John J. Cannon Catherine Playoust
School of Mathematics and Statistics, University of Sydney

An Introduction to Algebraic Programming with Magma (Draft)

Volume I: The Magma Language
Volume II: The Magma Categories

February 16, 2001

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Copyright ©1996. All rights reserved.
No part of this book may be reproduced without written permission.

Typeset in the LaTeX macro package *CLMono01* developed by Springer-Verlag Heidelberg as a free service to its authors and editors preparing camera-ready copy.

Preface

What Is Magma?

MAGMA is a programming language designed for the investigation of algebraic, geometric and combinatorial structures, or *magmas*. The syntax of the language resembles that of many well-known programming languages. What is special about MAGMA is the provision of mathematical data types such as groups, rings, fields, sets, sequences and mappings, together with a large collection of functions for performing standard tasks in algebra. Information about the magmas and their elements is stored in a mathematically powerful way, making advanced symbolic algebraic computation feasible. MAGMA is a sophisticated tool for experimentation, education, and computer-aided proof, useful for both students and professional mathematicians.

About This Book

This book is an introductory manual for MAGMA. It presumes no knowledge of computer programming, and its examples are chosen to illustrate language and algorithmic features as simply as possible. Volume I explains the language and user environment in detail, and Volume II deals with the major algebraic, geometrical and combinatorial structures implemented in the system. Note carefully that this book does not attempt to give comprehensive coverage of the algorithms implemented in MAGMA. Before undertaking serious computations in MAGMA, the user should also consult the reference manual, *Handbook of Magma Functions* [BoC96]; within the present book, it is referred to simply as *Handbook*.

Although this book is designed as an introduction, the short booklet *First Steps in Magma* may be more appropriate for those requiring only a passing acquaintance with the system.

System Updates and Release Notes

The version of the system documented in this edition is MAGMA V2, released in 1996. Release notes for later versions may be obtained from the developers of MAGMA.

Acknowledgements

Many people assisted us in the writing of this book. We would particularly like to thank Wieb Bosma and Allan Steel, who wrote early versions of the chapters on number fields and polynomial rings, respectively, and the current members of the Computational Algebra Group for their valiant proof-reading.

We also wish to thank our families for their support.

Computational Algebra Group
School of Mathematics and Statistics
University of Sydney
NSW 2006
Australia

Email: magma@maths.usyd.edu.au
Telephone: +61–2–9351–3338
Fax: +61–2–9351–4534
Home Page:
http://magma.maths.usyd.edu.au/

Sydney, September 1996 John J. Cannon, Catherine Playoust

Table of Contents

Preface V

Part I. Overview

1. Getting Started With Magma 3
1.1 Entering the Magma System 3
1.2 Input and Output 3
1.3 Creating Structures and Their Elements 4
1.4 Online Help and Environment 5
1.5 Quitting Magma 5

2. Developed Examples 7
2.1 Affine Plane from a Projective Plane by Derivation 7
2.2 Constructing an Endo-trivial Module 9
2.3 Molien Series and Primary Invariants 13
2.4 Galois Group and Action 16

Part II. The Language

3. Basic Ideas 23
3.1 Evaluating and Printing Expressions 23
3.1.1 Arithmetic 24
3.1.2 Function Calls 25
3.1.3 Printing Text 26
3.2 Identifiers 26
3.2.1 Labelling Information 26
3.2.2 Showing All Identifiers and Their Values 28
3.2.3 Deletion and Unassigned Identifiers 28
3.3 Assignment 29
3.3.1 The Simple Assignment Statement 29
3.3.2 The Mutation Assignment Statement 31

3.3.3 Multi-Valued Expressions and the Multiple Assignment Statement . . . 31
3.4 The **where**-construction . . . 33
3.5 Integers, Rationals, Reals, and Complex Numbers . . . 35
3.6 Booleans . . . 37
3.7 Comments . . . 40
3.8 Recalling Previously Printed Values . . . 40
3.9 Loading Input from a File . . . 42

