ using `PermCharInfo` (see 70.13.1).

1 ▶ `PermCharInfo( tbl, permchars[, "LaTeX" ] )` F
▶ `PermCharInfo( tbl, permchars[, "HTML" ] )` F

Let *tbl* be the ordinary character table of the group G, and `permchars` either the permutation character $(1_U)^G$, for a subgroup U of G, or a list of such permutation characters. `PermCharInfo` returns a record with the following components.

`contained`:
: a list containing, for each character $\psi = (1_U)^G$ in *permchars*, a list containing at position i the number $\psi[i]|U|/$`SizesCentralizers`$(tbl)[i]$, which equals the number of those elements of U that are contained in class i of *tbl*,

`bound`:
: a list containing, for each character $\psi = (1_U)^G$ in *permchars*, a list containing at position i the number $|U|/\gcd(|U|,$ `SizesCentralizers`$(tbl)[i])$, which divides the class length in U of an element in class i of *tbl*,

`display`:
: a record that can be used as second argument of `Display` to display each permutation character in *permchars* and the corresponding components `contained` and `bound`, for those classes where at least one character of *permchars* is nonzero,

`ATLAS`:
: a list of strings describing the decomposition of the permutation characters in *permchars* into the irreducible characters of *tbl*, given in an ATLAS-like notation. This means that the irreducible constituents are indicated by their degrees followed by lower case letters `a`, `b`, `c`, ..., which indicate the successive irreducible characters of *tbl* of that degree, in the order in which they appear in `Irr(` *tbl* `)`. A sequence of small letters (not necessarily distinct) after a single number indicates a sum of irreducible constituents all of the same degree, an exponent n for the letter *lett* means that *lett* is repeated n times. The default notation for exponentiation is *lett*`^`n, this is also chosen if the optional third argument is the string `"LaTeX"`; if the third argument is the string `"HTML"` then exponentiation is denoted by *lett*`<sup>`n`<sup>`.

```
gap> t:= CharacterTable( "A6" );;
gap> psi:= Sum( Irr( t ){ [ 1, 3, 6 ] } );
Character( CharacterTable( "A6" ), [ 15, 3, 0, 3, 1, 0, 0 ] )
gap> info:= PermCharInfo( t, psi );
rec( contained := [ [ 1, 9, 0, 8, 6, 0, 0 ] ],
  bound := [ [ 1, 3, 8, 8, 6, 24, 24 ] ],
  display := rec( classes := [ 1, 2, 4, 5 ],
      chars := [ [ 15, 3, 0, 3, 1, 0, 0 ], [ 1, 9, 0, 8, 6, 0, 0 ],
          [ 1, 3, 8, 8, 6, 24, 24 ] ], letter := "I" ),
  ATLAS := [ "1a+5b+9a" ] )
gap> Display( t, info.display );
A6
```

```
      2  3  3  .  2
      3  2  .  2  .
      5  1  .  .  .

        1a 2a 3b 4a
     2P 1a 1a 3b 2a
     3P 1a 2a 1a 4a
     5P 1a 2a 3b 4a

I.1    15  3  3  1
I.2     1  9  8  6
I.3     1  3  8  6
gap> j1:= CharacterTable( "J1" );;
gap> psi:= TrivialCharacter( CharacterTable( "7:6" ) )^j1;
Character( CharacterTable( "J1" ), [ 4180, 20, 10, 0, 0, 2, 1, 0, 0, 0, 0, 0,
  0, 0, 0 ] )
gap> PermCharInfo( j1, psi ).ATLAS;
[ "1a+56aabb+76aaab+77aabbcc+120aaabbbccc+133a^{4}bbcc+209a^{5}" ]
```

2 ▶ `PermCharInfoRelative( `*tbl*`, `*tbl2*`, `*permchars*` )` F

Let *tbl* and *tbl2* be the ordinary character tables of two groups H and G, respectively, where H is of index 2 in G, and *permchars* either the permutation character $(1_U)^G$, for a subgroup U of G, or a list of such permutation characters. `PermCharInfoRelative` returns a record with the same components as `PermCharInfo` (see 70.13.1), the only exception is that the entries of the `ATLAS` component are names relative to *tbl*.

More precisely, the i-th entry of the `ATLAS` component is a string describing the decomposition of the i-th entry in *permchars*. The degrees and distinguishing letters of the constituents refer to the irreducibles of *tbl*, as follows. The two irreducible characters of *tbl2* of degree N, say, that extend the irreducible character N`a` of *tbl* are denoted by $N\mathtt{a}^+$ and $N\mathtt{a}^-$. The irreducible character of *tbl2* of degree $2N$, say, whose restriction to *tbl* is the sum of the irreducible characters N`a` and N`b` is denoted as N`ab`. Multiplicities larger than 1 of constituents are denoted by exponents.

(This format is useful mainly for multiplicity free permutation characters.)

```
gap> t:= CharacterTable( "A5" );;
gap> t2:= CharacterTable( "A5.2" );;
gap> List( Irr( t2 ), x -> x[1] );
[ 1, 1, 6, 4, 4, 5, 5 ]
gap> List( Irr( t ), x -> x[1] );
[ 1, 3, 3, 4, 5 ]
gap> permchars:= List( [ [1], [1,2], [1,7], [1,3,4,4,6,6,7] ],
>                      l -> Sum( Irr( t2 ){ l } ) );
[ Character( CharacterTable( "A5.2" ), [ 1, 1, 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5.2" ), [ 2, 2, 2, 2, 0, 0, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 6, 2, 0, 1, 0, 2, 0 ] ),
  Character( CharacterTable( "A5.2" ), [ 30, 2, 0, 0, 6, 0, 0 ] ) ]
gap> info:= PermCharInfoRelative( t, t2, permchars );;
gap> info.ATLAS;
[ "1a^+", "1a^{\\pm}", "1a^++5a^-", "1a^++3ab+4(a^+)^{2}+5a^{\\pm}a^+" ]
```

70.14 Computing Possible Permutation Characters

1 ▶ `PermChars( tbl )` F
▶ `PermChars( tbl, degree )` F
▶ `PermChars( tbl, arec )` F

GAP provides several algorithms to determine possible permutation characters from a given character table. They are described in detail in [BP98]. The algorithm is selected from the choice of the record components of the optional argument record *arec*. The user is encouraged to try different approaches, especially if one choice fails to come to an end.

Regardless of the algorithm used in a specific case, `PermChars` returns a list of **all** possible permutation characters with the properties described by *arec*. There is no guarantee that a character of this list is in fact a permutation character. But an empty list always means there is no permutation character with these properties (e.g., of a certain degree).

In the first form `PermChars` returns the list of all possible permutation characters of the group with character table *tbl*. This list might be rather long for big groups, and its computation might take much time. The algorithm is described in Section 3.2 in [BP98]; it depends on a preprocessing step, where the inequalities arising from the condition $\pi(g) \geq 0$ are transformed into a system of inequalities that guides the search (see 70.14.5). So the following commands compute the list of 39 possible permutation characters of the Mathieu group M_{11}.

```
gap> m11:= CharacterTable( "M11" );;
gap> SetName( m11, "m11" );
gap> perms:= PermChars( m11 );;
gap> Length( perms );
39
```

There are two different search strategies for this algorithm. The default strategy simply constructs all characters with nonnegative values and then tests for each such character whether its degree is a divisor of the order of the group. The other strategy uses the inequalities to predict whether a character of a certain degree can lie in the currently searched part of the search tree. To choose this strategy, use the third form of `PermChars` and set the component `degree` to the range of degrees (which might also be a range containing all divisors of the group order) you want to look for; additionally, the record component `ineq` can take the inequalities computed by `Inequalities` if they are needed more than once.

In the second form `PermChars` returns the list of all possible permutation characters of *tbl* that have degree *degree*. For that purpose, a preprocessing step is performed where essentially the rational character table is inverted in order to determine boundary points for the simplex in which the possible permutation characters of the given degree must lie (see 70.14.3). The algorithm is described at the end of Section 3.2 in [BP98]; Note that inverting big integer matrices needs a lot of time and space. So this preprocessing is restricted to groups with less than 100 classes, say.

```
gap> deg220:= PermChars( m11, 220 );
[ Character( m11, [ 220, 4, 4, 0, 0, 4, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 12, 4, 4, 0, 0, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ) ]
```

In the third form `PermChars` returns the list of all possible permutation characters that have the properties described by the argument record *arec*. One such situation has been mentioned above. If *arec* contains a degree as value of the record component `degree` then `PermChars` will behave exactly as in the second form.

```
gap> deg220 = PermChars( m11, rec( degree:= 220 ) );
true
```

For the meaning of additional components of *arec* besides `degree`, see 70.14.4.

Instead of `degree`, *arec* may have the component `torso` bound to a list that contains some known values of the required characters at the right positions; at least the degree *arec*`.torso[1]` must be an integer. In this case, the algorithm described in Section 3.3 in [BP98] is chosen. The component `chars`, if present, holds a list of all those **rational** irreducible characters of *tbl* that might be constituents of the required characters.

(**Note**: If *arec*`.chars` is bound and does not contain **all** rational irreducible characters of *tbl*, GAP checks whether the scalar products of all class functions in the result list with the omitted rational irreducible characters of *tbl* are nonnegative; so there should be nontrivial reasons for excluding a character that is known to be not a constituent of the desired possible permutation characters.)

```
gap> PermChars( m11, rec( torso:= [ 220 ] ) );
[ Character( m11, [ 220, 4, 4, 0, 0, 4, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ),
  Character( m11, [ 220, 12, 4, 4, 0, 0, 0, 0, 0, 0 ] ) ]
gap> PermChars( m11, rec( torso:= [ 220,,,,, 2 ] ) );
[ Character( m11, [ 220, 20, 4, 0, 0, 2, 0, 0, 0, 0 ] ) ]
```

An additional restriction on the possible permutation characters computed can be forced if *arec* contains, in addition to `torso`, the components `normalsubgroup` and `nonfaithful`, with values a list of class positions of a normal subgroup N of the group G of *tbl* and a possible permutation character π of G, respectively, such that N is contained in the kernel of π. In this case, `PermChars` returns the list of those possible permutation characters ψ of *tbl* coinciding with `torso` wherever its values are bound and having the property that no irreducible constituent of $\psi - \pi$ has N in its kernel. If the component `chars` is bound in *arec* then the above statements apply. An interpretation of the computed characters is the following. Suppose there exists a subgroup V of G such that $\pi = (1_V)^G$; Then $N \leq V$, and if a computed character is of the form $(1_U)^G$ for a subgroup U of G then $V = UN$.

```
gap> s4:= CharacterTable( "Symmetric", 4 );;
gap> nsg:= ClassPositionsOfDerivedSubgroup( s4 );;
gap> pi:= TrivialCharacter( s4 );;
gap> PermChars( s4, rec( torso:= [ 12 ], normalsubgroup:= nsg,
>                        nonfaithful:= pi ) );
[ Character( CharacterTable( "Sym(4)" ), [ 12, 2, 0, 0, 0 ] ) ]
gap> pi:= Sum( Filtered( Irr( s4 ),
>              chi -> IsSubset( ClassPositionsOfKernel( chi ), nsg ) ) );
Character( CharacterTable( "Sym(4)" ), [ 2, 0, 2, 2, 0 ] )
gap> PermChars( s4, rec( torso:= [ 12 ], normalsubgroup:= nsg,
>                        nonfaithful:= pi ) );
[ Character( CharacterTable( "Sym(4)" ), [ 12, 0, 4, 0, 0 ] ) ]
```

The class functions returned by `PermChars` have the properties tested by `TestPerm1`, `TestPerm2`, and `TestPerm3`. So they are possible permutation characters. See 70.14.2 for criteria whether a possible permutation character can in fact be a permutation character.

2 ▶ `TestPerm1(` *tbl*, *char* `)` F
▶ `TestPerm2(` *tbl*, *char* `)` F
▶ `TestPerm3(` *tbl*, *chars* `)` F
▶ `TestPerm4(` *tbl*, *chars* `)` F
▶ `TestPerm5(` *tbl*, *chars*, *modtbl* `)` F

The first three of these functions implement tests of the properties of possible permutation characters listed in Section 70.13, The other two implement test of additional properties. Let *tbl* be the ordinary character table of a group G, say, *char* a rational character of *tbl*, and *chars* a list of rational characters of *tbl*. For applying `TestPerm5`, the knowledge of a p-modular Brauer table *modtbl* of G is required. `TestPerm4` and

`TestPerm5` expect the characters in *chars* to satisfy the conditions checked by `TestPerm1` and `TestPerm2` (see below).

The return values of the functions were chosen parallel to the tests listed in [NPP84].

`TestPerm1` return `1` or `2` if *char* fails because of (T1) or (T2), respectively; this corresponds to the criteria (b) and (d). Note that only those power maps are considered that are stored on *tbl*. If *char* satisfies the conditions, `0` is returned.

`TestPerm2` returns `1` if *char* fails because of the criterion (c), it returns `3`, `4`, or `5` if *char* fails because of (T3), (T4), or (T5), respectively; these tests correspond to (g), a weaker form of (h), and (j). If *char* satisfies the conditions, `0` is returned.

`TestPerm3` returns the list of all those class functions in the list *chars* that satisfy criterion (h); this is a stronger version of (T6).

`TestPerm4` returns the list of all those class functions in the list *chars* that satisfy (T8) and (T9) for each prime divisor p of the order of G; these tests use modular representation theory but do not require the knowledge of decomposition matrices (cf. `TestPerm5` below).

(T8) implements the test of the fact that in the case that p divides $|G|$ and the degree of a transitive permutation character π exactly once, the projective cover of the trivial character is a summand of π. (This test is omitted if the projective cover cannot be identified.)

Given a permutation character π of a group G and a prime integer p, the restriction π_B to a p-block B of G has the following property, which is checked by (T9). For each $g \in G$ such that g^n is a p-element of G, $\pi_B(g^n)$ is a nonnegative integer that satisfies $|\pi_B(g)| \leq \pi_B(g^n) \leq \pi(g^n)$. (This is Corollary A on p. 113 of [Sco73].)

`TestPerm5` requires the p-modular Brauer table *modtbl* of G, for some prime p dividing the order of G, and checks whether those characters in the list *chars* whose degree is divisible by the p-part of the order of G can be decomposed into projective indecomposable characters; `TestPerm5` returns the sublist of all those characters in *chars* that either satisfy this condition or to which the test does not apply.

```
gap> tbl:= CharacterTable( "A5" );;
gap> rat:= RationalizedMat( Irr( tbl ) );
[ Character( CharacterTable( "A5" ), [ 1, 1, 1, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 6, -2, 0, 1, 1 ] ),
  Character( CharacterTable( "A5" ), [ 4, 0, 1, -1, -1 ] ),
  Character( CharacterTable( "A5" ), [ 5, 1, -1, 0, 0 ] ) ]
gap> tup:= Filtered( Tuples( [ 0, 1 ], 4 ), x -> not IsZero( x ) );
[ [ 0, 0, 0, 1 ], [ 0, 0, 1, 0 ], [ 0, 0, 1, 1 ], [ 0, 1, 0, 0 ],
  [ 0, 1, 0, 1 ], [ 0, 1, 1, 0 ], [ 0, 1, 1, 1 ], [ 1, 0, 0, 0 ],
  [ 1, 0, 0, 1 ], [ 1, 0, 1, 0 ], [ 1, 0, 1, 1 ], [ 1, 1, 0, 0 ],
  [ 1, 1, 0, 1 ], [ 1, 1, 1, 0 ], [ 1, 1, 1, 1 ] ]
gap> lincomb:= List( tup, coeff -> coeff * rat );;
gap> List( lincomb, psi -> TestPerm1( tbl, psi ) );
[ 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0 ]
gap> List( lincomb, psi -> TestPerm2( tbl, psi ) );
[ 0, 5, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1 ]
gap> Set( List( TestPerm3( tbl, lincomb ), x -> Position( lincomb, x ) ) );
[ 1, 4, 6, 7, 8, 9, 10, 11, 13 ]
gap> tbl:= CharacterTable( "A7" );
CharacterTable( "A7" )
gap> perms:= PermChars( tbl, rec( degree:= 315 ) );
[ Character( CharacterTable( "A7" ), [ 315, 3, 0, 0, 3, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "A7" ), [ 315, 15, 0, 0, 1, 0, 0, 0, 0 ] ) ]
gap> TestPerm4( tbl, perms );
```

```
[ Character( CharacterTable( "A7" ), [ 315, 15, 0, 0, 1, 0, 0, 0, 0 ] ) ]
gap> perms:= PermChars( tbl, rec( degree:= 15 ) );
[ Character( CharacterTable( "A7" ), [ 15, 3, 0, 3, 1, 0, 0, 1, 1 ] ),
  Character( CharacterTable( "A7" ), [ 15, 3, 3, 0, 1, 0, 3, 1, 1 ] ) ]
gap> TestPerm5( tbl, perms, tbl mod 5 );
[ Character( CharacterTable( "A7" ), [ 15, 3, 0, 3, 1, 0, 0, 1, 1 ] ) ]
```

3 ► `PermBounds(` *tbl*`,` *d* `)` F

Let *tbl* be the ordinary character table of the group G. All G-characters π satisfying $\pi(g) > 0$ and $\pi(1) = d$, for a given degree d, lie in a simplex described by these conditions. `PermBounds` computes the boundary points of this simplex for $d = 0$, from which the boundary points for any other d are easily derived. (Some conditions from the power maps of *tbl* are also involved.) For this purpose, a matrix similar to the rational character table of G has to be inverted. These boundary points are used by `PermChars` (see 70.14.1) to construct all possible permutation characters (see 70.13) of a given degree. `PermChars` either calls `PermBounds` or takes this information from the `bounds` component of its argument record.

4 ► `PermComb(` *tbl*`,` *arec* `)` F

`PermComb` computes possible permutation characters of the character table *tbl* by the improved combinatorial approach described at the end of Section 3.2 in [BP98].

For computing the possible linear combinations **without** prescribing better bounds (i.e., when the computation of bounds shall be suppressed), enter *arec*`:= rec( degree :=` *degree*`, bounds := false )`, where *degree* is the character degree; this is useful if the multiplicities are expected to be small, and if this is forced by high irreducible degrees.

A list of upper bounds on the multiplicities of the rational irreducibles characters can be explicitly prescribed as a `maxmult` component in *arec*.

5 ► `Inequalities(` *tbl*`,` *chars*`[,` *option*`] )` O

Let *tbl* be the ordinary character table of a group G. The condition $\pi(g) \geq 0$ for every possible permutation character π of G places restrictions on the multiplicities a_i of the irreducible constituents χ_i of $\pi = \sum_{i=1}^{r} a_i\chi_i$. For every element $g \in G$, we have $\sum_{i=1}^{r} a_i\chi_i(g) \geq 0$. The power maps provide even stronger conditions.

This system of inequalities is kind of diagonalized, resulting in a system of inequalities restricting a_i in terms of a_j, $j < i$. These inequalities are used to construct characters with nonnegative values (see 70.14.1). `PermChars` either calls `Inequalities` or takes this information from the `ineq` component of its argument record.

The number of inequalities arising in the process of diagonalization may grow very strongly.

There are two ways to organize the projection. The first, which is chosen if no *option* argument is present, is the straight approach which takes the rational irreducible characters in their original order and by this guarantees the character with the smallest degree to be considered first. The other way, which is chosen if the string `"small"` is entered as third argument *option*, tries to keep the number of intermediate inequalities small by eventually changing the order of characters.

```
gap> tbl:= CharacterTable( "M11" );;
gap> PermComb( tbl, rec( degree:= 110 ) );
[ Character( CharacterTable( "M11" ), [ 110, 6, 2, 2, 0, 0, 2, 2, 0, 0 ] ),
  Character( CharacterTable( "M11" ), [ 110, 6, 2, 6, 0, 0, 0, 0, 0, 0 ] ),
  Character( CharacterTable( "M11" ), [ 110, 14, 2, 2, 0, 2, 0, 0, 0, 0 ] ) ]
gap> # Now compute only multiplicity free permutation characters.
gap> bounds:= List( RationalizedMat( Irr( tbl ) ), x -> 1 );;
gap> PermComb( tbl, rec( degree:= 110, maxmult:= bounds ) );
[ Character( CharacterTable( "M11" ), [ 110, 6, 2, 2, 0, 0, 2, 2, 0, 0 ] ) ]
```

70.15 Operations for Brauer Characters

1 ▶ `FrobeniusCharacterValue( value, p )` F

Let *value* be a cyclotomic whose coefficients over the rationals are in the ring $\mathbb{Z}_p$ of p-local numbers, where p is a prime integer. Assume that *value* lies in $\mathbb{Z}_p[\zeta]$ for $\zeta = \texttt{E}(p^n)$.

`FrobeniusCharacterValue` returns the image of *value* under the ring homomorphism from $\mathbb{Z}_p[\zeta]$ to the field with p^n elements that is defined with the help of Conway polynomials (see 57.5.1), more information can be found in Sections 2–5 of [JLPW95].

If *value* is a Brauer character value in characteristic p then the result can be described as the corresponding value of the Frobenius character, that is, as the trace of a representing matrix with the given Brauer character value.

If the result of `FrobeniusCharacterValue` cannot be expressed as an element of a finite field in GAP (see Chapter 57) then `FrobeniusCharacterValue` returns `fail`.

If the Conway polynomial of degree n is required for the computation then it is computed only if `IsCheapConwayPolynomial` returns `true` when it is called with p and n, otherwise `fail` is returned.

2 ▶ `BrauerCharacterValue( mat )` A

For an invertible matrix *mat* over a finite field F, `BrauerCharacterValue` returns the Brauer character value of *mat* if the order of *mat* is coprime to the characteristic of F, and `fail` otherwise.

The **Brauer character value** of a matrix is the sum of complex lifts of its eigenvalues.

```
gap> g:= SL(2,4);;           # 2-dim. irreducible representation of A5
gap> ccl:= ConjugacyClasses( g );;
gap> rep:= List( ccl, Representative );;
gap> List( rep, Order );
[ 1, 2, 5, 5, 3 ]
gap> phi:= List( rep, BrauerCharacterValue );
[ 2, fail, E(5)^2+E(5)^3, E(5)+E(5)^4, -1 ]
gap> List( phi{ [ 1, 3, 4, 5 ] }, x -> FrobeniusCharacterValue( x, 2 ) );
[ 0*Z(2), Z(2^2), Z(2^2)^2, Z(2)^0 ]
gap> List( rep{ [ 1, 3, 4, 5 ] }, TraceMat );
[ 0*Z(2), Z(2^2), Z(2^2)^2, Z(2)^0 ]
```

3 ▶ `SizeOfFieldOfDefinition( val, p )` F

For a cyclotomic or a list of cyclotomics *val*, and a prime integer p, `SizeOfFieldOfDefinition` returns the size of the smallest finite field in characteristic p that contains the p-modular reduction of *val*.

The reduction map is defined as in [JLPW95], that is, the complex (p^d-1)-th root of unity $\texttt{E}(q^d-1)$ is mapped to the residue class of the indeterminate, modulo the ideal spanned by the Conway polynomial (see 57.5.1) of degree d over the field with p elements.

If *val* is a Brauer character then the value returned is the size of the smallest finite field in characteristic p over which the corresponding representation lives.

4 ▶ `RealizableBrauerCharacters( matrix, q )` F

For a list *matrix* of absolutely irreducible Brauer characters in characteristic p, and a power q of p, `RealizableBrauerCharacters` returns a duplicate-free list of sums of Frobenius conjugates of the rows of *matrix*, each irreducible over the field with q elements.

```
gap> irr:= Irr( CharacterTable( "A5" ) mod 2 );
[ Character( BrauerTable( "A5", 2 ), [ 1, 1, 1, 1 ] ),
  Character( BrauerTable( "A5", 2 ), [ 2, -1, E(5)+E(5)^4, E(5)^2+E(5)^3 ] ),
  Character( BrauerTable( "A5", 2 ), [ 2, -1, E(5)^2+E(5)^3, E(5)+E(5)^4 ] ),
  Character( BrauerTable( "A5", 2 ), [ 4, 1, -1, -1 ] ) ]
gap> List( irr, phi -> SizeOfFieldOfDefinition( phi, 2 ) );
[ 2, 4, 4, 2 ]
gap> RealizableBrauerCharacters( irr, 2 );
[ Character( BrauerTable( "A5", 2 ), [ 1, 1, 1, 1 ] ),
  ClassFunction( BrauerTable( "A5", 2 ), [ 4, -2, -1, -1 ] ),
  Character( BrauerTable( "A5", 2 ), [ 4, 1, -1, -1 ] ) ]
```

70.16 Domains Generated by Class Functions

GAP supports groups, vector spaces, and algebras generated by class functions.

71 Maps Concerning Character Tables

Besides the characters, **power maps** (see 71.1) are an important part of a character table. Often their computation is not easy, and if the table has no access to the underlying group then in general they cannot be obtained from the matrix of irreducible characters; so it is useful to store them on the table.

If not only a single table is considered but different tables of a group and a subgroup or of a group and a factor group are used, also **class fusion maps** (see 71.2) must be known to get information about the embedding or simply to induce or restrict characters (see 70.9).

These are examples of functions from conjugacy classes which will be called **maps** in the following. (This should not be confused with the term mapping, see 31.) In GAP, maps are represented by lists. Also each character, each list of element orders, centralizer orders, or class lengths are maps, and for a permutation *perm* of classes, `ListPerm( perm )` is a map.

When maps are constructed without access to a group, often one only knows that the image of a given class is contained in a set of possible images, e.g., that the image of a class under a subgroup fusion is in the set of all classes with the same element order. Using further information, such as centralizer orders, power maps and the restriction of characters, the sets of possible images can be restricted further. In many cases, at the end the images are uniquely determined.

Because of this approach, many functions in this chapter work not only with maps but with **parametrized maps** (or paramaps for short). More about parametrized maps can be found in Section 71.3.

The implementation follows [Bre91], a description of the main ideas together with several examples can be found in [Bre99].

Several examples in this chapter require the GAP Character Table Library to be available. If it is not yet loaded then we load it now.

```
gap> LoadPackage( "ctbllib" );
true
```

71.1 Power Maps

The n-th power map of a character table is represented by a list that stores at position i the position of the class containing the n-th powers of the elements in the i-th class. The n-th power map can be composed from the power maps of the prime divisors p of n, so usually only power maps for primes p are actually stored in the character table.

For an ordinary character table *tbl* with access to its underlying group G, the p-th power map of *tbl* can be computed using the identification of the conjugacy classes of G with the classes of *tbl*. For an ordinary character table without access to a group, in general the p-th power maps (and hence also the element orders) for prime divisors p of the group order are not uniquely determined by the matrix of irreducible characters. So only necessary conditions can be checked in this case, which in general yields only a list of several possibilities for the desired power map. Character tables of the GAP character table library store all p-th power maps for prime divisors p of the group order.

Power maps of Brauer tables can be derived from the power maps of the underlying ordinary tables.

For (computing and) accessing the n-th power map of a character table, `PowerMap` (see 71.1.1) can be used; if the n-th power map cannot be uniquely determined then `PowerMap` returns `fail`.

The list of all possible p-th power maps of a table in the sense that certain necessary conditions are satisfied can be computed with `PossiblePowerMaps` (see 71.1.2). This provides a default strategy, the subroutines are listed in Section 71.4.

1 ▶ `PowerMap(` *tbl*`,` *n*`[,` *class*`] )` O
▶ `PowerMapOp(` *tbl*`,` *n*`[,` *class*`] )` O
▶ `ComputedPowerMaps(` *tbl*` )` A

Called with first argument a character table *tbl* and second argument an integer n, `PowerMap` returns the n-th power map of *tbl*. This is a list containing at position i the position of the class of n-th powers of the elements in the i-th class of *tbl*.

If the additional third argument *class* is present then the position of n-th powers of the *class*-th class is returned.

If the n-th power map is not uniquely determined by *tbl* then `fail` is returned. This can happen only if *tbl* has no access to its underlying group.

The power maps of *tbl* that were computed already by `PowerMap` are stored in *tbl* as value of the attribute `ComputedPowerMaps`, the n-th power map at position n. `PowerMap` checks whether the desired power map is already stored, computes it using the operation `PowerMapOp` if it is not yet known, and stores it. So methods for the computation of power maps can be installed for the operation `PowerMapOp`.

```
gap> tbl:= CharacterTable( "L3(2)" );;
gap> ComputedPowerMaps( tbl );
[ , [ 1, 1, 3, 2, 5, 6 ], [ 1, 2, 1, 4, 6, 5 ],,,, [ 1, 2, 3, 4, 1, 1 ] ]
gap> PowerMap( tbl, 5 );
[ 1, 2, 3, 4, 6, 5 ]
gap> ComputedPowerMaps( tbl );
[ , [ 1, 1, 3, 2, 5, 6 ], [ 1, 2, 1, 4, 6, 5 ],, [ 1, 2, 3, 4, 6, 5 ],,
  [ 1, 2, 3, 4, 1, 1 ] ]
gap> PowerMap( tbl, 137, 2 );
2
```

2 ▶ `PossiblePowerMaps(` *tbl*`,` *p*`[,` *options*`] )` O

For the ordinary character table *tbl* of the group G, say, and a prime integer p, `PossiblePowerMaps` returns the list of all maps that have the following properties of the p-th power map of *tbl*. (Representative orders are used only if the `OrdersClassRepresentatives` value of *tbl* is known, see 69.8.5.)

1. For class i, the centralizer order of the image is a multiple of the i-th centralizer order; if the elements in the i-th class have order coprime to p then the centralizer orders of class i and its image are equal.
2. Let n be the order of elements in class i. If *prime* divides n then the images have order n/p; otherwise the images have order n. These criteria are checked in `InitPowerMap` (see 71.4.1).
3. For each character χ of G and each element g in G, the values $\chi(g^p)$ and `GaloisCyc`$(\chi(g), p)$ are algebraic integers that are congruent modulo p; if p does not divide the element order of g then the two values are equal. This congruence is checked for the characters specified below in the discussion of the *options* argument; For linear characters λ among these characters, the condition $\chi(g)^p = \chi(g^p)$ is checked. The corresponding function is `Congruences` (see 71.4.2).
4. For each character χ of G, the kernel is a normal subgroup N, and $g^p \in N$ for all $g \in N$; moreover, if N has index p in G then $g^p \in N$ for all $g \in G$, and if the index of N in G is coprime to p then $g^p \notin N$ for each $g \notin N$. These conditions are checked for the kernels of all characters χ specified below, the corresponding function is `ConsiderKernels` (see 71.4.3).

5. If p is larger than the order m of an element $g \in G$ then the class of g^p is determined by the power maps for primes dividing the residue of p modulo m. If these power maps are stored in the `ComputedPowerMaps` value (see 71.1.1) of *tbl* then this information is used. This criterion is checked in `ConsiderSmallerPowerMaps` (see 71.4.4).
6. For each character χ of G, the symmetrization ψ defined by $\psi(g) = (\chi(g)^p - \chi(g^p))/p$ is a character. This condition is checked for the kernels of all characters χ specified below, the corresponding function is `PowerMapsAllowedBySymmetrizations` (see 71.4.6).

If *tbl* is a Brauer table, the possibilities are computed from those for the underlying ordinary table.

The optional argument *options* must be a record that may have the following components:

`chars`:
: a list of characters which are used for the check of the criteria 3., 4., and 6.; the default is `Irr(` *tbl* `)`,

`powermap`:
: a parametrized map which is an approximation of the desired map

`decompose`:
: a Boolean; a `true` value indicates that all constituents of the symmetrizations of `chars` computed for criterion 6. lie in `chars`, so the symmetrizations can be decomposed into elements of `chars`; the default value of `decompose` is `true` if `chars` is not bound and `Irr(` *tbl* `)` is known, otherwise `false`,

`quick`:
: a Boolean; if `true` then the subroutines are called with value `true` for the argument *quick*; especially, as soon as only one possibility remains this possibility is returned immediately; the default value is `false`,

`parameters`:
: a record with components `maxamb`, `minamb` and `maxlen` which control the subroutine `PowerMapsAllowedBySymmetrizations`; it only uses characters with current indeterminateness up to `maxamb`, tests decomposability only for characters with current indeterminateness at least `minamb`, and admits a branch according to a character only if there is one with at most `maxlen` possible symmetrizations.

```
gap> tbl:= CharacterTable( "U4(3).4" );;
gap> PossiblePowerMaps( tbl, 2 );
[ [ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4, 5, 6,
      8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18, 18, 20, 20,
      20, 20, 22, 22, 24, 24, 25, 26, 28, 28, 29, 29 ] ]
```

3 ▶ `ElementOrdersPowerMap(` *powermap* `)` F

Let *powermap* be a nonempty list containing at position p, if bound, the p-th power map of a character table or group. `ElementOrdersPowerMap` returns a list of the same length as each entry in *powermap*, with entry at position i equal to the order of elements in class i if this order is uniquely determined by *powermap*, and equal to an unknown (see Chapter 19) otherwise.

```
gap> tbl:= CharacterTable( "U4(3).4" );;
gap> known:= ComputedPowerMaps( tbl );;
gap> Length( known );
7
gap> sub:= ShallowCopy( known );;  Unbind( sub[7] );
gap> ElementOrdersPowerMap( sub );
[ 1, 2, 3, 3, 3, 4, 4, 5, 6, 6, Unknown(1), Unknown(2), 8, 9, 12, 2, 2, 4, 4,
  6, 6, 6, 8, 10, 12, 12, 12, Unknown(3), Unknown(4), 4, 4, 4, 4, 4, 4, 8, 8,
  8, 8, 12, 12, 12, 12, 12, 12, 20, 20, 24, 24, Unknown(5), Unknown(6),
```

```
  Unknown(7), Unknown(8) ]
gap> ord:= ElementOrdersPowerMap( known );
[ 1, 2, 3, 3, 3, 4, 4, 5, 6, 6, 7, 7, 8, 9, 12, 2, 2, 4, 4, 6, 6, 6, 8, 10,
  12, 12, 12, 14, 14, 4, 4, 4, 4, 4, 4, 8, 8, 8, 8, 12, 12, 12, 12, 12, 12,
  20, 20, 24, 24, 28, 28, 28, 28 ]
gap> ord = OrdersClassRepresentatives( tbl );
true
```

4 ▶ `PowerMapByComposition( `*tbl*`, `*n*` )` F

tbl must be a nearly character table, and n a positive integer. If the power maps for all prime divisors of n are stored in the `ComputedPowerMaps` list of *tbl* then `PowerMapByComposition` returns the n-th power map of *tbl*. Otherwise `fail` is returned.

```
gap> tbl:= CharacterTable( "U4(3).4" );;  exp:= Exponent( tbl );
2520
gap> PowerMapByComposition( tbl, exp );
[ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1 ]
gap> Length( ComputedPowerMaps( tbl ) );
7
gap> PowerMapByComposition( tbl, 11 );
fail
gap> PowerMap( tbl, 11 );;
gap> PowerMapByComposition( tbl, 11 );
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
  22, 23, 24, 26, 25, 27, 28, 29, 31, 30, 33, 32, 35, 34, 37, 36, 39, 38, 41,
  40, 43, 42, 45, 44, 47, 46, 49, 48, 51, 50, 53, 52 ]
```

The permutation group of matrix automorphisms (see 69.20.1) acts on the possible power maps returned by `PossiblePowerMaps` (see 71.1.2) by permuting a list via `Permuted` (see 21.20.16) and then mapping the images via `OnPoints` (see 39.2.1). Note that by definition, the group of table automorphisms acts trivially.

5 ▶ `OrbitPowerMaps( `*map*`, `*permgrp*` )` F

returns the orbit of the power map *map* under the action of the permutation group *permgrp* via a combination of `Permuted` (see 21.20.16) and `OnPoints` (see 39.2.1).

6 ▶ `RepresentativesPowerMaps( `*listofmaps*`, `*permgrp*` )` F

returns a list of orbit representatives of the power maps in the list *listofmaps* under the action of the permutation group *permgrp* via a combination of `Permuted` (see 21.20.16) and `OnPoints` (see 39.2.1).

```
gap> tbl:= CharacterTable( "3.McL" );;
gap> grp:= MatrixAutomorphisms( Irr( tbl ) );  Size( grp );
<permutation group with 5 generators>
32
gap> poss:= PossiblePowerMaps( CharacterTable( "3.McL" ), 3 );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17, 4, 4,
      4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 9, 8, 37, 37, 37, 40,
      40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14, 14, 14, 14,
      14, 14, 37, 37, 37, 37, 37, 37 ],
  [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17, 4, 4,
      4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37, 37, 37, 40,
      40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14, 14, 14, 14,
```

```
      14, 14, 37, 37, 37, 37, 37, 37 ] ]
gap> reps:= RepresentativesPowerMaps( poss, grp );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17, 4, 4,
      4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37, 37, 37, 40,
      40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14, 14, 14, 14,
      14, 14, 37, 37, 37, 37, 37, 37 ] ]
gap> orb:= OrbitPowerMaps( reps[1], grp );
[ [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17, 4, 4,
      4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 8, 9, 37, 37, 37, 40,
      40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14, 14, 14, 14,
      14, 14, 37, 37, 37, 37, 37, 37 ],
  [ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17, 4, 4,
      4, 4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, 9, 8, 37, 37, 37, 40,
      40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14, 14, 14, 14,
      14, 14, 37, 37, 37, 37, 37, 37 ] ]
gap> Parametrized( orb );
[ 1, 1, 1, 4, 4, 4, 1, 1, 1, 1, 11, 11, 11, 14, 14, 14, 17, 17, 17, 4, 4, 4,
  4, 4, 4, 29, 29, 29, 26, 26, 26, 32, 32, 32, [ 8, 9 ], [ 8, 9 ], 37, 37,
  37, 40, 40, 40, 43, 43, 43, 11, 11, 11, 52, 52, 52, 49, 49, 49, 14, 14, 14,
  14, 14, 14, 37, 37, 37, 37, 37, 37 ]
```

71.2 Class Fusions between Character Tables

For a group G and a subgroup H of G, the fusion map between the character table of H and the character table of G is represented by a list that stores at position i the position of the i-th class of the table of H in the classes list of the table of G.

For ordinary character tables *tbl1* and *tbl2* of H and G, with access to the groups H and G, the class fusion between *tbl1* and *tbl2* can be computed using the identifications of the conjugacy classes of H with the classes of *tbl1* and the conjugacy classes of G with the classes of *tbl2*. For two ordinary character tables without access to its underlying group, or in the situation that the group stored in *tbl1* is not physically a subgroup of the group stored in *tbl2* but an isomorphic copy, in general the class fusion is not uniquely determined by the information stored on the tables such as irreducible characters and power maps. So only necessary conditions can be checked in this case, which in general yields only a list of several possibilities for the desired class fusion. Character tables of the GAP character table library store various class fusions that are regarded as important, for example fusions from maximal subgroups (see 71.2.2 and "ctbllib:maxes" in the manual for the GAP Character Table Library).

Class fusions between Brauer tables can be derived from the class fusions between the underlying ordinary tables. The class fusion from a Brauer table to the underlying ordinary table is stored when the Brauer table is constructed from the ordinary table, so no method is needed to compute such a fusion.

For (computing and) accessing the class fusion between two character tables, `FusionConjugacyClasses` (see 71.2.1) can be used; if the class fusion cannot be uniquely determined then `FusionConjugacyClasses` returns `fail`.

The list of all possible class fusion between two tables in the sense that certain necessary conditions are satisfied can be computed with `PossibleClassFusions` (see 71.2.6). This provides a default strategy, the subroutines are listed in Section 71.5.

It should be noted that all the following functions except `FusionConjugacyClasses` (see 71.2.1) deal only with the situation of class fusions from subgroups. The computation of **factor fusions** from a character table to the table of a factor group is not dealt with here. Since the ordinary character table of a group G determines the character tables of all factor groups of G, the factor fusion to a given character table of

a factor group of G is determined up to table automorphisms (see 69.8.8) once the class positions of the kernel of the natural epimorphism have been fixed.

1 ▶ `FusionConjugacyClasses(` *tbl1*`,` *tbl2* `)` O
▶ `FusionConjugacyClasses(` *H*`,` *G* `)` O
▶ `FusionConjugacyClasses(` *hom*`[,` *tbl1*`,` *tbl2*`] )` O
▶ `FusionConjugacyClassesOp(` *tbl1*`,` *tbl2* `)` O
▶ `FusionConjugacyClassesOp(` *hom* `)` A

Called with two character tables *tbl1* and *tbl2*, `FusionConjugacyClasses` returns the fusion of conjugacy classes between *tbl1* and *tbl2*. (If one of the tables is a Brauer table, it will delegate this task to the underlying ordinary table.)

Called with two groups H and G where H is a subgroup of G, `FusionConjugacyClasses` returns the fusion of conjugacy classes between H and G. This is done by delegating to the ordinary character tables of H and G, since class fusions are stored only for character tables and not for groups.

Note that the returned class fusion refers to the ordering of conjugacy classes in the character tables if the arguments are character tables and to the ordering of conjugacy classes in the groups if the arguments are groups (see 69.6.2).

Called with a group homomorphism *hom*, `FusionConjugacyClasses` returns the fusion of conjugacy classes between the preimage and the image of *hom*; contrary to the two cases above, also factor fusions can be handled by this variant. If *hom* is the only argument then the class fusion refers to the ordering of conjugacy classes in the groups. If the character tables of preimage and image are given as *tbl1* and *tbl2*, respectively (each table with its group stored), then the fusion refers to the ordering of classes in these tables.

If no class fusion exists or if the class fusion is not uniquely determined, `fail` is returned; this may happen when `FusionConjugacyClasses` is called with two character tables that do not know compatible underlying groups.

Methods for the computation of class fusions can be installed for the operation `FusionConjugacyClassesOp`.

```
gap> s4:= SymmetricGroup( 4 );
Sym( [ 1 .. 4 ] )
gap> tbls4:= CharacterTable( s4 );;
gap> d8:= SylowSubgroup( s4, 2 );
Group([ (1,2), (3,4), (1,3)(2,4) ])
gap> FusionConjugacyClasses( d8, s4 );
[ 1, 2, 3, 3, 5 ]
gap> tbls5:= CharacterTable( "S5" );;
gap> FusionConjugacyClasses( CharacterTable( "A5" ), tbls5 );
[ 1, 2, 3, 4, 4 ]
gap> FusionConjugacyClasses( CharacterTable( "A5" ), CharacterTable( "J1" ) );
fail
gap> PossibleClassFusions( CharacterTable( "A5" ), CharacterTable( "J1" ) );
[ [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 5, 4 ] ]
```

2 ▶ `ComputedClassFusions(` *tbl* `)` A

The class fusions from the character table *tbl* that have been computed already by `FusionConjugacyClasses` (see 71.2.1) or explicitly stored by `StoreFusion` (see 71.2.4) are stored in the `ComputedClassFusions` list of *tbl1*. Each entry of this list is a record with the following components.

`name`
: the `Identifier` value of the character table to which the fusion maps,

`map`
: the list of positions of image classes,

`text` (optional)
: a string giving additional information about the fusion map, for example whether the map is uniquely determined by the character tables,

`specification` (optional, rarely used)
: a value that distinguishes different fusions between the same tables.

Note that stored fusion maps may differ from the maps returned by `GetFusionMap` and the maps entered by `StoreFusion` if the table *destination* has a nonidentity `ClassPermutation` value. So if one fetches a fusion map from a table *tbl1* to a table *tbl2* via access to the data in the `ComputedFusionMaps` list *tbl1* then the stored value must be composed with the `ClassPermutation` value of *tbl2* in order to obtain the correct class fusion. (If one handles fusions only via `GetFusionMap` and `StoreFusion` (see 71.2.3, 71.2.4) then this adjustment is made automatically.)

Fusions are identified via the `Identifier` value of the destination table and not by this table itself because many fusions between character tables in the GAP character table library are stored on library tables, and it is not desirable to load together with a library table also all those character tables that occur as destinations of fusions from this table.

For storing fusions and accessing stored fusions, see also 71.2.3, 71.2.4. For accessing the identifiers of tables that store a fusion into a given character table, see 71.2.5.

3 ▶ `GetFusionMap(` *source*`,` *destination* `)` F
▶ `GetFusionMap(` *source*`,` *destination*`,` *specification* `)` F

For two ordinary character tables *source* and *destination*, `GetFusionMap` checks whether the `ComputedClassFusion` list of *source* (see 71.2.2) contains a record with `name` component `Identifier(` *destination* `)`, and returns returns the `map` component of the first such record. `GetFusionMap(` *source*`,` *destination*`,` *specification* `)` fetches that fusion map for which the record additionally has the `specification` component *specification*.

If both *source* and *destination* are Brauer tables, first the same is done, and if no fusion map was found then `GetFusionMap` looks whether a fusion map between the ordinary tables is stored; if so then the fusion map between *source* and *destination* is stored on *source*, and then returned.

If no appropriate fusion is found, `GetFusionMap` returns `fail`. For the computation of class fusions, see 71.2.1.

4 ▶ `StoreFusion(` *source*`,` *fusion*`,` *destination* `)` F

For two character tables *source* and *destination*, `StoreFusion` stores the fusion *fusion* from *source* to *destination* in the `ComputedClassFusions` list (see 71.2.2) of *source*, and adds the `Identifier` string of *destination* to the `NamesOfFusionSources` list (see `NamesOfFusionSources`) of *destination*.

fusion can either be a fusion map (that is, the list of positions of the image classes) or a record as described in 71.2.2.

If fusions to *destination* are already stored on *source* then another fusion can be stored only if it has a record component `specification` that distinguishes it from the stored fusions. In the case of such an ambiguity, `StoreFusion` raises an error.

```
gap> ComputedClassFusions( CharacterTable( d8 ) );
[ rec( name := "CT1", map := [ 1, 2, 3, 3, 5 ] ) ]
gap> Identifier( tbls4 );
"CT1"
gap> GetFusionMap( CharacterTable( d8 ), tbls4 );
[ 1, 2, 3, 3, 5 ]
gap> GetFusionMap( tbls4, tbls5 );
fail
gap> poss:= PossibleClassFusions( tbls4, tbls5 );
```

```
[ [ 1, 5, 2, 3, 6 ] ]
gap> StoreFusion( tbls4, poss[1], tbls5 );
gap> GetFusionMap( tbls4, tbls5 );
[ 1, 5, 2, 3, 6 ]
```

5 ▶ `NamesOfFusionSources(` *tbl* `)` A

For a character table *tbl*, `NamesOfFusionSources` returns the list of identifiers of all those character tables that are known to have fusions to *tbl* stored. The `NamesOfFusionSources` value is updated whenever a fusion to *tbl* is stored using `StoreFusion` (see 71.2.4).

```
gap> NamesOfFusionSources( tbls5 );
[ "2.A5.2", "Isoclinic(2.A5.2)", "A5", "S3x2", "(A5x3):2", "2^4:s5",
  "2.M22M5", "4.M22M5", "M22.2M4", "2.M12M8", "2.2.2^4+6:S5", "2.2^4+6:S5",
  "4.2^4.S5", "2.HSM10", "3^1+4:2^1+4.s5", "2^(1+4).S5", "(2^2xA5):2",
  "2^10:(2^5:s5)", "3^4:S5", "M24C2B", "gl25", "mo62", "s2wrs5", "s4",
  "twd5a", "w(d5)", "5:4", "CT1" ]
```

6 ▶ `PossibleClassFusions(` *subtbl*, *tbl*`[,` *options*`] )` O

For two ordinary character tables *subtbl* and *tbl* of the groups H and G, say, `PossibleClassFusions` returns the list of all maps that have the following properties of class fusions from *subtbl* to *tbl*.

1. For class i, the centralizer order of the image in G is a multiple of the i-th centralizer order in H, and the element orders in the i-th class and its image are equal. These criteria are checked in `InitFusion` (see 71.5.1).
2. The class fusion commutes with power maps. This is checked using `TestConsistencyMaps` (see 71.3.12).
3. If the permutation character of G corresponding to the action of G on the cosets of H is specified (see the discussion of the *options* argument below) then it prescribes for each class C of G the number of elements of H fusing into C. The corresponding function is `CheckPermChar` (see 71.5.2).
4. The table automorphisms of *tbl* (see 69.8.8) are used in order to compute only orbit representatives. (But note that the list returned by `PossibleClassFusions` contains the full orbits.)
5. For each character χ of G, the restriction to H via the class fusion is a character of H. This condition is checked for all characters specified below, the corresponding function is `FusionsAllowedByRestrictions` (see 71.5.4).
6. The class multiplication coefficients in *subtbl* do not exceed the corresponding coefficients in *tbl*. This is checked in `ConsiderStructureConstants` (see 71.2.9, and see also the comment on the parameter `verify` below).

If *subtbl* and *tbl* are Brauer tables then the possibilities are computed from those for the underlying ordinary tables.

The optional argument *options* must be a record that may have the following components:

`chars`
: a list of characters of *tbl* which are used for the check of 5.; the default is `Irr(` *tbl* `)`,

`subchars`
: a list of characters of *subtbl* which are constituents of the retrictions of `chars`, the default is `Irr(` *subtbl* `)`,

`fusionmap`
: a parametrized map which is an approximation of the desired map,

`decompose`
: a Boolean; a `true` value indicates that all constituents of the restrictions of `chars` computed for criterion 5. lie in `subchars`, so the restrictions can be decomposed into elements of `subchars`; the default value of `decompose` is `true` if `subchars` is not bound and `Irr(` *subtbl* `)` is known, otherwise `false`,

`permchar`
: (a values list of) a permutation character; only those fusions affording that permutation character are computed,

`quick`
: a Boolean; if `true` then the subroutines are called with value `true` for the argument *quick*; especially, as soon as only one possibility remains then this possibility is returned immediately; the default value is `false`,

`verify`
: a Boolean; if `false` then `ConsiderStructureConstants` is called only if more than one orbit of possible class fusions exists, under the action of the groups of table automorphisms; the default value is `false` (because the computation of the structure constants is usually very time comsuming, compared with checking the other criteria),

`parameters`
: a record with components `maxamb`, `minamb` and `maxlen` which control the subroutine `FusionsAllowedByRestrictions`; it only uses characters with current indeterminateness up to `maxamb`, tests decomposability only for characters with current indeterminateness at least `minamb`, and admits a branch according to a character only if there is one with at most `maxlen` possible restrictions.

```
gap> subtbl:= CharacterTable( "U3(3)" );;  tbl:= CharacterTable( "J4" );;
gap> PossibleClassFusions( subtbl, tbl );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 5, 5, 6, 10, 13, 12, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 16, 16, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 13, 12, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 13, 12, 16, 16, 22, 22 ] ]
```

The permutation groups of table automorphisms (see 69.8.8) of the subgroup table *subtbl* and the supergroup table *tbl* act on the possible class fusions returned by `PossibleClassFusions` (see 71.2.6), the former by permuting a list via `Permuted` (see 21.20.16), the latter by mapping the images via `OnPoints` (see 39.2.1).

If the set of possible fusions with certain properties was computed that are not invariant under the full groups of table automorphisms then only a smaller group acts. This may happen for example if a permutation character or if an explicit approximation of the fusion map is prescribed in the call of `PossibleClassFusions`.

7 ▶ `OrbitFusions(` *subtblautomorphisms*`,` *fusionmap*`,` *tblautomorphisms* `)` F

returns the orbit of the class fusion map *fusionmap* under the actions of the permutation groups *subtblautomorphisms* and *tblautomorphisms* of automorphisms of the character table of the subgroup and the supergroup, respectively.

8 ▶ `RepresentativesFusions(` *subtblautomorphisms*`,` *listofmaps*`,` *tblautomorphisms* `)` F
▶ `RepresentativesFusions(` *subtbl*`,` *listofmaps*`,` *tbl* `)` F

returns a list of orbit representatives of class fusion maps in the list *listofmaps* under the action of maximal admissible subgroups of the table automorphisms *subtblautomorphisms* of the subgroup table and *tblautomorphisms* of the supergroup table. Both groups of table automorphisms must be permutation groups.

Instead of the groups of table automorphisms, also the character tables *subtbl* and *tbl* may be entered. In this case, the `AutomorphismsOfTable` values of the tables are used.

```
gap> fus:= GetFusionMap( subtbl, tbl );
[ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ]
gap> orb:= OrbitFusions( AutomorphismsOfTable( subtbl ), fus,
>              AutomorphismsOfTable( tbl ) );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 5, 5, 6, 10, 13, 12, 14, 14, 21, 21 ] ]
gap> rep:= RepresentativesFusions( AutomorphismsOfTable( subtbl ), orb,
>              AutomorphismsOfTable( tbl ) );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ] ]
```

9 ▶ `ConsiderStructureConstants(` *subtbl*`,` *tbl*`,` *fusions*`,` *quick* `)` F

Let *subtbl* and *tbl* be ordinary character tables and *fusions* be a list of possible class fusions from *subtbl* to *tbl*. `ConsiderStructureConstants` returns the list of those maps σ in *fusions* with the property that for all triples (i, j, k) of class positions, `ClassMultiplicationCoefficient`$(subtbl, i, j, k)$ is not bigger than `ClassMultiplicationCoefficient`$(tbl, \sigma[i], \sigma[j], \sigma[k])$; see 69.10.6 for the definition of class multiplication coefficients/structure constants.

The argument *quick* must be a Boolean; if it is `true` then only those triples are checked for which for which at least two entries in *fusions* have different images.

71.3 Parametrized Maps

A **parametrized map** is a list whose i-th entry is either unbound (which means that nothing is known about the image(s) of the i-th class) or the image of the i-th class (i.e., an integer for fusion maps, power maps, element orders etc., and a cyclotomic for characters), or a list of possible images of the i-th class. In this sense, maps are special parametrized maps. We often identify a parametrized map *paramap* with the set of all maps *map* with the property that either *map*`[i] =` *paramap*`[i]` or *map*`[i]` is contained in the list *paramap*`[i]`; we say then that *map* is contained in *paramap*.

This definition implies that parametrized maps cannot be used to describe sets of maps where lists are possible images. An exception are strings which naturally arise as images when class names are considered. So strings and lists of strings are allowed in parametrized maps, and character constants (see Chapter 26) are not allowed in maps.

1 ▶ `CompositionMaps(` *paramap2*`,` *paramap1*`[,` *class*`] )` F

The composition of two parametrized maps *paramap1*, *paramap2* is defined as the parametrized map *comp* that contains all compositions $f_2 \circ f_1$ of elements f_1 of *paramap1* and f_2 of *paramap2*. For example, the composition of a character χ of a group G by a parametrized class fusion map from a subgroup H to G is the parametrized map that contains all restrictions of χ by elements of the parametrized fusion map.

`CompositionMaps(`*paramap2*`,` *paramap1*`)` is a parametrized map with entry `CompositionMaps(`*paramap2*`,` *paramap1*`,` *class*`)` at position *class*. If *paramap1*`[`*class*`]` is an integer then `CompositionMaps(`*paramap2*`,` *paramap1*`,` *class*`)` is equal to *paramap2*`[` *paramap1*`[` *class* `] ]`. Otherwise it is the union of *paramap2*`[`i`]` for i in *paramap1*`[` *class* `]`.

```
gap> map1:= [ 1, [ 2 .. 4 ], [ 4, 5 ], 1 ];;
gap> map2:= [ [ 1, 2 ], 2, 2, 3, 3 ];;
gap> CompositionMaps( map2, map1 );
[ [ 1, 2 ], [ 2, 3 ], 3, [ 1, 2 ] ]
gap> CompositionMaps( map1, map2 );
[ [ 1, 2, 3, 4 ], [ 2, 3, 4 ], [ 2, 3, 4 ], [ 4, 5 ], [ 4, 5 ] ]
```

2 ▶ `InverseMap(` *paramap* `)` F

For a parametrized map *paramap*, `InverseMap` returns a mutable parametrized map whose i-th entry is unbound if i is not in the image of *paramap*, equal to j if i is (in) the image of *paramap*`[`j`]` exactly for j, and equal to the set of all preimages of i under *paramap* otherwise.

We have `CompositionMaps(` *paramap*`, InverseMap(` *paramap* `) )` the identity map.

```
gap> tbl:= CharacterTable( "2.A5" );;  f:= CharacterTable( "A5" );;
gap> fus:= GetFusionMap( tbl, f );
[ 1, 1, 2, 3, 3, 4, 4, 5, 5 ]
gap> inv:= InverseMap( fus );
[ [ 1, 2 ], 3, [ 4, 5 ], [ 6, 7 ], [ 8, 9 ] ]
gap> CompositionMaps( fus, inv );
[ 1, 2, 3, 4, 5 ]
gap> # transfer a power map ‘‘up’’ to the factor group
gap> pow:= PowerMap( tbl, 2 );
[ 1, 1, 2, 4, 4, 8, 8, 6, 6 ]
gap> CompositionMaps( fus, CompositionMaps( pow, inv ) );
[ 1, 1, 3, 5, 4 ]
gap> last = PowerMap( f, 2 );
true
gap> # transfer a power map of the factor group ‘‘down’’ to the group
gap> CompositionMaps( inv, CompositionMaps( PowerMap( f, 2 ), fus ) );
[ [ 1, 2 ], [ 1, 2 ], [ 1, 2 ], [ 4, 5 ], [ 4, 5 ], [ 8, 9 ], [ 8, 9 ],
  [ 6, 7 ], [ 6, 7 ] ]
```

3 ▶ `ProjectionMap(` *fusionmap* `)` F

For a map *fusionmap*, `ProjectionMap` returns a parametrized map whose i-th entry is unbound if i is not in the image of *fusionmap*, and equal to j if j is the smallest position such that i is the image of *fusionmap*`[`j`]`.

We have `CompositionMaps(` *fusionmap*`, ProjectionMap(` *fusionmap* `) )` the identity map, i.e., first projecting and then fusing yields the identity. Note that *fusionmap* must **not** be a parametrized map.

```
gap> ProjectionMap( [ 1, 1, 1, 2, 2, 2, 3, 4, 5, 5, 5, 6, 6, 6 ] );
[ 1, 4, 7, 8, 9, 12 ]
```

4 ▶ `Indirected(` *character*`,` *paramap* `)` O

For a map *character* and a parametrized map *paramap*, `Indirected` returns a parametrized map whose entry at position i is *character*`[` *paramap*`[`i`] ]` if *paramap*`[`i`]` is an integer, and an unknown (see Chapter 19) otherwise.

```
gap> tbl:= CharacterTable( "M12" );;
gap> fus:= [ 1, 3, 4, [ 6, 7 ], 8, 10, [ 11, 12 ], [ 11, 12 ],
>            [ 14, 15 ], [ 14, 15 ] ];;
gap> List( Irr( tbl ){ [ 1 .. 6 ] }, x -> Indirected( x, fus ) );
[ [ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ],
  [ 11, 3, 2, Unknown(9), 1, 0, Unknown(10), Unknown(11), 0, 0 ],
  [ 11, 3, 2, Unknown(12), 1, 0, Unknown(13), Unknown(14), 0, 0 ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, Unknown(15), Unknown(16) ],
  [ 16, 0, -2, 0, 1, 0, 0, 0, Unknown(17), Unknown(18) ],
  [ 45, -3, 0, 1, 0, 0, -1, -1, 1, 1 ] ]
```

5 ▶ `Parametrized( `*list*` )` F

For a list *list* of (parametrized) maps of the same length, `Parametrized` returns the smallest parametrized map containing all elements of *list*.

`Parametrized` is the inverse function to `ContainedMaps` (see 71.3.6).

```
gap> Parametrized( [ [ 1, 2, 3, 4, 5 ], [ 1, 3, 2, 4, 5 ],
>                    [ 1, 2, 3, 4, 6 ] ] );
[ 1, [ 2, 3 ], [ 2, 3 ], 4, [ 5, 6 ] ]
```

6 ▶ `ContainedMaps( `*paramap*` )` F

For a parametrized map *paramap*, `ContainedMaps` returns the set of all maps contained in *paramap*.

`ContainedMaps` is the inverse function to `Parametrized` (see 71.3.5) in the sense that `Parametrized( ContainedMaps( `*paramap*` ) )` is equal to *paramap*.

```
gap> ContainedMaps( [ 1, [ 2, 3 ], [ 2, 3 ], 4, [ 5, 6 ] ] );
[ [ 1, 2, 2, 4, 5 ], [ 1, 2, 2, 4, 6 ], [ 1, 2, 3, 4, 5 ], [ 1, 2, 3, 4, 6 ],
  [ 1, 3, 2, 4, 5 ], [ 1, 3, 2, 4, 6 ], [ 1, 3, 3, 4, 5 ], [ 1, 3, 3, 4, 6 ] ]
```

7 ▶ `UpdateMap( `*character*`, `*paramap*`, `*indirected*` )` F

Let *character* be a map, *paramap* a parametrized map, and *indirected* a parametrized map that is contained in `CompositionMaps( `*character*`, `*paramap*` )`.

Then `UpdateMap` changes *paramap* to the parametrized map containing exactly the maps whose composition with *character* is equal to *indirected*.

If a contradiction is detected then `false` is returned immediately, otherwise `true`.

```
gap> subtbl:= CharacterTable( "S4(4).2" );;  tbl:= CharacterTable( "He" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> fus;
[ 1, 2, 2, [ 2, 3 ], 4, 4, [ 7, 8 ], [ 7, 8 ], 9, 9, 9, [ 10, 11 ],
  [ 10, 11 ], 18, 18, 25, 25, [ 26, 27 ], [ 26, 27 ], 2, [ 6, 7 ], [ 6, 7 ],
  [ 6, 7, 8 ], 10, 10, 17, 17, 18, [ 19, 20 ], [ 19, 20 ] ]
gap> chi:= Irr( tbl )[2];
Character( CharacterTable( "He" ), [ 51, 11, 3, 6, 0, 3, 3, -1, 1, 2, 0,
  3*E(7)+3*E(7)^2+3*E(7)^4, 3*E(7)^3+3*E(7)^5+3*E(7)^6, 2,
  E(7)+E(7)^2+2*E(7)^3+E(7)^4+2*E(7)^5+2*E(7)^6,
  2*E(7)+2*E(7)^2+E(7)^3+2*E(7)^4+E(7)^5+E(7)^6, 1, 1, 0, 0,
  -E(7)-E(7)^2-E(7)^4, -E(7)^3-E(7)^5-E(7)^6, E(7)+E(7)^2+E(7)^4,
  E(7)^3+E(7)^5+E(7)^6, 1, 0, 0, -1, -1, 0, 0, E(7)+E(7)^2+E(7)^4,
  E(7)^3+E(7)^5+E(7)^6 ] )
gap> filt:= Filtered( Irr( subtbl ), x -> x[1] = 50 );
```

```
[ Character( CharacterTable( "S4(4).2" ), [ 50, 10, 10, 2, 5, 5, -2, 2, 0, 0,
      0, 1, 1, 0, 0, 0, 0, -1, -1, 10, 2, 2, 2, 1, 1, 0, 0, 0, -1, -1 ] ),
  Character( CharacterTable( "S4(4).2" ), [ 50, 10, 10, 2, 5, 5, -2, 2, 0, 0,
      0, 1, 1, 0, 0, 0, 0, -1, -1, -10, -2, -2, -2, -1, -1, 0, 0, 0, 1, 1 ] )
 ]
gap> UpdateMap( chi, fus, filt[1] + TrivialCharacter( subtbl ) );
true
gap> fus;
[ 1, 2, 2, 3, 4, 4, 8, 7, 9, 9, 9, 10, 10, 18, 18, 25, 25, [ 26, 27 ],
  [ 26, 27 ], 2, [ 6, 7 ], [ 6, 7 ], [ 6, 7 ], 10, 10, 17, 17, 18,
  [ 19, 20 ], [ 19, 20 ] ]
```

8 ▶ `MeetMaps(` *paramap1*`,` *paramap2* `)` F

For two parametrized maps *paramap1* and *paramap2*, `MeetMaps` changes *paramap1* such that the image of class i is the intersection of *paramap1*`[`i`]` and *paramap2*`[`i`]`.

If this implies that no images remain for a class, the position of such a class is returned. If no such inconsistency occurs, `MeetMaps` returns `true`.

```
gap> map1:= [ [ 1, 2 ], [ 3, 4 ], 5, 6, [ 7, 8, 9 ] ];;
gap> map2:= [ [ 1, 3 ], [ 3, 4 ], [ 5, 6 ], 6, [ 8, 9, 10 ] ];;
gap> MeetMaps( map1, map2 );  map1;
true
[ 1, [ 3, 4 ], 5, 6, [ 8, 9 ] ]
```

9 ▶ `CommutativeDiagram(` *paramap1*`,` *paramap2*`,` *paramap3*`,` *paramap4*`[,` *improvements*`] )` F

Let *paramap1*, *paramap2*, *paramap3*, *paramap4* be parametrized maps covering parametrized maps f_1, f_2, f_3, f_4 with the property that `CompositionMaps`(f_2, f_1) is equal to `CompositionMaps`(f_4, f_3).

`CommutativeDiagram` checks this consistency, and changes the arguments such that all possible images are removed that cannot occur in the parametrized maps f_i.

The return value is `fail` if an inconsistency was found. Otherwise a record with the components `imp1`, `imp2`, `imp3`, `imp4` is returned, each bound to the list of positions where the corresponding parametrized map was changed,

The optional argument *improvements* must be a record with components `imp1`, `imp2`, `imp3`, `imp4`. If such a record is specified then only diagrams are considered where entries of the i-th component occur as preimages of the i-th parametrized map.

When an inconsistency is detected, `CommutativeDiagram` immediately returns `fail`. Otherwise a record is returned that contains four lists `imp1`, ..., `imp4`: `imp`i is the list of classes where *paramap_*i was changed.

```
gap> map1:= [ [ 1, 2, 3 ], [ 1, 3 ] ];;  map2:= [ [ 1, 2 ], 1, [ 1, 3 ] ];;
gap> map3:= [ [ 2, 3 ], 3 ];;  map4:= [ , 1, 2, [ 1, 2 ] ];;
gap> imp:= CommutativeDiagram( map1, map2, map3, map4 );
rec( imp1 := [ 2 ], imp2 := [ 1 ], imp3 := [  ], imp4 := [  ] )
gap> map1;  map2;  map3;  map4;
[ [ 1, 2, 3 ], 1 ]
[ 2, 1, [ 1, 3 ] ]
[ [ 2, 3 ], 3 ]
[ , 1, 2, [ 1, 2 ] ]
gap> imp2:= CommutativeDiagram( map1, map2, map3, map4, imp );
rec( imp1 := [  ], imp2 := [  ], imp3 := [  ], imp4 := [  ] )
```

10 ▶ `CheckFixedPoints(` *inside1*`,` *between*`,` *inside2* `)` F

Let *inside1*, *between*, *inside2* be parametrized maps, where *between* is assumed to map each fixed point of *inside1* (that is, *inside1*`[`i`] =` i) to a fixed point of *inside2* (that is, *between*$[i]$ is either an integer that

is fixed by *inside2* or a list that has nonempty intersection with the union of its images under *inside2*). `CheckFixedPoints` changes *between* and *inside2* by removing all those entries violate this condition.

When an inconsistency is detected, `CheckFixedPoints` immediately returns `fail`. Otherwise the list of positions is returned where changes occurred.

```
gap> subtbl:= CharacterTable( "L4(3).2_2" );;
gap> tbl:= CharacterTable( "O7(3)" );;
gap> fus:= InitFusion( subtbl, tbl );;  fus{ [ 48, 49 ] };
[ [ 54, 55, 56, 57 ], [ 54, 55, 56, 57 ] ]
gap> CheckFixedPoints( ComputedPowerMaps( subtbl )[5], fus,
>        ComputedPowerMaps( tbl )[5] );
[ 48, 49 ]
gap> fus{ [ 48, 49 ] };
[ [ 56, 57 ], [ 56, 57 ] ]
```

11 ▶ `TransferDiagram(` *inside1*`,` *between*`,` *inside2*`[,` *improvements*`] )` F

Let *inside1*, *between*, *inside2* be parametrized maps covering parametrized maps m_1, f, m_2 with the property that `CompositionMaps`(m_2, f) is equal to `CompositionMaps`(f, m_1).

`TransferDiagram` checks this consistency, and changes the arguments such that all possible images are removed that cannot occur in the parametrized maps m_i and f.

So `TransferDiagram` is similar to `CommutativeDiagram` (see 71.3.9), but *between* occurs twice in each diagram checked.

If a record *improvements* with fields `impinside1`, `impbetween` and `impinside2` is specified, only those diagrams with elements of `impinside1` as preimages of *inside1*, elements of `impbetween` as preimages of *between* or elements of `impinside2` as preimages of *inside2* are considered.

When an inconsistency is detected, `TransferDiagram` immediately returns `fail`. Otherwise a record is returned that contains three lists `impinside1`, `impbetween`, and `impinside2` of positions where the arguments were changed.

```
gap> subtbl:= CharacterTable( "2F4(2)" );;  tbl:= CharacterTable( "Ru" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> permchar:= Sum( Irr( tbl ){ [ 1, 5, 6 ] } );;
gap> CheckPermChar( subtbl, tbl, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ], [ 18, 19 ],
  [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> tr:= TransferDiagram( PowerMap( subtbl, 2 ), fus, PowerMap( tbl, 2 ) );
rec( impinside1 := [  ], impbetween := [ 12, 23 ], impinside2 := [  ] )
gap> tr:= TransferDiagram( PowerMap( subtbl, 3 ), fus, PowerMap( tbl, 3 ) );
rec( impinside1 := [  ], impbetween := [ 14, 24, 25 ], impinside2 := [  ] )
gap> tr:= TransferDiagram( PowerMap( subtbl, 3 ), fus, PowerMap( tbl, 3 ),
>             tr );
rec( impinside1 := [  ], impbetween := [  ], impinside2 := [  ] )
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, [ 25, 26 ], [ 25, 26 ],
  5, 5, 6, 8, 14, 13, 19, 19, [ 25, 26 ], [ 25, 26 ], 27, 27 ]
```

12 ▶ `TestConsistencyMaps(` *powermap1*`,` *fusionmap*`,` *powermap2*`[,` *fus_imp*`] )` F

Let *powermap1* and *powermap2* be lists of parametrized maps, and *fusionmap* a parametrized map, such that for each i, the i-th entry in *powermap1*, *fusionmap*, and the i-th entry in *powermap2* (if bound) are valid arguments for `TransferDiagram` (see 71.3.11). So a typical situation for applying `TestConsistencyMaps` is

that *fusionmap* is an approximation of a class fusion, and *powermap1*, *powermap2* are the lists of power maps of the subgroup and the group.

`TestConsistencyMaps` repeatedly applies `TransferDiagram` to these arguments for all i until no more changes occur.

If a list *fus_imp* is specified then only those diagrams with elements of *fus_imp* as preimages of *fusionmap* are considered.

When an inconsistency is detected, `TestConsistencyMaps` immediately returns `false`. Otherwise `true` is returned.

```
gap> subtbl:= CharacterTable( "2F4(2)" );;  tbl:= CharacterTable( "Ru" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> permchar:= Sum( Irr( tbl ){ [ 1, 5, 6 ] } );;
gap> CheckPermChar( subtbl, tbl, fus, permchar );; fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ], [ 18, 19 ],
  [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> TestConsistencyMaps( ComputedPowerMaps( subtbl ), fus,
>        ComputedPowerMaps( tbl ) );
true
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, 15, 16, 18, 20, [ 25, 26 ], [ 25, 26 ],
  5, 5, 6, 8, 14, 13, 19, 19, [ 25, 26 ], [ 25, 26 ], 27, 27 ]
gap> Indeterminateness( fus );
16
```

13 ▶ `Indeterminateness(` *paramap* `)` F

For a parametrized map *paramap*, `Indeterminateness` returns the number of maps contained in *paramap*, that is, the product of lengths of lists in *paramap* denoting lists of several images.

```
gap> Indeterminateness( [ 1, [ 2, 3 ], [ 4, 5 ], [ 6, 7, 8, 9, 10 ], 11 ] );
20
```

14 ▶ `PrintAmbiguity(` *list*, *paramap* `)` F

For each map in the list *list*, `PrintAmbiguity` prints its position in *list*, the indeterminateness (see 71.3.13) of the composition with the parametrized map *paramap*, and the list of positions where a list of images occurs in this composition.

```
gap> paramap:= [ 1, [ 2, 3 ], [ 3, 4 ], [ 2, 3, 4 ], 5 ];;
gap> list:= [ [ 1, 1, 1, 1, 1 ], [ 1, 1, 2, 2, 3 ], [ 1, 2, 3, 4, 5 ] ];;
gap> PrintAmbiguity( list, paramap );
1 1 [  ]
2 4 [ 2, 4 ]
3 12 [ 2, 3, 4 ]
```

15 ▶ `ContainedSpecialVectors(` *tbl*, *chars*, *paracharacter*, *func* `)` F
▶ `IntScalarProducts(` *tbl*, *chars*, *candidate* `)` F
▶ `NonnegIntScalarProducts(` *tbl*, *chars*, *candidate* `)` F
▶ `ContainedPossibleVirtualCharacters(` *tbl*, *chars*, *paracharacter* `)` F
▶ `ContainedPossibleCharacters(` *tbl*, *chars*, *paracharacter* `)` F

Let *tbl* be an ordinary character table, *chars* a list of class functions (or values lists), *paracharacter* a parametrized class function of *tbl*, and *func* a function that expects the three arguments *tbl*, *chars*, and a values list of a class function, and that returns either `true` or `false`.

`ContainedSpecialVectors` returns the list of all those elements *vec* of *paracharacter* that have integral norm, have integral scalar product with the principal character of *tbl*, and that satisfy *func*`( `*tbl*`, `*chars*`, `*vec*` ) = true`,

Two special cases of *func* are the check whether the scalar products in *tbl* between the vector *vec* and all lists in *chars* are integers or nonnegative integers, respectively. These functions are accessible as global variables `IntScalarProducts` and `NonnegIntScalarProducts`, and `ContainedPossibleVirtualCharacters` and `ContainedPossibleCharacters` provide access to these special cases of `ContainedSpecialVectors`.

```
gap> subtbl:= CharacterTable( "HSM12" );;  tbl:= CharacterTable( "HS" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> rest:= CompositionMaps( Irr( tbl )[8], fus );
[ 231, [ -9, 7 ], [ -9, 7 ], [ -9, 7 ], 6, 15, 15, [ -1, 15 ], [ -1, 15 ], 1,
  [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ -2, 0 ], [ 1, 2 ], [ 1, 2 ],
  [ 1, 2 ], 0, 0, 1, 0, 0, 0, 0 ]
gap> irr:= Irr( subtbl );;
gap> # no further condition
gap> cont1:= ContainedSpecialVectors( subtbl, irr, rest,
>                function( tbl, chars, vec ) return true; end );;
gap> Length( cont1 );
24
gap> # require scalar products to be integral
gap> cont2:= ContainedSpecialVectors( subtbl, irr, rest,
>                IntScalarProducts );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,
      0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0,
      0, 0, 0 ],
  [ 231, 7, -9, -9, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,
      0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0,
      0, 0, 0 ] ]
gap> # additionally require scalar products to be nonnegative
gap> cont3:= ContainedSpecialVectors( subtbl, irr, rest,
>                NonnegIntScalarProducts );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,
      0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0,
      0, 0, 0 ] ]
gap> cont2 = ContainedPossibleVirtualCharacters( subtbl, irr, rest );
true
gap> cont3 = ContainedPossibleCharacters( subtbl, irr, rest );
true
```

16 ▶ `CollapsedMat(` *mat*`, `*maps*` )` F

is a record with components

`fusion`
: fusion that collapses those columns of *mat* that are equal in *mat* and also for all maps in the list *maps*,

`mat`
: the image of *mat* under that fusion.

```
gap> mat:= [ [ 1, 1, 1, 1 ], [ 2, -1, 0, 0 ], [ 4, 4, 1, 1 ] ];;
gap> coll:= CollapsedMat( mat, [] );
rec( mat := [ [ 1, 1, 1 ], [ 2, -1, 0 ], [ 4, 4, 1 ] ],
  fusion := [ 1, 2, 3, 3 ] )
gap> List( last.mat, x -> x{ last.fusion } ) = mat;
true
gap> coll:= CollapsedMat( mat, [ [ 1, 1, 1, 2 ] ] );
rec( mat := [ [ 1, 1, 1, 1 ], [ 2, -1, 0, 0 ], [ 4, 4, 1, 1 ] ],
  fusion := [ 1, 2, 3, 4 ] )
```

17 ▶ `ContainedDecomposables( `*constituents*`, `*moduls*`, `*parachar*`, `*func*` )` F

▶ `ContainedCharacters( `*tbl*`, `*constituents*`, `*parachar*` )` F

For these functions, let *constituents* be a list of **rational** class functions, *moduls* a list of positive integers, *parachar* a parametrized rational class function, *func* a function that returns either `true` or `false` when called with (a values list of) a class function, and *tbl* a character table.

`ContainedDecomposables` returns the set of all elements χ of *parachar* that satisfy $func(\chi)$ = `true` and that lie in the $\mathbb{Z}$-lattice spanned by *constituents*, modulo *moduls*. The latter means they lie in the $\mathbb{Z}$-lattice spanned by *constituents* and the set

$$\{moduls[i] \cdot e_i; 1 \leq i \leq n\},$$

where n is the length of *parachar* and e_i is the i-th standard basis vector.

One application of `ContainedDecomposables` is the following. *constituents* is a list of (values lists of) rational characters of an ordinary character table *tbl*, *moduls* is the list of centralizer orders of *tbl* (see 69.8.6), and *func* checks whether a vector in the lattice mentioned above has nonnegative integral scalar product in *tbl* with all entries of *constituents*. This situation is handled by `ContainedCharacters`. Note that the entries of the result list are **not** necessary linear combinations of *constituents*, and they are **not** necessarily characters of *tbl*.

```
gap> subtbl:= CharacterTable( "HSM12" );;  tbl:= CharacterTable( "HS" );;
gap> rat:= RationalizedMat( Irr( subtbl ) );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> rest:= CompositionMaps( Irr( tbl )[8], fus );
[ 231, [ -9, 7 ], [ -9, 7 ], [ -9, 7 ], 6, 15, 15, [ -1, 15 ], [ -1, 15 ], 1,
  [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ 1, 6 ], [ -2, 0 ], [ 1, 2 ], [ 1, 2 ],
  [ 1, 2 ], 0, 0, 1, 0, 0, 0, 0 ]
gap> # compute all vectors in the lattice
gap> ContainedDecomposables( rat, SizesCentralizers( subtbl ), rest,
>        ReturnTrue );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,
      0, 0, 0, 0 ],
  [ 231, 7, -9, -9, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,
      0, 0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0,
      0, 0, 0 ],
  [ 231, 7, -9, 7, 6, 15, 15, 15, 15, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0,
      0, 0, 0 ] ]
gap> # compute only those vectors that are characters
gap> ContainedDecomposables( rat, SizesCentralizers( subtbl ), rest,
>        x -> NonnegIntScalarProducts( subtbl, Irr( subtbl ), x ) );
[ [ 231, 7, -9, -9, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1,
      0, 0, 0, 0 ],
```

```
[ 231, 7, -9, 7, 6, 15, 15, -1, -1, 1, 6, 6, 1, 1, -2, 1, 2, 2, 0, 0, 1, 0,
    0, 0, 0 ] ]
```

71.4 Subroutines for the Construction of Power Maps

1 ▶ `InitPowerMap(` *tbl*`,` *prime* `)` F

For an ordinary character table *tbl* and a prime *prime*, `InitPowerMap` returns a parametrized map that is a first approximation of the *prime*-th powermap of *tbl*, using the conditions 1. and 2. listed in the description of `PossiblePowerMaps` (see 71.1.2).

If there are classes for which no images are possible, according to these criteria, then `fail` is returned.

```
gap> t:= CharacterTable( "U4(3).4" );;
gap> pow:= InitPowerMap( t, 2 );
[ 1, 1, 3, 4, 5, [ 2, 16 ], [ 2, 16, 17 ], 8, 3, [ 3, 4 ], [ 11, 12 ],
  [ 11, 12 ], [ 6, 7, 18, 19, 30, 31, 32, 33 ], 14, [ 9, 20 ], 1, 1, 2, 2, 3,
  [ 3, 4, 5 ], [ 3, 4, 5 ], [ 6, 7, 18, 19, 30, 31, 32, 33 ], 8, 9, 9,
  [ 9, 10, 20, 21, 22 ], [ 11, 12 ], [ 11, 12 ], 16, 16, [ 2, 16 ],
  [ 2, 16 ], 17, 17, [ 6, 18, 30, 31, 32, 33 ], [ 6, 18, 30, 31, 32, 33 ],
  [ 6, 7, 18, 19, 30, 31, 32, 33 ], [ 6, 7, 18, 19, 30, 31, 32, 33 ], 20, 20,
  [ 9, 20 ], [ 9, 20 ], [ 9, 10, 20, 21, 22 ], [ 9, 10, 20, 21, 22 ], 24, 24,
  [ 15, 25, 26, 40, 41, 42, 43 ], [ 15, 25, 26, 40, 41, 42, 43 ], [ 28, 29 ],
  [ 28, 29 ], [ 28, 29 ], [ 28, 29 ] ]
```

In the argument lists of the functions `Congruences`, `ConsiderKernels`, and `ConsiderSmallerPowerMaps`, *tbl* is an ordinary character table, *chars* a list of (values lists of) characters of *tbl*, *prime* a prime integer, *approxmap* a parametrized map that is an approximation for the *prime*-th power map of *tbl* (e.g., a list returned by `InitPowerMap`, see 71.4.1), and *quick* a Boolean.