4. Algebraic Structures . . . 45
4.1 Magmas, Categories and Varieties . . . 45
4.2 Creation of Magmas . . . 48
4.3 Automatically-Created Magmas . . . 49
4.4 Creating a Free Magma . . . 50
4.5 Creating and Operating on Elements of a Magma . . . 51
4.5.1 Creating Literal Elements . . . 52
4.5.2 Arithmetic Operators . . . 54
4.5.3 Relational Operators . . . 56
4.5.4 Canonical Forms of Elements . . . 57
4.5.5 Generators and Element Representation . . . 58
4.6 Submagmas, Normal Closures and Ideals . . . 60
4.6.1 Creating Submagmas and Ideals . . . 61
4.6.2 The Embedding in the Free Magma . . . 63
4.6.3 Inclusion Monomorphism and Principle of Locality . . . 64
4.7 Magma Generators . . . 66
4.7.1 Generating Sets . . . 66
4.7.2 Determination of the Generators . . . 68
4.7.3 Remarks on Generator Names . . . 72
4.7.4 The Pseudo-Identifier $ . . . 74
4.8 Quotient Magmas . . . 75
4.9 Shorthand Constructors . . . 78
4.9.1 Finitely-Presented Magmas . . . 78
4.9.2 Submagmas of Generic Magmas . . . 81
4.10 Some Other Operations on Magmas . . . 82
4.11 Warning About Large Magmas . . . 85

5. Conditional Statements and Expressions . . . 87
5.1 The **if**-statement . . . 87
5.1.1 Basic Form of the **if**-statement . . . 87
5.1.2 Special Prompt Symbols . . . 88
5.1.3 Short Form of the **if**-statement . . . 89
5.1.4 Avoiding Nested **if**-statements . . . 89
5.1.5 Longer Example: Area of a Triangle . . . 90
5.2 The **select**-expression . . . 92
5.3 The **case**-statement . . . 92

5.4 The **case**-expression 94

6. Aggregate Structures 97
6.1 Properties of the Aggregate Categories 97
6.2 Construction of Iterable Homogeneous Aggregates 100
6.2.1 Construction from an Element-List 100
6.2.2 The Universe of a Homogeneous Aggregate 102
6.2.3 Empty and Null Sets and Sequences 104
6.2.4 Construction of Arithmetic Progressions 105
6.2.5 Construction from an Element-Description 107
6.2.6 Recursive Definition of Enumerated Sequences 112
6.2.7 Enumerated Sequences with Undefined Terms 113
6.3 Operations on Iterable Homogeneous Aggregates 114
6.3.1 Index Position, Cardinality and Length 114
6.3.2 Testing Equality, Membership, and Subsetness 115
6.3.3 Including or Excluding an Element 116
6.3.4 Maximum and Minimum Elements 116
6.3.5 Reduction 117
6.4 Further Operations on Set Categories 118
6.5 Further Operations on Enumerated Sequences 120
6.5.1 Changing a Term of a Sequence 121
6.5.2 Obtaining Several Terms of a Sequence 121
6.5.3 Subsequences and Concatenation 122
6.5.4 Inserting and Removing Objects 123
6.5.5 Rearranging the Terms of a Sequence 124
6.5.6 Multi-Dimensional Sequences 125
6.6 Representative and Random Elements 126
6.7 Existential and Universal Quantifiers 127
6.8 Creating and Operating on Formal Sets 129
6.9 Lists 130
6.10 Cartesian Products and Tuples 132
6.11 Coproducts 136
6.12 Record Formats and Records 141
6.12.1 Creating Record Formats and Records 141
6.12.2 Example: Using a Function to Create Records 143
6.13 Transfer Functions Between Aggregates 146

7. Mappings and Homomorphisms 149
7.1 Standard Mappings 150
7.2 Operations on Mappings and Homomorphisms 152
7.2.1 Calculating Images 152
7.2.2 Calculating Preimages 154
7.2.3 Functions on Mappings and Homomorphisms 156
7.3 Constructing Mappings and Homomorphisms 156
7.3.1 Mapping Rule Given as Expression 157

7.3.2 Element-by-Element Rule for General Mappings 160
7.3.3 Generator-Image Mapping Rule for Homomorphisms . . 161
7.4 Composing Mappings 163
7.5 Recursive Definitions 163

8. Functions and Procedures 167
8.1 Arguments and Invocations 168
8.2 User-Defined Functions 170
8.2.1 Constructor Form of Function Expression 171
8.2.2 Statement Form of Function Expression 174
8.3 User-Defined Procedures 176
8.3.1 Constructor Form of Procedure Expression 176
8.3.2 Statement Form of Procedure Expression 177
8.3.3 Reference Arguments in a Procedure 178
8.3.4 Early Return from a Procedure Invocation 180
8.4 Local and Non-Local Identifiers 181
8.5 Functions and Procedures as First-Class Objects 183
8.6 Recursion 183
8.7 The **forward**-declaration 185
8.8 Function Assignment and Procedure Assignment 187
8.9 Error Messages 188
8.10 Parameters 191
8.10.1 Parameters in Function or Procedure Calls 191
8.10.2 Parameters in Function or Procedure Definitions 193
8.11 The **assert**-statement 195