The *quick* value `true` means that only those classes are considered for which *approxmap* lists more than one possible image.

2 ▶ `Congruences(` *tbl*`,` *chars*`,` *approxmap*`,` *prime*`,` *quick* `)` F

`Congruences` replaces the entries of *approxmap* by improved values, according to condition 3. listed in the description of `PossiblePowerMaps` (see 71.1.2).

For each class for which no images are possible according to the tests, the new value of *approxmap* is an empty list. `Congruences` returns `true` if no such inconsistencies occur, and `false` otherwise.

```
gap> Congruences( t, Irr( t ), pow, 2, false );  pow;
true
[ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, [ 6, 7 ], 14, 9, 1, 1, 2, 2, 3, 4, 5,
  [ 6, 7 ], 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, [ 18, 19 ],
  [ 18, 19 ], 20, 20, 20, 20, 22, 22, 24, 24, [ 25, 26 ], [ 25, 26 ], 28, 28,
  29, 29 ]
```

3 ▶ `ConsiderKernels(` *tbl*`,` *chars*`,` *approxmap*`,` *prime*`,` *quick* `)` F

`ConsiderKernels` replaces the entries of *approxmap* by improved values, according to condition 4. listed in the description of `PossiblePowerMaps` (see 71.1.2).

`Congruences` returns `true` if the orders of the kernels of all characters in *chars* divide the order of the group of *tbl*, and `false` otherwise.

```
gap> t:= CharacterTable( "A7.2" );;  init:= InitPowerMap( t, 2 );
[ 1, 1, 3, 4, [ 2, 9, 10 ], 6, 3, 8, 1, 1, [ 2, 9, 10 ], 3, [ 3, 4 ], 6,
  [ 7, 12 ] ]
gap> ConsiderKernels( t, Irr( t ), init, 2, false );
true
gap> init;
[ 1, 1, 3, 4, 2, 6, 3, 8, 1, 1, 2, 3, [ 3, 4 ], 6, 7 ]
```

4 ► `ConsiderSmallerPowerMaps( tbl, approxmap, prime, quick )` F

`ConsiderSmallerPowerMaps` replaces the entries of *approxmap* by improved values, according to condition 5. listed in the description of `PossiblePowerMaps` (see 71.1.2).

`ConsiderSmallerPowerMaps` returns `true` if each class admits at least one image after the checks, otherwise `false` is returned. If no element orders of *tbl* are stored (see 69.8.5) then `true` is returned without any tests.

```
gap> t:= CharacterTable( "3.A6" );;  init:= InitPowerMap( t, 5 );
[ 1, [ 2, 3 ], [ 2, 3 ], 4, [ 5, 6 ], [ 5, 6 ], [ 7, 8 ], [ 7, 8 ], 9,
  [ 10, 11 ], [ 10, 11 ], 1, [ 2, 3 ], [ 2, 3 ], 1, [ 2, 3 ], [ 2, 3 ] ]
gap> Indeterminateness( init );
4096
gap> ConsiderSmallerPowerMaps( t, init, 5, false );
true
gap> Indeterminateness( init );
256
```

5 ► `MinusCharacter( character, prime_powermap, prime )` F

Let *character* be (the list of values of) a class function χ, *prime* a prime integer p, and *prime_powermap* a parametrized map that is an approximation of the p-th power map for the character table of χ. `MinusCharacter` returns the parametrized map of values of χ^{p-}, which is defined by $\chi^{p-}(g) = (\chi(g)^p - \chi(g^p))/p$.

```
gap> tbl:= CharacterTable( "S7" );;  pow:= InitPowerMap( tbl, 2 );;
gap> pow;
[ 1, 1, 3, 4, [ 2, 9, 10 ], 6, 3, 8, 1, 1, [ 2, 9, 10 ], 3, [ 3, 4 ], 6,
  [ 7, 12 ] ]
gap> chars:= Irr( tbl ){ [ 2 .. 5 ] };;
gap> List( chars, x -> MinusCharacter( x, pow, 2 ) );
[ [ 0, 0, 0, 0, [ 0, 1 ], 0, 0, 0, 0, 0, [ 0, 1 ], 0, 0, 0, [ 0, 1 ] ],
  [ 15, -1, 3, 0, [ -2, -1, 0 ], 0, -1, 1, 5, -3, [ 0, 1, 2 ], -1, 0, 0,
      [ 0, 1 ] ],
  [ 15, -1, 3, 0, [ -1, 0, 2 ], 0, -1, 1, 5, -3, [ 1, 2, 4 ], -1, 0, 0, 1 ],
  [ 190, -2, 1, 1, [ 0, 2 ], 0, 1, 1, -10, -10, [ 0, 2 ], -1, -1, 0,
      [ -1, 0 ] ] ]
```

6 ► `PowerMapsAllowedBySymmetrizations( tbl, subchars, chars, approxmap, prime, parameters )` F

Let *tbl* be an ordinary character table, *prime* a prime integer, *approxmap* a parametrized map that is an approximation of the *prime*-th power map of *tbl* (e.g., a list returned by `InitPowerMap`, see 71.4.1), *chars* and *subchars* two lists of (values lists of) characters of *tbl*, and *parameters* a record with components `maxlen`, `minamb`, `maxamb` (three integers), `quick` (a Boolean), and `contained` (a function). Usual values of `contained` are `ContainedCharacters` or `ContainedPossibleCharacters`.

`PowerMapsAllowedBySymmetrizations` replaces the entries of *approxmap* by improved values, according to condition 6. listed in the description of `PossiblePowerMaps` (see 71.1.2).

More precisely, the strategy used is as follows.

First, for each $\chi \in$ *chars*, let `minus:= MinusCharacter(`χ`,` *approxmap*`,` *prime*`)`.

- If `Indeterminateness(minus)` $= 1$ and *parameters*`.quick = false` then the scalar products of `minus` with *subchars* are checked; if not all scalar products are nonnegative integers then an empty list is returned, otherwise χ is deleted from the list of characters to inspect.
- Otherwise if `Indeterminateness( minus )` is smaller than *parameters*`.minamb` then χ is deleted from the list of characters.
- If *parameters*`.minamb` $\leq$ `Indeterminateness( minus )` $\leq$ *parameters*`.maxamb` then construct the list of contained class functions `poss:=` *parameters*`.contained(`*tbl*`,` *subchars*`, minus)` and `Parametrized( poss )`, and improve the approximation of the power map using `UpdateMap`.

If this yields no further immediate improvements then we branch. If there is a character from *chars* left with less or equal *parameters*`.maxlen` possible symmetrizations, compute the union of power maps allowed by these possibilities. Otherwise we choose a class C such that the possible symmetrizations of a character in *chars* differ at C, and compute recursively the union of all allowed power maps with image at C fixed in the set given by the current approximation of the power map.

```
gap> tbl:= CharacterTable( "U4(3).4" );;
gap> pow:= InitPowerMap( tbl, 2 );;
gap> Congruences( tbl, Irr( tbl ), pow, 2 );;  pow;
[ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, [ 6, 7 ], 14, 9, 1, 1, 2, 2, 3, 4, 5,
  [ 6, 7 ], 8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, [ 18, 19 ],
  [ 18, 19 ], 20, 20, 20, 20, 22, 22, 24, 24, [ 25, 26 ], [ 25, 26 ], 28, 28,
  29, 29 ]
gap> PowerMapsAllowedBySymmetrizations( tbl, Irr( tbl ), Irr( tbl ),
>       pow, 2, rec( maxlen:= 10, contained:= ContainedPossibleCharacters,
>       minamb:= 2, maxamb:= infinity, quick:= false ) );
[ [ 1, 1, 3, 4, 5, 2, 2, 8, 3, 4, 11, 12, 6, 14, 9, 1, 1, 2, 2, 3, 4, 5, 6,
      8, 9, 9, 10, 11, 12, 16, 16, 16, 16, 17, 17, 18, 18, 18, 18, 20, 20,
      20, 20, 22, 22, 24, 24, 25, 26, 28, 28, 29, 29 ] ]
```

71.5 Subroutines for the Construction of Class Fusions

1 ▶ `InitFusion(` *subtbl*`,` *tbl* `)` F

For two ordinary character tables *subtbl* and *tbl*, `InitFusion` returns a parametrized map that is a first approximation of the class fusion from *subtbl* to *tbl*, using condition 1. listed in the description of `PossibleClassFusions` (see 71.2.6).

If there are classes for which no images are possible, according to this criterion, then `fail` is returned.

```
gap> subtbl:= CharacterTable( "2F4(2)" );;  tbl:= CharacterTable( "Ru" );;
gap> fus:= InitFusion( subtbl, tbl );
[ 1, 2, 2, 4, [ 5, 6 ], [ 5, 6, 7, 8 ], [ 5, 6, 7, 8 ], [ 9, 10 ], 11, 14,
  14, [ 13, 14, 15 ], [ 16, 17 ], [ 18, 19 ], 20, [ 25, 26 ], [ 25, 26 ],
  [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], [ 5, 6, 7, 8 ], [ 13, 14, 15 ],
  [ 13, 14, 15 ], [ 18, 19 ], [ 18, 19 ], [ 25, 26 ], [ 25, 26 ],
  [ 27, 28, 29 ], [ 27, 28, 29 ] ]
```

2 ▶ `CheckPermChar(` *subtbl*`,` *tbl*`,` *approxmap*`,` *permchar* `)` F

`CheckPermChar` replaces the entries of the parametrized map *approxmap* by improved values, according to condition 3. listed in the description of `PossibleClassFusions` (see 71.2.6).

`CheckPermChar` returns `true` if no inconsistency occurred, and `false` otherwise.

```
gap> permchar:= Sum( Irr( tbl ){ [ 1, 5, 6 ] } );;
gap> CheckPermChar( subtbl, tbl, fus, permchar ); fus;
true
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20,
  [ 25, 26 ], [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ], [ 18, 19 ],
  [ 25, 26 ], [ 25, 26 ], 27, 27 ]
```

3 ▶ `ConsiderTableAutomorphisms( `*approxmap*`, `*grp*` )` F

`ConsiderTableAutomorphisms` replaces the entries of the parametrized map *approxmap* by improved values, according to condition 4. listed in the description of `PossibleClassFusions` (see 71.2.6).

Afterwards exactly one representative of fusion maps (contained in *approxmap*) in each orbit under the action of the permutation group *grp* is contained in the modified parametrized map.

`ConsiderTableAutomorphisms` returns the list of positions where *approxmap* was changed.

```
gap> ConsiderTableAutomorphisms( fus, AutomorphismsOfTable( tbl ) );
[ 16 ]
gap> fus;
[ 1, 2, 2, 4, 5, 7, 8, 9, 11, 14, 14, [ 13, 15 ], 16, [ 18, 19 ], 20, 25,
  [ 25, 26 ], 5, 5, 6, 8, 14, [ 13, 15 ], [ 18, 19 ], [ 18, 19 ], [ 25, 26 ],
  [ 25, 26 ], 27, 27 ]
```

4 ▶ `FusionsAllowedByRestrictions( `*subtbl*`, `*tbl*`, `*subchars*`, `*chars*`, `*approxmap*`, `*parameters*` )` F

Let *subtbl* and *tbl* be ordinary character tables, *subchars* and *chars* two lists of (values lists of) characters of *subtbl* and *tbl*, respectively, *approxmap* a parametrized map that is an approximation of the class fusion of *subtbl* in *tbl*, and *parameters* a record with components `maxlen`, `minamb`, `maxamb` (three integers), *quick* (a Boolean), and `contained` (a function). Usual values of `contained` are `ContainedCharacters` or `ContainedPossibleCharacters`.

`FusionsAllowedByResrictions` replaces the entries of *approxmap* by improved values, according to condition 5. listed in the description of `PossibleClassFusions` (see 71.2.6).

More precisely, the strategy used is as follows.

First, for each $\chi \in$ *chars*, let `restricted:= CompositionMaps(` χ`, `*approxmap*` )`.

- If `Indeterminateness(restricted)` $= 1$ and *parameters*`.quick = false` then the scalar products of `restricted` with *subchars* are checked; if not all scalar products are nonnegative integers then an empty list is returned, otherwise χ is deleted from the list of characters to inspect.
- Otherwise if `Indeterminateness( minus )` is smaller than *parameters*`.minamb` then χ is deleted from the list of characters.
- If *parameters*`.minamb` $\leq$ `Indeterminateness( restricted )` $\leq$ *parameters*`.maxamb` then construct `poss:=` *parameters*`.contained(` *subtbl*`, `*subchars*`, restricted )` and `Parametrized( poss )`, and improve the approximation of the fusion map using `UpdateMap`.

If this yields no further immediate improvements then we branch. If there is a character from *chars* left with less or equal *parameters*`.maxlen` possible restrictions, compute the union of fusion maps allowed by these possibilities. Otherwise we choose a class C such that the possible restrictions of a character in *chars* differ at C, and compute recursively the union of all allowed fusion maps with image at C fixed in the set given by the current approximation of the fusion map.

```
gap> subtbl:= CharacterTable( "U3(3)" );;  tbl:= CharacterTable( "J4" );;
gap> fus:= InitFusion( subtbl, tbl );;
gap> TestConsistencyMaps( ComputedPowerMaps( subtbl ), fus,
>        ComputedPowerMaps( tbl ) );
true
gap> fus;
[ 1, 2, 4, 4, [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], 10, [ 12, 13 ], [ 12, 13 ],
  [ 14, 15, 16 ], [ 14, 15, 16 ], [ 21, 22 ], [ 21, 22 ] ]
gap> ConsiderTableAutomorphisms( fus, AutomorphismsOfTable( tbl ) );
[ 9 ]
gap> fus;
[ 1, 2, 4, 4, [ 5, 6 ], [ 5, 6 ], [ 5, 6 ], 10, 12, [ 12, 13 ],
  [ 14, 15, 16 ], [ 14, 15, 16 ], [ 21, 22 ], [ 21, 22 ] ]
gap> FusionsAllowedByRestrictions( subtbl, tbl, Irr( subtbl ),
>        Irr( tbl ), fus, rec( maxlen:= 10,
>        contained:= ContainedPossibleCharacters, minamb:= 2,
>        maxamb:= infinity, quick:= false ) );
[ [ 1, 2, 4, 4, 5, 5, 6, 10, 12, 13, 14, 14, 21, 21 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 15, 15, 22, 22 ],
  [ 1, 2, 4, 4, 6, 6, 6, 10, 12, 13, 16, 16, 22, 22 ] ]
```

72 Monomiality Questions

This chapter describes functions dealing with the monomiality of finite (solvable) groups and their characters.

All these functions assume **characters** to be class function objects as described in Chapter 70, lists of character **values** are not allowed.

The usual **property tests** of GAP that return either `true` or `false` are not sufficient for us. When we ask whether a group character χ has a certain property, such as quasiprimitivity, we usually want more information than just yes or no. Often we are interested in the reason **why** a group character χ was proved to have a certain property, e.g., whether monomiality of χ was proved by the observation that the underlying group is nilpotent, or whether it was necessary to construct a linear character of a subgroup from which χ can be induced. In the latter case we also may be interested in this linear character. Therefore we need test functions that return a record containing such useful information. For example, the record returned by the function `TestQuasiPrimitive` (see 72.2.3) contains the component `isQuasiPrimitive` (which is the known boolean property flag), and additionally the component `comment`, a string telling the reason for the value of the `isQuasiPrimitive` component, and in the case that the argument χ was **not** quasiprimitive also the component `character`, which is an irreducible constituent of a nonhomogeneous restriction of χ to a normal subgroup. Besides these test functions there are also the known properties, e.g., the property `IsQuasiPrimitive` which will call the attribute `TestQuasiPrimitive`, and return the value of the `isQuasiPrimitive` component of the result.

A few words about how to use the monomiality functions seem to be necessary. Monomiality questions usually involve computations in many subgroups and factor groups of a given group, and for these groups often expensive calculations such as that of the character table are necessary. So one should be careful not to construct the same group over and over again, instead the same group object should be reused, such that its character table need to be computed only once. For example, suppose you want to restrict a character to a normal subgroup N that was constructed as a normal closure of some group elements, and suppose that you have already computed with normal subgroups (by calls to `NormalSubgroups` or `MaximalNormalSubgroups`) and their character tables. Then you should look in the lists of known normal subgroups whether N is contained, and if so you can use the known character table. A mechanism that supports this for normal subgroups is described in 69.21.

Also the following hint may be useful in this context. If you know that sooner or later you will compute the character table of a group G then it may be advisable to compute it as soon as possible. For example, if you need the normal subgroups of G then they can be computed more efficiently if the character table of G is known, and they can be stored compatibly to the contained G-conjugacy classes. This correspondence of classes list and normal subgroup can be used very often.

Several **examples** in this chapter use the symmetric group S_4 and the special linear group $SL(2,3)$. For running the examples, you must first define the groups, for example as follows.

```
gap> S4:= SymmetricGroup( 4 );;  SetName( S4, "S4" );
gap> Sl23:= SL( 2, 3 );;
```

1 ► `InfoMonomial` V

Most of the functions described in this chapter print some (hopefully useful) **information** if the info level of the info class `InfoMonomial` is at least 1 (see 7.4 for details).

72.1 Character Degrees and Derived Length

1 ▶ `Alpha( G )` A

For a group G, `Alpha` returns a list whose i-th entry is the maximal derived length of groups $G/\ker(\chi)$ for $\chi \in Irr(G)$ with $\chi(1)$ at most the i-th irreducible degree of G.

2 ▶ `Delta( G )` A

For a group G, `Delta` returns the list `[ 1, alp[2] - alp[1], ..., alp[`n`] - alp[`n`-1] ]`, where `alp = Alpha( G )` (see 72.1.1).

3 ▶ `IsBergerCondition( G )` P
▶ `IsBergerCondition( chi )` P

Called with an irreducible character *chi* of the group G, `IsBergerCondition` returns `true` if *chi* satisfies $M' \leq \ker(\chi)$ for every normal subgroup M of G with the property that $M \leq \ker(\psi)$ for all $\psi \in Irr(G)$ with $\psi(1) < \chi(1)$, and `false` otherwise.

Called with a group G, `IsBergerCondition` returns `true` if all irreducible characters of G satisfy the inequality above, and `false` otherwise.

For groups of odd order the result is always `true` by a theorem of T. R. Berger (see [Ber76], Thm. 2.2).

In the case that `false` is returned `InfoMonomial` tells about a degree for which the inequality is violated.

```
gap> Alpha( Sl23 );
[ 1, 3, 3 ]
gap> Alpha( S4 );
[ 1, 2, 3 ]
gap> Delta( Sl23 );
[ 1, 2, 0 ]
gap> Delta( S4 );
[ 1, 1, 1 ]
gap> IsBergerCondition( S4 );
true
gap> IsBergerCondition( Sl23 );
false
gap> List( Irr( Sl23 ), IsBergerCondition );
[ true, true, true, false, false, false, true ]
gap> List( Irr( Sl23 ), Degree );
[ 1, 1, 1, 2, 2, 2, 3 ]
```

72.2 Primitivity of Characters

1 ▶ `TestHomogeneous( chi, N )` F

For a group character *chi* of the group G, say, and a normal subgroup N of G, `TestHomogeneous` returns a record with information whether the restriction of *chi* to N is homogeneous, i.e., is a multiple of an irreducible character.

N may be given also as list of conjugacy class positions w.r.t. the character table of G.

The components of the result are

`isHomogeneous`
: `true` or `false`,

`comment`
: a string telling a reason for the value of the `isHomogeneous` component,

`character`
irreducible constituent of the restriction, only bound if the restriction had to be checked,

`multiplicity`
multiplicity of the `character` component in the restriction of *chi*.

```
gap> n:= DerivedSubgroup( S123 );;
gap> chi:= Irr( S123 )[7];
Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] )
gap> TestHomogeneous( chi, n );
rec( isHomogeneous := false, comment := "restriction checked",
  character := Character( CharacterTable( Group(
    [ [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ],
      [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ],
      [ [ Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0 ] ] ]) ), [ 1, -1, 1, -1, 1 ] ),
  multiplicity := 1 )
gap> chi:= Irr( S123 )[4];
Character( CharacterTable( SL(2,3) ), [ 2, -2, 1, 1, -1, -1, 0 ] )
gap> cln:= ClassPositionsOfNormalSubgroup( CharacterTable( S123 ), n );
[ 1, 2, 7 ]
gap> TestHomogeneous( chi, cln );
rec( isHomogeneous := true, comment := "restricts irreducibly" )
```

2 ▶ `IsPrimitiveCharacter(` *chi* `)` P

For a character *chi* of the group G, say, `IsPrimitiveCharacter` returns `true` if *chi* is not induced from any proper subgroup, and `false` otherwise.

```
gap> IsPrimitive( Irr( S123 )[4] );
true
gap> IsPrimitive( Irr( S123 )[7] );
false
```

3 ▶ `TestQuasiPrimitive(` *chi* `)` A
▶ `IsQuasiPrimitive(` *chi* `)` P

`TestQuasiPrimitive` returns a record with information about quasiprimitivity of the group character *chi*, i.e., whether *chi* restricts homogeneously to every normal subgroup of its group. The result record contains at least the components `isQuasiPrimitive` (with value either `true` or `false`) and `comment` (a string telling a reason for the value of the component `isQuasiPrimitive`). If *chi* is not quasiprimitive then there is additionally a component `character`, with value an irreducible constituent of a nonhomogeneous restriction of *chi*.

`IsQuasiPrimitive` returns `true` or `false`, depending on whether the character *chi* is quasiprimitive.

Note that for solvable groups, quasiprimitivity is the same as primitivity (see 72.2.2).

```
gap> chi:= Irr( S123 )[4];
Character( CharacterTable( SL(2,3) ), [ 2, -2, 1, 1, -1, -1, 0 ] )
gap> TestQuasiPrimitive( chi );
rec( isQuasiPrimitive := true, comment := "all restrictions checked" )
gap> chi:= Irr( S123 )[7];
Character( CharacterTable( SL(2,3) ), [ 3, 3, 0, 0, 0, 0, -1 ] )
gap> TestQuasiPrimitive( chi );
rec( isQuasiPrimitive := false, comment := "restriction checked",
  character := Character( CharacterTable( Group(
```

```
      [ [ [ 0*Z(3), Z(3) ], [ Z(3)^0, 0*Z(3) ] ],
        [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ],
        [ [ Z(3), Z(3)^0 ], [ Z(3)^0, Z(3)^0 ] ] ]) ), [ 1, -1, 1, -1, 1 ] ) )
```

4 ▶ `TestInducedFromNormalSubgroup( `*chi*`[, `*N*`] )` F
▶ `IsInducedFromNormalSubgroup( `*chi*` )` P

`TestInducedFromNormalSubgroup` returns a record with information whether the irreducible character *chi* of the group G, say, is induced from a proper normal subgroup of G. If the second argument N is present, which must be a normal subgroup of G or the list of class positions of a normal subgroup of G, it is checked whether *chi* is induced from N.

The result contains always the components `isInduced` (either `true` or `false`) and `comment` (a string telling a reason for the value of the component `isInduced`). In the `true` case there is a component `character` which contains a character of a maximal normal subgroup from which *chi* is induced.

`IsInducedFromNormalSubgroup` returns `true` if *chi* is induced from a proper normal subgroup of G, and `false` otherwise.

```
gap> List( Irr( S123 ), IsInducedFromNormalSubgroup );
[ false, false, false, false, false, false, true ]
gap> List( Irr( S4 ){ [ 1, 3, 4 ] },
>          TestInducedFromNormalSubgroup );
[ rec( isInduced := false, comment := "linear character" ),
  rec( isInduced := true, comment := "induced from component '.character'",
      character := Character( CharacterTable( Alt( [ 1 .. 4 ] ) ),
        [ 1, 1, E(3)^2, E(3) ] ) ),
  rec( isInduced := false, comment := "all maximal normal subgroups checked"
     ) ]
```

72.3 Testing Monomiality

A character χ of a finite group G is called **monomial** if χ is induced from a linear character of a subgroup of G. A finite group G is called **monomial** (or M**-group**) if each ordinary irreducible character of G is monomial.

There are GAP properties `IsMonomialGroup` (see 37.15.9) and `IsMonomialCharacter`, but one can use `IsMonomial` instead.

1 ▶ `TestMonomial( `*chi*` )` A
▶ `TestMonomial( `*G*` )` A
▶ `TestMonomial( `*chi*`, `*uselattice*` )` O
▶ `TestMonomial( `*G*`, `*uselattice*` )` O

Called with a group character *chi* of a group G, `TestMonomial` returns a record containing information about monomiality of the group G or the group character *chi*, respectively.

If `TestMonomial` proves the character *chi* to be monomial then the result contains components `isMonomial` (with value `true`), `comment` (a string telling a reason for monomiality), and if it was necessary to compute a linear character from which *chi* is induced, also a component `character`.

If `TestMonomial` proves *chi* or G to be nonmonomial then the value of the component `isMonomial` is `false`, and in the case of G a nonmonomial character is the value of the component `character` if it had been necessary to compute it.

A Boolean can be entered as the second argument *uselattice*; if the value is `true` then the subgroup lattice of the underlying group is used if necessary, if the value is `false` then the subgroup lattice is used only for groups of order at most `TestMonomialUseLattice` (see 72.3.2). The default value of *uselattice* is `false`.

For a group whose lattice must not be used, it may happen that `TestMonomial` cannot prove or disprove monomiality; then the result record contains the component `isMonomial` with value `"?"`. This case occurs in the call for a character *chi* if and only if *chi* is not induced from the inertia subgroup of a component of any reducible restriction to a normal subgroup. It can happen that *chi* is monomial in this situation. For a group, this case occurs if no irreducible character can be proved to be nonmonomial, and if no decision is possible for at least one irreducible character.

```
gap> TestMonomial( S4 );
rec( isMonomial := true, comment := "abelian by supersolvable group" )
gap> TestMonomial( Sl23 );
rec( isMonomial := false, comment := "list Delta( G ) contains entry > 1" )
```

2 ▶ `TestMonomialUseLattice` V

This global variable controls for which groups the operation `TestMonomial` (see 72.3.1) may compute the subgroup lattice. The value can be set to a positive integer or `infinity`, the default is 1000.

3 ▶ `IsMonomialNumber(` *n* `)` P

For a positive integer n, `IsMonomialNumber` returns `true` if every solvable group of order n is monomial, and `false` otherwise. One can also use `IsMonomial` instead.

Let $\nu_p(n)$ denote the multiplicity of the prime p as factor of n, and $ord(p, q)$ the multiplicative order of p (mod q).

Then there exists a solvable nonmonomial group of order n if and only if one of the following conditions is satisfied.

1. $\nu_2(n) \geq 2$ and there is a p such that $\nu_p(n) \geq 3$ and $p \equiv -1 \pmod 4$,
2. $\nu_2(n) \geq 3$ and there is a p such that $\nu_p(n) \geq 3$ and $p \equiv 1 \pmod 4$,
3. there are odd prime divisors p and q of n such that $ord(p, q)$ is even and $ord(p, q) < \nu_p(n)$ (especially $\nu_p(n) \geq 3$),
4. there is a prime divisor q of n such that $\nu_2(n) \geq 2ord(2, q) + 2$ (especially $\nu_2(n) \geq 4$),
5. $\nu_2(n) \geq 2$ and there is a p such that $p \equiv 1 \pmod 4$, $ord(p, q)$ is odd, and $2ord(p, q) < \nu_p(n)$ (especially $\nu_p(n) \geq 3$).

These five possibilities correspond to the five types of solvable minimal nonmonomial groups (see 72.4.2) that can occur as subgroups and factor groups of groups of order n.

```
gap> Filtered( [ 1 .. 111 ], x -> not IsMonomial( x ) );
[ 24, 48, 72, 96, 108 ]
```

4 ▶ `TestMonomialQuick(` *chi* `)` A
▶ `TestMonomialQuick(` *G* `)` A

`TestMonomialQuick` does some cheap tests whether the irreducible character *chi* resp. the group *G* is monomial. Here "cheap" means in particular that no computations of character tables are involved. The return value is a record with components

`isMonomial`
: either `true` or `false` or the string `"?"`, depending on whether (non)monomiality could be proved, and

`comment`
: a string telling the reason for the value of the `isMonomial` component.

A group G is proved to be monomial by `TestMonomialQuick` if G is nilpotent or Sylow abelian by supersolvable, or if G is solvable and its order is not divisible by the third power of a prime, Nonsolvable groups are proved to be nonmonomial by `TestMonomialQuick`.

An irreducible character *chi* is proved to be monomial if it is linear, or if its codegree is a prime power, or if its group knows to be monomial, or if the factor group modulo the kernel can be proved to be monomial by `TestMonomialQuick`.

```
gap> TestMonomialQuick( Irr( S4 )[3] );
rec( isMonomial := true, comment := "whole group is monomial" )
gap> TestMonomialQuick( S4 );
rec( isMonomial := true, comment := "abelian by supersolvable group" )
gap> TestMonomialQuick( S123 );
rec( isMonomial := "?", comment := "no decision by cheap tests" )
```

5 ▶ `TestSubnormallyMonomial( G )` A
▶ `TestSubnormallyMonomial( chi )` A
▶ `IsSubnormallyMonomial( G )` P
▶ `IsSubnormallyMonomial( chi )` P

A character of the group G is called **subnormally monomial** (SM for short) if it is induced from a linear character of a subnormal subgroup of G. A group G is called SM if all its irreducible characters are SM.

`TestSubnormallyMonomial` returns a record with information whether the group G or the irreducible character *chi* of G is SM.

The result has components `isSubnormallyMonomial` (either `true` or `false`) and `comment` (a string telling a reason for the value of the component `isSubnormallyMonomial`); in the case that the `isSubnormallyMonomial` component has value `false` there is also a component `character`, with value an irreducible character of G that is not SM.

`IsSubnormallyMonomial` returns `true` if the group G or the group character *chi* is subnormally monomial, and `false` otherwise.

```
gap> TestSubnormallyMonomial( S4 );
rec( isSubnormallyMonomial := false,
  character := Character( CharacterTable( S4 ), [ 3, -1, -1, 0, 1 ] ),
  comment := "found non-SM character" )
gap> TestSubnormallyMonomial( Irr( S4 )[4] );
rec( isSubnormallyMonomial := false,
  comment := "all subnormal subgroups checked" )
gap> TestSubnormallyMonomial( DerivedSubgroup( S4 ) );
rec( isSubnormallyMonomial := true, comment := "all irreducibles checked" )
```

6 ▶ `TestRelativelySM( G )` A
▶ `TestRelativelySM( chi )` A
▶ `TestRelativelySM( G, N )` O
▶ `TestRelativelySM( chi, N )` O
▶ `IsRelativelySM( chi )` P
▶ `IsRelativelySM( G )` P

In the first two cases, `TestRelativelySM` returns a record with information whether the argument, which must be a SM group G or an irreducible character *chi* of a SM group G, is relatively SM with respect to every normal subgroup of G.

In the second two cases, a normal subgroup N of G is the second argument. Here `TestRelativelySM` returns a record with information whether the first argument is relatively SM with respect to N, i.e, whether there is

a subnormal subgroup H of G that contains N such that the character *chi* resp. every irreducible character of G is induced from a character ψ of H such that the restriction of ψ to N is irreducible.

The result record has the components `isRelativelySM` (with value either `true` or `false`) and `comment` (a string that describes a reason). If the argument is a group G that is not relatively SM with respect to a normal subgroup then additionally the component `character` is bound, with value a not relatively SM character of such a normal subgroup.

`IsRelativelySM` returns `true` if the SM group G or the irreducible character *chi* of the SM group G is relatively SM with respect to every normal subgroup of G, and `false` otherwise.

Note that it is **not** checked whether G is SM.

```
gap> IsSubnormallyMonomial( DerivedSubgroup( S4 ) );
true
gap> TestRelativelySM( DerivedSubgroup( S4 ) );
rec( isRelativelySM := true,
  comment := "normal subgroups are abelian or have nilpotent factor group" )
```

72.4 Minimal Nonmonomial Groups

1 ▶ `IsMinimalNonmonomial( G )` P

A group G is called **minimal nonmonomial** if it is nonmonomial, and all proper subgroups and factor groups are monomial.

```
gap> IsMinimalNonmonomial( Sl23 );  IsMinimalNonmonomial( S4 );
true
false
```

2 ▶ `MinimalNonmonomialGroup( p, factsize )` F

is a solvable minimal nonmonomial group described by the parameters *factsize* and p if such a group exists, and `false` otherwise.

Suppose that the required group K exists. Then *factsize* is the size of the Fitting factor $K/F(K)$, and this value is 4, 8, an odd prime, twice an odd prime, or four times an odd prime. In the case that *factsize* is twice an odd prime, the centre $Z(K)$ is cyclic of order 2^{p+1}. In all other cases p is the (unique) prime that divides the order of $F(K)$.

The solvable minimal nonmonomial groups were classified by van der Waall, see [vdW76].

```
gap> MinimalNonmonomialGroup(  2,  3 ); # the group SL(2,3)
2^(1+2):3
gap> MinimalNonmonomialGroup(  3,  4 );
3^(1+2):4
gap> MinimalNonmonomialGroup(  5,  8 );
5^(1+2):Q8
gap> MinimalNonmonomialGroup( 13, 12 );
13^(1+2):2.D6
gap> MinimalNonmonomialGroup(  1, 14 );
2^(1+6):D14
gap> MinimalNonmonomialGroup(  2, 14 );
(2^(1+6)Y4):D14
```

73 Installing GAP

GAP runs on a large number of different operating systems. It behaves slightly different on each of those. This chapter describes the behaviour of GAP, the installation, and the options on some of those operating systems.

Currently it contains instructions for **UNIX** (which includes the popular variant **Linux**), for Apple Macintosh computers under **OS X** (see 73.14) as well under the "classic" **MacOS** (see 73.15), and finally for **Windows**.

For other systems the section 73.13 gives hints how to approach such a port.

73.1 Installation Overview

To permit compatibility over a wide range of operating systems, the installation of GAP might differ from what you are accustomed to for you particular operating system. In particular, there is no "Installer" program.

Installing the GAP distribution alone takes about 150MB of disk space. The packages add another 100MB. (These are upper limits. Unix is more efficient in storing a large number of small files than Windows.) You also should have at least 64MB of memory to run GAP.

The installation consists of 5 easy steps:

- Get the archive(s) suitable for your system
- Unpack
- Compile (unless a binary has been provided already)
- Test the installation
- Install packages. (Some packages will only work under Unix and OS X).

Installation will always install the full version of GAP. There is no "Upgrade" mode. If you are worried about losing the old version, you can keep an existing installation of GAP in another directory, it will not be overwritten.

Section 73.8 contains information about the manual, where to find and how to print it. Section 73.9 lists common problems with the installation.

73.2 Get the Archives

You can get archives for the GAP distribution from

`http://www.gap-system.org` . As different operating systems use different archive formats, GAP is available in a variety of archives. These archives slightly differ in the treatment of text or binary files. If you get the wrong archive you might get error messages during compilation or not be able to look at text files in an editor.

If you use

Unix
: you can use the `.tar.gz`, `.tar.bz2` or `.zoo` archives.

Macintosh OS X
: you can use the `.tar.gz` or `.zoo` archives (or install the `MacOS` version, see also 73.14).

MacOS
: Use the `.zoo` archive. Also make sure you get the `.sit` StuffIt archives for the `unzoo` uncompressor.

Windows
: Use the `-win.zip` or the `.zoo` archives.

Now get the installation archives of the right kind, according to your operating system. You want to get the archives:

`gap4r4p5`
: which contains the main GAP installation

`packages4r4p5`
: which contains GAP packages that provide further functionality

`unzoo`
: Only if you use the `.zoo` archives: Unix and OS X users get the source code `unzoo.c`, MacOS users get the `.sit` archive and Windows users the `.exe` binary.

`htmie4r4p5`
: An alternative version of the HTML manual which uses a nonstandard "symbol" font instead of Unicode characters. If you use a webbrowser such as `Internet Explorer` that has difficulties with rendering Unicode correctly, you might want to replace the HTML documentation with this version. (See section 73.8.)

`xtom4r4p5`
: Contains about 80MB of further tables of marks. (You can always install this later if the need arises.)

Note that starting with release 4.4, the distribution archives for GAP will always contain the most recent bugfix. Thus if you install anew from scratch, you will not need to get any bugfixes.

73.3 Unpacking

The concrete act of unpacking will vary slightly, depending on the operating system and the type of archive used.

Unix, OS X
: Under Unix or OS X unpack the archive `gap4r4p5` in whatever place you want GAP to reside. If you use the `.zoo` archive, you will have to compile the `unzoo` program first

```
cc -o unzoo -DSYS_IS_UNIX -O unzoo.c
./unzoo -x gap4r4p5.zoo
```

(If you unpack the archive as `root` user under UNIX, make sure that you issue the command `umask 022` before, to ensure that users will have permissions to read the files.)

MacOS
: if the `.sit` archive did not extract autmatically, click it to force extraction. You will end up with an applications, `unzoo 4.4 PPC`. Now move this applications, as well as the `gap4r4p5.zoo` archive to the Folder in which you want to install GAP. Drag the archive `gap4r4p5.zoo onto the icon of 'unzoo 4.4 PPC`. You will get a window with many lines of text output. This process will create a folder `gap4r4` in the current folder.

Windows
: The archive must be extracted to the **main** directory of the `C:` drive. (If you do not have permissions or sufficient free space to create directories there, see section 73.17.) If you use the `.zoo` archives we provide move `unzoo.exe` and `gap4r4p5.zoo` in the `C:` directory, open the `MS-DOS` (or `Command Prompt`) window. (You can find this under "Start/Programs/Accessories".) In this window issue the commands

```
cd c:\
unzoo -x gap4r4p5.zoo
```

(It might be necessary to use upper case letters instead)
: If you prefer to use the `-win.zip` archive use a `ZIP` extractor. Make sure that you specify extraction to the `c:/` folder (with **no** extra directory name – the directory is part of the archive) to avoid extraction in a wrong place or in a separate directory. After extraction you can start GAP with the file

```
C:\GAP4R4\bin\gapw95.exe
```

73.4 Compilation

For the MacOS and Windows version the unpacking process will already have put binaries in place. Under Unix and OS X you will have to compile (OS X users please see section 73.14 for information about compilation) such a binary yourself.

Go into the directory `gap4r4` (which you just created by unpacking). Issue the two commands

```
./configure
make
```

Both will produce a lot of text output. You should end up with a shell script `bin/gap.sh` which you can use to start GAP. (If you want, you can copy this script later to a directory that is listed in your search path.)

OS X users please note that this script must be started from within the `Terminal` Application. It is not possible to start GAP by clicking this script.

If you get weird error messages from these commands, make sure that you got the Unix version of GAP (i.e. not the `-win.zip` format archive) and that you extracted the archive on the machine on which you compile. Also see section 73.10 below for further information.

If you use OS X in the "Panther" release (version 10.3), you might want to change the call to `make` to

```
make COPTS="-fast -mcpu=7450"
```

on a G4 system or to

```
make COPTS="-O3 -mtune=G5 -mcpu=G5 -mpowerpc64"
```

on a G5 system (please note that the `-fast` compiler option causes problems on a G5 at the time of this writing – February 2004). Initial tests indicate that this will give you substantially improved performance.

Unless you want to use the same installation of GAP also under Windows or MacOS (not OS X), issue the command

```
make removewin
```

to delete unnecessary files that are Windows-only and only take up about 2MB of space.

73.5 Test of the installation

You are now at a point where you can start GAP for the first time. Unix and OS X users type

```
./bin/gap.sh
```

MacOS users click the Application `GAP 4 PPC` in the `gap4r4` directory, Windows users start

```
C:\GAP4R4\bin\gapw95.exe
```

GAP should start up with its banner and after a little while give you a command prompt `>`.

Try a few commands to see if the compilation succeeded.

```
gap> 2 * 3 + 4;
10
gap> Factorial( 30 );
265252859812191058636308480000000
gap> m11 := Group((1,2,3,4,5,6,7,8,9,10,11),(3,7,11,8)(4,10,5,6));;
gap> Size( m11 );
7920
gap> Length( ConjugacyClasses( m11 ) );
10
gap> Factors( 10^42 + 1 );
#I  IsPrimeInt: probably prime, but not proven: 4458192223320340849
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
```

If you get the error message "`hmm, I cannot find lib/init.g`" you are likely to have installed only the binary (or on Windows have not installed GAP in the root directory of the `C:` drive).

If GAP starts but you get error messages for the commands you issued, the files in the `lib` directory are likely to be corrupt or incomplete. Make sure that you used the proper archive and that extraction proceeded without errors.

Especially try the command line editing and history facilities, because they are probably the most machine dependent feature of GAP. Enter a few commands and then make sure that *ctr*-`P` redisplays the last command, that *ctr*-`E` moves the cursor to the end of the line, that *ctr*-`B` moves the cursor back one character, and that *ctr*-`D` deletes single characters. So, after entering the above commands, typing

ctr-`P` *ctr*-`E` *ctr*-`B` *ctr*-`B` *ctr*-`B` *ctr*-`D` `2` *return*

should give the following lines:

```
gap> Factors( 10^42 + 2 );
#I  IsPrimeInt: probably prime, but not proven: 3145594690908701990242740067
[ 2, 3, 433, 953, 128400049, 3145594690908701990242740067 ]
```

If you want to run a more thorough test (this is not required), you can read in a test script that exercises more of GAPs capabilities.

```
gap> Read( Filename( DirectoriesLibrary( "tst" ), "testall.g" ) );
```

The test requires about 60-70MB of memory and runs about 2 minutes on a Pentium III/1 GHz machine. You will get a large number of lines with output about the progress of the tests.

```
test file         GAP4stones     time(msec)
-----------------------------------------
testing: /home/fac/a/hulpke/gap4/tst/zlattice.tst
zlattice.tst               0              0
testing: /home/fac/a/hulpke/gap4/tst/gaussian.tst
gaussian.tst               0             10
[ further lines deleted ]
```

Windows users should note that the MS-DOS/Command Prompt user interface provided by Microsoft might not offer history scrolling or cut and paste with the mouse. See section 73.17 for a way around this.

73.6 Packages

To extract the packages, extract the `package4r4p5` archive **in** the "gap4r4/pkg' directory by the same method used to extracted the main archive. (If you use `unzoo` or under MacOS move both the `unzoo` binary and the archive in the `pkg` directory and extract there.)

For packages that consist only of GAP code no further installation is necessary. Some packages however contain external binaries that will require separate compilation. (If you use the MacOS version or Windows you will not be able to use external binaries anyhow any you can skip the rest of this section.) You can skip this compilation now and do it later – GAP will work fine, just the capabilities of the affected packages won't be available.

In general, each package contains a `README` file that contains information about the package and the necessary installation. Typically the installation for a package consists of changing in the packages directory and issuing the commands `./configure; make` in the packages directory. (This has to be done separately for exery package).

If you have problems with package installations please contact the package authors as listed in the packages `README` file.

73.7 Finish Installation and Cleanup

Congratulations, your installation is finished.

Once the installation is complete, we would like to ask you to send us a short note to `support@gap-system.org`, telling us about the installation. (This is just a courtesy; we like to know how many people are using GAP and get feedback regarding difficulties (hopefully none) that users may have had with installation.)

We also suggest that you subscribe to our `GAP Forum` mailing list; see the GAP web pages for details. Whenever there is a bug fix or new release of GAP this is where it is announced. The `GAP Forum` also deals with user questions of a general nature; bug reports and other problems you have while installing and/or using GAP should be sent to `support@gap-system.org`.

At this point you can also delete all archive files. (You might want to keep `unzoo` if you used it in case a bugfix will be released.)

The directories `trans`, `small` and `prim` within `gap4r4` contain data libraries. If you are short of disk space you can erase some of them. Similarly, you can erase and any of the GAP package directories in `pkg` that you have decided you don't need, but then of course you will not be able to access these data or packages. (You should do this only if you have diskspace problems as you might find out later that you need certain packages.)

If you are new to GAP, you might want to read through the following two sections for information about the documentation.

73.8 The Documentation

The GAP manual is distributed in various "books". The standard distribution contains four of them (as well as a comprehensive index). GAP packages (see Chapter 74 and, in particular, Section 74.2) provide their own documentation in their own `doc` directories.

All documentation will be available automatically for the online help (see Section 2.8 in the Tutorial and Chapter 2 in this manual for more details).

There also is (if installed) an HTML version of some books that can be viewed with an HTML browser, see 2.3. Some of these use **unicode characters** for mathematical formulae. If your browser is not able to display these (older versions of `Internet Explorer` do not), get the `htmie4r4p5` archive which provides math symbols in a different encoding. (`Mozilla`, `Konqueror` and `Safari` all support unicode characters.)

The manual is also available in `pdf` format. In the full UNIX distribution (`gap4r4p5.zoo`) these files are included in the directory `gap4r4/doc` in the subdirectories `tut` (a beginner's tutorial), `ref` (the reference manual), `prg` (programmer's tutorial), `ext` (programmer's reference) and `new` (new material that might still change in future versions).

If you want to use these manual files with the online help you may check (or make sure) that your system provides some additional software like

`xpdf`
: see

  `http://www.foolabs.com/xpdf/`

`xdvi`
: see any of the CTAN sites/mirrors; the main site is:

  `http://www.ctan.org/`
  : and the mirrors are listed at:

  `http://www.ctan.org/tex-archive/CTAN.sites`
  : At any of the mirrors the path of the file containing the `xdvi` archive (after the main site name) is `tex-archive/dviware/xdvi/xdvi.tar.gz`.

`acroread`
: see

  `http://www.adobe.com/products/acrobat/readstep.html`

As a complete beginner, we suggest you read the tutorial first for an introduction to GAP 4. Then start to use the system with extensive use of the online help system (see Section 2.8 in the Tutorial and Chapter 2 in this manual).

If you have experience with GAP 3, it might be still worthwhile to at least glance over the first chapters of the tutorial. You however should read the last chapter of the tutorial, "Migrating to GAP4". This chapter gives a summary of changes between GAP 3 and GAP 4 that will affect the user. It also explains a "compatibility mode" you may turn on to make GAP 4 behave a bit more like GAP 3.

As some of the manuals are quite large, you should not immediately print them. If you start using GAP it can be helpful to print the tutorial (and probably the first chapters of the reference manual). There is no compelling reason to print the whole of the reference manual, better use the online help which provides useful search features.

73.9 If Things Go Wrong

This section lists a few common problems when installing or running GAP and their remedies. Also see the FAQ list on the GAP web pages at

```
http://www.gap-system.org/Info/Faq/faq.html
```

GAP starts with a warning "`hmm, I cannot find 'lib/init.g'`".
: You either started only the binary or did not edit the shell script/batch file to give the correct library path. You must start the binary with the command line option `-l` *path* where *path* is the path to the GAP home directory. See section 3.1 in the reference manual.

When starting, GAP produces error messages about undefined variables.
: You might have a `.gaprc` file that was intended for GAP 3 but is not compatible with GAP 4. See section 3.4 in chapter 3 of the reference manual.

GAP complains: "`corrupted completion file`".
: Some library files got changed without rebuilding the completion files. This is often a sign that earlier a bugfix was not installed properly or that you changed the library yourself. In the latter case, start GAP with command line option `-N` and see section 3.5.

GAP stops with an error message "`exceeded the permitted memory`".
: Your job got bigger than what is permitted by default (256MB). (This is a safety feature to avoid single jobs wrecking a multi-user system.) You can type `return;` to continue, if the error message happens repeatedly you better start the job anew and use the command line option `-o` to set a higher memory limit.

GAP stops with an error message: "`cannot extend the workspace any more`".
: Your calculation exceeded the available memory. Most likely you asked GAP to do something which required more memory than you have (as listing all elements of S_{15} for example). You can use the command line option `-g` (see section 3.1 in the reference manual) to display how much memory GAP uses. If this is below what your machine has available extending the workspace is impossible. Start GAP with more memory or use the `-a` option to pre-allocate initially a large piece of workspace.

GAP is not able to allocate memory above a certain limit
: Being a 32 bit program, GAP currently is unable to use over 4GB of memory. Since the address space also has to keep the operating system, 3GB probably are an upper limit for a GAP workspace.

Depending on the operating system, it also might be necessary to compile the GAP binary statically (i.e. to include all system libraries) to avoid collisions with system libraries located by default at an address within the workspace. (Under Linux for example, 1GB is a typical limit.) You can compile a static binary using `make static`.

Windows users also see below for a built-in limit.

`make` complains about not being able to find files in `cnf` or `src` which exist.
: The dates of the new files were not extracted properly (Alpha-OSF machines are prone to this). Call

```
touch * cnf/* src/*
```

from the main GAP directory (this ought to reset the date of all relevant files to "now") and try again.

Recompilation does not actually compile changed files.
: The dates of the new files were not extracted properly. Go in the source directory and `touch` (UNIX command to change date) the new files.

Recompilation fails or the new binary crashes.
: Call `make clean` and restart the `configure` / `make` process completely from scratch. (It is possible that the operating system and/or compiler got upgraded in the meantime and so the existing `.o` files cannot be used any longer.

A calculation runs into an error "no method found".
: GAP is not able to execute a certain operation with the given arguments. Besides the possibility of bugs in the library this means two things: Either GAP truly is incapable of coping with this task (the objects might be too complicated for the existing algorithms or there might be no algorithm that can cope with the input). Another possibility is that GAP does not **know** that the objects have certain nice properties (like being finite) which are required for the available algorithms. See sections 7.2.1 and 13.7.1.

Problems specific to Windows

Command line editing does not work under Windows.
: The default key commands are UNIX-like. GAP also tries to emulate some of the special keys under Windows, however if the key repeat is set too high, Windows loses parts of the codes for these keys and thus GAP cannot recognize them. Windows98 produces the same scan code for all cursor keys. As GAP does not interface directly with the Windows machinery, there is no known way around this problem. Use the Unix-style cursor commands.

The `^`-key or `"`-key cannot be entered.
: This is a problem if you are running a keyboard driver for some non-english languages. These drivers catch the `^` character to produce the French circumflex accent and do not pass it properly to GAP. No fix is known.

GAP does not start or cannot expand memory
: You will have to edit a registry entry to be able to use more than 127MB of memory. See 73.17.

Cut and Paste does not work
: You might want to start GAP under `rxvt` – see 73.17. Also

`http://www.gap-system.org/Faq/Hardware-OS/hardware-os.html` might give a remedy.

You get an error message about the `cygwin1.dll`
: GAP comnes with a version of this dynamic library. If you have another version installed (use "Find"), delete the older one (and probably copy the newer one in both places).

If all these remedies fail or you encountered a bug please send a mail to `support@gap-system.org`. Please give:

- a (short, if possible) self-contained excerpt of a GAP session containing both input and output that illustrates your problem (including comments of why you think it is a bug); and
- state the type of machine, operating system, (compiler used, if UNIX/Linux) and version of GAP (for example "gap4r4p5, fix1") you are using (the line after the GAP 4 banner starting:

```
GAP4, Version: 4...
```

when your GAP 4 starts up, supplies the information required).

73.10 Known Problems of the Configure Process

If `make` complains "Do not know how to make *xyz*" but *xyz* is an existing file, it is likely that the dates of the files were not extracted properly (Alpha-OSF machines are prone to this). Call

```
touch * cnf/* src/*
```

from the main GAP directory (this ought to reset the date of all relevant files to "now") and try again.

Sometimes the `configure` process does not properly figure out the "inline" compiler command. If you get error messages that complain that "inline" is unknown, edit the file `config.h` in the `bin/`*target* subdirectory and replace the line

```
/* #undef inline */
```

by

```
#define inline
```

and then try to compile again.

The `configure` script respects compiler settings given in environment variables. However such settings may conflict with the automatic configuration process. If `configure` produces strange error messages about not being able to run the compiler, check whether environment variables that might affect the compilation (in particular `CC`, `LD`, `CFLAGS`, `LDFLAGS` and `C_INCLUDE_PATH`) are set and reset them using `unsetenv`.

Some users reported problems with `make`, while the GNU version `gmake` worked. Thus if problems occur you should try `gmake` instead if it is installed on your machine.

73.11 Problems on Particular Systems

The highest levels of optimization of the OSF/4 C compiler `cc` on the Compaq alpha chip make assumptions about the use of pointers which are not valid for GAP, and produce executables that can crash; `-O3` seems to be safe, but `-O4` and `-fast` are not.

On Sun and IRIX systems which are capable of running in 32 or 64 bit modes, it is possible to build a 64 bit version of GAP, but special procedures are needed (and, on Suns, a compiler bug must be circumvented). If you wish to compile on such a system, please send an email to `support@gap-system.org`.

73.12 Optimization and Compiler Options

Because of the large variety of different versions of UNIX and different compilers it is possible that the `configure` process will not chose best possible optimization level, but you might need to tell `make` about it.

If you want to compile GAP with further compiler options (for example specific processor optimizations) you will have to assign them to the variable `COPTS` as in the following example when calling `make`:

```
make COPTS=-option
```

If there are several compiler options or if they contain spaces you might have to enclose them by quotes to avoid depending on the shell you are using.

The `configure` process also introduces some default compiler options. (See the `Makefile` in the `bin` directory for details.) You can eliminate these by assigning the variable `CFLAGS` (which contains the default options and `COPTS`) to the desired list of compiler options in the same way as you would assign `COPTS`.

The **recommended** C compiler for GAP is the GNU C compiler `gcc`, or a related compiler such as `egcs`. There are two reasons for this recommendation: firstly we use `gcc` in GAP development and so this combination has been far more heavily tested than any other and secondly, we have found that it generally produces code which is faster than that produced by other compilers.

If you do wish to use another compiler, you should remove `config.cache` and `config.status` in the GAP root directory, set the environment variable `CC` to the name of your preferred compiler and then rerun `configure` and `make`. You may have to experiment to determine the best values for `CFLAGS` and/or `COPTS` as described above. Please let us (`support@gap-system.org`) know the results of your experiments.

73.13 Porting GAP

Porting GAP to a new operating system should not be very difficult. However, GAP expects some features from the operating system and the compiler and porting GAP to a system or with a compiler that do not have those features may prove very difficult.

The design of GAP makes it quite portable. GAP consists of a small kernel written in the programming language C and a large library written in the programming language provided by the GAP kernel, which is also called GAP.

Once the kernel has been ported, the library poses no additional problem, because all those functions only need the kernel to work, they need no additional support from the environment.

The kernel itself is separated into a large part that is largely operating system and compiler independent, and one file that contains all the operating system and compiler dependent functions. Usually only this file must be modified to port GAP to a new operating system.

Now let us take a look at the minimal support that GAP needs from the operating system and the machine:

You need enough main memory in your computer. The size of the GAP kernel varies between 1.5 and 2.5 MByte (depending on the machine). The GAP library additionally takes a minimum of 10MByte and the library of functions that GAP loads takes up another 1.5 MByte. So it is clear that at least 16 MByte of main memory are required to do any serious work with GAP.

Additionally, the GAP kernel needs a flat address space, that is all the memory is available in one contiguous chunk.

Note that this implies that there is no point in trying to port GAP to plain MS-DOS running on IBM PCs and compatibles. The version of GAP for IBM PC compatibles that we provide runs on machines with the Intel 80486, Pentium or beyond processor under 32-bit Windows. (This is also necessary, because, as just mentioned, GAP wants to view its memory as a large flat address space.)

Next let us turn to the requirements for the C compiler and its library.

As was already mentioned, the GAP kernel is written in the C language. We have tried to use as few features of the C language as possible. GAP has been compiled without problems with compilers that adhere to the old definition from Kernighan and Ritchie, and with compilers that adhere to the new definition from the ANSI-C standard.

Porting GAP to another UNIX should not be hard. You need some very basic understanding of C and UNIX. If you plan to port GAP to a non-UNIX system please contact `support@gap-system.org`.

The configuration script runs various tests to determine the configuration of your system. It produces a file `bin/`*architecture*`/config.h` which contains definitions according to the test results. It might be, however, that the tests used don't produce on your machine the results they are expected to or that further tests are necessary. If this is the case the easiest way is to edit the `config.h` script, remove all object files and call `make` in the `bin/`*architecture* subdirectory. If you have to resort to changing or amending this file, please tell us what had to be changed (mail to `support@gap-system.org`). If you had to add further definitions please also tell what properties of your system these `define`s represent.

If GAP compiles but crashes while reading the library or during a garbage collection with a bus error it is possible that the configuration script did not guess the permitted pointer alignment correctly. This value is stored in the line

```
#define C_STACK_ALIGN      2
```

of `config.h`. Increase the value to the next power of 2 (≤ 8) and compile GAP anew.

There is still a `Makefile` in the `src` directory, but it is not used by the configuration process any longer. As a last resort you might want to try this file, but please still report your problems to `support`.

73.14 GAP for Macintosh OS X

OS X, the new version of the Macintosh operating system, is built on top of a variant of Unix. Thus there are two ways to run GAP under this operating system.

The first way is simply to follow the instructions in section 73.15 below and to run the binary we provide in the "Classic" environment.

The second way is to install the Unix version of GAP.

We recommend to use this (Unix) version because you will be able to use all features of GAP as well as all packages. However for installation you might need a basic knowledge of Unix. Note also that the Unix version of GAP uses Unix style text files. (These files differ from older style macintosh text files in that lines do not contain a trailing `CR` character. Depending on what text editor you use you might not be able to inspect GAP library files and potentially might run into problems with program files you create if they contain strings that go over several lines.)

The following are a couple of notes and remarks about this:

You will need a compiler. The `gcc` C compiler is not installed by default, but is part of the "Developer tools" package. This package might be in an installer package already on your system (look at the `Installer` folder under `Applications`), if not you can get it for free from Apple by registering as a developer. See

`http://developer.apple.com` for details. For the "Panther" release (10.3) of OS X, `gcc` is part of the "XCode" development tools which you should install.

To compile and run GAP you will have to open the `Terminal` application and type the Unix command into its window. The `Terminal` application can be found in the `Utilities` folder in the `Applications` folder. GAP also will run in such a window.

The next thing to note is that you should get the Unix type GAP archives, i.e. usually the tar archive, **not** the zip archive (You won't be able to compile the program as given in the `.zip` archive).

(If you prefer to use the `.zoo` type archive over `.tar` you can use this as well. However then you will need to use the Unix version of `unzoo`, which you will need to compile first by the command

```
gcc -o unzoo -DSYS_IS_UNIX -O unzoo.c
```

If you use the Macintosh version of the `unzoo` extractor, the text files will be converted to MacOS format and you will have problems with the compilation.)

Go (using the `cd` command in the terminal window) into the directory where you want to install GAP, and copy the archives (and the unzoo program if you want to use `zoo`) in this directory. Then extract the archive by the command

```
tar zxvf gap4r4p5.tar.gz
```

(respectively – if you prefer `zoo` –

```
gcc -o unzoo -DSYS_IS_UNIX -O unzoo.c
./unzoo -x gap4r4p5.zoo
```

)

Then simply follow the Unix installation instructions to compile GAP.

73.15 GAP for MacOS

This section contains information about GAP that is specific to the port of GAP for Apple Macintosh systems under MacOS (simply called GAP for MacOS below).

To run GAP for MacOS you need an Apple Macintosh with a Motorola M68020, M68030, or M68040 processor, or a Power Macintosh. The computer must have at least 16MByte of (physical) memory and a harddisk. For serious calculations, much more may be needed. The operating system must be System 7 or higher. GAP for MacOS runs under System X, however only in Classic Mode.

The section 73.18 describes the copyright as it applies to the executable version that we distribute. The section 73.16 describes how you install GAP for MacOS.

Please refer to the relevant sections of Chapter 3 in the GAP reference manual (included with the GAP distribution for an overview over the features of GAP for MacOS.

73.16 Installation of GAP for MacOS

Installing GAP under MacOS is fairly easy. First, decide into which folder you want to install GAP 4.4.5. GAP will be installed in a subfolder `gap4r4` of this folder. You can later move GAP to a different location.

Note that certain parts of the output in the examples should only be taken as rough outline, especially file sizes and file dates are **not** to be taken literally.

If you encounter problems please also see section 73.9 of this document.

Get the Mac-specific files described in 73.2, that is, the distribution `gap4r4p5.zoo` and the binary archive `unzoo4r4-PPC.sit`,

How you can get those files is described in the section 73.2. Remember that the distribution consists of binary files and that you must transmit them in binary mode.

If the `sit` files did not extract automatically click on them to extract them. If even this fails use one of the standard decompression utilities, such as `Stuffit Expander`.

After this process you should end up with two applications, `GAP 4 PPC` and `unzoo 4.4 PPC`.

The latter is used to uncompress the `.zoo` archives which contain most of GAP. The `zoo` archives we provide for GAP contain comments which indicate whether files are text or binary files. The `unzoo` we provide uses these comments. If you use another zoo extractor you might lose this information and end up with files that contain text but cannot be opened as text files.

The following installation example assumes that you are installing GAP in the folder `Applications` on a PowerPC Macintosh. (For a 68k Macintosh you should replace all references to `PPC` to ones referring to `68K`

Move the file `gap4r4p5.zoo` into the folder `Applications` and drag it onto the icon of `unzoo 4.4 PPC`. You will get many lines of output in this window.

This should have created a folder `gap4r4` in the current folder.

(You will not need the file `gap4r4p5.zoo` any longer. If you are short of disk space you can remove it now.)

If you got not the full distribution file but several small files, extract all of them (**except the** GAP **packages!**) in this way.

Move `GAP 4 PPC` and the bugfix file (if there is one) in the folder `gap4r4`. Drag the bugfix file onto the icon of `unzoo 4.4 PPC` to decompress it.

If you got any GAP packages, move them into the `pkg` folder in the folder folder `gap4r4` extract them there, in the same way as the bugfix.

After extraction you may discard all `.zoo` files if you are short of disk space.

The folders `trans`, `small` and `prim` contain data libraries. If you are short of disk space you can erase some of them and any GAP package directories in the `pkg` directory that you don't need, but then of course

you will not be able to access these data and packages. (Any GAP package that has a C code component is essentially UNIX-dependent and you may as well delete those; such packages typically describe in their `README` files that they require `configure` and `make` to complete their installation or have a `src` directory.)

Before you use GAP, you should set up GAP's memory allocation, by setting appropriate values by selecting the GAP application and `Get Info...` in the Finder's `File` menu (in order to be able to modify the values there, you have to do this **before** you launch GAP).

The maximum amount of workspace GAP can use depends on the amount of memory the Finder allocates to GAP when it is launched. The maximum amount of GAP workspace is this value, minus a certain amount used internally by the GAP application (for the PPC version, currently around 1.7 Megabytes, plus the size of the GAP application if you do not use virtual memory, and 2.9 Megabytes for the 68K version), minus any additional amount set with the `-a`, `-P` or `-W` command line options (see below).

You can find information about the amount of free GAP workspace, the total amount of available workspace, and the remaining free memory, by choosing `About GAP` in the `Apple` menu.

To ensure efficient operation, you should not allocate more memory to GAP than the amount of physical memory in your computer. If you are not using virtual memory, the amount may have to be considerably less (depending on your system and the number of other applications which you may want to run at the same time).

If you notice heavy disk use during garbage collections, this is a clear indication that you have allocated too much memory to GAP.

In order to test your installation now run the `GAP` application by clicking on `GAP 4 PPC`. You should get the GAP banner and then the GAP prompt in a window titled `GAP log`. (The process of starting GAP may take a while.)

Try a few things to see if the installation succeeded.

```
gap> 2 * 3 + 4;
10
gap> Factorial( 30 );
265252859812191058636308480000000
gap> Factors( 10^42 + 1 );
[ 29, 101, 281, 9901, 226549, 121499449, 4458192223320340849 ]
gap> m11 := Group((1,2,3,4,5,6,7,8,9,10,11),(3,7,11,8)(4,10,5,6));;
gap> Size( m11 );
7920
gap> Length( ConjugacyClasses( m11 ) );
10
```

A set of test files is provided, running them all probably takes some 40 minutes on a 200 MHz PPC machine. This is not a necessary part of the installation; it only serves as a confirmation that everything went OK. The full test suite takes some time and uses quite a bit of memory (around 70MB), so you may wish to skip this step or run only part of the tests. This does no harm.

Initially we must ensure that the print width of GAP is 80 characters per line which we achieve with the `SizeScreen` command (otherwise we will be swamped with error messages).

```
gap> SizeScreen([80,]);;
gap> Filename( DirectoriesLibrary("tst"), "testall.g" );
"./tst/testall.g"
gap> Read(last);
[many output lines omitted]
```

The information about the manual is system independent; you can find it in section 73.8.

A few final reminders:

- We would appreciate after installation your sending us a short note at `support@gap-system.org` (even if you have installed GAP 3 before). Generally, we do not reply to such emails; we only use them to gain some idea of how many people use GAP and of the machines/operating systems on which GAP has been successfully installed.
- We also suggest that you subscribe to our `GAP Forum` mailing list; see the GAP web pages for details. Whenever there is a bug fix or new release of GAP this is where it is announced. The `GAP Forum` also deals with user questions of a general nature; bug reports and other problems you have while installing and/or using GAP should be sent to `support@gap-system.org`.

That's all, your installation should be complete. Please refer to Chapter 3 in the GAP reference manual for a description of some special features and options of GAP for MacOS.

We hope that you will enjoy using GAP. Remember, if you have problems, do not hesitate to contact us at `support@gap-system.org`. See Section 73.9 for what to include in a bug report.

73.17 Expert Windows installation

This section decribes how to get a better shell for GAP and how to install GAP in another directory. These tasks are slightly complicated due to problems in the design of Windows, if you have not edited a batch file before you might want to contact your system administrator for help.

Some users report that the `rxvt` shell (see

`http://www.rxvt.org`) gives a better windows environment for cut/paste etc.

You can find a copy of this program in the `bin` subdirectory of the GAP installation. **Please note that this program is not part of the GAP distribution and that we cannot offer any support for it**. You can start GAP under this program via the `gaprxvt.bat` script in the `bin` subdirectory.

(The program has been tested only under a particular version of Windows98. It might not work under other releases. It also might be necessary to adapt paths in the batch file.) Under `rxvt` the standard Unix XWindows cut-and-paste operations (left mouse button cuts, middle mouse button pastes) work. After you terminate GAP a text window might stay on which you can safely delete.

If you decide to install GAP in another directory than `C:` you can do so, but you will have to edit a batch file and use this file to start GAP.

First unpack the GAP distribution in the directory you want.

Lets suppose you want GAP to reside in the directory

```
C:\MY PROGRAMS\GAP
```

Extract GAP (as described in the previous section for `C:`) in this directory. (alternatively, you can also first unpack it in

```
C:\GAP4R4
```

test it there first, and afterwards move it in the desired location.)

You now will have to edit the provided batch file, that will be used to start GAP. This batch file is needed, since GAP otherwise will not find its library directories. The file sits in the `bin` directory of the GAP distribution, i.e. in our example

```
C:\MY PROGRAMS\GAP\GAPGAP4R4\bin\gap.bat
```

This file should contain the following **single** line (which might be broken over in several lines in this manual as the page width is limited):

```
"C:\MY PROGRAMS\GAP\GAPGAP4R4\bin\gapw95.exe" -m 14m
  -l "C:\MY PROGRAMS\GAP\GAPGAP4R4" %1 %2 %3 %4 %5 %6 %7 %8
```

You now should be able to start GAP by clicking this `gap.bat` file.

If you also want to use `rxvt` you have to edit the `gaprxvt.bat` file to take care of the changed path in two places for the GAP binary as well as for the GAP library.

By default, the "cygwin" environment we use limits a programs workspace to 128MB of memory. To increase this limit, it is necessary to edit the Windows registry.

WARNING: Editing the registry is the Windows equivalent of open heart surgery. Do not attempt this change if you have no previous experience in doing this. The web page

`http://www.cygwin.com/cygwin-ug-net/setup-maxmem.html` gives further details.

Before changing the entries, you might have to run GAP once first to create the appropriate registry keys.

The shell script `usemem.bat` in the `bin` directory sets a registry entry

`/HKEY_LOCAL_MACHINE/Software/Cygnus Solutions/Cygwin/heap_chunk_in_mb`

to decimal 1024.

If you prefer to do the change by hand, open `regedit` and go to the `Cygwin` Key listed above. Then choose `new value` and add `heap_chunk_in_mb`. `Modify` it to contain decimal 1024.

73.18 Copyrights

In addition to the general copyright for GAP set forth in the Copyright the following terms apply to the versions of GAP for Windows and MacOS.

The executable of GAP for Windows that we distribute was compiled with the `gnuwin32` compiler of the `cygwin` package. This compiler can be obtained by anonymous `ftp` from a variety of general public FTP archives. Many thanks to the Free Software Foundation and RedHat Software for this amazing piece of work.

The GNU C compiler is

Copyright (C) 2002 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA

under the terms of the GNU General Public License (GPL).

The Cygwin32 API library is also covered by the GNU GPL. The executable we provide is linked against this library (and in the process includes GPL'd Cygwin32 glue code). This means that the executable falls under the GPL too, which it does anyhow.

The `cygwin1.dll`, `libW11.dll`, `rxvt.exe` and `regtool.exe` binaries are taken unmodified from the Cygwin distribution. They are copyright by RedHat Software and released under the GPL. You can find more information about `cygwin` under

`http://www.cygwin.com` . You also will be able to obtain the sources for `cygwin` from this place.

The system dependent part of GAP for MacOS was written by Burkhard Höfling (his email address is `b.hoefling@tu-bs.de`). He assigns the copyright to the GAP group. Many thanks to Burkhard for his help! Burkhard Höfling's port was partly based on an earlier port of GAP for the Mac, which was done by Dave Bayer (`dab@math.columbia.edu`) and used the Mac Programmers Workshop (MPW) compiler. Many thanks to Dave for his work. Moreover, the built-in editor is based upon the freeware text editor `PlainText` by Mel Park which, in turn, uses `TE32K`, a `TextEdit` replacement by Roy Wood. It also uses `Internet Config`.

For technical reasons we do not distribute the Macintosh specific source and project files as part of the standard archives. If you are interested in compiling GAP yourself, we are happy to provide you with the appropriate files (contact us at `support@gap-system.org`). The source can be compiled with CodeWarrior Pro 5 with Apple's Universal Headers 3.3 installed.

Please contact the author `b.hoefling@tu-bs.de` or `support@gap-system.org` if you need further information.

74 GAP Packages

The functionality of GAP can be extended by loading GAP packages. Many packages are distributed together with the core system of GAP consisting of the GAP kernel, the GAP library and the various data libraries.

GAP packages are written by (groups of) GAP users which may not be members of the GAP developer team. The responsibility and copyright of a GAP package remains with the original author(s).

GAP packages have their own documentation which is smoothly integrated into the GAP help system.

All GAP users who develop new code are invited to share the results of their efforts with other GAP users by making the code and its documentation available in form of a package. Information how to do this is available from the GAP Web pages (

`http://www.gap-system.org`

) and in the extension manual 4. There are possibilities to get a package distributed together with GAP and it is possible to submit a package to a formal refereeing process.

In this Chapter we describe how to use existing packages.

74.1 Installing a GAP Package

Before a package can be used it must be installed. With a standard installation of GAP there should be quite a few packages already available. But since GAP packages are released independently of the main GAP system it may be sensible to upgrade or install new packages between upgrades of your GAP installation.

A package consists of a collection of files within a single directory that must be a subdirectory of the `pkg` directory in one of the GAP root directories, see 9.2. (If you don't have access to the `pkg` directory in your main GAP installation you can add private root directories as explained in that section.)

Whenever you get from somewhere an archive of a GAP package it should be accompanied with a `README` file that explains its installation. Some packages just consist of GAP code and the installation is done by unpacking the archive in one of the places described above. There are also packages that need further installation steps, there may be for example some external programs which have to be compiled (this is often done by just saying `./configure; make` inside the unpacked package directory, but check the individual `README` files).

74.2 Loading a GAP Package

Some GAP packages are prepared for **automatic loading**, that is they will be loaded automatically with GAP, others must in each case be separately loaded by a call to `LoadPackage`.

1 ▶ `LoadPackage(` *name*`[,` *version*`] )` F
▶ `LoadPackage(` *name*`[,` *version*`,` *banner*`[,` *outercalls*`]] )` F

loads the GAP package with name *name*. If the optional version string *version* is given, the package will only be loaded in a version number at least as large as *version*, or equal to *version* if its first character is `=` (see 4.14 in "Extending GAP"). The argument *name* is case insensitive.

`LoadPackage` will return `true` if the package has been successfully loaded and will return `fail` if the package could not be loaded. The latter may be the case if the package is not installed, if necessary binaries have not been compiled, or if the version number of the available version is too small.

If the package *name* has already been loaded in a version number at least or equal to *version*, respectively, `LoadPackage` returns `true` without doing anything else.

If the optional third argument *banner* is `false` then no package banner is printed. The fourth argument *outercalls* is used only for recursive calls of `LoadPackage`, when the loading process for a package triggers the loading of other packages.

After a package has been loaded its code and documentation should be available as other parts of the GAP library are.

The documentation of each GAP package will tell you if the package loads automatically or not. Also, GAP prints the list of names of all GAP packages which have been loaded (either by automatic loading or via `LoadPackage` commands in one's `.gaprc` file or the like) at the end of the initialization process.

A GAP package may also install only its documentation automatically but still need loading by `LoadPackage`. In this situation the online help displays `(not loaded)` in the header lines of the manual pages belonging to this GAP package.

If for some reason you don't want certain packages to be automatically loaded, GAP provides three levels for disabling autoloading:

The autoloading of specific packages can be overwritten for the whole GAP installation by putting a file `NOAUTO` into a `pkg` directory that contains lines with the names of packages which should not be automatically loaded.

Furthermore, individual users can disable the autoloading of specific packages by using the following command in their `.gaprc` file (see 3.4).

```
ExcludeFromAutoload( pkgnames );
```

where *pkgnames* is the list of names of the GAP packages in question.

Using the `-A` command line option when starting up GAP (see 3.1), automatic loading is switched off, and the scanning of the `pkg` directories containing the installed packages is delayed until the first call of 74.2.1.

74.3 Functions for GAP Packages

The following functions are mainly used in files contained in a package and not by users of a package.

1 ▶ `ReadPackage( name, file )` F
▶ `ReadPackage( pkg-file )` F
▶ `RereadPackage( name, file )` F
▶ `RereadPackage( pkg-file )` F

In the first form, `ReadPackage` reads the file *file* of the GAP package *name*, where *file* is given as a path relative to the home directory of *name*. In the second form where only one argument *pkg-file* is given, this should be the path of a file relative to the `pkg` subdirectory of GAP root paths (see 9.2 in the GAP Reference Manual). Note that in this case, the package name is assumed to be equal to the first part of *pkg-file*, **so this form is not recommended**.

The absolute path is determined as follows. If the package in question has already been loaded then the file in the directory of the loaded version is read. If the package is available but not yet loaded then the directory given by `TestPackageAvailability` (see 74.3.2), without prescribed version number, is used. (Note that the `ReadPackage` call does **not** force the package to be loaded.)

If the file is readable then `true` is returned, otherwise `false`.

Each of *name*, *file* and *pkg-file* should be a string. The *name* argument is case insensitive.

`RereadPackage` does the same as `ReadPackage`, except that also read-only global variables are overwritten (cf 9.7.13 in the GAP Reference Manual).

2 ▶ `TestPackageAvailability( name, version )` F
▶ `TestPackageAvailability( name, version, intest )` F

For strings *name* and *version*, `TestPackageAvailability` tests whether the GAP package *name* is available for loading in a version that is at least *version*, or equal to *version* if the first character of *version* is `=`, see Section 4.14 of "Extending GAP" for details about version numbers.

The result is `true` if the package is already loaded, `fail` if it is not available, and the string denoting the GAP root path where the package resides if it is available, but not yet loaded. A test function (the value of the component `AvailabilityTest` in the `PackageInfo.g` file of the package) should therefore test for the result of `TestPackageAvailability` being not equal to `fail`.

The argument *name* is case insensitive.

The optional argument *intest* is a list of pairs `[ pkgnam, pkgversion ]` such that the function has been called with these arguments on outer levels. (Note that several packages may require each other, with different required versions.)

3 ▶ `InstalledPackageVersion( name )` F

If the GAP package with name *name* has already been loaded then `InstalledPackageVersion` returns the string denoting the version number of this version of the package. If the package is available but has not yet been loaded then the version number string for that version of the package that currently would be loaded. (Note that loading **another** package might force loading another version of the package *name*, so the result of `InstalledPackageVersion` will be different afterwards.) If the package is not available then `fail` is returned.

The argument *name* is case insensitive.

4 ▶ `DirectoriesPackageLibrary( name[, path] )` F

takes the string *name*, a name of a GAP package and returns a list of directory objects for the sub-directory/ies containing the library functions of this GAP package, up to one for each `pkg` sub-directory of a path in `GAPInfo.RootPaths`. The default is that the library functions are in the subdirectory `lib` of the GAP package's home directory. If this is not the case, then the second argument *path* needs to be present and must be a string that is a path name relative to the home directory of the GAP package with name *name*.

As an example, the following returns a directory object for the library functions of the GAP package `Example`:

```
gap> DirectoriesPackageLibrary( "Example", "gap" );
[ dir("/home/werner/gap/4.0/pkg/example/gap/") ]
```

Observe that we needed the second argument `"gap"` here, since `Example`'s library functions are in the sub-directory `gap` rather than `lib`.

In order to find a subdirectory deeper than one level in a package directory, the second argument is again necessary whether or not the desired subdirectory relative to the package's directory begins with `lib`. The directories in *path* should be separated by `/` (even on systems, like Windows, which use `\` as the directory separator). For example, suppose there is a package `somepackage` with a subdirectory `m11` in the directory `data`, then we might expect the following:

```
gap> DirectoriesPackageLibrary( "somepackage", "data/m11" );
[ dir("/home/werner/gap/4.0/pkg/somepackage/data/m11") ]
```

5 ▶ `DirectoriesPackagePrograms(` *name* `)` F

returns a list of the `bin/`*architecture* subdirectories of all packages *name* where *architecture* is the architecture on which GAP has been compiled. The directories returned by `DirectoriesPackagePrograms` are the place where external binaries for the GAP package *name* and the current architecture should be located.

```
gap> DirectoriesPackagePrograms( "nq" );
[ dir("/home/werner/gap/4.0/pkg/nq/bin/i686-unknown-linux2.0.30-gcc/") ]
```

6 ▶ `CompareVersionNumbers(` *supplied*`,` *required* `)` F
▶ `CompareVersionNumbers(` *supplied*`,` *required*`, "equal" )` F

compares two version numbers, given as strings. They are split at non-digit characters, the resulting integer lists are compared lexicographically. The routine tests whether *supplied* is at least as large as *required*, and returns `true` or `false` accordingly. A version number ending in `dev` is considered to be infinite. See Section 4.14 of "Extending GAP" for details about version numbers.

75 Replaced and Removed Command Names

In general we try to keep GAP 4 compatible with former releases as much as possible. Nevertheless, from time to time it seems appropriate to remove some commands or to change the names of some commands or variables. There are various reasons for that: Some functionality was improved and got another (hopefully better) interface, names turned out to be too special or too general for the underlying functionality, or names are found to be unintuitive or inconsistent with other names.

In this chapter we collect such old names while pointing to the sections which explain how to substitute them. Usually, old names will be available for several releases; they may be removed when they don't seem to be used any more.

75.1 Group Actions - Name Changes

The concept of a group action is sometimes referred to as a "group operation". In GAP 3 as well as in older versions of GAP 4 the term `Operation` was used instead of `Action`. We decided to change the names to avoid confusion with the term "operation" as in `DeclareOperation` and "Operations for Xyz".

Here are some examples of such name changes.

OLD	NOW USE
`Operation`	`Action`
`RepresentativeOperation`	`RepresentativeAction`
`OperationHomomorphism`	`ActionHomomorphism`
`FunctionOperation`	`FunctionAction`

75.2 Package Interface - Obsolete Functions and Name Changes

With GAP 4.4 the package interface was changed. Thereby some functions became obsolete and the names of some others were made more consistent.