9. Iterative Statements 197
9.1 Methods of Iteration 197
9.2 The **while**-statement 199
9.3 The **repeat**-statement 202
9.4 The **for**-statement 205
9.4.1 Properties of Loop Identifiers 207
9.4.2 Using Sequences as Domains of Iteration 208
9.5 Exiting an Iteration or Loop Quickly 210
9.5.1 Exiting the Current Iteration Quickly 210
9.5.2 Exiting the Whole Loop Quickly 210
9.5.3 Exits in Nested Loops 212
9.6 Iterations Without Iterative Statements 213

10. Intrinsics, Signatures and Packages 215
10.1 Signatures of Intrinsics 216
10.2 Operators Regarded as Intrinsics 217
10.3 Listing Signatures 218
10.4 Creating User Intrinsics 218
10.5 Package Files 221

10.6 Attaching a Package File 222
10.7 Changing and Updating a Package File 224
10.8 Testing for Errors in Intrinsics 225
10.9 Package Spec Files 231
10.9.1 Structure of Spec Files 231
10.9.2 Using Spec Files 232
10.10 Sharing Constants Among Package Files 233

Part III. The User Interface

11. Online Help 237
11.1 Single Online Help Requests 238
11.2 The Online Help Browser System 239
11.2.1 The Online Help Nodes 239
11.2.2 Finding a Node 242
11.2.3 Walking the Online Help Tree 247
11.2.4 The Kinds of Help Nodes 248
11.2.5 Cross-References 249
11.2.6 Tab Completion in the Browser 250
11.2.7 Recalling Nodes 252
11.2.8 Summary of Browser Commands 254
11.3 Other Features of Single Online Help Requests 255

12. Strings 257
12.1 Creating a String 257
12.2 Operations on Strings 258
12.3 Special Characters in Strings 262
12.4 Analysis of Strings 263
12.4.1 String Conversions 263
12.4.2 Splitting into Substrings 264
12.4.3 Regular Expressions 265

13. Printing and User Input 269
13.1 Levels of Printing in the **print**-statement 269
13.2 Printing an Object to a String 270
13.3 Verbose Printing 271
13.4 Formatted Printing 272
13.5 Reading Input from the User 274

14. Files and External Processes 277
14.1 The Current Directory 277
14.2 Loading Files During a Session 277
14.3 Files and Identifiers at Start-Up 278
14.3.1 Loading Files at Start-Up 278

14.3.2 Assigning Identifiers at Start-Up 279
14.3.3 Using a Standard Start-Up File 279
14.4 Echoing the Input from Files 280
14.5 The Output File and Log File 281
14.6 Reading and Writing to Data Files 281
14.6.1 Read/Write Operations on a Whole File 281
14.6.2 Read/Write Operations on Parts of a File 282
14.7 Saving the Workspace of a Session 285
14.8 External Processes 285
14.8.1 System Calls 285
14.8.2 Pipes 286
14.8.3 Temporary Names and MAGMA Processes 287

15. The User Environment 289
15.1 Tab Completion 289
15.2 Editing Lines of Input 290
15.2.1 Editing the Current Input Line 290
15.2.2 Editing a Previous Input Line 291
15.3 Interrupting a Computation or Forcing a Stop 293
15.4 Timing MAGMA Operations 294
15.5 Scrolling Output 294
15.6 Memory Limit 295

Part IV. Appendices

16. Reserved Words 299

17. Precedence of Operators 301

18. Summary of the Grammar 303

Part V. Rings and Fields

19. Overview of Rings and Fields 317
19.1 Creating Rings and Fields 317
19.1.1 Standard creation functions 317
19.1.2 Recursive ring constructions 319
19.1.3 Ring constructors 319
19.1.4 Other associated ring constructions 322
19.2 Operations on Ring Elements 324
19.3 Testing Properties of a Ring 328
19.4 Operations on Various Kinds of Rings 330
19.4.1 Finite Rings 330

19.4.2 Ordered Rings 331
19.4.3 Euclidean Rings 331
19.4.4 Unique Factorization Domains 332

20. Ring of Integers and Field of Rationals 335
20.1 Creating Integers and Rationals 335
20.2 Manipulating Integers and Rationals 337
20.2.1 Arithmetic 337
20.2.2 Elementary Functions 339
20.2.3 Arithmetical Functions 343
20.2.4 Factorizations 345
20.2.5 Modular Arithmetic 348
20.3 Residue Class Rings 350
20.4 Primes and Factorization 353
20.4.1 Primality versus Compositeness 354
20.4.2 Factorization 356
20.4.3 Other Functions Related to Primes and Factorization . 360
20.5 Notes. 361

21. Univariate Polynomial Rings 363
21.1 Constructing Polynomial Rings 364
21.2 Creating Polynomials 364
21.3 Univariate Rational Function Fields 365
21.4 Operations on Univariate Polynomials 366
21.5 Functions for Special Types of Polynomial Rings 367
21.6 Factorization and Root-Finding 369
21.6.1 Factorization 369
21.6.2 Hensel Lifting 372
21.6.3 Roots of Univariate Polynomials 374

22. Multivariate Polynomial Rings 377
22.1 Polynomial Rings and Monomial Orders 377
22.2 Polynomial Creation and Access 379
22.3 Factorization, Resultants and Derivatives 382
22.4 Multivariate Function Fields 385
22.5 Ideals 386
22.6 Ideal Access and Arithmetic 390
22.7 Quotient Rings 391
22.8 Varieties and Elimination 393
22.9 Radical and Decomposition of Ideals 398
22.10Graded Polynomial Rings and Hilbert Series 401
22.11Dimension of Ideals 404

23. Power Series and Laurent Series 407
23.1 Power Series and Laurent Series 407
23.2 Constructing Series Rings and Their Elements 408
23.2.1 Free Precision Series Rings 409
23.2.2 Fixed Absolute Precision Power Series Rings 412
23.2.3 Limited Relative Precision Laurent Series Rings 412
23.3 Operations on Series 413
23.4 Transcendental Functions 415
23.5 Series as Generating Functions 417

24. Finite Fields 419
24.1 Finite Fields by Cardinality 419
24.1.1 Constructing Finite Fields 419
24.1.2 Creation of Elements 420
24.2 Finding Special Elements 422
24.3 Extensions of Finite Fields 423
24.3.1 Extensions by a Given Defining Polynomial 423
24.3.2 Extensions by a Given Degree 425
24.3.3 Splitting Fields 426
24.3.4 Roots of Unity in Extension Fields 428
24.4 Subfields of Finite Fields 428
24.5 Embedding a Finite Field in Another Finite Field 429
24.6 Associated Polynomials 432
24.7 Operations on Finite Field Elements 434
24.8 Printing Finite Field Elements 436
24.9 Finite Fields as Vector Spaces 437
24.10Finite Fields as Matrix Algebras 439

25. Number Fields 443
25.1 Elementary Constructions of Number Fields 444
25.1.1 Number Fields 444
25.1.2 Quadratic Fields 446
25.1.3 Cyclotomic Fields 447
25.2 Equation Orders and Maximal Orders 449
25.2.1 Orders in Quadratic Fields 451
25.2.2 Orders in Number Fields 453
25.3 Elements of Number Fields and Their Orders 455
25.4 Advanced Constructions of Fields and Elements 458
25.4.1 Cyclotomic Fields and Roots of Unity 458
25.4.2 Subfields of Cyclotomic Fields 460
25.4.3 Relative Extensions and Subfields 464
25.5 Ideals of General Number Fields 470
25.6 Unit Group and Class Group 473
25.6.1 Units in Quadratic Orders 475
25.6.2 Class Groups of Quadratic Fields 476

25.6.3 Units of General Number Fields 479
25.6.4 Class Groups of General Number Fields 481

26. The Real and Complex Fields 485
26.1 The Two Kinds of Real and Complex Fields 485
26.2 Working in the Default Real Field 486
26.3 The Free Real Field 488
26.4 The Fixed Precision Real Fields 491
26.5 The Complex Fields 494
26.6 Elementary Real and Complex Operations 495
26.7 Standard Real Constants 497
26.8 Advanced Real and Complex Functions 497
26.8.1 Transcendental Functions 497
26.8.2 Trigonometric and Hyperbolic Functions 498
26.8.3 Summation of Infinite Series 499
26.8.4 Continued Fractions 500
26.8.5 Special Functions 500
26.8.6 Example: The Prime Number Theorem 502