The following functions are no longer needed: `DeclarePackage`, `DeclareAutoPackage`, `DeclarePackageDocumentation` and `DeclarePackageAutoDocumentation`. They are substituted by entries in the `PackageInfo.g` files, see "ref:the packageinfo.g file".

The following function names were changed.

OLD	NOW USE
`RequirePackage`	`LoadPackage`
`ReadPkg`	`ReadPackage`
`RereadPkg`	`RereadPackage`
`CreateCompletionFilesPkg`	`CreateCompletionFilesPackage`

75.3 Normal Forms of Integer Matrices - Name Changes

Former versions of GAP 4 documented several functions for computing the Smith or Hermite normal form of integer matrices. Some of them were never implemented and it was unclear which commands to use. The functionality of all of these commands is now available with `NormalFormIntMat` (see 25.2.9) and a few interface functions.

75.4 Miscellaneous Name Changes or Removed Names

In former releases of GAP 4 there were some global variable names bound to general information about the running GAP (path names, command line options, ...). Although they were not officially documented they were used by several users and in some packages. We mention here `BANNER` and `QUIET`. This type of information is now collected in a record with name `GAPInfo` and will become documented after a test phase.

Here are some further name changes.

OLD	NOW USE
`MonomialTotalDegreeLess`	`MonomialExtGrlexLess`
`NormedVectors`	`NormedRowVectors`

Bibliography

[AMW82] D[avid] G. Arrell, S[anjiv] Manrai, and M[ichael] F. Worboys. A procedure for obtaining simplified defining relations for a subgroup. In Campbell and Robertson [CR82], pages 155–159.

[AR84] D[avid] G. Arrell and E[dmund] F. Robertson. A modified Todd-Coxeter algorithm. In Atkinson [Atk84], pages 27–32.

[Art68] E[mil] Artin. *Galoissche Theorie*. Verlag Harri Deutsch, Frankfurt/Main, 1968.

[Atk84] Michael D. Atkinson, editor. *Computational Group Theory, Proceedings LMS Symposium on Computational Group Theory, Durham 1982*. Academic Press, 1984.

[Bak84] Alan Baker. *A concise introduction to the theory of numbers*. Cambridge University Press, 1984.

[BC76] M[ichael] J. Beetham and C[olin] M. Campbell. A note on the Todd-Coxeter coset enumeration algorithm. *Proc. Edinburgh Math. Soc. Edinburgh Math. Notes*, 20:73–79, 1976.

[BC89] Richard P. Brent and Graeme L. Cohen. A new lower bound for odd perfect numbers. *Math. Comp.*, 53:431–437, 1989.

[BC94] Ulrich Baum and Michael Clausen. Computing irreducible representations of supersolvable groups. *Math. Comput.*, 207:351–359, 1994.

[BCFS91] L[ászló] Babai, G[ene] Cooperman, L[arry] Finkelstein, and Á[kos] Seress. Nearly linear time algorithms for permutation groups with a small base. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC'91), Bonn 1991*, pages 200–209. ACM Press, 1991.

[BE99a] Hans Ulrich Besche and Bettina Eick. Construction of finite groups. *J. Symbolic Comput.*, 27(4):387–404, 1999.

[BE99b] Hans Ulrich Besche and Bettina Eick. The groups of order at most 1000 except 512 and 768. *J. Symbolic Comput.*, 27(4):405–413, 1999.

[BE01] Hans Ulrich Besche and Bettina Eick. The groups of order $q^n \cdot p$. *Comm. Alg.*, 29(4):1759–1772, 2001.

[BEOon] Hans Ulrich Besche, Bettina Eick, and E. A. O'Brien. A millennium project: constructing small groups, In preparation.

[Ber76] T. R. Berger. Characters and derived length in groups of odd order. *J. Algebra*, 39:199–207, 1976.

[Bes92] Hans Ulrich Besche. Die Berechnung von Charaktergraden und Charakteren endlicher auflösbarer Gruppen im Computeralgebrasystem GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992.

[BL96] Francis Buekenhout and Dimitri Leemans. On the list of finite primitive permutation groups of degree ≤ 50. *J. Symbolic Comput.*, 22(2):215–225, 1996.

[BL98] Thomas Breuer and Steve Linton. The GAP 4 type system. organizing algebraic algorithms. In Gloor [Glo98], pages 38–45.

[BM83] Gregory Butler and John McKay. The transitive groups of degree up to 11. *Comm. Algebra*, 11:863–911, 1983.

[Bou70] N. Bourbaki. *Éléments de Mathématique, Algèbre I*, volume 1. Hermann, Paris, 1970.

[BP98] Thomas Breuer and Götz Pfeiffer. Finding Possible Permutation Characters. *J. Symbolic Comput.*, 26:343–354, 1998.

[Bre91] Thomas Breuer. Potenzabbildungen, Untergruppenfusionen, Tafel-Automorphismen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1991.

[Bre97] Thomas Breuer. Integral bases for subfields of cyclotomic fields. *AAECC*, 8:279–289, 1997.

[Bre99] Thomas Breuer. Computing Possible Class Fusions from Character Tables. *Comm. Algebra*, 27(6):2733–2748, 1999.

[BTW93] Bernhard Beauzamy, Vilmar Trevisan, and Paul S. Wang. Polynomial factorization: Sharp bounds, Efficient algorithms. *J. Symbolic Comput.*, 15:393–413, 1993.

[Bur55] W[illiam S.] Burnside. *Theory of Groups of Finite Order*. Dover Publications, New York, 1955. Unabridged republication of the second edition, published in 1911.

[But93] Gregory Butler. The transitive groups of degree fourteen and fifteen. *J. Symbolic Comput.*, pages 413–422, 1993.

[Can73] John J. Cannon. Construction of defining relators for finite groups. *Discrete Math.*, pages 105–129, 1973.

[Car72] R. W. Carter. *Simple groups of Lie type*, volume 28 of *Pure and Applied Mathematics*. John Wiley and Sons, 1972.

[CCN+85] J[ohn] H. Conway, R[obert] T. Curtis, S[imon] P. Norton, R[ichard] A. Parker, and R[obert] A. Wilson. *Atlas of finite groups*. Oxford University Press, 1985.

[CHM98] John H. Conway, Alexander Hulpke, and John McKay. On transitive permutation groups. *LMS J. Comput. Math.*, 1:1–8, 1998.

[CLO97] David Cox, John Little, and Donal O'Shea. *Ideals, varieties, and algorithms*. Undergraduate Texts in Mathematics. Springer-Verlag, New York, second edition, 1997. An introduction to computational algebraic geometry and commutative algebra.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin, Heidelberg and New York, 1993.

[Con90a] S[am] B. Conlon. Calculating characters of p-groups. *J. Symbolic Comput.*, 9(5 & 6):535–550, 1990.

[Con90b] S[am] B. Conlon. Computing modular and projective character degrees of soluble groups. *J. Symbolic Comput.*, 9(5 & 6):551–570, 1990.

[CR82] Colin M. Campbell and Edmund F. Robertson, editors. *Groups-St.Andrews 1981, Proceedings of a conference, St.Andrews 1981*, volume 71 of *London Math. Soc. Lecture Note Series*. Cambridge University Press, 1982.

[Dix67] J[ohn] D. Dixon. High speed computations of group characters. *Numer. Math.*, 10:446–450, 1967.

[Dix93] John D. Dixon. Constructing representations of finite groups. In Larry Finkelstein and William M. Kantor, editors, *Proceedings of the 1991 DIMACS Workshop on Groups and Computation*, volume 11 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 105–112. American Mathematical Society, 1993.

[DM88] John D. Dixon and Brian Mortimer. The primitive permutation groups of degree less than 1000. *Math. Proc. Cambridge Philos. Soc.*, 103:213–238, 1988.

[Dre69] Andreas [W. M.] Dress. A characterization of solvable groups. *Math. Z.*, 110:213–217, 1969.

[EH02] Bettina Eich and Burkhard Höfling. The solvable primitive permutation groups of degree at most 6560. submitted, 2002.

[Eic97] Bettina Eick. Special presentations for finite soluble groups and computing (pre-)Frattini subgroups. In Larry Finkelstein and William M. Kantor, editors, *Proceedings of the 2nd DIMACS Workshop held at Rutgers University, New Brunswick, NJ, June 7–10, 1995*, volume 28 of *DIMACS Series in*

Discrete Mathematics and Theoretical Computer Science, pages 101–112. American Mathematical Society, 1997.

[FJNT95] V[olkmar] Felsch, D[avid] L. Johnson, J[oachim] Neubüser, and S[ergey] V. Tsaranov. The structure of certain Coxeter groups. In C[olin] M. Campbell, T[haddeus] C. Hurley, E[dmund] F. Robertson, S[ean] J. Tobin, and J[ames] J. Ward, editors, *Groups '93 Galway / St.Andrews, Galway 1993, Volume 1*, volume 211 of *London Math. Soc. Lecture Note Series*, pages 177–190. Cambridge University Press, 1995.

[FN79] Volkmar Felsch and Joachim Neubüser. An algorithm for the computation of conjugacy classes and centralizers in p-groups. In Edward W. Ng, editor, *Symbolic and Algebraic Computation (Proceedings of EUROSAM 79, An International Symposium on Symbolic and Algebraic Manipulation, Marseille, 1979)*, Lecture Notes in Computer Science, 72, pages 452–465. Springer, Berlin, Heidelberg and New York, 1979.

[Fra82] J[ames] S. Frame. Recursive computation of tensor power components. *Bayreuther Math. Schr.*, 10:153–159, 1982.

[Glo98] Oliver Gloor, editor. *Proceedings of the 1998 International Symposium on Symbolic and Algebraic Computation*. The Association for Computing Machinery, ACM Press, 1998.

[Hal59] Marshall Hall, Jr. *The theory of Groups*. Macmillan, 1959.

[Hav69] George Havas. Symbolic and algebraic calculation. Basser Computing Dept., Technical Report 89, Basser Department of Computer Science, University of Sydney, Sydney, Australia, 1969.

[Hav74] George Havas. A Reidemeister-Schreier program. In M[ichael] F. Newman, editor, *Proceedings of the Second International Conference on the Theory of Groups, Canberra, 1973*, volume 372 of *Lecture Notes in Math.*, pages 347–356. Springer, Berlin, Heidelberg and New York, 1974.

[HB82] B[ertram] Huppert and N[orman] Blackburn. *Endliche Gruppen II*, volume 1242 of *Grundlehren Math. Wiss.* Springer, Berlin, Heidelberg and New York, 1982.

[HIÖ89] Trevor [O.] Hawkes, I. M[artin] Isaacs, and M. Özaydin. On the Möbius function of a finite group. *Rocky Mountain J. Math.*, 19:1003–1034, 1989.

[HJLP] Gerhard Hiss, Christoph Jansen, Klaus Lux, and Richard [A.] Parker. Computational modular character theory.

`http://www.math.rwth-aachen.de/LDFM/homes/MOC/CoMoChaT/`.

[HKRR84] George Havas, P[eter] E. Kenne, J[ames] S. Richardson, and E[dmund] F. Robertson. A Tietze transformation program. In Atkinson [Atk84], pages 67–71.

[How76] J. M. Howie. *An introduction to semigroup theory*. Academic Press [Harcourt Brace Jovanovich Publishers], London, 1976. L.M.S. Monographs, No. 7.

[HP89] Derek F. Holt and W[ilhelm] Plesken. *Perfect Groups*. Oxford Math. Monographs. Oxford University Press, 1989.

[HR94] Derek [F.] Holt and Sarah Rees. Testing modules for irreducibility. *J. Austral. Math. Soc. Ser. A*, 57:1–16, 1994.

[Hul93] Alexander Hulpke. Zur Berechnung von Charaktertafeln. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, 1993.

[Hul96] Alexander Hulpke. *Konstruktion transitiver Permutationsgruppen*. Dissertation, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1996.

[Hul98] Alexander Hulpke. Computing normal subgroups. In Gloor [Glo98], pages 194–198.

[Hul99] Alexander Hulpke. Computing subgroups invariant under a set of automorphisms. *J. Symbolic Comput.*, 27(4):415–427, 1999.

[Hul00] Alexander Hulpke. Conjugacy classes in finite permutation groups via homomorphic images. *Math. Comp.*, 69(232):1633–1651, 2000.

[Hul01] Alexander Hulpke. Representing subgroups of finitely presented groups by quotient subgroups. *Experimental Mathematics*, 10(3):369–381, 2001.

[Hular] Alexander Hulpke. Constructing transitive permutation groups. *J. Symbolic Comput.*, to appear.

[Hum72] James E. Humphreys. *Introduction to Lie algebras and representation theory*. Springer-Verlag, New York, 1972. Graduate Texts in Mathematics, Vol. 9.

[Hum78] James E. Humphreys. *Introduction to Lie algebras and representation theory*. Springer-Verlag, New York, 1978. Second printing, revised.

[Hup67] B[ertram] Huppert. *Endliche Gruppen I*, volume 134 of *Grundlehren Math. Wiss.* Springer, Berlin, Heidelberg and New York, 1967.

[IE94] Hiroyuki Ishibashi and A. G. Earnest. Two-element generation of orthogonal groups over finite fields. *J. Algebra*, 165(1):164–171, 1994.

[Isa76] I. M. Isaacs. *Character theory of finite groups*, volume 69 of *Pure and applied mathematics*. Academic Press, New York, 1976. xii+303 pp., ISBN 0-12-374550-0.

[JLPW95] Christoph Jansen, Klaus Lux, Richard [A.] Parker, and Robert [A.] Wilson. *An Atlas of Brauer Characters*, volume 11 of *London Math. Soc. Monographs*. Oxford University Press, 1995.

[Joh97] D. L. Johnson. *Presentations of groups*. Cambridge University Press, Cambridge, second edition, 1997.

[Kau92] Ansgar Kaup. Gitterbasen und Charaktere endlicher Gruppen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992.

[KL90] Peter Kleidman and Martin Liebeck. *The subgroup structure of the finite classical groups*. Cambridge University Press, 1990.

[Kli66] A. U. Klimyk. Decomposition of the direct product of irreducible representations of semisimple Lie algebras into irreducible representations. *Ukrain. Mat. Ž.*, 18(5):19–27, 1966.

[Kli68] A. U. Klimyk. Decomposition of a direct product of irreducible representations of a semisimple Lie algebra into irreducible representations. In *American Mathematical Society Translations. Series 2*, volume 76, pages 63–73. American Mathematical Society, Providence, R.I., 1968.

[KLM01] Gregor Kemper, Frank Lübeck, and Kay Magaard. Matrix generators for the Ree groups ${}^2G_2(q)$. *Comm. Algebra*, 29(1):407–413, 2001.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, third edition, 1998.

[Lüb] Frank Lübeck. Conway polynomials for finite fields.

`http://www.math.rwth-aachen.de:8001/~Frank.Luebeck/data/ConwayPol`.

[Leo91] Jeffrey S. Leon. Permutation group algorithms based on partitions, I: theory and algorithms. *J. Symbolic Comput.*, 12:533–583, 1991.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:513–534, 1982.

[LNS84] R[einhard] Laue, J[oachim] Neubüser, and U[lrich] Schoenwaelder. Algorithms for finite soluble groups and the SOGOS system. In Atkinson [Atk84], pages 105–135.

[LP91] Klaus Lux and Herbert Pahlings. Computational aspects of representation theory of finite groups. In G. O. Michler and C. R. Ringel, editors, *Representation theory of finite groups and finite-dimensional algebras*, volume 95 of *Progress in Mathematics*, pages 37–64. Birkhäuser, Basel, 1991.

[Mac81] I. G. Macdonald. Numbers of conjugacy classes in some finite classical groups. *Bull. Austral. Math. Soc.*, 23(1):23–48, 1981.

[MN89] M[atthias] Mecky and J[oachim] Neubüser. Some remarks on the computation of conjugacy classes of soluble groups. *Bull. Austral. Math. Soc.*, 40(2):281–292, 1989.

[MS85] John McKay and Leonard H. Soicher. Computing Galois groups over the rationals. *J. Number Theory*, 20:273–281, 1985.

[Mur58] F[rancis] D. Murnaghan. The orthogonal and symplectic groups. Communications Series A 13, Dublin Inst. Adv. Studies, 1958.

[MV97] Meena Mahajan and V. Vinay. Determinant: combinatorics, algorithms, and complexity. *Chicago J. Theoret. Comput. Sci.*, pages Article 5, 26 pp. (electronic), 1997.

[Neb95] Gabriele Nebe. *Endliche rationale Matrixgruppen vom Grad 24*. Dissertation, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1995.

[Neb96] Gabriele Nebe. Finite subgroups of $GL_n(\mathbb{Q})$ for $25 \leq n \leq 31$. *Comm. Alg.*, 24 (7):2341–2397, 1996.

[Neu82] Joachim Neubüser. An elementary introduction to coset table methods in computational group theory. In Campbell and Robertson [CR82], pages 1–45.

[Neu92] Jürgen Neukirch. *Algebraische Zahlentheorie*. Springer, Berlin, Heidelberg and New York, 1992.

[New77] M[ichael] F. Newman. Determination of groups of prime-power order. In R. A. Bryce, J. Cossey, and M[ichael] F. Newman, editors, *Group theory, Proc. Miniconf., Austral. Nat. Univ., Canberra, 1975*, volume 573 of *Lecture Notes in Math.*, pages 73–84. Springer, Berlin, Heidelberg and New York, 1977.

[New90] M[ichael] F. Newman. Proving a group infinite. *Arch. Math. (Basel)*, 54(3):209–211, 1990.

[NP95a] G[abriele] Nebe and W[ilhelm] Plesken. *Finite rational matrix groups*, volume 556 of *AMS Memoirs*. American Mathematical Society, 1995.

[NP95b] G[abriele] Nebe and W[ilhelm] Plesken. *Finite rational matrix groups of degree 16*, pages 74–144. Volume 556 of *AMS Memoirs* [NP95a], 1995.

[NPP84] J[oachim] Neubüser, H[erbert] Pahlings, and W[ilhelm] Plesken. CAS; design and use of a system for the handling of characters of finite groups. In Atkinson [Atk84], pages 195–247.

[O'B90] E[amonn] A. O'Brien. The p-group generation algorithm. *J. Symbolic Comput.*, 9:677–698, 1990.

[O'B91] E[amonn] A. O'Brien. The groups of order 256. *J. Algebra*, 143:219–235, 1991.

[Pah93] Herbert Pahlings. On the Möbius function of a finite group. *Arch. Math. (Basel)*, 60:7–14, 1993.

[Par84] Richard Parker. The Computer Calculation of Modular Characters (the MeatAxe). In Atkinson [Atk84], pages 267–274.

[Pfe91] G. Pfeiffer. Von Permutationscharakteren und Markentafeln. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, 1991.

[Pfe97] G. Pfeiffer. The Subgroups of M_{24}, or How to Compute the Table of Marks of a Finite Group. *Experiment. Math.*, 6(3):247–270, 1997.

[Ple85] W[ilhelm] Plesken. Finite unimodular groups of prime degree and circulants. *J. Algebra*, 97:286–312, 1985.

[Ple90] W[ilhelm] Plesken. Additive decompositions of positive integral quadratic forms. The paper is available at Lehrstuhl B für Mathematik, Rheinisch Westfälische Technische Hochschule Aachen, may be it will be published in the near future, 1990.

[PN95] W[ilhelm] Plesken and G[abriele] Nebe. *Finite rational matrix groups*, pages 1–73. Volume 556 of *AMS Memoirs* [NP95a], 1995.

[Poh87] M[ichael] Pohst. A modification of the LLL reduction algorithm. *J. Symbolic Comput.*, 4:123–127, 1987.

[PP77] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of GL(n,Z). I. the five and seven dimensional cases, II. the six dimensional case. *Math. Comp.*, 31:536–576, 1977.

[PP80] Wilhelm Plesken and Michael Pohst. On maximal finite irreducible subgroups of GL(n,Z). III. the nine dimensional case, IV. remarks on even dimensions with application to n = 8, V. the eight dimensional case and a complete description of dimensions less than ten. *Math. Comp.*, 34:245–301, 1980.

[RDU02] Colva M. Roney-Dougal and William R. Unger. The affine primitive permutation groups of degree less than 1000. preprint, 2002.

[Rin93] Michael Ringe. *The C MeatAxe, Release 1.5*. Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1993.

[Rob88] E[dmund] F. Robertson. Tietze transformations with weighted substring search. *J. Symbolic Comput.*, 6:59–64, 1988.

[Roy87] Gordon F. Royle. The transitive groups of degree twelve. *J. Symbolic Comput.*, pages 255–268, 1987.

[Sch90] Gerhard J. A. Schneider. Dixon's character table algorithm revisited. *J. Symbolic Comput.*, 9:601–606, 1990.

[Sch92] Michael Scherner. Erweiterung einer Arithmetik von Kreisteilungskörpern auf deren Teilkörper und deren Implementation in GAP. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1992.

[Sch94] Ute Schiffer. Cliffordmatrizen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1994.

[Sco73] L. L. Scott. Modular permutation representations. *Trans. Amer. Math. Soc.*, 175:101–121, 1973.

[Sho92] Mark W. Short. *The Primitive Soluble Permutation Groups of Degree less than 256*, volume 1519 of *Lecture Notes in Math.* Springer, Berlin, Heidelberg and New York, 1992.

[Sim70] Charles C. Sims. Computational methods in the study of permutation groups. In John Leech, editor, *Computational Problems in Abstract Algebra, Proc. Conf. Oxford, 1967*, pages 169–183. Pergamon Press, Oxford, 1970.

[Sim90] Charles C. Sims. Computing the order of a solvable permutation group. *J. Symbolic Comput.*, 9:699–705, 1990.

[Sim94] C. C. Sims. *Computation with Finitely Presented Groups*. Cambridge University Press, 1994.

[Sim97] Charles C. Sims. Computing with subgroups of automorphism groups of finite groups. In Wolfgang Küchlin, editor, *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, pages 40–403. The Association for Computing Machinery, ACM Press, 1997.

[Sou94] Bernd Souvignier. Irreducible finite integral matrix groups of degree 8 and 10. *Math. Comp.*, 63:335–350, 1994.

[SPA89] Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany. *SPAS - Subgroup Presentation Algorithms System, version 2.5, User's reference manual*, 1989.

[Tay87] D. E. Taylor. Pairs of generators for matrix groups. I. *The Cayley Bulletin*, 3, 1987.

[The93] Heiko Theißen. Methoden zur Bestimmung der rationalen Konjugiertheit in endlichen Gruppen. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, 1993.

[The97] Heiko Theißen. *Eine Methode zur Normalisatorberechnung in Permutationsgruppen mit Anwendungen in der Konstruktion primitiver Gruppen*. Dissertation, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1997.

[vdW76] Robert W. van der Waall. On symplectic primitive modules and monomial groups. *Indagationes Math.*, 38:362–375, 1976.

[Wag90] Stan Wagon. Editor's corner: the Euclidean algorithm strikes again. *Amer. Math. Monthly*, 97(2):125–129, 1990.

[Wil] Robert A. Wilson. ATLAS of Finite Group Representations.

`http://www.mat.bham.ac.uk/atlas/`.

[Wil96] R[obert] A. Wilson. Standard generators for sporadic simple groups. *J. Algebra*, 184:505–515, 1996.

[Zag90] D. Zagier. A one-sentence proof that every prime $p \equiv 1 \pmod 4$ is a sum of two squares. *Amer. Math. Monthly*, 97(2):144, 1990.

[Zum89] Matthias Zumbroich. Grundlagen einer Arithmetik in Kreisteilungskörpern und ihre Implementation in CAS. Diplomarbeit, Lehrstuhl D für Mathematik, Rheinisch Westfälische Technische Hochschule, Aachen, Germany, 1989.

Index

This index covers only this manual. A page number in *italics* refers to a whole section which is devoted to the indexed subject. Keywords are sorted with case and spaces ignored, e.g., "`PermutationCharacter`" comes before "permutation group".

(Near-)Additive Magma Categories, *550*
(Near-)Additive Magma Generation, *551*
`+`, 48
`-`, 48
`-A`, 29
`-B`, 30
`-C`, 30
`-D`, 30
`-K`, 28
`-L`, 29
`-M`, 30
`-N`, 30
`-O`, 30
`-P`, 30
  on Macintosh, 31
`-R`, 29
`-T`, 30
`-U`, 30
`-W`, 30
  on Macintosh, 31
`-X`, 30
`-Y`, 30
`-a`, 29
  on Macintosh, 31
`-b`, 27
`-e`, 27
  on Macintosh, 31
`-f`, 27
  on Macintosh, 31
`-g`, 28
`-g -g`, 28
`-h`, 27
`-i`, 30
`-l`, 28
`-m`, 28
`-n`, 27
  on Macintosh, 31
`-o`, 28
  on Macintosh, 31
`-q`, 27
`-r`, 29
`-x`, 27
`-y`, 28
`-z`, 30
  on Macintosh, 31
`.gaprc`, 33
`/`, 48
  for character tables, 723
`*`, 48
  for character tables, 723
`\'`, 247
`\XYZ`, 247
`\"`, 247
`\\`, 247
`\b`, 247
`\c`, 247
`\in`, operation for testing membership, 271
`\n`, 247
`\r`, 247
`^`, 48
  for class functions, 761
1-Cohomology, *374*
2-Cohomology and Extensions, *452*

A

`AbelianGroup`, 504
`AbelianInvariants`, for character tables, 726
  for groups, 363
Abelian Invariants for Subgroups, *472*
`AbelianInvariantsMultiplier`, 377
`AbelianInvariantsNormalClosureFpGroup`, 473
`AbelianInvariantsNormalClosureFpGroupRrs`, 473
`AbelianInvariantsOfList`, 239
`AbelianInvariantsSubgroupFpGroup`, 472
`AbelianInvariantsSubgroupFpGroupMtc`, 472
`AbelianInvariantsSubgroupFpGroupRrs`, 472

`AbelianNumberField`, 582
abelian number field, 583
abelian number fields, canonicalbasis, 585
abelian number fields, Galois group, 586
`AbelianSubfactorAction`, 399
About Group Actions, *391*
`AbsInt`, 127
`AbsoluteIrreducibleModules`, 742
`AbsoluteValue`, 157
absolute value of an integer, 127
`AbsolutIrreducibleModules`, 742
abstract word, 321
`AbstractWordTietzeWord`, 481
accessing, list elements, 172
  record elements, 257
Accessing a Module, *687*
Accessing Record Elements, *257*
Accessing Subgroups via Tables of Marks, *708*
`AClosestVectorCombinationsMatFFEVecFFE`, 215
`AClosestVectorCombinationsMatFFEVec-FFECoords`, 215
`ActingAlgebra`, 625
`ActingDomain`, 403
Acting OnRight and OnLeft, *428*
`Action`, 398
action, by conjugation, 392
  on blocks, 392
  on sets, 392
`ActionHomomorphism`, 397
Action of a group on itself, *399*
Action on Subfactors Defined by a Pcgs, *443*
actions, 392
Actions of Matrix Groups, *425*
`ActorOfExternalSet`, 405
`Add`, 175
add, an element to a set, 193
`AddCoeffs`, 213
`AddGenerator`, 483
`AddGenerators`, 340
`AddGeneratorsExtendSchreierTree`, 420
addition, 48
  list and non-list, 183
  matrices, 219
  matrix and scalar, 219
  operation, 292
  rational functions, 661
  scalar and matrix, 219
  scalar and matrix list, 220
  scalar and vector, 211
  vector and scalar, 211
  vectors, 210
Additive Arithmetic for Lists, *182*
`AdditiveInverse`, 290
`AdditiveInverseAttr`, 290
`AdditiveInverseImmutable`, 290
`AdditiveInverseMutable`, 290
`AdditiveInverseOp`, 290
`AdditiveInverseSameMutability`, 290
`AdditiveInverseSM`, 290
`AdditiveNeutralElement`, 552
`AddRelator`, 483
`AddRowVector`, 213
`AddRule`, 340
`AddRuleReduced`, 340
`AddSet`, 193
`AdjointAssociativeAlgebra`, 642
`AdjointBasis`, 614
`AdjointMatrix`, 642
`AdjointModule`, 626
Advanced Features of GAP, *29*
Advanced List Manipulations, *201*
Advanced Methods for Dixon-Schneider Calculations, *743*
`AffineAction`, 444
`AffineActionLayer`, 444
`AffineOperation`, 444
`AffineOperationLayer`, 444
`Agemo`, 360
`Algebra`, 605
`AlgebraByStructureConstants`, 608
`AlgebraGeneralMappingByImages`, 619
`AlgebraHomomorphismByImages`, 619
`AlgebraHomomorphismByImagesNC`, 619
`AlgebraicExtension`, 682
`AlgebraWithOne`, 605
`AlgebraWithOneGeneralMappingByImages`, 619
`AlgebraWithOneHomomorphismByImages`, 620
`AlgebraWithOneHomomorphismByImagesNC`, 620
`AllBlocks`, 402
`AllIrreducibleSolvableGroups`, 522
`All`*Library*`Groups`, 510
`AllPrimitiveGroups`, 510
`AllSmallGroups`, 513
`AllTransitiveGroups`, 510
`Alpha`, 812
`AlternatingGroup`, 504

and, 168
  for filters, 117, 169
An Example of Advanced Dixon-Schneider Calculations, *744*
ANFAutomorphism, 587
AntiSymmetricParts, 778
antisymmetric relation, 311
Append, 175
AppendTo, 95
  for streams, 101
Apple, 828, 829
ApplicableMethod, 77
ApplicableMethod, *77*
ApplicableMethodTypes, 77
Apply, 196
ApplySimpleReflection, 639
ApproximateSuborbitsStabilizerPermGroup, 419
ARCH_IS_MAC, 34
ARCH_IS_UNIX, 34
ARCH_IS_WINDOWS, 34
arg, special function argument, 46, 56
Arithmetic for External Representations of Polynomials, *680*
Arithmetic for Lists, *180*
Arithmetic Operations for Class Functions, *759*
Arithmetic Operations for Elements, *292*
Arithmetic Operations for General Mappings, *304*
Arithmetic Operators, *48*
Arrangements, 147
arrow notation for functions, 57
AsAlgebra, 614
AsAlgebraWithOne, 615
AsBinaryRelationOnPoints, 312
AsBlockMatrix, 234
AscendingChain, 366
AsDivisionRing, 571
AsDuplicateFreeList, 195
AsField, 571
AsFreeLeftModule, 568
AsGroup, 345
AsGroupGeneralMappingByImages, 380
AsLeftIdeal, 558
AsLeftModule, 566
AsList, 267
AsMagma, 316
AsMonoid, 538
AsPolynomial, 663
AsRightIdeal, 558
AsRing, 555
AsSemigroup, 531
Assert, 80
AssertionLevel, 80
Assertions, *80*
AsSet, 267
AssignGeneratorVariables, 328
assignment, to a list, 173
  to a record, 258
  variable, 49
Assignments, *49*
AssignNiceMonomorphismAutomorphismGroup, 387
AssociatedPartition, 152
AssociatedReesMatrixSemigroupOfDClass, 537
Associates, 561
associativity, 48
AssocWordByLetterRep, 333
AsSortedList, 267
AsSSortedList, 267
As*Struct*, 284
AsSubalgebra, 615
AsSubalgebraWithOne, 615
AsSubgroup, 346
AsSubgroupOfWholeGroupByQuotient, 470
AsSubmagma, 316
AsSubmonoid, 538
AsSubsemigroup, 531
AsSubspace, 590
AsSub*struct*, 287
AsTransformation, 548
AsTransformationNC, 548
AsTwoSidedIdeal, 558
AsVectorSpace, 589
at exit functions, 72
ATLAS Irrationalities, *159*
AtlasIrrationality, 161
atomic irrationalities, 159
Attributes, *121*
Attributes and Operations for Algebras, *613*
Attributes and Properties for (Near-)Additive Magmas, *552*
Attributes and Properties for Collections, *268*
Attributes and Properties for Magmas, *318*
Attributes and Properties for Matrix Groups, *424*
Attributes and Properties of Character Tables, *724*
Attributes and Properties of Elements, *288*
Attributes of and Operations on Equivalence Relations, *313*

Attributes of Tables of Marks, *698*
`AttributeValueNotSet`, 122
`AugmentationIdeal`, 654
`AugmentedCosetTableInWholeGroup`, 465
`AugmentedCosetTableMtc`, 465
`AugmentedCosetTableRrs`, 465
Augmented Coset Tables and Rewriting, *465*
automatic loading of gap packages, 834
`AutomorphismDomain`, 386
`AutomorphismGroup`, 386
  for groups with pcgs, 445
automorphism group, of number fields, 586
Automorphisms and Equivalence of Character Tables, *751*
`AutomorphismsOfTable`, 727

B

b_N, 159
backslash character, 247
backspace character, 247
`Backtrace`, GAP3 name for `Where`, 70
Backtrack, *421*
`BANNER`, 839
`BaseFixedSpace`, 226
`BaseIntersectionIntMats`, 236
`BaseIntMat`, 236
`BaseMat`, 228
`BaseMatDestructive`, 228
`BaseOfGroup`, 418
`BaseOrthogonalSpaceMat`, 229
Bases of Vector Spaces, *591*
`BaseStabChain`, 418
`BaseSteinitzVectors`, 229
Basic Actions, *392*
Basic Groups, *503*
Basic Operations for Class Functions, *758*
Basic Operations for Lists, *171*
`BasicWreathProductOrdering`, 280
`Basis`, 592
`BasisNC`, 592
`BasisVectors`, 593
`Bell`, 145
`Bernoulli`, 146
`BestQuoInt`, 129
`BestSplittingMatrix`, 744
`BiAlgebraModule`, 623
`BiAlgebraModuleByGenerators`, 623
`BilinearFormMat`, 637
binary relation, 310
`BinaryRelationByElements`, 310
`BinaryRelationOnPoints`, 311
`BinaryRelationOnPointsNC`, 311
Binary Relations on Points, *311*
`BinaryRelationTransformation`, 549
`BindGlobal`, 45
`Binomial`, 145
blank, 40
`BlistList`, 206
Block Matrices, *234*
`BlockMatrix`, 234
`Blocks`, 402
`BlocksInfo`, 732
Block Systems, *402*
`BlownUpMat`, 231
`BlownUpVector`, 231
`BlowUpIsomorphism`, 425
BNF, 58
body, 55
`BombieriNorm`, 669
Boolean Lists Representing Subsets, *206*
bound, 42
Brauer character, 765
`BrauerCharacterValue`, 788
`BrauerTable`, 718
`BrauerTableOp`, 718
`BravaisGroup`, 428
`BravaisSubgroups`, 428
`BravaisSupergroups`, 428
Break, *54*
break loop message, 69
Break Loops, *67*
`break` statement, 54
browsing backwards, 22
browsing backwards one chapter, 22
browsing forward, 22
browsing forward one chapter, 22
browsing the next section browsed, 22
browsing the previous section browsed, 22
Browsing through the Sections, *22*
bug reports, see If Things Go Wrong, 826
Building new orderings, *276*

C

c_N, 159
Calculating with Group Automorphisms, *387*
Calendar Arithmetic, *254*

`CallFuncList`, 62
Calling a function with a list argument that is interpreted as several arguments, *62*
Cancellation Tests for Rational Functions, *680*
`CanComputeIndex`, 378
`CanComputeIsSubset`, 378
`CanComputeSize`, 377
`CanComputeSizeAnySubgroup`, 378
candidates, for permutation characters, 781, 783
`CanEasilyCompareElements`, 291
`CanEasilyCompareElementsFamily`, 291
`CanEasilyComputePcgs`, 431
`CanEasilySortElements`, 291
`CanEasilySortElementsFamily`, 291
`CanEasilyTestMembership`, 377
`CanonicalBasis`, 592
canonical basis, for matrix spaces, 599
  for row spaces, 599
`CanonicalGenerators`, 637
`CanonicalPcElement`, 433
`CanonicalPcgs`, 436
`CanonicalPcgsByGeneratorsWithImages`, 438
`CanonicalRepresentativeDeterminatorOfExternalSet`, 405
`CanonicalRepresentativeOfExternalSet`, 404
`CanonicalRightCosetElement`, 352
Carmichael's lambda function, 136
carriage return character, 247
`CartanMatrix`, 637
`CartanSubalgebra`, 634
`Cartesian`, 197
Categories, *118*
Categories and Properties of Algebras, *612*
Categories for Streams and the StreamsFamily, *97*
Categories of Associative Words, *326*
Categories of Matrices, *218*
`CategoriesOfObject`, 120
Categories of Words and Nonassociative Words, *321*
`CategoryCollections`, 263
`Center`, 319
center, 318
`CentralCharacter`, 768
central character, 768
`CentralIdempotentsOfAlgebra`, 617
centraliser, 318
`Centralizer`, 318
  for groups with pcgs, 445
`CentralizerInGLnZ`, 428
`CentralizerModulo`, 367
`CentralizerSizeLimitConsiderFunction`, 446
`CentralNormalSeriesByPcgs`, 439
`Centre`, 319
  for groups with pcgs, 445
centre, of a character, 767
`CentreOfCharacter`, 767
`CF`, 582
`ChangeStabChain`, 420
Changing Presentations, *482*
Changing the Help Viewer, *23*
Changing the Representation, *285*
Changing the Structure, *284*
`CHAR_INT`, 252
`CHAR_SINT`, 253
`Character`, 763
Character Conversion, *252*
`CharacterDegrees`, 724
Character Degrees and Derived Length, *812*
`Characteristic`, 288
characteristic, for class functions, 762
`CharacteristicPolynomial`, 230
characteristic polynomial, for field elements, 573
`CharacterNames`, 728
characters, 756
  permutation, 781, 783
  symmetrizations of, 777
`CharacterTable`, 718
Character Table Categories, *719*
`CharacterTableDirectProduct`, 746
`CharacterTableFactorGroup`, 747
`CharacterTableIsoclinic`, 748
character tables, 717
  access to, 717
  calculate, 717
  infix operators, 723
  of groups, 717
`CharacterTableWithSortedCharacters`, 749
`CharacterTableWithSortedClasses`, 750
`CharacterTableWreathSymmetric`, 748
character value, of group element using powering operator, 761
`CharsFamily`, 248
`CheckFixedPoints`, 802
`CheckForHandlingByNiceBasis`, 604
`CheckPermChar`, 809
`ChevalleyBasis`, 636
`ChiefNormalSeriesByPcgs`, 440

ChiefSeries, 364
ChiefSeriesThrough, 364
ChiefSeriesUnderAction, 364
ChineseRem, 130
Chinese remainder, 130
Chomp, 252
CIUnivPols, 661
ClassElementLattice, 369
classes, real, 729
ClassesSolvableGroup, 445
ClassFunction, 763
class function, 756
class function objects, 756
class functions, 799
  as ring elements, 760
ClassFunctionSameType, 764
Class Fusions between Character Tables, *794*
Classical Groups, *505*
ClassMultiplicationCoefficient, for character tables, 736
class multiplication coefficient, 736, 737
ClassNames, 728
ClassNamesTom, 700
ClassOrbit, 729
ClassPermutation, 750
ClassPositionsOfAgemo, 730
ClassPositionsOfCentre, for characters, 767
  for character tables, 730
ClassPositionsOfDerivedSubgroup, 730
ClassPositionsOfDirectProduct-Decompositions, 730
ClassPositionsOfElementaryAbelianSeries, 730
ClassPositionsOfFittingSubgroup, 730
ClassPositionsOfKernel, 767
ClassPositionsOfLowerCentralSeries, 730
ClassPositionsOfMaximalNormalSubgroups, 730
ClassPositionsOfNormalClosure, 730
ClassPositionsOfNormalSubgroup, 754
ClassPositionsOfNormalSubgroups, 730
ClassPositionsOfSupersolvableResiduum, 730
ClassPositionsOfUpperCentralSeries, 730
ClassRoots, 729
ClassStructureCharTable, 736
ClassTypesTom, 699
CleanedTailPcElement, 433
ClearCacheStats, 83
ClearProfile, 82
clone, an object, 114
CloseMutableBasis, 597
CloseStream, 98
ClosureGroup, 347
ClosureGroupAddElm, 348
ClosureGroupCompare, 348
ClosureGroupDefault, 348
ClosureGroupIntest, 348
ClosureLeftModule, 567
ClosureNearAdditiveGroup, 552
Closure Operations and Other Constructors, *312*
ClosureRing, 555
Closures of (Sub)groups, *347*
Closure*Struct*, 284
ClosureSubgroup, 348
ClosureSubgroupNC, 348
Coboundaries, 647
Cochain, 646
CochainSpace, 646
Cocycles, 647
cocycles, 374
CodePcGroup, 456
CodePcgs, 455
Coding a Pc Presentation, *455*
coefficient, binomial, 145
Coefficient List Arithmetic, *213*
Coefficients, 593
coefficients, for cyclotomics, 157
CoefficientsAndMagmaElements, 655
CoefficientsFamily, 677
CoefficientsMultiadic, 130
CoefficientsOfLaurentPolynomial, 669
CoefficientsOfUnivariatePolynomial, 663
CoefficientsOfUnivariateRationalFunction, 663
CoefficientsQadic, 130
CoefficientsRing, 670
CoeffsCyc, 157
CoeffsMod, 214
cohomology, 374
COHORTS_PRIMITIVE_GROUPS, 520
CoKernelOfAdditiveGeneralMapping, 307
CoKernelOfMultiplicativeGeneralMapping, 306
CollapsedMat, 805
Collected, 195
Collection Families, *263*
CollectionsFamily, 263
Coloring the Prompt and Input, *37*
ColorPrompt, 38

ColumnIndexOfReesMatrixSemigroupElement, 537
ColumnIndexOfReesZeroMatrixSemigroup-
Element, 537
Combinations, 147
Combinations, Arrangements and Tuples, *147*
CombinatorialCollector, 450
Combinatorial Numbers, *145*
Comm, 292
for words, 329
Command Line Options, *27*
comments, 40
CommutativeDiagram, 802
CommutatorFactorGroup, 367
CommutatorLength, 357
for character tables, 726
CommutatorSubgroup, 357
Compacted, 195
CompanionMat, 231
CompareVersionNumbers, 837
comparison, fp semigroup elements, 543
operation, 291
rational functions, 661
Comparison of Associative Words, *329*
Comparison of Class Functions, *759*
Comparison of Elements of Finitely Presented Groups, *458*
Comparison of Elements of Finitely Presented Semigroups, *543*
Comparison of Permutations, *407*
Comparison of Rational Functions, *661*
Comparison of Words, *322*
Comparison Operations for Elements, *291*
Comparisons, *47*
comparisons, of booleans, 167
of lists, 179
Comparisons of Booleans, *167*
Comparisons of Cyclotomics, *159*
Comparisons of Lists, *179*
Comparisons of Records, *260*
Comparisons of Strings, *249*
CompatibleConjugacyClasses, 722
CompatiblePairs, 454
Compilation, *821*
Compiling Library Code, *35*
Complementclasses, 357
ComplementclassesEA, 376
ComplementIntMat, 237
ComplementSystem, 360
Completion Files, *33*
ComplexConjugate, 162
for class functions, 762
ComplexificationQuat, 609
Components of a Dixon Record, *744*
CompositionMapping, 300
for Frobenius automorphisms, 580
CompositionMapping2, 300
CompositionMaps, 799
CompositionOfStraightLinePrograms, 337
CompositionSeries, 364
for groups with pcgs, 445
ComputedBrauerTables, 718
ComputedClassFusions, 795
ComputedIndicators, 735
ComputedIsPSolvableCharacterTables, 735
ComputedPowerMaps, 791
ComputedPrimeBlockss, 731
Computing a Pcgs, *430*
Computing a Permutation Representation, *410*
Computing Pc Groups, *451*
Computing Possible Permutation Characters, *783*
Computing the Irreducible Characters of a Group, *740*
Concatenation, 195
concatenation, of lists, 195
Conductor, 157
ConfluentRws, 340
Congruences, for character tables, 807
Congruences for semigroups, *533*
ConjugacyClass, 354
Conjugacy Classes, *354*
ConjugacyClasses, attribute, 354
for character tables, 721
for groups with pcgs, 445
for linear groups, 508
ConjugacyClassesByOrbits, 355
ConjugacyClassesByRandomSearch, 355
Conjugacy Classes in Classical Groups, *508*
Conjugacy Classes in Solvable Groups, *445*
ConjugacyClassesMaximalSubgroups, 368
ConjugacyClassesPerfectSubgroups, 371
ConjugacyClassesSubgroups, 368
ConjugacyClassSubgroups, 367
conjugate, matrix, 220
of a word, 329
ConjugateDominantWeight, 639
ConjugateDominantWeightWithWord, 639

`ConjugateGroup`, 345
`Conjugates`, 574
`ConjugateSubgroup`, 347
`ConjugateSubgroups`, 347
conjugation, 392
`ConjugatorAutomorphism`, 384
`ConjugatorAutomorphismNC`, 384
`ConjugatorIsomorphism`, 384
`ConjugatorOfConjugatorIsomorphism`, 385
`ConnectGroupAndCharacterTable`, 722
`ConsiderKernels`, 807
`ConsiderSmallerPowerMaps`, 808
`ConsiderStructureConstants`, 799
`ConsiderTableAutomorphisms`, 810
`ConstantTimeAccessList`, 191
constituent, of a group character, 766
`ConstituentsCompositionMapping`, 300
`ConstituentsOfCharacter`, 767
Constructing Algebras as Free Algebras, *606*
Constructing Algebras by Generators, *605*
Constructing Algebras by Structure Constants, *607*
Constructing Character Tables from Others, *746*
Constructing Domains, *283*
Constructing Lie algebras, *631*
Constructing Pc Groups, *449*
Constructing Subdomains, *287*
Constructing Tables of Marks, *694*
Constructing Vector Spaces, *589*
Construction of Abelian Number Fields, *582*
Construction of Stabilizer Chains, *415*
Constructors for Basic Groups, *509*
`ContainedCharacters`, 806
`ContainedDecomposables`, 806
`ContainedMaps`, 801
`ContainedPossibleCharacters`, 804, 805
`ContainedPossibleVirtualCharacters`, 804, 805
`ContainedSpecialVectors`, 804, 805
`ContainedTom`, 704
`ContainingTom`, 704
`ContinuedFractionApproximationOfRoot`, 141
`ContinuedFractionExpansionOfRoot`, 141
Continued Fractions, *141*
`continue` statement, 55
Conventions for Character Tables, *721*
convert, to a string, 248
Converting Groups to Finitely Presented Groups, *466*
`ConvertToCharacterTable`, 719
`ConvertToCharacterTableNC`, 719
`ConvertToMatrixRep`, 232
`ConvertToMatrixRepNC`, 232
`ConvertToRangeRep`, 204
`ConvertToStringRep`, 248
`ConvertToTableOfMarks`, 698
`ConvertToVectorRep`, 212
`ConvertToVectorRepNC`, 212
`ConwayPolynomial`, 580
Conway Polynomials, *580*
coprime, 48
copy, 114
  an object, 114
`CopyOptionsDefaults`, 420
Copyrights, *833*
`CopyStabChain`, 419
`Core`, 356
`CorrespondingGeneratorsByModuloPcgs`, 438
coset, 351
`CosetLeadersMatFFE`, 215
Cosets, *351*
`CosetTable`, 461
`CosetTableBySubgroup`, 462
`CosetTableDefaultLimit`, 463
`CosetTableDefaultMaxLimit`, 462
`CosetTableFromGensAndRels`, 462
`CosetTableInWholeGroup`, 465
`CosetTableOfFpSemigroup`, 546
Coset Tables and Coset Enumeration, *461*
Coset tables for subgroups in the whole group, *465*
`CosetTableStandard`, 464
CRC, 36
`CrcFile`, 95
  example, 36
CRC Numbers, *36*
`CreateCompletionFiles`, 34
`CreateCompletionFilesPkg`, 838
Creating Character Tables, *717*
Creating Class Functions from Values Lists, *763*
Creating Class Functions using Groups, *764*
Creating Finite Fields, *578*
Creating Finitely Presented Groups, *458*
Creating Finitely Presented Semigroups, *542*
Creating Group Homomorphisms, *379*
Creating Groups, *344*
Creating Mappings, *299*
Creating Permutations, *408*
Creating Presentations, *475*

Creation of Algebraic Extensions, *682*
Creation of Rational Functions, *679*
Credit, *21*
`CrystGroupDefaultAction`, 429
`Cycle`, 400
`CycleLength`, 400
`CycleLengths`, 400
`Cycles`, 400
`CycleStructureClass`, 768
`CycleStructurePerm`, 408
`CyclicExtensionsTom`, 704
`CyclicGroup`, 503
`CyclotomicField`, 582
cyclotomic field elements, 155
cyclotomic fields, canonicalbasis, 584
`CyclotomicPolynomial`, 667
Cyclotomic Polynomials, *667*
`Cyclotomics`, 155
cyclotomics, defaultfield, 583

D

d_N, 159
Darstellungsgruppe, see `EpimorphismSchurCover`, 376
`DataType`, 125
data type, unknown, 165
`DayDMY`, 254
`DaysInMonth`, 254
`DaysInYear`, 254
Debugging Recursion, *84*
DEC, 240
`DeclareAutoPackage`, 838
`DeclareHandlingByNiceBasis`, 603
`DeclareInfoClass`, 79
`DeclarePackage`, 838
`DeclarePackageAutoDocumentation`, 838
`DeclarePackageDocumentation`, 838
`DecodeTree`, 492
DecodeTree, *492*
decompose, a group character, 766
`DecomposedFixedPointVector`, 704
`DecomposeTensorProduct`, 647
`Decomposition`, 240
`DecompositionInt`, 241
`DecompositionMatrix`, 733
decomposition matrix, 240
Decompositions, *240*
`Decreased`, 775
`DefaultField`, 570
  for cyclotomics, 158
  for finite field elements, 578
`DefaultFieldByGenerators`, 571
`DefaultFieldOfMatrix`, 221
`DefaultFieldOfMatrixGroup`, 424
`DefaultRing`, 554
  for finite field elements, 578
`DefaultRingByGenerators`, 555
`DefaultStabChainOptions`, 417
Defining a Pcgs Yourself, *431*
`DefiningPolynomial`, 572
`DefiningQuotientHomomorphism`, 470
`DegreeFFE`, 577
`DegreeIndeterminate`, 665
`DegreeOfBinaryRelation`, 311
`DegreeOfCharacter`, 766
`DegreeOfLaurentPolynomial`, 664
`DegreeOfTransformation`, 547
`DegreeOfTransformationSemigroup`, 533
`DegreeOverPrimeField`, 572
`Delta`, 813
denominator, of a rational, 144
`DenominatorCyc`, 157
`DenominatorOfModuloPcgs`, 437
`DenominatorOfRationalFunction`, 662
`DenominatorRat`, 144
deprecated, 838
`DepthOfPcElement`, 433
`DepthOfUpperTriangularMatrix`, 230
`Derangements`, 149
`Derivations`, 632
`Derivative`, 666
`DerivedLength`, 365
`DerivedSeriesOfGroup`, 365
`DerivedSubgroup`, 357
`DerivedSubgroupsTom`, 703
`DerivedSubgroupsTomPossible`, 703
`DerivedSubgroupsTomUnique`, 703
`DerivedSubgroupTom`, 703
`DescriptionOfRootOfUnity`, 158
`Determinant`, 221
determinant character, 768
`DeterminantIntMat`, 240
`DeterminantMat`, 221
`DeterminantMatDestructive`, 222
`DeterminantMatDivFree`, 222
Determinant of an integer matrix, *240*

DeterminantOfCharacter, 768
Developing rewriting systems, *342*
DiagonalizeIntMat, 238
DiagonalizeMat, 227
DiagonalMat, 223
DiagonalOfMat, 229
Difference, 271
DifferenceBlist, 207
DihedralGroup, 504
Dimension, 568
DimensionOfHighestWeightModule, 648
DimensionOfMatrixGroup, 424
DimensionOfVectors, 598
DimensionsLoewyFactors, 366
DimensionsMat, 221
Directories, *91*
DirectoriesLibrary, 91
DirectoriesPackageLibrary, 836
DirectoriesPackagePrograms, 836
DirectoriesSystemPrograms, 92
Directory, 91
DirectoryContents, 92
DirectoryCurrent, 91
DirectoryTemporary, 91
DirectProduct, 498
DirectProductOp, 498
Direct Products, *498*
DirectSumDecomposition, 617
  for Lie algebras, 636
Direct Sum Decompositions, *635*
DirectSumOfAlgebraModules, 628
  for Lie algebras, 651
DirectSumOfAlgebras, 616
DisableAttributeValueStoring, 123
disable automatic loading, 834
Discriminant, 666
Display, 66
  for character tables, 737, 762
  for tables of marks, 696
DisplayCacheStats, 83
DisplayCompositionSeries, 364
DisplayEggBoxOfDClass, 535
DisplayImfInvariants, 524
DisplayInformationPerfectGroups, 517
DisplayOptions, 738
DisplayOptionsStack, 89
DisplayProfile, 82
DisplayRevision, 83
DistancesDistributionMatFFEVecFFE, 214
DistancesDistributionVecFFEsVecFFE, 214
DistanceVecFFE, 214
Distinguished Subalgebras, *633*
division, 48
  operation, 292
DivisionRingByGenerators, 571
division rings, 570
divisors, of an integer, 133
DivisorsInt, 133
Dixon-Schneider algorithm, 742
DixonInit, 743
DixonRecord, 743
DixonSplit, 743
DixontinI, 743
DMYDay, 254
DMYhmsSeconds, 255
DnLattice, 776
DnLatticeIterative, 777
do, 52
document formats (text, dvi, ps, pdf, html), 23
Domain, 288
DomainByGenerators, 288
Domain Categories, *285*
Domains, *111*
Domains Generated by Class Functions, *789*
Domains of Subspaces of Vector Spaces, *590*
DominantCharacter, 647
DominantWeights, 647
DoubleCoset, 353
DoubleCosetRepsAndSizes, 354
Double Cosets, *353*
DoubleCosets, operation, 353
DoubleCosetsNC, operation, 353
doublequote character, 247
doublequotes, 245
DownEnv, 70
Dummy Streams, *106*
duplicate free, 190
DuplicateFreeList, 195
Duplication of Lists, *177*
Duplication of Objects, *114*
DxIncludeIrreducibles, 744

E

E, 155
e_N, 159
EANormalSeriesByPcgs, 439

Earns, 401
EB, 159
EC, 159
Echelonized Matrices, *227*
ED, 159
Edit, 74
Editing Files, *74*
Editor Support, *74*
EE, 159
EF, 159
Efficiency of Homomorphisms, *382*
EG, 159
EggBoxOfDClass, 535
EH, 159
EI, 160
Eigenspaces, 226
Eigenvalues, 226
EigenvaluesChar, 769
Eigenvectors, 226
Eigenvectors and eigenvalues, *226*
EJ, 160
EK, 160
EL, 160
ElementaryAbelianGroup, 504
ElementaryAbelianSeries, 365
ElementaryAbelianSeriesLargeSteps, 365
Elementary Divisors, *226*
ElementaryDivisorsMat, 227
ElementaryDivisorsMatDestructive, 227
Elementary Operations for a Pcgs, *432*
Elementary Operations for a Pcgs and an Element, *432*
Elementary Operations for Integers, *126*
Elementary Operations for Rationals, *143*
Elementary Tietze Transformations, *486*
ElementOfFpGroup, 460
ElementOfFpSemigroup, 544
ElementOfMagmaRing, 655
ElementOrdersPowerMap, 792
ElementProperty, 421
Elements, 268
elements, definition, 110
  of a list or collection, 268
Elements as equivalence classes, *110*
ElementsFamily, 263
Elements in Algebraic Extensions, *682*
Elements of Free Magma Rings, *654*
Elements of pc groups, *448*
ElementsStabChain, 419
Elements with Prescribed Images, *396*
element test, for lists, 179
elif, 51
EliminatedWord, 331
EliminationOrdering, 675
else, 51
EM, 160
emacs, 74
Embedding, 301
  example for direct products, 498
  example for semidirect products, 500
  example for wreath products, 501
  for group products, 502
  for Lie algebras, 631
  for magma rings, 655
embeddings, find all, 388
Embeddings and Projections for Group Products, *502*
EmptyBinaryRelation, 310
EmptyMatrix, 223
EmptySCTable, 607
EmptyStabChain, 420
EnableAttributeValueStoring, 123
End, 601
end, 55
Enlarging Internally Represented Lists, *179*
Enumerator, 264
EnumeratorByBasis, 594
EnumeratorByFunctions, 264
Enumerators, *205*
EnumeratorSorted, 264
environment, 55
EpimorphismNilpotentQuotient, 472
EpimorphismPGroup, 472
EpimorphismQuotientSystem, 471
epimorphisms, find all, 388
EpimorphismSchurCover, 376
equality, associative words, 329
  elements of finitely presented groups, 458
  nonassociative words, 322
  of records, 260
  operation, 291
  pcwords, 448
Equality and Comparison of Domains, *283*
equality test, 47
  for permutations, 407
equivalence class, 313

Equivalence Classes, *313*
EquivalenceClasses, attribute, 314
EquivalenceClassOfElement, 314
EquivalenceClassOfElementNC, 314
EquivalenceClassRelation, 314
equivalence relation, 311, 312
EquivalenceRelationByPairs, 313
EquivalenceRelationByPairsNC, 313
EquivalenceRelationByPartition, 313
EquivalenceRelationByPartitionNC, 313
EquivalenceRelationByProperty, 313
EquivalenceRelationByRelation, 313
EquivalenceRelationPartition, 313
Equivalence Relations, *312*
ER, 160
Error, 72
Error, *72*
ErrorCount, 72
ErrorCount, *72*
ErrorNoTraceBack, 68
errors, syntax, 64
ES, 160
escaped characters, 247
escaping non-special characters, 247
ET, 160
EU, 160
EuclideanDegree, 562
EuclideanQuotient, 562
EuclideanRemainder, 562
Euclidean Rings, *562*
Euler's totient function, 136
EulerianFunction, 363
EulerianFunctionByTom, 704
EV, 160
EvalStraightLineProgElm, 338
EvalString, 254
evaluation, 42
  strings, 253
EW, 160
EX, 160
ExactSizeConsiderFunction, 373
Exec, 109
Exec, *109*
execution, 49
exit, 72
expanded form of monomials, 678
Expert Windows installation, *832*
Exponent, 363
  for character tables, 726
exponent, of the prime residue group, 136
exponentiation, operation, 292
ExponentOfPcElement, 432
ExponentsConjugateLayer, 434
ExponentsOfCommutator, 434
ExponentsOfConjugate, 434
ExponentsOfPcElement, 432
ExponentsOfRelativePower, 434
Exponents of Special Products, *434*
ExponentSumWord, 330
ExponentSyllable, 331
Expressing Group Elements as Words in Generators, *348*
Expressions, *42*
ExtendedPcgs, 436
ExtendStabChain, 420
Extension, 453
ExtensionNC, 453
ExtensionRepresentatives, 454
Extensions, 453
Extensions of the p-adic Numbers, *685*
ExteriorPowerOfAlgebraModule, 651
ExternalOrbit, 404
ExternalOrbits, 404
ExternalOrbitsStabilizers, 404
External Representation for Nonassociative Words, *325*
external representation of polynomials, 678
ExternalSet, 403
External Sets, *403*
ExternalSubset, 404
Extract, 773
ExtraspecialGroup, 504
ExtRepDenominatorRatFun, 679
ExtRepNumeratorRatFun, 679
ExtRepOfObj, external representation, for cyclotomics, 158
ExtRepPolynomialRatFun, 679
EY, 160

F

f_N, 159
FactorCosetAction, 399
  for fp groups, 462
FactorCosetOperation, 462
FactorFreeSemigroupByRelations, 542
FactorGroup, 367

`FactorGroupFpGroupByRels`, 458
`FactorGroupNC`, 367
`FactorGroupNormalSubgroupClasses`, 754
Factor Groups, *366*
Factor Groups of Polycyclic Groups - Modulo Pcgs, *436*
Factor Groups of Polycyclic Groups in their Own Representation, *438*
`FactorGroupTom`, 705
`Factorial`, 145
`Factorization`, 349
factorization, 348
`Factors`, 562
  of univariate polynomial, 667
`FactorsInt`, 132
`FactorsOfDirectProduct`, 747
`FactorsSquarefree`, 668
Fail, *167*
`fail`, 167
`FaithfulModule`, 627
  for Lie algebras, 645
Families, *116*
`FamiliesOfGeneralMappingsAndRanges`, 309
`FamiliesOfRows`, 753
`FamilyForOrdering`, 277
`FamilyObj`, 116
`FamilyPcgs`, 447
`FamilyRange`, 309
`FamilySource`, 309
FAQ, 824
features, under UNIX, 27
`fi`, 51
`Fibonacci`, 153
Fibonacci and Lucas Sequences, *153*
`Field`, 570
`FieldExtension`, 572
field homomorphisms, Frobenius, 580
`FieldOfMatrixGroup`, 424
`FieldOverItselfByGenerators`, 571
fields, 570
File Access, *93*
`FileDescriptorOfStream`, 98
`Filename`, 92
Filename, *92*
File Operations, *94*
File Streams, *103*
`Filtered`, 198
Filters, *117*
Filters Controlling the Arithmetic Behaviour of Lists, *181*
Finding Positions in Lists, *187*
Finding Submodules, *688*
`FindSl2`, 643
Finish Installation and Cleanup, *823*
Finite Field Elements, *576*
Finitely Presented Lie Algebras, *643*
Finitely presented monoids, *544*
finiteness test, for a list or collection, 268
Finite Perfect Groups, *515*
`First`, 199
`FittingSubgroup`, 358
`Flat`, 196
flush character, 247
For, *52*
`ForAll`, 200
`ForAny`, 200
`for` loop, 52
`FpElmComparisonMethod`, 459
`FpGroupPresentation`, 475
`FpGrpMonSmgOfFpGrpMonSmgElement`, 542
`FpLieAlgebraByCartanMatrix`, 644
frame, 778
`FrattiniSubgroup`, 358
  for groups with pcgs, 445
`FreeAlgebra`, 606
`FreeAlgebraWithOne`, 606
`FreeAssociativeAlgebra`, 606
`FreeAssociativeAlgebraWithOne`, 606
`FreeGeneratorsOfFpGroup`, 459
`FreeGeneratorsOfFpSemigroup`, 543
`FreeGeneratorsOfWholeGroup`, 459
`FreeGroup`, 326
`FreeGroupOfFpGroup`, 459
Free Groups, Monoids and Semigroups, *326*
`FreeLeftModule`, 568
`FreeLieAlgebra`, 632
`FreeMagma`, 324
`FreeMagmaRing`, 653
Free Magma Rings, *653*
Free Magmas, *324*
`FreeMagmaWithOne`, 324
Free Modules, *567*
`FreeMonoid`, 538
  with example, 327
`FreeMonoidOfRewritingSystem`, 546
`FreeSemigroup`, 327

with examples, 532
FreeSemigroupOfFpSemigroup, 543
FreeSemigroupOfRewritingSystem, 546
Frobenius automorphism, 580
FrobeniusAutomorphism, 580
FrobeniusAutomorphism, *580*
FrobeniusCharacterValue, 787
FullMatrixAlgebra, 609
FullMatrixAlgebraCentralizer, 617
FullMatrixLieAlgebra, 632
FullMatrixModule, 569
FullMatrixSpace, 598
FullRowModule, 569
FullRowSpace, 598
FullTransformationSemigroup, 533
Function, *55*
function, 55
FunctionAction, 403
function call, 46
with arguments, 46
with options, 46
Function Calls, *46*
FunctionOperation, 838
functions, 61, 299
definition by arrow notation, 57
definition of, 55
recursive, 55
with a variable number of arguments, 46, 56
FunctionsFamily, 63
Functions for Coding Theory, *214*
Functions for GAP Packages, *835*
Functions that do nothing, *62*
Function that Modify Boolean Lists, *208*
Function Types, *63*
FusionCharTableTom, 711
FusionConjugacyClasses, 794
FusionConjugacyClassesOp, 794
fusions, 794
FusionsAllowedByRestrictions, 810
FusionsTom, 700

G

G-sets, 391, 403
g_N, 159
gac, 34
Galois Action, *572*
Galois Conjugacy of Cyclotomics, *161*
GaloisCyc, 161
for class functions, 762
GaloisField, 579
GaloisGroup, of field, 573
of rational class of a group, 356
Galois Groups of Abelian Number Fields, *586*
GaloisMat, 163
GaloisStabilizer, 584
GaloisType, 668
gap.rc, 31, 33
GAP3, 33
Gap3CatalogueIdGroup, 513
GAP for Macintosh OS X, *828*
GAP for MacOS, *829*
GAPInfo, 839
GAPInfo.RootPaths, 29
GapInputPcGroup, 452
GapInputSCTable, 607
GAPKB_REW, 545
GAP Root Directory, *90*
GasmanLimits, 86
GasmanMessageStatus, 86
GasmanStatistics, 86
Gaussian algorithm, 225
GaussianIntegers, 588
GaussianRationals, 583
Gaussians, *588*
Gcd, 563
Gcd and Lcm, *563*
Gcdex, 130
GcdInt, 129
GcdOp, 563
GcdRepresentation, 563
GcdRepresentationOp, 563
General Binary Relations, *310*
GeneralisedEigenspaces, 226
GeneralisedEigenvalues, 226
generalized characters, 756
GeneralizedEigenspaces, 226
GeneralizedEigenvalues, 226
GeneralLinearGroup, 505
GeneralMappingByElements, 299
General Mappings, *308*
GeneralMappingsFamily, 309
GeneralOrthogonalGroup, 507
GeneralUnitaryGroup, 506
Generating Fields, *570*
Generating modules, *566*
Generating Rings, *554*

generator, of the prime residue group, 138
GeneratorsOfAdditiveGroup, 552
GeneratorsOfAdditiveMagma, 552
GeneratorsOfAdditiveMagmaWithZero, 552
GeneratorsOfAlgebra, 613
GeneratorsOfAlgebraModule, 624
GeneratorsOfAlgebraWithOne, 613
GeneratorsOfDivisionRing, 571
GeneratorsOfDomain, 287
GeneratorsOfEquivalenceRelationPartition, 313
GeneratorsOfField, 571
GeneratorsOfGroup, 345
GeneratorsOfIdeal, 557
GeneratorsOfLeftIdeal, 558
GeneratorsOfLeftModule, 566
GeneratorsOfLeftOperatorAdditiveGroup, 566
GeneratorsOfLeftVectorSpace, 590
GeneratorsOfMagma, 318
GeneratorsOfMagmaWithInverses, 318
GeneratorsOfMagmaWithOne, 318
GeneratorsOfMonoid, 538
GeneratorsOfNearAdditiveGroup, 552
GeneratorsOfNearAdditiveMagma, 552
GeneratorsOfNearAdditiveMagmaWithZero, 552
GeneratorsOfPresentation, 475
GeneratorsOfRightIdeal, 558
GeneratorsOfRightModule, 567
GeneratorsOfRightOperatorAdditiveGroup, 566
GeneratorsOfRing, 555
GeneratorsOfRingWithOne, 559
GeneratorsOfRws, 340
GeneratorsOfSemigroup, 531
GeneratorsOf*Struct*, 284
GeneratorsOfTwoSidedIdeal, 557
GeneratorsOfVectorSpace, 590
GeneratorsPrimeResidues, 137
GeneratorsSmallest, 374
GeneratorsSubgroupsTom, 709
GeneratorSyllable, 331
Generic Construction of Tables of Marks, *712*
GetFusionMap, 796
Get the Archives, *819*
getting help, 22
GF, 579
GL, 505
GL and SL, *425*
Global Memory Information, *86*
GModuleByMats, 687
GO, 507
GQuotients, 388
Grading, 618
Green's Relations, *534*
GreensDClasses, 535
GreensDClassOfElement, 535
GreensDRelation, 534
GreensHClasses, 535
GreensHClassOfElement, 535
GreensHRelation, 534
GreensJClasses, 535
GreensJClassOfElement, 535
GreensJRelation, 534
GreensLClasses, 535
GreensLClassOfElement, 535
GreensLRelation, 534
GreensRClasses, 535
GreensRClassOfElement, 535
GreensRRelation, 534
Groebner Bases, *676*
GroebnerBasis, 676
GroebnerBasisNC, 676
Group, 344
group actions, 391, 392
  operations syntax, 391
Group Actions - Name Changes, *838*
group algebra, 653
Group Automorphisms, *384*
GroupByRws, 450
GroupByRwsNC, 450
group characters, 756
Group Elements, *344*
GroupGeneralMappingByImages, 380
GroupHClassOfGreensDClass, 536
GroupHomomorphismByFunction, 380
GroupHomomorphismByImages, 379
GroupHomomorphismByImagesNC, 379
GroupOfPcgs, 432
group operations, 392, 838
Group Properties, *361*
GroupRing, 653
group ring, 653
Groups of Automorphisms, *386*
GroupStabChain, 418
GroupWithGenerators, 345
GU, 506

H

h_N, 159
HallSubgroup, 359
HallSystem, 360
  for groups with pcgs, 445
Handling of Streams in the Background, *106*
HasAbelianFactorGroup, 367
HasElementaryAbelianFactorGroup, 367
HasIndeterminateName, 660
HasParent, 286
HasseDiagramBinaryRelation, 312
HeadPcElementByNumber, 434
HenselBound, 669
hermite normal form, 838
HermiteNormalFormIntegerMat, 238
HermiteNormalFormIntegerMatTransform, 238
HeuristicCancelPolynomials, 681
HexStringInt, 250
HighestWeightModule, 650
History of Character Theory Stuff in GAP, *716*
HMSMSec, 255
Hom, 601
HomeEnumerator, 403
Homomorphism for very large groups, *383*
HomomorphismQuotientSemigroup, 534
homomorphisms, find all, 387
homomorphisms, Frobenius, field, 580
Homomorphisms of Algebras, *618*
HomomorphismTransformationSemigroup, 533
How to Implement New Kinds of Vector Spaces, *603*
HumanReadableDefinition, 707

I

i_N, 160
Ideal, 556
IdealByGenerators, 557
IdealNC, 557
Ideals, *611*
Ideals in Rings, *556*
Ideals of semigroups, *533*
Idempotents, 319
IdempotentsTom, 700
IdempotentsTomInfo, 700
Identical Lists, *176*
Identical Objects, *111*
Identical Records, *259*
IdentificationOfConjugacyClasses, 722
Identifier, for character tables, 728
  for tables of marks, 701
Identifiers, *41*
Identity, 288
IdentityBinaryRelation, 310
IdentityFromSCTable, 608
IdentityMapping, 301
IdentityMat, 222
IdentityTransformation, 547
IdFunc, 63
IdGap3SolvableGroup, 513
IdGroup, 513
IdSmallGroup, 513
IdsOfAllSmallGroups, 513
If, *51*
if statement, 51
If Things Go Wrong, *824*
Image, 302
  for Frobenius automorphisms, 580
image, vector under matrix, 220
ImageElm, 302
ImageListOfTransformation, 547
Images, 303
ImagesElm, 302
ImageSetOfTransformation, 548
ImagesRepresentative, 302
ImagesSet, 302
ImagesSmallestGenerators, 383
ImagesSource, 302
Images under Mappings, *302*
ImfInvariants, 526
ImfMatrixGroup, 527
ImfNumberQClasses, 524
ImfNumberQQClasses, 524
ImfNumberZClasses, 524
Immutable, 113
ImmutableBasis, 596
ImmutableMatrix, 232
in, for collections, 271
  for lists, 179
  for strictly sorted lists, 193
  operation for, 271
IndependentGeneratorsOfAbelianGroup, 374
Indeterminate, 659
IndeterminateName, 660
Indeterminateness, 804
IndeterminateNumberOfLaurentPolynomial, 670
IndeterminateNumberOfUnivariateRational-
  Function, 660

IndeterminateOfUnivariateRationalFunction, 660
Indeterminates, *659*
IndeterminatesOfPolynomialRing, 670
Index, 346
IndexInWholeGroup, 346
IndexNC, 346
Index numbers of primitive groups, *521*
Indicator, 735
IndicatorOp, 735
IndicesCentralNormalSteps, 439
IndicesChiefNormalSteps, 440
IndicesEANormalSteps, 439
IndicesInvolutaryGenerators, 464
IndicesNormalSteps, 441
IndicesOfAdjointBasis, 614
IndicesPCentralNormalStepsPGroup, 440
IndicesStabChain, 419
Indirected, 800
Induced Actions, *689*
InducedAutomorphism, 386
InducedClassFunction, 770
InducedClassFunctions, 770
InducedCyclic, 771
InducedPcgs, 435
InducedPcgsByGenerators, 435
InducedPcgsByGeneratorsNC, 435
InducedPcgsByPcSequence, 435
InducedPcgsByPcSequenceAndGenerators, 435
InducedPcgsByPcSequenceNC, 435
InducedPcgsWrtFamilyPcgs, 447
InducedPcgsWrtSpecialPcgs, 443
Inequalities, 787
inequality, of records, 260
inequality test, 47
InertiaSubgroup, 767
Infinity, *158*
infinity, 158
inflated class functions, 769
Info, 79
InfoAlgebra, 605
InfoAttributes, 123
InfoBckt, 421
InfoCharacterTable, 720
InfoCoh, 376
InfoComplement, 357
InfoCoset, 354
InfoFpGroup, 458
Info Functions, *79*
InfoGroebner, 677
InfoGroup, 345
InfoLattice, 371
InfoLevel, 79
InfoMatrix, 218
InfoMonomial, 812
InfoNumtheor, 136
InfoOptions, 89
InfoPcSubgroup, 373
Information about a function, *61*
Information about the version used, *83*
InfoText, 729
InfoTom, 698
InfoWarning, 80
InitFusion, 809
InitPowerMap, 807
InjectionZeroMagma, 317
InnerAutomorphism, 385
InnerAutomorphismNC, 385
InnerAutomorphismsAutomorphismGroup, 386
inner product, of group characters, 766
Input-Output Streams, *104*
InputLogTo, 95
  for streams, 102
  stop logging input, 95
InputOutputLocalProcess, 105
InputTextFile, 103
InputTextNone, 106
InputTextString, 104
InputTextUser, 103
InsertTrivialStabilizer, 420
InstallAtExit, 72
installation, 819
Installation of GAP for MacOS, *830*
Installation Overview, *819*
InstallCharReadHookFunc, 106
InstalledPackageVersion, 836
InstallFactorMaintenance, 294
InstallHandlingByNiceBasis, 603
Installing a GAP Package, *834*
InstallIsomorphismMaintenance, 294
InstallSubsetMaintenance, 294
Int, 126
  for cyclotomics, 156
  for strings, 253
INT_CHAR, 252
integer part of a quotient, 129

`Integers`, 126
Integral Bases of Abelian Number Fields, *584*
`IntegralizedMat`, 241
`IntegratedStraightLineProgram`, 337
`IntermediateGroup`, 366
`IntermediateSubgroups`, 366
Internally Represented Cyclotomics, *164*
Internally Represented Strings, *248*
`InterpolatedPolynomial`, 565
`IntersectBlist`, 208
`Intersection`, 270
  for groups with pcgs, 445
intersection, of collections, 270
  of sets, 194
`Intersection2`, 270
`IntersectionBlist`, 207
`IntersectionsTom`, 705
`IntersectSet`, 194
`IntFFE`, 578
`IntFFESymm`, 578
`IntHexString`, 253
`IntScalarProducts`, 804, 805
`IntVecFFE`, 578
`InvariantBilinearForm`, 426
`InvariantElementaryAbelianSeries`, 365
Invariant Forms, *426*, *690*
`InvariantLattice`, 428
`InvariantQuadraticForm`, 426
`InvariantSesquilinearForm`, 426
`InvariantSubgroupsElementaryAbelianGroup`, 372
`Inverse`, 290
inverse, group homomorphism, 381
  matrix, 220
  of class function, 761
`InverseAttr`, 290
`InverseClasses`, 729
`InverseGeneralMapping`, 300
`InverseImmutable`, 290
`InverseMap`, 799
`InverseMatMod`, 234
`InverseMutable`, 290
`InverseOp`, 290
`InverseRepresentative`, 419
`InverseSameMutability`, 290
`InverseSM`, 290
Invoking the Help, *22*
`Irr`, 724
irrationalities, 155
`IrrBaumClausen`, 740
`IrrConlon`, 740
`IrrDixonSchneider`, 740
Irreducibility Tests, *688*
irreducible character, 765
irreducible characters, computation, 743
`IrreducibleDifferences`, 772
Irreducible Maximal Finite Integral Matrix Groups, *522*
`IrreducibleModules`, 742
  for groups with pcgs, 445
`IrreducibleRepresentations`, 741
`IrreducibleRepresentationsDixon`, 742
`IrreducibleSolvableGroup`, 522
`IrreducibleSolvableGroupMS`, 521
Irreducible Solvable Matrix Groups, *521*
`Is16BitsFamily`, 333
`Is32BitsFamily`, 333
`Is8BitsFamily`, 333
`IsAbelian`, 319
  for character tables, 726
`IsAbelianNumberField`, 583
`IsAbelianNumberFieldPolynomialRing`, 672
`IsAbelianTom`, 702
`IsAdditiveElement`, 295
`IsAdditiveElementWithInverse`, 295
`IsAdditiveElementWithZero`, 295
`IsAdditiveGroup`, 551
`IsAdditiveGroupGeneralMapping`, 307
`IsAdditiveGroupHomomorphism`, 307
`IsAdditivelyCommutative`, 552
`IsAdditivelyCommutativeElement`, 297
`IsAdditivelyCommutativeElementCollColl`, 297
`IsAdditivelyCommutativeElementCollection`, 297
`IsAdditivelyCommutativeElementFamily`, 297
`IsAdditiveMagma`, 550
`IsAdditiveMagmaWithInverses`, 551
`IsAdditiveMagmaWithZero`, 550
`IsAlgebra`, 612
`IsAlgebraGeneralMapping`, 308
`IsAlgebraHomomorphism`, 308
`IsAlgebraicElement`, 683
`IsAlgebraicExtension`, 682
`IsAlgebraModuleElement`, 624
`IsAlgebraModuleElementCollection`, 624
`IsAlgebraModuleElementFamily`, 624

IsAlgebraWithOne, 612
IsAlgebraWithOneGeneralMapping, 308
IsAlgebraWithOneHomomorphism, 308
IsAlphaChar, 249
IsAlternatingGroup, 412
IsAnticommutative, 560
IsAntisymmetricBinaryRelation, 311
IsAssociated, 560
IsAssociative, 319
IsAssociativeElement, 297
IsAssociativeElementCollColl, 297
IsAssociativeElementCollection, 297
IsAssocWord, 326
IsAssocWordWithInverse, 326
IsAssocWordWithOne, 326
IsAutomorphismGroup, 386
IsBasicWreathLessThanOrEqual, 329
IsBasicWreathProductOrdering, 281
IsBasis, 592
IsBasisByNiceBasis, 603
IsBasisOfAlgebraModuleElementSpace, 625
IsBergerCondition, 813
IsBijective, 302
IsBinaryRelation, 310
  same as IsEndoGeneralMapping, 310
IsBLetterAssocWordRep, 332
IsBLetterWordsFamily, 332
IsBlist, 206
IsBlockMatrixRep, 234
IsBool, 167
IsBound, for lists, 175
IsBound and Unbind for Lists, *175*
IsBound and Unbind for Records, *261*
IsBoundGlobal, 45
IsBrauerTable, 719
IsBravaisGroup, 428
IsBuiltFromAdditiveMagmaWithInverses, 341
IsBuiltFromGroup, 341
IsBuiltFromMagma, 341
IsBuiltFromMagmaWithInverses, 341
IsBuiltFromMagmaWithOne, 341
IsBuiltFromSemigroup, 341
IsCanonicalBasis, 595
IsCanonicalBasisFullMatrixModule, 599
IsCanonicalBasisFullRowModule, 599
IsCanonicalNiceMonomorphism, 384
IsCanonicalPcgs, 436
IsCentral, 319
IsChar, 245
IsCharacter, 765
IsCharacteristicSubgroup, 346
IsCharacterTable, 719
IsCharacterTableInProgress, 719
IsCharCollection, 245
IsCheapConwayPolynomial, 581
IsClassFunction, 756
IsClassFusionOfNormalSubgroup, 735
IsClosedStream, 97
IsCochain, 645
IsCochainCollection, 645
IsCollection, 263
IsCollectionFamily, 263
IsCommutative, 319
IsCommutativeElement, 297
IsCommutativeElementCollColl, 297
IsCommutativeElementCollection, 297
IsCompositionMappingRep, 300
IsConfluent, 339
  for pc groups, 450
IsConjugacyClassSubgroupsByStabilizerRep, 368
IsConjugacyClassSubgroupsRep, 368
IsConjugate, 356
IsConjugatorAutomorphism, 385
IsConjugatorIsomorphism, 385
IsConstantRationalFunction, 663
IsConstantTimeAccessGeneralMapping, 308
IsConstantTimeAccessList, 171
IsContainedInSpan, 597
IsCopyable, 112
IsCyc, 155
IsCyclic, 361
  for character tables, 726
IsCyclicTom, 702
IsCyclotomic, 155
IsCyclotomicField, 584
IsCyclotomicMatrixGroup, 427
IsDenseList, 170
IsDiagonalMat, 222
IsDigitChar, 249
IsDirectoryPath, 93
IsDistributive, 560
IsDivisionRing, 570
IsDomain, 287
IsDoneIterator, 274
IsDoubleCoset, 354