Part VI. Modules and Algebras

27. Vector Spaces and Matrix Spaces 507
27.1 Constructing the Full Vector Space 507
27.2 Constructing the Full Matrix Space 508
27.3 Tuple Spaces and Matrix Spaces over a Ring 510
27.4 Vectors and Matrices 510
27.4.1 Creating Vectors and Matrices 510
27.4.2 Arithmetic and Functions 512
27.4.3 Indexing Vectors and Matrices 512
27.4.4 Blocks within Matrices 513
27.5 Subspaces and Quotient Spaces 515
27.6 Operations on Vector Spaces 516
27.7 Operations with Linear Transformations 517
27.8 Row and Column Operations 521
27.9 Simultaneous Systems of Linear Equations 522
27.10Changing the Basis 523
27.10.1Defining the Chosen Basis 523
27.10.2Constructing a Basis Gradually 525
27.11Changing the Coefficient Field 526

28. Matrix Algebras 527
28.1 The Generic Matrix Algebra and Its Elements 527
28.1.1 Constructing the Generic Matrix Algebra 527
28.1.2 Constructing Matrices 527
28.1.3 Arithmetic on Matrices 529
28.2 Constructing Other Matrix Algebras 530
28.2.1 Subalgebras 530
28.2.2 Ideals 531
28.2.3 Quotients 533
28.2.4 Other Matrix Algebra Constructions 534
28.3 General Facts About a Matrix Algebra 535
28.4 Linear Algebra Concepts 537

Part VII. Groups

29. Overview of Groups 543
29.1 Construction of Groups 544
29.1.1 Abstract Groups 544
29.1.2 Concrete Groups 545
29.1.3 Some Standard Groups 547
29.2 Operations on Group Elements 549
29.2.1 Arithmetic 549
29.2.2 Boolean Operations 551
29.2.3 Order of an Element 551
29.3 Generating Random Elements 552
29.4 Constructing a Subgroup 553
29.4.1 Defining a General Subgroup 553
29.4.2 Constructing a Normal Closure 554
29.4.3 Constructing a Standard Subgroup 554
29.5 Constructing a Quotient Group 555
29.5.1 General Quotients 555
29.5.2 Special Quotients 556
29.6 Order and Index Functions 557
29.7 Properties of Groups and Subgroups 559
29.8 Characteristic Subgroups, Series and Normal Structure 559
29.8.1 Characteristic Subgroups 559
29.8.2 Series 561
29.9 Abstract Structure of a Group 563
29.10Conjugacy Classes of Elements 566
29.10.1Testing Conjugacy 566
29.10.2Constructing One Class 567
29.10.3Constructing All the Classes 567
29.10.4The Class Map and Power Map 569
29.11Conjugacy Classes of Subgroups 571

29.11.1Determining the Conjugacy Classes 571
29.11.2The Poset of Subgroup Classes 574
29.11.3Normal Subgroups 578
29.12Characters 578
29.12.1Class Functions and Characters 578
29.12.2Creating Individual Characters 579
29.12.3Operations on Characters 579
29.12.4Properties of Characters 581
29.12.5Automatic Determination of Irreducible Characters . . . 581
29.12.6Standard Constructions for Characters 584
29.12.7Decomposition of Tensor Powers 585
29.12.8Interactive Determination of Irreducible Characters . . . 587
29.13Transferring Between Group Categories 587
29.13.1Transferring to the Polycyclic Group Category 588
29.13.2Transferring to the Permutation Group Category 588

30. Finitely-Presented Groups 591
30.1 Free Groups and Words 592
30.1.1 Constructing the Free Group 592
30.1.2 Computing with Words 593
30.2 Quotients and Construction of Finitely-Presented Groups 595
30.2.1 Relations 595
30.2.2 The **quo**-constructor 596
30.2.3 The Group Constructor 598
30.2.4 Extensions, Products and Standard Groups 599
30.2.5 Presentations for Concrete Groups 600
30.2.6 Access Functions for Finitely Presented Groups 601
30.3 Special Quotients 601
30.3.1 Abelian Quotients 601
30.3.2 p-Quotients 602
30.3.3 Other Quotients 605
30.4 Subgroups 605
30.4.1 The **sub**-constructor 605
30.4.2 The Todd-Coxeter Algorithm 606
30.4.3 Coset Spaces 609
30.4.4 Subgroup Constructions 611
30.5 Permutation Representations 613
30.6 Presentations for Subgroups 615
30.7 Simplifying a Presentation 618
30.8 Locating Subgroups of Finite Index 619
30.8.1 Subgroups of Low Index 619
30.8.2 The Low Index Subgroups Process 622
30.9 Finite Groups 624
30.9.1 Determining the Order 624
30.9.2 Constructing a Representation 625