IsDuplicateFree, 190
IsDuplicateFreeList, 190
IsDxLargeGroup, 744
IsElementaryAbelian, 361
IsElementOfFpMonoid, 542
IsElementOfFpSemigroup, 542
IsElementOfFreeMagmaRing, 654
IsElementOfFreeMagmaRingCollection, 654
IsElementOfFreeMagmaRingFamily, 654
IsElementOfMagmaRingModuloRelations, 656
IsElementOfMagmaRingModuloRelations-Collection, 656
IsElementOfMagmaRingModuloRelationsFamily, 656
IsElementOfMagmaRingModuloSpanOfZeroFamily, 656
IsEmpty, 268
IsEmptyString, 248
IsEndOfStream, 100
IsEndoGeneralMapping, 308
  same as IsBinaryRelation, 310
IsEqualSet, 193
IsEquivalenceClass, 313
IsEquivalenceRelation, 311
IsEuclideanRing, 562
IsEvenInt, 127
IsExecutableFile, 93
IsExistingFile, 93
IsExtAElement, 295
IsExternalOrbit, 404
IsExternalSet, 403
IsExternalSubset, 404
IsExtLElement, 295
IsExtRElement, 295
IsFamilyPcgs, 447
IsFFE, 576
IsFFECollColl, 576
IsFFECollection, 576
IsField, 570
IsFieldControlledByGaloisGroup, 573
IsFieldHomomorphism, 308
IsFinite, 268
  for character tables, 726
IsFiniteDimensional, 568
  for matrix algebras, 613
IsFiniteFieldPolynomialRing, 672
IsFinitelyGeneratedGroup, 362
IsFiniteOrderElement, 297
IsFiniteOrderElementCollColl, 297
IsFiniteOrderElementCollection, 297
IsFiniteOrdersPcgs, 432
IsFixedStabilizer, 420
IsFLMLOR, 612
IsFLMLORWithOne, 612
IsFpGroup, 458
IsFpMonoid, 542
IsFpSemigroup, 541
IsFreeGroup, 327
IsFreeLeftModule, 567
IsFreeMagmaRing, 654
IsFreeMagmaRingWithOne, 654
IsFromFpGroupGeneralMappingByImages, 390
IsFromFpGroupHomomorphismByImages, 390
IsFromFpGroupStdGensGeneralMappingByImages, 390
IsFromFpGroupStdGensHomomorphismByImages, 390
IsFullHomModule, 602
IsFullMatrixModule, 569
IsFullRowModule, 569
IsFullSubgroupGLorSLRespectingBilinearForm, 426
IsFullSubgroupGLorSLRespectingQuadratic-Form, 427
IsFullSubgroupGLorSLRespectingSesquilinear-Form, 426
IsFullTransformationSemigroup, 533
IsFunction, 63
IsGaussianIntegers, 588
IsGaussianRationals, 583
IsGaussianSpace, 597
IsGaussInt, 158
IsGaussRat, 158
IsGeneralizedDomain, 287
IsGeneralizedRowVector, 181
IsGeneralLinearGroup, 425
IsGeneralMapping, 308
IsGeneralMappingFamily, 309
IsGeneratorsOf*Struct*, 284
IsGL, 425
IsGreensClass, 535
IsGreensDClass, 535
IsGreensDRelation, 534
IsGreensHClass, 535
IsGreensHRelation, 534
IsGreensJClass, 535

IsGreensJRelation, 534
IsGreensLClass, 535
IsGreensLessThanOrEqual, 535
IsGreensLRelation, 534
IsGreensRClass, 535
IsGreensRelation, 534
IsGreensRRelation, 534
IsGroup, 345
IsGroupGeneralMapping, 306
IsGroupGeneralMappingByAsGroupGeneral-
MappingByImages, 389
IsGroupGeneralMappingByImages, 389
IsGroupGeneralMappingByPcgs, 390
IsGroupHClass, 536
IsGroupHomomorphism, 306
IsGroupOfAutomorphisms, 386
IsGroupRing, 654
IsHandledByNiceBasis, 569
for vector spaces, 603
IsHandledByNiceMonomorphism, 384
IsHasseDiagram, 311
IsHomogeneousList, 171
IsIdempotent, 290
IsIdenticalObj, 111
IsIncomparableUnder, 277
IsInducedFromNormalSubgroup, 815
IsInducedPcgs, 435
IsInducedPcgsWrtSpecialPcgs, 443
IsInfBitsFamily, 333
IsInfinity, 158
IsInjective, 301
IsInnerAutomorphism, 385
IsInputOutputStream, 104
IsInputStream, 97
IsInputTextNone, 97
IsInputTextStream, 97
IsInt, 126
IsIntegerMatrixGroup, 427
IsIntegers, 126
IsIntegralBasis, 595
IsIntegralCyclotomic, 156
IsIntegralRing, 559
IsInternallyConsistent, 115
for character tables, 734
for tables of marks, 702
IsIrreducibleCharacter, 765
IsIrreducibleRingElement, 561
IsIterator, 274
IsJacobianElement, 297
IsJacobianElementCollColl, 297
IsJacobianElementCollection, 297
IsJacobianRing, 560
IsLaurentPolynomial, 663
IsLaurentPolynomialDefaultRep, 679
IsLDistributive, 559
IsLeftAlgebraModuleElement, 624
IsLeftAlgebraModuleElementCollection, 624
IsLeftIdeal, 557
IsLeftIdealInParent, 557
IsLeftModule, 566
IsLeftModuleGeneralMapping, 307
IsLeftModuleHomomorphism, 307
IsLeftOperatorAdditiveGroup, 566
IsLeftSemigroupIdeal, 533
IsLeftVectorSpace, 589
IsLessThanOrEqualUnder, 277
IsLessThanUnder, 277
IsLetterAssocWordRep, 332
IsLetterWordsFamily, 332
IsLexicographicallyLess, 196
IsLieAbelian, 635
IsLieAlgebra, 612
IsLieMatrix, 219
IsLieNilpotent, 635
IsLieObject, 630
IsLieObjectCollection, 630
IsLieSolvable, 635
IsLinearMapping, 307
IsLinearMappingsModule, 602
IsList, 170
IsListDefault, 181
IsListOrCollection, 264
IsLowerAlphaChar, 249
IsLowerTriangularMat, 222
IsMagma, 315
IsMagmaHomomorphism, 305
IsMagmaRingModuloRelations, 656
IsMagmaRingModuloSpanOfZero, 656
IsMagmaWithInverses, 315
IsMagmaWithInversesIfNonzero, 315
IsMagmaWithOne, 315
IsMapping, 301
IsMatchingSublist, 190
IsMatrix, 218
IsMatrixGroup, 424
IsMatrixModule, 569

IsMatrixSpace, 597
IsMinimalNonmonomial, 818
IsModuloPcgs, 437
IsMonoid, 538
IsMonomial, for characters, 815
  for character tables, 726
  for groups, 815
  for positive integers, 816
IsMonomialGroup, 361
IsMonomialMatrix, 222
IsMonomialNumber, 816
IsMonomialOrdering, 672
IsMultiplicativeElement, 295
IsMultiplicativeElementWithInverse, 296
IsMultiplicativeElementWithOne, 295
IsMultiplicativeElementWithZero, 295
IsMultiplicativeGeneralizedRowVector, 181
IsMultiplicativeZero, 319
IsMutable, 113
IsMutableBasis, 596
IsNaturalAlternatingGroup, 411
IsNaturalGL, 425
IsNaturalGLnZ, 427
IsNaturalSL, 426
IsNaturalSLnZ, 428
IsNaturalSymmetricGroup, 411
IsNearAdditiveElement, 295
IsNearAdditiveElementWithInverse, 295
IsNearAdditiveElementWithZero, 295
IsNearAdditiveGroup, 550
IsNearAdditiveMagma, 550
IsNearAdditiveMagmaWithInverses, 550
IsNearAdditiveMagmaWithZero, 550
IsNearlyCharacterTable, 719
IsNearRingElement, 296
IsNearRingElementWithInverse, 296
IsNearRingElementWithOne, 296
IsNegRat, 144
IsNilpotent, for character tables, 726
  for groups with pcgs, 445
IsNilpotentElement, 642
IsNilpotentGroup, 361
IsNilpotentTom, 702
IsNonassocWord, 322
IsNonassocWordCollection, 322
IsNonassocWordWithOne, 322
IsNonassocWordWithOneCollection, 322
IsNonnegativeIntegers, 126
IsNonSPGeneralMapping, 309
IsNonTrivial, 268
IsNormal, 346
IsNormalBasis, 595
IsNotIdenticalObj, 112
IsNumberField, 583
IsObject, 110
IsOddInt, 127
isomorphic, pc group, 451, 452
IsomorphicSubgroups, 388
IsomorphismFpAlgebra, 621
IsomorphismFpGroup, 466
  for subgroups of fp groups, 469
IsomorphismFpGroupByGenerators, 467
IsomorphismFpGroupByGeneratorsNC, 467
IsomorphismFpGroupByPcgs, 449
IsomorphismFpSemigroup, 542
IsomorphismGroups, 387
IsomorphismMatrixAlgebra, 621
IsomorphismPcGroup, 452
IsomorphismPermGroup, 410
  for Imf matrix groups, 529
IsomorphismPermGroupImfGroup, 530
IsomorphismReesMatrixSemigroup, 537
IsomorphismRefinedPcGroup, 451
Isomorphism*RepStruct*, 285
isomorphisms, find all, 388
IsomorphismSCAlgebra, 622
IsomorphismSimplifiedFpGroup, 469
IsomorphismSpecialPcGroup, 452
IsomorphismTransformationSemigroup, 533
IsomorphismTypeInfoFiniteSimpleGroup, 361
IsOne, 290
IsOperation, 63
IsOrdering, 276
IsOrderingOnFamilyOfAssocWords, 277
IsOrdinaryMatrix, 218
IsOrdinaryTable, 719
IsOutputStream, 97
IsOutputTextNone, 98
IsOutputTextStream, 97
IsPadicExtensionNumber, 685
IsPadicExtensionNumberFamily, 685
IsParentPcgsFamilyPcgs, 447
IsPartialOrderBinaryRelation, 311
IsPcGroup, 448
IsPcGroupGeneralMappingByImages, 390
IsPcGroupHomomorphismByImages, 390

IsPcgs, 431
IsPcgsCentralSeries, 439
IsPcgsChiefSeries, 440
IsPcgsElementaryAbelianSeries, 439
IsPcgsPCentralSeriesPGroup, 440
IsPerfect, for character tables, 726
IsPerfectGroup, 361
IsPerfectTom, 702
IsPerm, 406
IsPermCollColl, 406
IsPermCollection, 406
IsPermGroup, 410
IsPermGroupGeneralMappingByImages, 390
IsPermGroupHomomorphismByImages, 390
IsPGroup, 362
IsPNilpotent, 363
IsPolycyclicGroup, 361
IsPolynomial, 663
IsPolynomialDefaultRep, 679
IsPolynomialFunction, 662
IsPolynomialFunctionsFamily, 677
IsPolynomialRing, 671
IsPosInt, 126
IsPositiveIntegers, 126
IsPosRat, 143
IsPreimagesByAsGroupGeneralMappingByImages, 390
IsPreOrderBinaryRelation, 311
IsPrime, 561
IsPrimeField, 572
IsPrimeInt, 131
IsPrimeOrdersPcgs, 432
IsPrimePowerInt, 132
IsPrimitive, 401
IsPrimitiveCharacter, 814
IsPrimitivePolynomial, 664
IsPrimitiveRootMod, 138
IsProbablyPrimeInt, 131
IsPseudoCanonicalBasisFullHomModule, 602
IsPSolvable, 363
IsPSolvableCharacterTable, 735
IsPSolvableCharacterTableOp, 735
IsPurePadicNumber, 684
IsPurePadicNumberFamily, 684
IsQuasiPrimitive, 814
IsQuaternion, 613
IsQuaternionCollColl, 613
IsQuaternionCollection, 613
IsQuickPositionList, 205
IsQuotientSemigroup, 534
IsRange, 204
IsRat, 143
IsRationalFunction, 662
IsRationalFunctionDefaultRep, 678
IsRationalFunctionsFamily, 677
IsRationalMatrixGroup, 427
IsRationals, 143
IsRationalsPolynomialRing, 672
IsRDistributive, 560
IsReadableFile, 93
IsReadOnlyGlobal, 44
IsRecord, 257
IsRecordCollColl, 257
IsRecordCollection, 257
IsReduced, 340
IsReductionOrdering, 278
IsReesCongruence, 534
IsReesCongruenceSemigroup, 532
IsReesMatrixSemigroup, 536
IsReesMatrixSemigroupElement, 536
IsReesZeroMatrixSemigroup, 536
IsReesZeroMatrixSemigroupElement, 536
IsReflexiveBinaryRelation, 310
IsRegular, 401
IsRegularDClass, 536
IsRegularSemigroup, 532
IsRegularSemigroupElement, 532
IsRelativelySM, 817
IsRestrictedLieAlgebra, 640
IsRewritingSystem, 339
IsRightAlgebraModuleElement, 624
IsRightAlgebraModuleElementCollection, 624
IsRightCoset, 352
IsRightIdeal, 557
IsRightIdealInParent, 557
IsRightModule, 566
IsRightOperatorAdditiveGroup, 566
IsRightSemigroupIdeal, 533
IsRing, 554
IsRingElement, 296
IsRingElementWithInverse, 296
IsRingElementWithOne, 296
IsRingGeneralMapping, 308
IsRingHomomorphism, 308
IsRingWithOne, 558
IsRingWithOneGeneralMapping, 308

IsRingWithOneHomomorphism, 308
IsRootSystem, 636
IsRootSystemFromLieAlgebra, 637
IsRowModule, 569
IsRowSpace, 597
IsRowVector, 210
IsScalar, 296
IsSemiEchelonized, 598
IsSemigroup, 531
IsSemigroupCongruence, 533
IsSemigroupIdeal, 533
IsSemiRegular, 401
IsSet, 191
IsShortLexLessThanOrEqual, 329
IsShortLexOrdering, 279
IsSimple, for character tables, 726
IsSimpleAlgebra, 612
IsSimpleGroup, 361
IsSimpleSemigroup, 532
IsSingleValued, 301
IsSL, 425
IsSolvable, for character tables, 726
IsSolvableGroup, 361
IsSolvableTom, 702
IsSortedList, 190
IsSpecialLinearGroup, 425
IsSpecialPcgs, 442
IsSPGeneralMapping, 309
IsSporadicSimple, for character tables, 726
IsSSortedList, 191
IsStandardGeneratorsOfGroup, 708
IsStraightLineProgElm, 338
IsStraightLineProgram, 334
IsStream, 97
IsString, 245
IsStringRep, 248
Is*Struct*, 285
IsSubgroup, 346
IsSubgroupFpGroup, 458
IsSubgroupOfWholeGroupByQuotientRep, 470
IsSubgroupSL, 426
IsSubmonoidFpMonoid, 541
IsSubnormal, 347
IsSubnormallyMonomial, 817
IsSubsemigroupFpSemigroup, 541
IsSubset, 270
IsSubsetBlist, 207
IsSubsetLocallyFiniteGroup, 362
IsSubsetSet, 193
IsSubspacesVectorSpace, 590
IsSub*struct*, 287
IsSupersolvable, for character tables, 726
  for groups with pcgs, 445
IsSupersolvableGroup, 361
IsSurjective, 302
IsSyllableAssocWordRep, 333
IsSyllableWordsFamily, 333
IsSymmetricBinaryRelation, 310
IsSymmetricGroup, 412
IsTable, 171
IsTableOfMarks, 698
IsTableOfMarksWithGens, 709
IsToPcGroupGeneralMappingByImages, 390
IsToPcGroupHomomorphismByImages, 390
IsToPermGroupGeneralMappingByImages, 390
IsToPermGroupHomomorphismByImages, 390
IsTotal, 301
IsTotalOrdering, 277
IsTransformation, 547
IsTransformationCollection, 547
IsTransformationMonoid, 533
IsTransformationSemigroup, 533
IsTransitive, for characters, 768
  for class functions, 768
  for group actions, 400
IsTransitiveBinaryRelation, 310
IsTranslationInvariantOrdering, 277
IsTrivial, 268
IsTuple, 299
IsTwoSidedIdeal, 557
IsTwoSidedIdealInParent, 557
IsUEALatticeElement, 648
IsUEALatticeElementCollection, 648
IsUEALatticeElementFamily, 648
IsUniqueFactorizationRing, 559
IsUnit, 560
IsUnivariatePolynomial, 663
IsUnivariatePolynomialRing, 672
IsUnivariateRationalFunction, 663
IsUnknown, 165
IsUpperAlphaChar, 249
IsUpperTriangularMat, 222
IsValidIdentifier, 42
IsVector, 296
IsVectorSpace, 589
IsVirtualCharacter, 765

IsWeightLexOrdering, 280
IsWeightRepElement, 649
IsWeightRepElementCollection, 649
IsWeightRepElementFamily, 649
IsWellFoundedOrdering, 276
IsWeylGroup, 638
IsWholeFamily, 269
IsWLetterAssocWordRep, 332
IsWLetterWordsFamily, 332
IsWord, 321
IsWordCollection, 322
IsWordWithInverse, 321
IsWordWithOne, 321
IsWreathProductOrdering, 281
IsWritableFile, 93
IsZero, 290
IsZeroGroup, 532
IsZeroSimpleSemigroup, 532
IsZeroSquaredElement, 297
IsZeroSquaredElementCollColl, 297
IsZeroSquaredElementCollection, 297
IsZeroSquaredRing, 560
IsZmodnZObj, 134
IsZmodnZObjNonprime, 134
IsZmodpZObj, 134
IsZmodpZObjLarge, 134
IsZmodpZObjSmall, 134
Iterated, 201
Iterator, 273
IteratorByBasis, 594
IteratorByFunctions, 275
IteratorList, 274
Iterators, *273*
IteratorSorted, 274

J

j_N, 160
Jacobi, 138
JenningsLieAlgebra, 641
JenningsSeries, 366
JoinEquivalenceRelations, 313
JoinStringsWithSeparator, 251
JordanDecomposition, 230

K

k_N, 160
KappaPerp, 642
KB_REW, 545
KernelOfAdditiveGeneralMapping, 307
KernelOfCharacter, 767
KernelOfMultiplicativeGeneralMapping, 306
KernelOfTransformation, 548
Keywords, *41*
KillingMatrix, 642
KnownAttributesOfObject, 121
Known Problems of the Configure Process, *826*
KnownPropertiesOfObject, 124
KnownTruePropertiesOfObject, 124
KnowsHowToDecompose, 378
KnuthBendixRewritingSystem, 545
Krasner-Kaloujnine theorem, 502
KroneckerProduct, 224
KuKGenerators, 502

L

l_N, 160
Lambda, 136
Language Overview, *39*
larger or equal, 47
larger test, 47
LargestElementGroup, 374
LargestElementStabChain, 419
LargestMovedPoint, 407
LargestUnknown, 165
last, 64
LastSystemError, 90
LaTeXStringDecompositionMatrix, 733
lattice base reduction, 241, 242
lattice basis reduction, for virtual characters, 772
LatticeByCyclicExtension, 371
LatticeGeneratorsInUEA, 648
Lattice Reduction, *241*
LatticeSubgroups, 369
LatticeSubgroupsByTom, 695
LaurentPolynomialByCoefficients, 669
LaurentPolynomialByExtRep, 680
Laurent Polynomials, *669*
LClassOfHClass, 535
Lcm, 564
LcmInt, 130
LcmOp, 564
LeadCoeffsIGS, 436
LeadingCoefficient, 665
LeadingCoefficientOfPolynomial, 673
LeadingExponentOfPcElement, 433
LeadingMonomial, 665
LeadingMonomialOfPolynomial, 673

`LeadingTermOfPolynomial`, 673
Leaving GAP, *72*
`LeftActingAlgebra`, 625
`LeftActingDomain`, 567
`LeftActingRingOfIdeal`, 558
`LeftAlgebraModule`, 623
`LeftAlgebraModuleByGenerators`, 622
left cosets, 352
`LeftDerivations`, 632
`LeftIdeal`, 556
`LeftIdealByGenerators`, 557
`LeftIdealNC`, 557
`LeftModuleByGenerators`, 567
`LeftModuleByHomomorphismToMatAlg`, 626
`LeftModuleGeneralMappingByImages`, 600
`LeftModuleHomomorphismByImages`, 600
`LeftModuleHomomorphismByImagesNC`, 600
`LeftModuleHomomorphismByMatrix`, 601
`LeftQuotient`, 292
  for words, 329
`LeftShiftRowVector`, 214
legacy, 838
`Legendre`, 139
`Length`, 191
  of an associative word, 330
length, of a word, 330
`LengthsTom`, 699
`LenstraBase`, 586
`LessThanFunction`, 277
`LessThanOrEqualFunction`, 277
`LetterRepAssocWord`, 333
`LevelsOfGenerators`, 281
`LeviMalcevDecomposition`, 618
  for Lie algebras, 636
Lexical Structure, *40*
`LexicographicOrdering`, 278
`LGFirst`, 443
`LGLayers`, 442
`LGLength`, 443
`LGWeights`, 442
library tables, 717
`LieAlgebra`, 631
`LieAlgebraByStructureConstants`, 631
`LieBracket`, 292
`LieCenter`, 633
`LieCentralizer`, 633
`LieCentre`, 633
`LieCoboundaryOperator`, 646
`LieDerivedSeries`, 634
`LieDerivedSubalgebra`, 633
`LieFamily`, 630
`LieLowerCentralSeries`, 634
`LieNilRadical`, 634
`LieNormalizer`, 633
`LieObject`, 630
Lie objects, *630*
`LieSolvableRadical`, 634
`LieUpperCentralSeries`, 635
`LiftedInducedPcgs`, 438
`LiftedPcElement`, 438
`LinearAction`, 444
`LinearActionLayer`, 444
`LinearCharacters`, 725
`LinearCombination`, 594
`LinearCombinationPcgs`, 433
Linear equations over the integers and Integral Matrices, *236*
`LinearIndependentColumns`, 240
Linear Mappings, *307*
`LinearOperation`, 444
`LinearOperationLayer`, 444
Line Editing, *73*
`LinesOfStraightLineProgram`, 335
`List`, 198
list and non-list, difference, 183
  left quotient, 186
  mod, 185
  product, 184
  quotient, 185
List Assignment, *173*
list assignment, operation, 172
`ListBlist`, 206
list boundedness test, operation, 172
List Categories, *170*
list element, access, 172
  assignment, 173
  operation, 172
List Elements, *172*
list equal, comparison, 179
`ListN`, 201
list of available books, 23
`ListPerm`, 408
Lists and Collections, *264*
list smaller, comparison, 180
`ListStabChain`, 419
list unbind, operation, 172

ListWithIdenticalEntries, 187
ListX, 202
LLL, 772
LLL algorithm, for Gram matrices, 242
  for vectors, 241
  for virtual characters, 772
LLLReducedBasis, 241
LLLReducedGramMat, 242
LoadDynamicModule, 35
Loading a GAP Package, *834*
loading a saved workspace, 37
LoadPackage, 834
local, 55
logarithm, discrete, 137
  of a root of unity, 158
LogFFE, 578
logical, 167
logical operations, 168
LogInt, 128
LogMod, 137
LogModShanks, 137
LogTo, 95
  for streams, 102
  stop logging, 95
LongestWeylWordPerm, 639
loop, read eval print, 64
loop, for, 52
  repeat, 52
  while, 51
loop over iterator, 54
loop over object, 54
loop over range, 53
loops, leaving, 54
  restarting, 55
LowercaseString, 250
LowerCentralSeriesOfGroup, 365
Low Index Subgroups, *465*
LowIndexSubgroupsFpGroup, 465
Low Level Routines to Modify and Create Stabilizer Chains, *419*
Lucas, 153

M

m_N, 160
Macintosh, 828, 829
MacOS, 829
Magma, 315
MagmaByGenerators, 316
MagmaByMultiplicationTable, 317
Magma Categories, *315*
MagmaElement, 317
Magma Generation, *315*
MagmaHomomorphismByFunctionNC, 305
Magma Homomorphisms, *305*
MagmaRingModuloSpanOfZero, 656
Magma Rings modulo Relations, *656*
Magma Rings modulo the Span of a Zero Element, *656*
Magmas Defined by Multiplication Tables, *317*
MagmaWithInverses, 316
MagmaWithInversesByGenerators, 316
MagmaWithInversesByMultiplicationTable, 317
MagmaWithOne, 315
MagmaWithOneByGenerators, 316
MagmaWithOneByMultiplicationTable, 317
Main Loop, *64*
MakeConfluent, 340
MakeImmutable, 113
MakeReadOnlyGlobal, 44
MakeReadWriteGlobal, 44
Making transformation semigroups, *533*
Manual Conventions, *20*
map, parametrized, 799
MappedWord, 323
MappingByFunction, 299
MappingPermListList, 409
Mappings that Respect Addition, *306*
Mappings that Respect Multiplication, *305*
Mappings which are Compatible with Algebraic Structures, *305*
maps, 790
MarksTom, 698
MatAlgebra, 609
MatClassMultCoeffsCharTable, 737
MathieuGroup, 505
MatLieAlgebra, 632
matrices, commutator, 220
Matrices as Basis of a Row Space, *228*
Matrices as Linear Mappings, *230*
Matrices over Finite Fields, *231*
Matrices Representing Linear Equations and the Gaussian Algorithm, *225*
MatrixAlgebra, 609
MatrixAutomorphisms, 751
matrix automorphisms, 793
MatrixByBlockMatrix, 235

Matrix Constructions, *222*
Matrix Groups in Characteristic 0, *427*
`MatrixLieAlgebra`, 632
`MatrixOfAction`, 625
matrix spaces, 597
`MatScalarProducts`, 766
`MatTom`, 701
`MaximalAbelianQuotient`, 367
`MaximalBlocks`, 402
`MaximalNormalSubgroups`, 369
`MaximalSubgroupClassReps`, 368
`MaximalSubgroups`, 368
  for groups with pcgs, 445
`MaximalSubgroupsLattice`, 369
`MaximalSubgroupsTom`, 705
`Maximum`, 197
`MaximumList`, 197
MeatAxe Modules, *687*
`MeetEquivalenceRelations`, 313
`MeetMaps`, 802
Membership Test for Collections, *271*
Membership Test for Lists, *179*
`MinimalElementCosetStabChain`, 419
`MinimalGeneratingSet`, 374
  for groups with pcgs, 445
`MinimalNonmonomialGroup`, 818
Minimal Nonmonomial Groups, *818*
`MinimalNormalSubgroups`, 369
`MinimalPolynomial`, 667
  over a field, 573
  over a ring, 667
Minimal Polynomials, *667*
`MinimalStabChain`, 417
`MinimalSupergroupsLattice`, 370
`MinimalSupergroupsTom`, 706
`MinimizedBombieriNorm`, 669
`Minimum`, 197
`MinimumList`, 197
`MinusCharacter`, 808
Miscellaneous, *142*
Miscellaneous Name Changes or Removed Names, *839*
mod, integers, 134
  laurent polynomials, 661
  lists, 185
  rationals, 48
`mod`, 48
  arithmetic operators, 48
  for character tables, 723
  residue class rings, 133
modular inverse, 48
modular remainder, 48
modular roots, 140
`ModuleByRestriction`, 627
Module Constructions, *687*
Module Homomorphisms, *689*
`ModuleOfExtension`, 453
Modules over Lie Algebras and Their Cohomology, *645*
Modules over Semisimple Lie Algebras, *647*
modulo, 48
  arithmetic operators, 48
  for pcgs, 437
  residue class rings, 133
`ModuloPcgs`, 437
`MoebiusMu`, 141
`MoebiusTom`, 701
Molien Series, *779*
`MolienSeries`, 779
`MolienSeriesInfo`, 780
`MolienSeriesWithGivenDenominator`, 781
`Monoid`, 538
`MonoidByGenerators`, 538
`MonoidByMultiplicationTable`, 539
`MonoidOfRewritingSystem`, 546
`MonomialComparisonFunction`, 673
`MonomialExtGrlexLess`, 675
`MonomialExtrepComparisonFun`, 674
`MonomialGrevlexOrdering`, 674
`MonomialGrlexOrdering`, 674
`MonomialLexOrdering`, 674
Monomial Orderings, *672*
`MonomialTotalDegreeLess`, 839
monomorphisms, find all, 388
`MorClassLoop`, 388
More about Boolean Lists, *208*
More About Global Variables, *44*
More about Tables of Marks, *693*
`MostFrequentGeneratorFpGroup`, 464
`MovedPoints`, 407
Moved Points of Permutations, *407*
`MTX.BasesCompositionSeries`, 689
`MTX.BasesMaximalSubmodules`, 688
`MTX.BasesMinimalSubmodules`, 688
`MTX.BasesMinimalSupermodules`, 689
`MTX.BasesSubmodules`, 688

MTX.BasisInOrbit, 690
MTX.BasisRadical, 688
MTX.BasisSocle, 688
MTX.CollectedFactors, 689
MTX.CompositionFactors, 689
MTX.DegreeSplittingField, 688
MTX.Dimension, 688
MTX.Distinguish, 690
MTX.Field, 688
MTX.Generators, 688
MTX.Homomorphism, 690
MTX.Homomorphisms, 690
MTX.InducedAction, 689
MTX.InducedActionFactorMatrix, 689
MTX.InducedActionFactorModule, 689
MTX.InducedActionMatrix, 689
MTX.InducedActionMatrixNB, 689
MTX.InducedActionSubmodule, 689
MTX.InducedActionSubmoduleNB, 689
MTX.InvariantBilinearForm, 690
MTX.InvariantQuadraticForm, 690
MTX.InvariantSesquilinearForm, 690
MTX.IsAbsolutelyIrreducible, 688
MTX.IsEquivalent, 689
MTX.IsIrreducible, 688
MTX.Isomorphism, 690
MTX.NormedBasisAndBaseChange, 689
MTX.OrthogonalSign, 690
MTX.ProperSubmoduleBasis, 688
MTX.SubGModule, 688
MTX.SubmoduleGModule, 688
multiplication, 48
  matrices, 220
  matrix and matrix list, 221
  matrix and scalar, 219
  matrix and vector, 220
  operation, 292
  scalar and matrix, 219
  scalar and matrix list, 220
  scalar and vector, 211
  vector and matrix, 219
  vector and matrix list, 221
  vector and scalar, 211
  vectors, 211
MultiplicationTable, 317
Multiplicative Arithmetic for Lists, *184*
Multiplicative Arithmetic Functions, *140*
MultiplicativeNeutralElement, 319
multiplicative order of an integer, 137
MultiplicativeZero, 319
MultiplicativeZeroOp, 289
multiplicity, of constituents of a group character, 766
multiplier, 377
multisets, 192
Multivariate Polynomials, *666*
MultRowVector, 214
Murnaghan components, 778, 779
Mutability and Copyability, *112*
Mutability Status and List Arithmetic, *186*
Mutable Bases, *595*
MutableBasis, 596
MutableBasisOfClosureUnderAction, 615
MutableBasisOfIdealInNonassociativeAlgebra, 616
MutableBasisOfNonassociativeAlgebra, 616
MutableIdentityMat, 224
MutableNullMat, 224

N

n_k, 160
Name, 115
NameFunction, 61
NameRNam, 262
NamesFilter, 118
NamesGVars, 45
NamesLocalVariablesFunction, 61
NamesOfFusionSources, 797
NamesSystemGVars, 45
NamesUserGVars, 45
NaturalCharacter, 764
Natural Embeddings related to Magma Rings, *655*
NaturalHomomorphismByGenerators, 305
NaturalHomomorphismByIdeal, 620
NaturalHomomorphismByNormalSubgroup, 366
NaturalHomomorphismByNormalSubgroupNC, 366
NaturalHomomorphismBySubAlgebraModule, 627
NaturalHomomorphismBySubspace, 601
NearAdditiveGroup, 551
NearAdditiveGroupByGenerators, 551
NearAdditiveMagma, 551
NearAdditiveMagmaByGenerators, 551
NearAdditiveMagmaWithZero, 551
NearAdditiveMagmaWithZeroByGenerators, 551
NearlyCharacterTablesFamily, 721
negative number, 48

NegativeRoots, 637
NegativeRootVectors, 637
NestingDepthA, 182
NestingDepthM, 182
NewInfoClass, 79
newline, 40
newline character, 247
NewmanInfinityCriterion, 474
New Presentations and Presentations for Subgroups, *469*
NextIterator, 274
NextPrimeInt, 132
NF, 582
NiceBasis, 603
NiceBasisFiltersInfo, 604
NiceFreeLeftModule, 602
NiceFreeLeftModuleInfo, 603
NiceMonomorphism, 384
NiceMonomorphismAutomGroup, 387
Nice Monomorphisms, *383*
NiceObject, 384
NiceVector, 602
NilpotencyClassOfGroup, 361
NilpotentQuotientOfFpLieAlgebra, 644
NK, 160
NOAUTO, 835
NonnegativeIntegers, 126
NonnegIntScalarProducts, 804, 805
NonNilpotentElement, 642
Norm, 573
  of character, 766
NormalBase, 574
NormalClosure, 356
NormalFormIntMat, 239
Normal Forms of Integer Matrices - Name Changes, *838*
Normal Forms over the Integers, *237*
NormalIntersection, 357
NormalizedElementOfMagmaRingModulo-Relations, 656
NormalizedWhitespace, 251
Normalizer, 356
normalizer, 356
NormalizerInGLnZ, 428
NormalizerInGLnZBravaisGroup, 428
NormalizersTom, 703
NormalizerTom, 703
NormalizeWhitespace, 251
NormalSeriesByPcgs, 441
Normal Structure, *356*
NormalSubgroupClasses, 754
NormalSubgroupClassesInfo, 753
NormalSubgroups, 369
NormedRowVector, 212
NormedRowVectors, 599
NormedVectors, 839
not, 169
NrArrangements, 147
NrBasisVectors, 596
NrCombinations, 147
NrConjugacyClasses, 355
  for character tables, 726
NrConjugacyClassesGL, 509
NrConjugacyClassesGU, 509
NrConjugacyClassesPGL, 509
NrConjugacyClassesPGU, 509
NrConjugacyClassesPSL, 509
NrConjugacyClassesPSU, 509
NrConjugacyClassesSL, 509
NrConjugacyClassesSLIsogeneous, 509
NrConjugacyClassesSU, 509
NrConjugacyClassesSUIsogeneous, 509
NrDerangements, 150
NrInputsOfStraightLineProgram, 335
NrMovedPoints, 407
NrOrderedPartitions, 151
NrPartitions, 151
NrPartitionsSet, 150
NrPartitionTuples, 153
NrPermutationsList, 149
NrPolyhedralSubgroups, 736
NrPrimitiveGroups, 520
NrRestrictedPartitions, 152
NrSubsTom, 699
NrTransitiveGroups, 511
NrTuples, 149
NrUnorderedTuples, 148
NullAlgebra, 609
NullMat, 222
NullspaceIntMat, 236
NullspaceMat, 225
NullspaceMatDestructive, 225
NullspaceModQ, 234
Number, 199
number, Bell, 146
  binomial, 145

Stirling, of the first kind, 146
Stirling, of the second kind, 147
`NumberArgumentsFunction`, 61
`NumberFFVector`, 213
number field, 583
number fields, Galois group, 586
`NumberIrreducibleSolvableGroups`, 522
`NumberPerfectGroups`, 516
`NumberPerfectLibraryGroups`, 516
`NumberSmallGroups`, 513
`NumberSyllables`, 331
numerator, of a rational, 144
`NumeratorOfModuloPcgs`, 437
`NumeratorOfRationalFunction`, 662
`NumeratorRat`, 144
Numerical Group Attributes, *363*

O

$O_p(G)$, see `PCore`, 356
`ObjByExtRep`, 649
Objects, *110*
obsolete, 838
`OCOneCocycles`, 376
octal character codes, 247
`OctaveAlgebra`, 609
`od`, 52
`OldGeneratorsOfPresentation`, 491
`Omega`, 360
`ONanScottType`, 412
`OnBreak`, 68
`OnBreakMessage`, 69
`One`, 288
`OneAttr`, 288
`OneCoboundaries`, 375
`OneCocycles`, 374
one cohomology, 374
`OneFactorBound`, 669
`OneImmutable`, 288
`OneIrreducibleSolvableGroup`, 522
`One`*Library*`Group`, 511
`OneMutable`, 288
`OneOfPcgs`, 432
`OneOp`, 288
`OnePrimitiveGroup`, 511
`OneSameMutability`, 288
`OneSM`, 288
`OneSmallGroup`, 513
`OneTransitiveGroup`, 511
`OnIndeterminates`, 667
as a permutation action, 393
`OnLeftInverse`, 392
`OnLines`, 393
example, 506
`OnPairs`, 392
`OnPoints`, 392
`OnRight`, 392
`OnSets`, 392
`OnSetsDisjointSets`, 392
`OnSetsSets`, 392
`OnSetsTuples`, 392
`OnSubspacesByCanonicalBasis`, 394
`OnTuples`, 392
`OnTuplesSets`, 392
`OnTuplesTuples`, 393
`Operation`, 838
`OperationAlgebraHomomorphism`, 621
Operational Structure of Domains, *282*
`OperationHomomorphism`, 838
operations, for booleans, 168
Operations and Attributes for Vector Spaces, *590*
Operations applicable to All Streams, *98*
Operations Concerning Blocks, *731*
Operations for (Near-)Additive Magmas, *552*
Operations for Abelian Number Fields, *583*
operations for algebraic elements, 682
Operations for Associative Words, *329*
Operations for Associative Words by their Syllables, *331*
Operations for Booleans, *168*
Operations for Brauer Characters, *787*
Operations for Class Functions, *765*
Operations for Collections, *270*
Operations for Cyclotomics, *155*
Operations for Domains, *287*
Operations for Finite Field Elements, *577*
Operations for Finitely Presented Groups, *460*
Operations for Group Homomorphisms, *381*
Operations for Input Streams, *98*
Operations for Lists, *194*
Operations for Output Streams, *101*
Operations for Pc Groups, *452*
Operations for Rational Functions, *661*
Operations for Special Kinds of Bases, *595*
Operations for Stabilizer Chains, *418*
Operations for Vector Space Bases, *593*
Operations for Words, *323*

Operations on elements of the algebra, *340*
Operations on rewriting systems, *339*
Operations to Evaluate Strings, *253*
Operations to Produce or Manipulate Strings, *250*
Operations which have Special Methods for Groups with Pcgs, *445*
operators, 42
  arithmetic, 48
  associativity, 49
  for cyclotomics, 159
  for lists, 180
  precedence, 48, 49
Operators for Character Tables, *723*
Operators for Matrices, *219*
Operators for Row Vectors, *210*
Optimization and Compiler Options, *827*
options, 27, 819
  command line, filenames, 29
  command line, internal, 30
options, under UNIX, 27
`or`, 168
`Orbit`, 394
`OrbitFusions`, 798
`OrbitLength`, 395
`OrbitLengths`, 395
`OrbitLengthsDomain`, 395
`OrbitPerms`, 410
`OrbitPowerMaps`, 793
`Orbits`, operation/attribute, 395
Orbits, *394*
`OrbitsDomain`, 395
`OrbitsPerms`, 410
`OrbitStabChain`, 419
`OrbitStabilizer`, 396
`OrbitStabilizerAlgorithm`, 396
Orbit Stabilizer Methods for Polycyclic Groups, *444*
`Order`, 291
  of a class function, 762
order, of a group, 344
  of a list, collection or domain, 269
  of the prime residue group, 136
`OrderedPartitions`, 151
ordering, booleans, 167
  of records, 260
`OrderingByLessThanFunctionNC`, 276
`OrderingByLessThanOrEqualFunctionNC`, 276
`OrderingOfRewritingSystem`, 339
`OrderingOnGenerators`, 278
`OrderingsFamily`, 276
Orderings on families of associative words, *277*
`OrderMod`, 137
`OrderOfRewritingSystem`, 339
`OrdersClassRepresentatives`, 727
`OrdersTom`, 699
`Ordinal`, 254
ordinary character, 765
`OrdinaryCharacterTable`, 725
`OrthogonalComponents`, 778
Orthogonal Embeddings, *242*
`OrthogonalEmbeddings`, 242
`OrthogonalEmbeddingsSpecialDimension`, 774
OSX, 828
Other Filters, *125*
Other Operations Applicable to any Object, *115*
Other Operations for Character Tables, *734*
Other Operations for Tables of Marks, *702*
output, suppressing, 64
`OutputLogTo`, 95
  for streams, 102
  stop logging output, 95
`OutputTextFile`, 103
`OutputTextNone`, 106
`OutputTextString`, 104
`OutputTextUser`, 104

P

p-group, 362
package, 834
Package Interface - Obsolete Functions and Name Changes, *838*
Packages, *823*
`PadicCoefficients`, 241
`PadicExtensionNumberFamily`, 685
`PadicNumber`, 685
  for pure padics, 684
`Pager`, 25
`Parametrized`, 801
Parametrized Maps, *799*
parametrized maps, 790
`Parent`, 286
`ParentPcgs`, 435
Parents, *286*
partial order, 311
`PartialOrderByOrderingFunction`, 312
`PartialOrderOfHasseDiagram`, 311
`Partitions`, 151

partitions, improper, of an integer, 151
ordered, of an integer, 151
restricted, of an integer, 152
PartitionsGreatestEQ, 152
PartitionsGreatestLE, 152
PartitionsSet, 150
PartitionTuples, 153
PcElementByExponents, 433
PcElementByExponentsNC, 433
PCentralLieAlgebra, 641
PCentralNormalSeriesByPcgsPGroup, 440
PCentralSeries, 366
PcGroupCode, 456
PcGroupCodeRec, 456
PcGroupFpGroup, 449
Pc groups versus fp groups, *448*
PcGroupWithPcgs, 451
Pcgs, 431
Pcgs_OrbitStabilizer, 445
Pcgs and Normal Series, *438*
PcgsByPcSequence, 431
PcgsByPcSequenceNC, 431
PcgsCentralSeries, 439
PcgsChiefSeries, 440
PcgsElementaryAbelianSeries, 439
PcgsPCentralSeriesPGroup, 440
PClassPGroup, 362
PCore, 356
PcSeries, 432
PerfectGroup, 515
perfect groups, 515
PerfectIdentification, 516
PerfectResiduum, 358
Perform, 196
Permanent, 154
Permanent of a Matrix, *154*
PermBounds, 787
PermCharInfo, 782
PermCharInfoRelative, 783
PermChars, 783
PermCharsTom, 712
PermComb, 787
PermLeftQuoTransformation, 549
PermList, 408
PermListList, 196
Permutation, 400
PermutationCharacter, 764
permutation character, 809
permutation characters, possible, 781, 783
PermutationCycle, 400
PermutationMat, 223
PermutationsFamily, 406
Permutations Induced by Elements and Cycles, *400*
PermutationsList, 149
PermutationTom, 697
Permuted, 198
as a permutation action, 393
for class functions, 762
PGL, 507
PGU, 508
Phi, 136
point stabilizer, 396
Polycyclic Generating Systems, *430*
PolynomialByExtRep, 679
PolynomialCoefficientsOfPolynomial, 665
PolynomialDivisionAlgorithm, 675
Polynomial Factorization, *667*
PolynomialModP, 668
PolynomialReducedRemainder, 675
PolynomialReduction, 675
PolynomialRing, 670
Polynomial Rings, *670*
Polynomials as Univariate Polynomials in one Indeterminate, *665*
polynomials over abelian number fields, factors, 583
Polynomials over the Rationals, *668*
PopOptions, 88
Portability, *90*
Porting GAP, *827*
Position, 187
PositionBound, 189
PositionCanonical, 187
PositionNonZero, 189
PositionNot, 189
PositionNthOccurrence, 188
PositionProperty, 189
PositionSet, 188
PositionSorted, 188
PositionStream, 100
PositionSublist, 190
PositionWord, 330
PositiveIntegers, 126
positive number, 48
PositiveRoots, 637
PositiveRootVectors, 637
PossibleClassFusions, 797

PossibleFusionsCharTableTom, 711
Possible Permutation Characters, *781*
possible permutation characters, 781, 783
PossiblePowerMaps, 791
power, 48
  matrix, 220
  meaning for class functions, 761
  of words, 329
PowerMap, 791
PowerMapByComposition, 793
PowerMapOp, 791
Power Maps, *790*
PowerMapsAllowedBySymmetrizations, 808
PowerMod, 564
PowerModCoeffs, 216
PowerModInt, 131
PowerPartition, 153
powerset, 147
PowerSubalgebraSeries, 614
PQuotient, 471
precedence, 48
precedence test, for permutations, 407
PrefrattiniSubgroup, 358
  for groups with pcgs, 445
PreImage, 304
PreImageElm, 303
PreImages, 304
PreImagesElm, 303
Preimages in the Free Group, *459*
Preimages in the Free Semigroup, *543*
PreimagesOfTransformation, 548
PreImagesRange, 303
PreImagesRepresentative, 304
PreImagesSet, 304
Preimages under Homomorphisms from an FpGroup, *470*
Preimages under Mappings, *303*
preorder, 311
PresentationFpGroup, 475
PresentationNormalClosure, 480
PresentationNormalClosureRrs, 480
PresentationSubgroup, 477
PresentationSubgroupMtc, 479
PresentationSubgroupRrs, 478
PresentationViaCosetTable, 476
previous result, 64
PrevPrimeInt, 132
PrimaryGeneratorWords, 479
primary subgroup generators, 493
PrimeBlocks, 731
PrimeBlocksOp, 731
PrimeField, 572
Prime Integers and Factorization, *131*
PrimePGroup, 362
PrimePowersInt, 133
prime residue group, 136
  exponent, 136
  generator, 138
  order, 136
Prime Residues, *136*
PrimeResidues, function, 136
Primes, 131
primitive, 401
PRIMITIVE_INDICES_MAGMA, 521
PrimitiveElement, 572
PrimitiveGroup, 520
Primitive Groups, *412*
PrimitiveGroupsIterator, 520
PrimitiveIdentification, 521
PrimitiveIndexIrreducibleSolvableGroup, 522
Primitive Permutation Groups, *519*
PrimitivePolynomial, 668
PrimitiveRoot, 579
PrimitiveRootMod, 138
primitive root modulo an integer, 138
Primitive Roots and Discrete Logarithms, *137*
Primitivity of Characters, *813*
Print, 65
PrintAmbiguity, 804
PrintArray, 224
PrintCharacterTable, 739
PrintFactorsInt, 133
PrintFormattingStatus, 102
Printing Character Tables, *737*
Printing Class Functions, *762*
Printing Presentations, *481*
Printing Tables of Marks, *696*
PrintObj, 66
  for character tables, 737, 762
  for tables of marks, 696
PrintTo, 95
  for streams, 101
ProbabilityShapes, 668
problems, 824
Problems on Particular Systems, *827*
procedure call, 50

Procedure Calls, *50*
procedure call with arguments, 50
`Process`, 108
Process, *108*
`PROD_GF2MAT_GF2MAT_ADVANCED`, 234
`PROD_GF2MAT_GF2MAT_SIMPLE`, 234
`Product`, 200
product, of words, 329
  rational functions, 661
`ProductCoeffs`, 216
`ProductSpace`, 614
`ProductX`, 203
`ProfileFunctions`, 82
`ProfileGlobalFunctions`, 82
`ProfileMethods`, 82
`ProfileOperations`, 81
`ProfileOperationsAndMethods`, 81
`PROFILETHRESHOLD`, 82
Profiling, *81*
`ProjectedInducedPcgs`, 438
`ProjectedPcElement`, 438
`Projection`, 301
  example for direct products, 498
  example for semidirect products, 500
  example for subdirect products, 500
  example for wreath products, 501
  for group products, 502
`ProjectionMap`, 800
projections, find all, 388
`ProjectiveActionHomomorphismMatrixGroup`, 425
`ProjectiveActionOnFullSpace`, 425
`ProjectiveGeneralLinearGroup`, 507
`ProjectiveGeneralUnitaryGroup`, 508
`ProjectiveOrder`, 233
`ProjectiveSpecialLinearGroup`, 508
`ProjectiveSpecialUnitaryGroup`, 508
`ProjectiveSymplecticGroup`, 508
prompt, 64
  partial, 64
Properties, *124*
Properties and Attributes for Lists, *190*
Properties and Attributes of (General) Mappings, *301*
Properties and Attributes of Binary Relations, *310*
Properties and Attributes of Matrices, *221*
Properties and Attributes of Rational Functions, *662*
Properties and basic functionality, *276*
Properties of a Lie Algebra, *635*
Properties of rewriting systems, *340*
Properties of Rings, *559*
Properties of Tables of Marks, *702*
`PRump`, 359
`PseudoRandom`, 272
`PSL`, 508
`PSP`, 508
`PSp`, 508
`PSU`, 508
`PthPowerImage`, 641
`PthPowerImages`, 640
Pure p-adic Numbers, *684*
`PurePadicNumberFamily`, 684
`PushOptions`, 88

Q

`Quadratic`, 162
quadratic residue, 138, 139
`QuaternionAlgebra`, 609
`QUIET`, 839
`QUIT`, emergency quit, 72
quit, in emergency, 72
`quit`, 67
`QUITTING`, 72
`QuoInt`, 129
`Quotient`, 555
quotient, for finitely presented groups, 458
  matrices, 220
  matrix and matrix list, 221
  matrix and scalar, 220
  of free monoid, 544
  of free semigroup, 542
  of words, 329
  rational functions, 661
  scalar and matrix, 220
  scalar and matrix list, 220
  vector and matrix, 220
`QuotientFromSCTable`, 608
Quotient Methods, *471*
`QuotientMod`, 564
`QuotientPolynomialsExtRep`, 680
`QuotientRemainder`, 563
Quotients, *534*
Quotients and Remainders, *129*
`QuotientSemigroupCongruence`, 534
`QuotientSemigroupHomomorphism`, 534
`QuotientSemigroupPreimage`, 534

QuotRemLaurpols, 670

R

r_N, 160
RadicalGroup, 358
RadicalOfAlgebra, 617
Random, 272
  for integers, 128
  for rationals, 144
RandomBinaryRelationOnPoints, 312
random element, of a list or collection, 272
Random Elements, *272*
RandomInvertibleMat, 224
RandomIsomorphismTest, 456
Random Isomorphism Testing, *456*
Randomized Methods for Permutation Groups, *414*
RandomList, 273
RandomMat, 224
Random Matrices, *224*
RandomPrimitivePolynomial, 581
random seed, 273
RandomTransformation, 547
RandomUnimodularMat, 224
Range, 302
range, 203
Ranges, *203*
RankAction, 401
RankFilter, 117
RankMat, 225
RankOfTransformation, 548
RankPGroup, 363
Rat, 144
  for strings, 253
RationalClass, 355
RationalClasses, 355
RationalFunctionByExtRep, 679
RationalFunctionByExtRepWithCancellation, 680
Rational Function Families, *677*
RationalFunctionsFamily, 677
RationalizedMat, 163
Rationals, 143
RClassOfHClass, 535
Read, 94
  for streams, 98
ReadAll, 100
ReadAllLine, 105
ReadAsFunction, 94
  for streams, 99
ReadByte, 99
read eval print loop, 64
ReadLine, 99
ReadPackage, 835
ReadPkg, 838
ReadTest, 84
  for streams, 99
RealClasses, 729
RealizableBrauerCharacters, 788
RecNames, 257
Recognizing Characters, *249*
record, component access, 257
  component assignment, 258
  component variable, 258
  component variable assignment, 259
Record Access Operations, *262*
Record Assignment, *258*
record assignment, operation, 262
record boundness test, operation, 262
record component, operation, 262
record unbind, operation, 262
Recovery from NoMethodFound-Errors, *76*
recursion, 55
redisplay a help section, 23
redisplay with next help viewer, 23
ReduceCoeffs, 216
ReduceCoeffsMod, 216
ReducedAdditiveInverse, 340
ReducedCharacters, 771
ReducedClassFunctions, 771
ReducedComm, 340
ReducedConfluentRewritingSystem, 544
ReducedConjugate, 340
ReducedDifference, 340
ReducedForm, 339
ReducedGroebnerBasis, 676
ReducedInverse, 340
ReducedLeftQuotient, 340
ReducedOne, 340
ReducedPcElement, 433
ReducedPower, 340
ReducedProduct, 340
ReducedQuotient, 340
ReducedScalarProduct, 340
ReducedSum, 340
ReducedZero, 340
ReduceRules, 340

ReduceStabChain, 420
Reducing Virtual Characters, *771*
Ree, 505
ReeGroup, 505
ReesCongruenceOfSemigroupIdeal, 533
ReesMatrixSemigroup, 536
ReesMatrixSemigroupElement, 536
Rees Matrix Semigroups, *536*
ReesZeroMatrixSemigroup, 536
ReesZeroMatrixSemigroupElement, 536
ReesZeroMatrixSemigroupElementIsZero, 537
RefinedPcGroup, 451
ReflectionMat, 224
ReflexiveClosureBinaryRelation, 312
reflexive relation, 310
regular, 401
regular action, 398
RegularActionHomomorphism, 399
RegularModule, 742
relations, 299
Relations Between Domains, *292*
RelationsOfFpSemigroup, 543
RelativeBasis, 593
RelativeBasisNC, 593
relatively prime, 48
RelativeOrderOfPcElement, 432
RelativeOrders, of a pcgs, 432
Relators in a Presentation, *480*
RelatorsOfFpGroup, 459
remainder, operation, 292
remainder of a quotient, 129
RemInt, 129
remove, an element from a set, 194
RemoveFile, 95
RemoveOuterCoeffs, 214
RemoveRelator, 483
RemoveSet, 194
RemoveStabChain, 420
Repeat, *52*
repeat loop, 52
ReplacedString, 251
Representation, *120*
representation, as a sum of two squares, 142
Representations for Associative Words, *332*
Representations for Group Homomorphisms, *389*
Representations given by modules, *742*
Representations of Algebras, *622*
RepresentationsOfObject, 121
Representative, 269
representative, of a list or collection, 269
RepresentativeAction, 396
RepresentativeLinearOperation, 622
RepresentativeOperation, 838
RepresentativesContainedRightCosets, 353
RepresentativesFusions, 798
RepresentativeSmallest, 269
RepresentativesMinimalBlocks, 402
RepresentativesPerfectSubgroups, 371
RepresentativesPowerMaps, 793
RepresentativesSimpleSubgroups, 371
RepresentativeTom, 710
RepresentativeTomByGenerators, 710
RepresentativeTomByGeneratorsNC, 710
RequirePackage, 838
Reread, 96
REREADING, 96
RereadPackage, 835
RereadPkg, 838
ResetOptionsStack, 88
residue, quadratic, 138, 139
Residue Class Rings, *133*
RespectsAddition, 306
RespectsAdditiveInverses, 306
RespectsInverses, 306
RespectsMultiplication, 305
RespectsOne, 306
RespectsScalarMultiplication, 307
RespectsZero, 307
RestoreStateRandom, 272
Restricted and Induced Class Functions, *769*
RestrictedClassFunction, 769
RestrictedClassFunctions, 770
Restricted Lie algebras, *640*
RestrictedMapping, 301
RestrictedPartitions, 152
RestrictedPerm, 409
RestrictedTransformation, 548
Resultant, 666
ResultOfStraightLineProgram, 335
Return, *58*
return, 67
  no value, 58
  with value, 58
ReturnFail, 63
ReturnFalse, 62
return from break loop, 67

`ReturnTrue`, 62
`Reversed`, 196
`RewindStream`, 100
`RewriteWord`, 465
Rewriting in Groups and Monoids, *341*
Rewriting Systems and the Knuth-Bendix Procedure, *544*
`RightActingAlgebra`, 625
`RightActingRingOfIdeal`, 558
`RightAlgebraModule`, 623
`RightAlgebraModuleByGenerators`, 623
`RightCoset`, 351
`RightCosets`, 351
right cosets, 351
`RightCosetsNC`, 351
`RightDerivations`, 632
`RightIdeal`, 556
`RightIdealByGenerators`, 557
`RightIdealNC`, 557
`RightModuleByHomomorphismToMatAlg`, 626
`RightShiftRowVector`, 214
`RightTransversal`, 352
`Ring`, 554
`RingByGenerators`, 555
Ring Homomorphisms, *308*
Rings With One, *558*
`RingWithOne`, 558
`RingWithOneByGenerators`, 558
`RNamObj`, 262
root, of 1 modulo an integer, 140
  of an integer, 128
  of an integer, smallest, 128
  of an integer modulo another, 139
`RootInt`, 128
`RootMod`, 139
`RootOfDefiningPolynomial`, 572
`RootsMod`, 139
Roots Modulo Integers, *138*
roots of unity, 155
`RootsOfUPol`, 665
`RootsUnityMod`, 140
`RootSystem`, 637
`RoundCyc`, 157
Row and Matrix Spaces, *597*
`RowIndexOfReesMatrixSemigroupElement`, 537
`RowIndexOfReesZeroMatrixSemigroupElement`, 537
row spaces, 597
Row Vectors over Finite Fields, *212*
`Rules`, 339
Running GAP under MacOS, *30*
`Runtime`, 81
`Runtimes`, 80

S

s_N, 160
`SameBlock`, 732
`SandwichMatrixOfReesMatrixSemigroup`, 537
`SandwichMatrixOfReesZeroMatrixSemigroup`, 537
save, 36
`SaveOnExitFile`, 73
`SaveWorkspace`, 36
Saving and Loading a Workspace, *36*
Saving a Pc Group, *452*
saving on exit, 72
`ScalarProduct`, for characters, 766
Schreier, 477
Schreier-Sims, random, 414
`SchurCover`, 377
Schur Covers and Multipliers, *376*
Schur multiplier, 377
scope, 42
`ScriptFromString`, 707
Searching for Homomorphisms, *387*
`SecHMSM`, 255
secondary subgroup generators, 493
`SecondsDMYhms`, 255
`SeekPositionStream`, 101
Selecting a Different MeatAxe, *687*
Selection Functions, *510*
`SemidirectProduct`, 499
Semidirect Products, *498*
`SemiEchelonBasis`, 599
`SemiEchelonBasisNC`, 599
`SemiEchelonMat`, 227
`SemiEchelonMatDestructive`, 227
`SemiEchelonMats`, 228
`SemiEchelonMatsDestructive`, 228
`SemiEchelonMatTransformation`, 228
`Semigroup`, 531
semigroup, 531
`SemigroupByGenerators`, 531
`SemigroupByMultiplicationTable`, 532
`SemigroupIdealByGenerators`, 533
`SemigroupOfRewritingSystem`, 546
semiregular, 401

Semisimple Lie Algebras and Root Systems, *636*
SemiSimpleType, 636
sequence, Bernoulli, 146
  Fibonacci, 153
  Lucas, 154
Series of Ideals, *634*
Set, 266
SetAssertionLevel, 80
SetCommutator, 450
SetConjugate, 450
SetCrystGroupDefaultAction, 429
set difference, of collections, 271
SetEntrySCTable, 607
SetGasmanMessageStatus, 86
SetHelpViewer, 24
SetIndeterminateName, 660
SetInfoLevel, 79
SetName, 115
Set Operations via Boolean Lists, *207*
SetParent, 286
SetPower, 450
SetPrintFormattingStatus, 102
SetRecursionTrapInterval, 85
SetReducedMultiplication, 459
Sets, *110*
sets, 170, 192
Sets of Subgroups, *367*
set stabilizer, 396
Setter, 122
setter, 121
Setter and Tester for Attributes, *121*
SetX, 202
ShallowCopy, 114
  for lists, 177
ShiftedCoeffs, 216
ShiftedPadicNumber, 684
Shifting and Trimming Coefficient Lists, *214*
ShortestVectors, 243
ShortLexOrdering, 279
short vectors spanning a lattice, 241, 772
ShowArgument, 76
ShowArguments, 76
ShowDetails, 76
ShowImpliedFilters, 118
ShowMethods, 76
ShowOtherMethods, 77
ShrinkCoeffs, 216
ShrinkRowVector, 214
SiftedPcElement, 433
SiftedPermutation, 419
SiftedVector, 600
Sigma, 140
sign, of an integer, 127
Sign and Cycle Structure, *408*
SignInt, 127
SignPartition, 152
SignPerm, 408
SimpleLieAlgebra, 632
SimpleSystem, 637
SimplifiedFpGroup, 477
SimplifiedFpGroup, *477*
SimplifyPresentation, 484
SimsNo, 521
SimultaneousEigenvalues, 233
SingleCollector, 450
singlequote character, 247
singlequotes, 245
SINT_CHAR, 253
Size, 269
  for character tables, 726
  for groups with pcgs, 445
size, of a list or collection, 269
SizeBlist, 207
SizeConsiderFunction, 373
SizeNumbersPerfectGroups, 516
SizeOfFieldOfDefinition, 788
SizesCentralizers, 727
SizesConjugacyClasses, 727
SizeScreen, 75
SizeScreen, *75*
SizesPerfectGroups, 515
SizeStabChain, 418
SL, 506
smaller, associative words, 329
  elements of finitely presented groups, 459
  nonassociative words, 322
  pcwords, 448
  rational functions, 661
SmallerDegreePermutationRepresentation, 411
smaller or equal, 47
smaller test, 47
SmallestGeneratorPerm, 407
SmallestMovedPoint, 407
SmallestRootInt, 128
SmallGeneratingSet, 374
SmallGroup, 513

Small Groups, *512*
SmallGroupsInformation, 513
Smash MeatAxe Flags, *691*
smith normal form, 838
SmithNormalFormIntegerMat, 238
SmithNormalFormIntegerMatTransforms, 238
SMTX.AbsoluteIrreducibilityTest, 691
SMTX.AlgEl, 691
SMTX.AlgElCharPol, 692
SMTX.AlgElCharPolFac, 692
SMTX.AlgElMat, 691
SMTX.AlgElNullspaceDimension, 692
SMTX.AlgElNullspaceVec, 692
SMTX.CentMat, 692
SMTX.CentMatMinPoly, 692
SMTX.CompleteBasis, 691
SMTX.Getter, 691
SMTX.GoodElementGModule, 691
SMTX.IrreducibilityTest, 691
SMTX.MatrixSum, 691
SMTX.MinimalSubGModule, 691
SMTX.MinimalSubGModules, 691
SMTX.RandomIrreducibleSubGModule, 691
SMTX.Setter, 691
SMTX.SortHomGModule, 691
SMTX.Subbasis, 691
SO, 507
Socle, 358
SocleTypePrimitiveGroup, 412
SolutionIntMat, 236
SolutionMat, 226
SolutionMatDestructive, 226
SolutionNullspaceIntMat, 236
Some Remarks about Character Theory in GAP, *715*
Some Special Algebras, *609*
Sort, 191
SortedCharacters, 749
SortedCharacterTable, 750
Sorted Character Tables, *749*
SortedList, 266
sorted list, 190
Sorted Lists and Sets, *192*
sorted lists as collections, 264
SortedSparseActionHomomorphism, 398
SortedTom, 697
Sortex, 192
Sorting Lists, *191*
SortingPerm, 192
Sorting Tables of Marks, *697*
SortParallel, 191
Source, 302
SP, 507
Sp, 507
space, 40
SparseActionHomomorphism, 398
SparseCartanMatrix, 638
Special Characters, *247*
special character sequences, 247
Special Filenames, *92*
Special Generating Sets, *374*
SpecialLinearGroup, 506
Special Multiplication Algorithms for Matrices over GF(2), *234*
SpecialOrthogonalGroup, 507
Special Pcgs, *441*
SpecialPcgs, attribute, 442
SpecialUnitaryGroup, 506
Specific and Parametrized Subgroups, *357*
Specific Methods for Subgroup Lattice Computations, *371*
SplitCharacters, 744
SplitExtension, 453
SplitExtensions, 455
SplitString, 250
SplittingField, 664
Sqrt, 292
square root, of an integer, 128
SquareRoots, 319
SSortedList, 266
StabChain, 415
StabChainBaseStrongGenerators, 417
StabChainImmutable, 415
StabChainMutable, 415
StabChainOp, 415
StabChainOptions, 417
Stabilizer, 396
Stabilizer Chain Records, *417*
Stabilizer Chains, *413*
StabilizerOfExternalSet, 404
StabilizerPcgs, 445
Stabilizers, *396*
StandardAssociate, 561
StandardGeneratorsFunctions, 707
StandardGeneratorsInfo, for groups, 706
for tables of marks, 710

`StandardGeneratorsOfGroup`, 708
Standard Generators of Groups, *706*
Standardization of coset tables, *464*
`StandardizeTable`, 464
`StarCyc`, 162
Statements, *49*
`StateRandom`, 272
`Stirling1`, 146
`Stirling2`, 146
Stirling number of the first kind, 146
Stirling number of the second kind, 147
`StoredGroebnerBasis`, 676
`StoreFusion`, 796
Storing Normal Subgroup Information, *753*
`StraightLineProgElm`, 338
`StraightLineProgGens`, 338
`StraightLineProgram`, 334
Straight Line Program Elements, *337*
`StraightLineProgramNC`, 334
Straight Line Programs, *334*
`StraightLineProgramsTom`, 709
`StreamsFamily`, 98
`StretchImportantSLPElement`, 338
strictly sorted list, 191
`String`, 250
  for cyclotomics, 156
`StringDate`, 255
`StringOfResultOfStraightLineProgram`, 336
`StringPP`, 250
strings, equality of, 249
  inequality of, 249
  lexicographic ordering of, 249
String Streams, *104*
`StringTime`, 255
`StrongGeneratorsStabChain`, 418
`StronglyConnectedComponents`, 312
Struct, 283
Struct`ByGenerators`, 284
`StructuralCopy`, 114
  for lists, 178
structure constant, 736, 737
`StructureConstantsTable`, 595
`StructureDescription`, 349
Structure Descriptions, *349*
Struct`WithGenerators`, 284
`SU`, 506
`Subalgebra`, 610
`SubAlgebraModule`, 626
`SubalgebraNC`, 610
Subalgebras, *610*
`SubalgebraWithOne`, 610
`SubalgebraWithOneNC`, 611
`SubdirectProduct`, 500
Subdirect Products, *500*
`SubdirectProducts`, 501
subdomains, 287
`Subfield`, 571
`SubfieldNC`, 571
`Subfields`, 572
Subfields of Fields, *571*
`Subgroup`, 346
`SubgroupByPcgs`, 436
`SubgroupByProperty`, 347
subgroup fusions, 794
subgroup generators tree, 493
Subgroup Lattice, *369*
`SubgroupNC`, 346
`SubgroupOfWholeGroupByCosetTable`, 465
`SubgroupOfWholeGroupByQuotientSubgroup`, 470
Subgroup Presentations, *477*
`SubgroupProperty`, 421
Subgroups, *345*
subgroups, polyhedral, 736
Subgroups characterized by prime powers, *360*
Subgroup Series, *363*
`SubgroupShell`, 347
Subgroups of Polycyclic Groups - Canonical Pcgs, *436*
Subgroups of Polycyclic Groups - Induced Pcgs, *434*
`SubgroupsSolvableGroup`, 372
sublist, 172
  access, 172
  assignment, 174
  operation, 173
sublist assignment, operation, 175
`Submagma`, 316
`SubmagmaNC`, 316
`SubmagmaWithInverses`, 316
`SubmagmaWithInversesNC`, 316
`SubmagmaWithOne`, 316
`SubmagmaWithOneNC`, 316
`Submodule`, 567
`SubmoduleNC`, 567
Submodules, *567*
`Submonoid`, 538
`SubmonoidNC`, 538

SubnearAdditiveGroup, 552
SubnearAdditiveGroupNC, 552
SubnearAdditiveMagma, 551
SubnearAdditiveMagmaNC, 551
SubnearAdditiveMagmaWithZero, 551
SubnearAdditiveMagmaWithZeroNC, 551
SubnormalSeries, 364
Subring, 555
SubringNC, 555
SubringWithOne, 559
SubringWithOneNC, 559
Subroutines for the Construction of Class Fusions, *809*
Subroutines for the Construction of Power Maps, *807*
Subsemigroup, 531
SubsemigroupNC, 531
subsets, 147
subset test, for collections, 270
Subspace, 589
SubspaceNC, 589
Subspaces, 590
SubstitutedWord, 331
SubsTom, 698
Sub*struct*, 287
Sub*struct*NC, 287
SubSyllables, 332
subtract, a set from another, 194
SubtractBlist, 208
subtraction, 48
  matrices, 219
  matrix and scalar, 219
  rational functions, 661
  scalar and matrix, 219
  scalar and matrix list, 220
  scalar and vector, 211
  vector and scalar, 211
  vectors, 211
SubtractSet, 194
Subword, 330
Successors, 311
Suitability for Compilation, *35*
Sum, 201
Sum and Intersection of Pcgs, *441*
SumFactorizationFunctionPcgs, 441
SumIntersectionMat, 229
SumX, 202
SupersolvableResiduum, 359
support, email address, 826
SupportedCharacterTableInfo, 719
SurjectiveActionHomomorphismAttr, 405
SuzukiGroup, 505
SylowComplement, 359
SylowSubgroup, 359
Sylow Subgroups and Hall Subgroups, *359*
SylowSystem, 360
Symbols, *40*
Symmetric and Alternating Groups, *411*
SymmetricClosureBinaryRelation, 312
SymmetricGroup, 505
symmetric group, powermap, 153
SymmetricParentGroup, 412
SymmetricParts, 778
SymmetricPowerOfAlgebraModule, 651
symmetric relation, 310
Symmetrizations, 777
symmetrizations, orthogonal, 778
  symplectic, 779
Symmetrizations of Class Functions, *777*
SymplecticComponents, 779
SymplecticGroup, 507
syntax errors, 64
system getter, 121
system setter, 121
Sz, 505

T

t_N, 160
TableAutomorphisms, 752
table automorphisms, 798, 810
table of chapters for help books, 23
TableOfMarks, 694
TableOfMarksByLattice, 695
TableOfMarksComponents, 698
TableOfMarksCyclic, 712
TableOfMarksDihedral, 713
TableOfMarksFamily, 698
TableOfMarksFrobenius, 713
Table of Marks Objects in GAP, *693*
table of sections for help books, 23
tables, 715, 717
tabulator, 40
Tau, 140
Technical Details about Tables of Marks, *698*
Technical Details about the Implementation of Magma Rings, *657*

Technical Matters Concerning General Mappings, *308*
`TemporaryGlobalVarName`, 45
`Tensored`, 769
`TensorProductGModule` , 687
`TensorProductOfAlgebraModules`, 650
Tensor Products and Exterior and Symmetric Powers, *650*
test, for a primitive root, 138
  for a rational, 143
  for records, 257
  for set equality, 193
`TestConsistencyMaps`, 803
`Tester`, 122
tester, 121
Test Files, *84*
`TestHomogeneous`, 813
`TestInducedFromNormalSubgroup`, 815
Testing Finiteness of Finitely Presented Groups, *474*
Testing for the System Architecture, *34*
Testing Monomiality, *815*
`TestJacobi`, 607
`TestMonomial`, 815
`TestMonomialQuick`, 816
`TestMonomialUseLattice`, 816
Test of the installation, *822*
`TestPackageAvailability`, 835
`TestPerm1`, 785
`TestPerm2`, 785
`TestPerm3`, 785
`TestPerm4`, 785
`TestPerm5`, 785
`TestQuasiPrimitive`, 814
`TestRelativelySM`, 817
Tests for Actions, *400*
Tests for the Availability of Methods, *377*
`TestSubnormallyMonomial`, 817
The .gaprc file, *33*
The Adjoint Representation, *641*
The Compiler, *34*
The Defining Attributes of Rational Functions, *678*
The Dixon-Schneider Algorithm, *742*
The Documentation, *823*
The External Representation for Associative Words, *333*
The family pcgs, *447*
The Interface between Character Tables and Groups, *721*
The Interface between Tables of Marks and Character Tables, *711*
The Library of Tables of Marks, *713*
`then`, 51
The Natural Action, *410*
The Pager Command, *25*
The Permutation Image of an Action, *397*
The Representations of Rational Functions, *677*
The Smash MeatAxe, *690*
The Syntax in BNF, *58*
`ThreeGroup` library, 512
Tietze Options, *496*
Tietze Transformations, *483*
Tietze Transformations that introduce new Generators, *487*
`TietzeWordAbstractWord`, 481
`time`, 81
Timing, *80*
Todd-Coxeter Procedure, *546*
`Trace`, 221
  for field elements, 573
  of a matrix, 221
`TracedCosetFpGroup`, 461
`TraceImmediateMethods`, 78
`TraceMat`, 221
`TraceMethods`, 78
`TracePolynomial`, 573
Tracing generator images through Tietze transformations, *491*
Tracing Methods, *78*
`TransferDiagram`, 803
`Transformation`, 547
`TransformationData`, 547
`TransformationFamily`, 547
`TransformationNC`, 547
`TransformationRelation`, 549
`TransformationType`, 547
`TransformingPermutations`, 752
`TransformingPermutationsCharacterTables`, 752
transitive, 400
`TransitiveClosureBinaryRelation`, 312
`TransitiveGroup`, 511
`TransitiveIdentification`, 511
Transitive Permutation Groups, *511*
transitive relation, 310
`Transitivity`, for characters, 768
  for class functions, 768
  for group actions, 401

TranslatorSubalgebra, 628
transporter, 396
TransposedMat, 223
TransposedMatAttr, 223
TransposedMatDestructive, 223
TransposedMatImmutable, 223
TransposedMatMutable, 223
TransposedMatOp, 223
TransposedMatrixGroup, 424
Transversals, *352*
Triangular Matrices, *229*
TriangulizedIntegerMat, 237
TriangulizedIntegerMatTransform, 237
TriangulizedNullspaceMat, 225
TriangulizedNullspaceMatDestructive, 225
TriangulizeIntegerMat, 237
TriangulizeMat, 225
TrivialCharacter, 764
TrivialGroup, 503
TrivialIterator, 275
TrivialSubalgebra, 611
TrivialSubgroup, 357
TrivialSubmagmaWithOne, 319
TrivialSubmodule, 567
TrivialSubmonoid, 538
TrivialSubnearAdditiveMagmaWithZero, 552
TrivialSubspace, 590
TryCosetTableInWholeGroup, 465
TryGcdCancelExtRepPolynomials, 680
Tuples, 149
tuple stabilizer, 396
TwoClosure, 421
TwoCoboundaries, 453
TwoCocycles, 453
TwoCohomology, 453
TwoGroup library, 512
TwoSidedIdeal, 556
TwoSidedIdealByGenerators, 557
TwoSidedIdealNC, 557
TwoSquares, 142
type, boolean, 167
  cyclotomic, 155
  records, 257
  strings, 245
TypeObj, 125
TypeOfDefaultGeneralMapping, 309
Types, *125*
TzEliminate, 486
TzFindCyclicJoins, 487
TzGo, 483
TzGoGo, 485
TzImagesOldGens, 491
TzInitGeneratorImages, 491
TzNewGenerator, 483
TzOptions, 496
TzPreImagesNewGens, 492
TzPrint, 482
TzPrintGeneratorImages, 492
TzPrintGenerators, 481
TzPrintLengths, 482
TzPrintOptions, 497
TzPrintPairs, 482
TzPrintPresentation, 482
TzPrintRelators, 481
TzPrintStatus, 482
TzSearch, 486
TzSearchEqual, 487
TzSort, 475
TzSubstitute, 488
TzSubstituteCyclicJoins, 491

U

u_N, 160
UglyVector, 602
Unbind, 43
  for lists, 176
UnbindGlobal, 45
UnderlyingCharacteristic, 728
UnderlyingCharacterTable, 758
UnderlyingElement, fp group elements, 460
  fp semigroup elements, 543
UnderlyingElementOfReesMatrixSemigroup-
  Element, 537
UnderlyingElementOfReesZeroMatrixSemigroup-
  Element, 537
UnderlyingExternalSet, 405
UnderlyingFamily, 631
UnderlyingGeneralMapping, 302
UnderlyingGroup, for character tables, 721
  for tables of marks, 700
UnderlyingLeftModule, 593
UnderlyingLieAlgebra, 637
UnderlyingMagma, 654
UnderlyingRelation, 302
UnInstallCharReadHookFunc, 106
Union, 271

union, of collections, 271
  of sets, 194
`Union2`, 271
`UnionBlist`, 207
`Unique`, 195
`UniteBlist`, 208
`UniteBlistList`, 208
`UniteSet`, 194
`Units`, 560
Units and Factorizations, *560*
`UnivariatenessTestRationalFunction`, 665
`UnivariatePolynomial`, 664
`UnivariatePolynomialByCoefficients`, 664
`UnivariatePolynomialRing`, 672
Univariate Polynomial Rings, *672*
Univariate Polynomials, *664*
`UnivariateRationalFunctionByCoefficients`, 670
Univariate Rational Functions, *670*
`UniversalEnvelopingAlgebra`, 643
Universal Enveloping Algebras, *643*
UNIX, features, 27
  options, 27
`UNIXSelect`, 98
`Unknown`, 165
`UnloadSmallGroupsData`, 513
`UnorderedTuples`, 148
Unpacking, *820*
`UnprofileFunctions`, 82
`UnprofileMethods`, 82
`until`, 52
`UntraceMethods`, 78
`UpdateMap`, 801
`UpEnv`, 70
`UpperCentralSeriesOfGroup`, 365
`UpperSubdiagonal`, 230
`UseBasis`, 568
`UseFactorRelation`, 293
Useful Categories for all Elements of a Family, *296*
Useful Categories of Elements, *294*
`UseIsomorphismRelation`, 293
User Streams, *103*
`UseSubsetRelation`, 293
utilities for editing GAP files, 74

V

v_N, 160
`Valuation`, 684
`Value`, 666
`ValueCochain`, 646
`ValueGlobal`, 44
`ValueMolienSeries`, 781
`ValueOption`, 88
`ValuePol`, 215
`ValuesOfClassFunction`, 758
Variable Access in a Break Loop, *70*
Variables, *42*
Vectors as coefficients of polynomials, *215*
`VectorSpace`, 589
`VectorSpaceByPcgsOfElementaryAbelianGroup`, 443
Vector Space Homomorphisms, *600*
Vector Spaces Handled By Nice Bases, *602*
`vi`, 74
`View`, 65
View and Print, *65*
`ViewObj`, 66
  for character tables, 737
  for class functions, 762
  for tables of marks, 696
`vim`, 74
`VirtualCharacter`, 763
virtual character, 765
virtual characters, 756

W

w_N, 160
`WedgeGModule` , 687
`WeekDay`, 255
`WeightLexOrdering`, 279
`WeightOfGenerators`, 280
`WeightsTom`, 702
`WeightVecFFE`, 214
`WeylGroup`, 638
`WeylOrbitIterator`, 640
`Where`, 70
While, *51*
`while` loop, 51
Whitespaces, *40*
Why Class Functions?, *756*
`WordAlp`, 250
words, in generators, 348
Working with large degree permutation groups, *421*
`WreathProduct`, 501
wreath product embedding, 502
`WreathProductImprimitiveAction`, 501

`WreathProductOrdering`, 281
`WreathProductProductAction`, 502
Wreath Products, *501*
`WriteAll`, 101
`WriteByte`, 101
`WriteLine`, 101

X

x_N, 160

Y

y_N, 160

Z

`Z`, 576
`ZClassRepsQClass`, 428
`Zero`, 289
`ZeroAttr`, 289
`ZeroCoefficient`, 655
`ZeroCoefficientRatFun`, 679
`ZeroImmutable`, 289
`ZeroMapping`, 300
`ZeroMutable`, 289
`ZeroOp`, 289
`ZeroSameMutability`, 289
`ZeroSM`, 289
`ZippedProduct`, 680
`ZippedSum`, 680
`ZmodnZ`, 134
`ZmodnZObj`, 134
`ZmodpZ`, 134
`ZmodpZNC`, 134
`ZumbroichBase`, 585
`Zuppos`, 371