31. Abelian Groups 627
31.1 Constructing the Free Abelian Group 627
31.2 Computing with Abelian Group Elements 627
31.3 Subgroups and Finitely-Presented Groups 629
31.4 Structure of Abelian Groups 631
31.5 Further Operations and Access Functions 634

32. Permutation Groups 637
32.1 Group Actions, Permutation Groups and G-Sets 638
32.2 The Symmetric Group and Permutations 638
32.2.1 Creating the Symmetric Group 638
32.2.2 Representation of Permutations 639
32.2.3 Arithmetic on Permutations 640
32.3 General Permutation Group Constructions 642
32.3.1 Subgroups and the Permutation Group Constructor . . 642
32.3.2 Standard Groups 643
32.3.3 Projective and Affine Groups 643
32.3.4 Product Constructions 645
32.4 Example: The Group of Symmetries of the Cube 646
32.5 Basic Operations with Permutation Groups 647
32.5.1 Computing Images 647
32.5.2 Accessing a Permutation Group 647
32.6 Orbits and Transitivity 648
32.6.1 Transitivity and Regularity 649
32.6.2 Orbits and Stabilizers 650
32.6.3 The Induced Action on an Orbit 654
32.7 Systems of Imprimitivity 655
32.7.1 Blocks and Block Systems 655
32.7.2 The Induced Action on a Block System 657
32.7.3 Example: A Reduction Algorithm 659
32.8 General G-sets 660
32.8.1 Construction of a G-set 660
32.8.2 Orbits and Stabilizers for General G-Sets 663
32.8.3 The Induced Action on a G-set 664
32.9 Primitive Groups and the O'Nan-Scott Theorem 665
32.9.1 Recognition of the Alternating and Symmetric Groups 665
32.9.2 Primitive Groups with Abelian Socle 666
32.9.3 Primitive Groups with Non-Abelian Socle 667
32.10Base and Strong Generating Sets 669

33. Matrix Groups 671
33.1 The General Linear Group and Its Elements 672
33.1.1 Creating the General Linear Group 672
33.1.2 Representation of Matrices 673
33.1.3 Arithmetic on Matrices 673

33.1.4 The Order of a Matrix 674
33.1.5 Matrix Invariants and Canonical Forms 675
33.2 General Matrix Group Constructions 676
33.2.1 Constructing Subgroups 676
33.2.2 The Matrix Group Constructor 678
33.2.3 Normal Subgroups and Quotients 679
33.2.4 Families of Classical Groups 680
33.2.5 Product Constructions 682
33.3 Access Functions for Matrix Groups and Elements 682
33.4 Matrix Rings and Matrix Groups 683
33.5 Finite Groups of Small Degree 684
33.5.1 Base and Strong Generating Set 684
33.6 The Natural R-module 686
33.6.1 Action on the Natural R-module 687
33.6.2 Orbits and Stabilizers 688
33.6.3 The Induced Action on an Orbit 690
33.7 The Natural $K[G]$-module 692
33.8 Submodules 692
33.9 Actions Induced from a Submodule 694
33.10Matrix Groups of Large Degree over a Finite Field 694
33.10.1Recognizing Classical Groups 694
33.10.2The Aschbacher Families 695
33.11Invariants for Finite Groups 696

Part VIII. Algebraic Geometry

34. Elliptic Curves 703
34.1 Some Algebraic Geometry 703
34.2 Elliptic Curves over Arbitrary Fields 705
34.2.1 Creation and Basic Quantities 705
34.2.2 Points on Elliptic Curves 705
34.3 Elliptic Curves over The Rationals 708
34.3.1 Minimal Models 708
34.3.2 Local Information 710
34.3.3 The Mordell-Weil Group 712

Part IX. Geometric and Combinatorial Structures

35. Enumerative Combinatorics 719
35.1 Elementary Counting Functions 719
35.2 Subsets of a Finite Set 720
35.3 Partitions 722
35.4 Example: Bell Numbers 724

36. Overview of Incidence Structures 727
36.1 Points and Blocks of Incidence Structures 727
36.2 Equality and Isomorphism 729
36.3 Automorphism Groups and Subgroups 730
36.3.1 Action on Points 730
36.3.2 Action on Blocks 732
36.3.3 Subgroups of the Automorphism Group 732

37. Graphs 735
37.1 Constructing Graphs 735
37.2 Sets of Vertices and Edges 737
37.3 Creating New Graphs 739
37.3.1 Addition and Subtraction 739
37.3.2 Subgraph 740
37.3.3 Quotient Graph 742
37.3.4 Some Standard Constructions 743
37.4 Testing Properties of a Graph 744
37.5 Incidence, Adjacency and Distance Matrices 744
37.6 Distance and Connectedness 746
37.7 Vertex Subsets and Colourings 747
37.8 Automorphism Group of a Graph 748
37.8.1 Edge Group 751
37.8.2 Canonical Graph and Isomorphic Graphs 752

38. Designs 753
38.1 The Categories of Incidence Structures 753
38.2 Creating Incidence Structures 754
38.3 Points and Blocks 759
38.4 Elementary Invariants of a Design 763
38.5 Incidence Structures with Particular Properties 764
38.6 Producing New Incidence Structures 766
38.7 Standard Constructions for Designs 767
38.8 Difference Sets and Their Development 769
38.9 Equality and Isomorphism 772
38.10Automorphism Group 774
38.11Relationship to Graphs and Linear Codes 776

39. Finite Planes 777
39.1 Constructing Projective and Affine Planes 777
39.1.1 Planes from Incidence Relations 777
39.1.2 Planes from Incidence Structures 780
39.1.3 Classical Planes from Vector Spaces 782
39.2 Equality and Isomorphism 783
39.3 Points and Lines 784
39.3.1 General Point and Line Operations 784

39.3.2 Operations for Classical Planes 790
39.4 Properties of a Plane 791
39.5 Subplanes 792
39.6 Constructing New Planes from Existing Planes 794
39.7 Arcs, Conics and Unitals 795
39.8 Group Actions and Collineation Groups 798
39.9 Relationship to Designs, Graphs and Codes 799

40. Error-Correcting Codes 801
40.1 Defining a Linear Code 801
40.1.1 Defining a Code from a Vector Space 802
40.1.2 Defining a Code from a Generator Matrix 803
40.1.3 Defining a Code from a Geometrical Structure 803
40.2 Calculations with Codewords and Vectors 804
40.3 General Facts about a Linear Code 806
40.4 Families of Linear Codes 808
40.4.1 Some Standard Codes 809
40.4.2 Cyclic Codes 810
40.4.3 Alternant and Generalized BCH Codes 811
40.5 Constructing Codes from Other Codes 813
40.6 The Weight Distribution 818
40.7 Group Actions, Automorphism and Equivalence 820
40.7.1 Group Actions on Codes 820
40.7.2 Equivalent and Isomorphic Codes 823
40.7.3 Automorphism Groups 824
40.8 Encoding and Decoding 827
40.8.1 Encoding 827
40.8.2 Errors in Transmission 829
40.8.3 Elementary Syndrome-Decoding Techniques 829
40.8.4 Automated Decoding 830
40.8.5 Simulation of Message Transmission 833
40.9 Advanced Facilities 835
40.10Notes on the Algorithms 836

References 837

Part I

Overview

1. Getting Started With MAGMA

1.1 Entering the MAGMA System

Access to MAGMA is dependent upon how it has been installed at the user's institution. Generally speaking, the user can enter MAGMA simply by going to a 'shell prompt' on the computer screen (the basic location for typing operating system commands) and typing the word

```
magma
```

then pressing the key marked 'return' or 'enter', which finishes the line. If this does not work, local experts should be consulted for advice.

1.2 Input and Output

Commands in MAGMA are known as *statements*. Each complete statement *must* finish with a semicolon (`;`). It is customary to place each statement on a separate line, by pressing the return key after the semicolon, except in the case of a succession of short statements. MAGMA will not *execute* (perform) the statement(s) on the current line until the return key has been pressed.

Whenever MAGMA is ready to receive a statement from the user, it displays a *prompt symbol* on the left of the input line. The prompt usually looks like this:

```
>
```

Statements may be typed following the prompt symbol. For instance, to find the sum of 2 and 4, the user should type `2 + 4;` after the prompt symbol, then press the return key. MAGMA will execute this statement as soon as the return key has been typed, and the screen will look like this:

```
> 2 + 4;
6
```

where the 6 is the output from MAGMA. MAGMA has evaluated the expression 2 + 4 and given the result as output.

If output different to the above appears on the user's screen, then a typing error has been made. The user should try again, correcting the error. If there is no output, the most likely reason is that the semicolon at the end of the line has been forgotten. In this case, the user should type a semicolon, and then press the return key.

Examples of MAGMA interactive sessions presented in this book will be normally be displayed as shown above. *Any line beginning with a prompt symbol has been typed by the user (not including the prompt itself), and any line without the prompt symbol at the beginning is* MAGMA*'s response to the user's input.*

In the following lines, the user first assigns the integer ring $\mathbb{Z}$ to the identifier Z, so that Z has the value $\mathbb{Z}$, then assigns the univariate polynomial ring over Z to P, naming its indeterminate x. After the assignments, the user indicates that P is to be printed by simply typing P. A polynomial product is then defined and printed in the one statement. Finally, the user asks for the factorization of $x^{12}-x^8-x^4+1$ to be printed (where the x is the indeterminate of P), and MAGMA prints the factorization $(x-1)^2(x+1)^2(x^2+1)^2(x^4+1)$:

```
> Z := IntegerRing();
> P<x> := PolynomialRing(Z);
> P;
Univariate Polynomial Algebra in x over Integer Ring
> (x^6 - 5*x^2 + 2) * (17*x^3 - 1);
17*x^9 - x^6 - 85*x^5 + 34*x^3 + 5*x^2 - 2
> Factorization(x^12 - x^8 - x^4 + 1);
[
    <x - 1, 2>,
    <x + 1, 2>,
    <x^2 + 1, 2>,
    <x^4 + 1, 1>
]
```

1.3 Creating Structures and Their Elements

Since all computation in MAGMA takes place in one or more algebraic structures, the first step in any computation involves defining the necessary *magmas* (algebraic structures). Once the magmas have defined, elements and other related objects of these magma may be created. Exceptions are certain automatically-created magmas, such as the ring of integers and the monoid of

character strings, which need not be defined explicitly prior to working with their elements. Some examples of the creation of magmas and their elements are given below:

```
> FF<t> := FunctionField(RationalField());
> FF;
Rational function field of rank 1 over Rational Field
Variables: t
> (3*t^2 - 6*t) / (9*t^2 - 9*t);
(1/3*t - 2/3)/(t - 1)

> Qm5<rt5i> := QuadraticField(-5);
> Qm5;
Quadratic Field Q(rt5i)
> rt5i ^ 2;
-5

> S6 := SymmetricGroup(6);
> S6;
Symmetric group S6 acting on a set of cardinality 6
Order = 720 = 2^4 * 3^2 * 5
> Random(S6);
(3, 5, 4, 6)
> Order(S6 ! (4,1,3)(2,6,5) );
3
```

1.4 Online Help and Environment

There are several short-cuts and other ways of making the user's task easier when using MAGMA. Since it would disturb the flow of the book to discuss them here, and none of the techniques are strictly essential for operating MAGMA, their description is deferred to Chapter 11, which explains the *online help system*, and Chapter 15, which explains other features of the *user environment*. The reader is advised to glance at these chapters soon, and study them in detail later.

1.5 Quitting MAGMA

The command for finishing a MAGMA session is

```
> quit;
```

The semicolon, followed by the return key, is compulsory. The MAGMA run will then be terminated.

An alternative method of quitting is to type the character control-d at the beginning of a line. In this case, typing the return key is not required.

2. Developed Examples

'Why,' said the Dodo, 'the best way to explain it is to do it.'

Alice in Wonderland
Lewis Carroll

This chapter consists of several developed examples of MAGMA, giving an informal taste of the language and algorithmic capabilities of the system. Readers who enjoy inductive learning may wish to try these examples themselves on their own copies of MAGMA.

2.1 Affine Plane from a Projective Plane by Derivation

In the area of finite geometry, MAGMA currently offers facilities for affine and projective planes. This example begins with a projective plane and constructs an affine plane from it by derivation. Apart from finite geometry, it demonstrates set operations, group actions, and linear codes.

We begin by creating the projective plane $PP = \mathrm{PG}(2, q)$, where $q = 16$, along with the point-set and line-set of the plane. Then the collineation group $G = \mathrm{PGL}(3, q)$ is formed in its action on the points of PP:

```
> q := 16;
> F<w> := FiniteField(q);
> PP, Pts, Lns := ProjectivePlane(F);
> PP;
Projective Plane PG(2, 16)
> #Pts, #Lns;
273 273

> G, gspt, gsln := CollineationGroup(PP);
> G;
Permutation group G acting on a set of cardinality 273
```

```
Order = 2^14 * 3^3 * 5^2 * 7 * 13 * 17
```

We next define a certain Hall oval, and print its points. We choose points P and Q on the oval and find the line PQ that contains P and Q. Then we choose a point X not on the line PQ: this is done by choosing another point on the oval:

```
> oval := { Pts | [1, x, w^4*x^14 + w^24*x^12 + w^12*x^10
>   + w^18*x^8 + w^10*x^6 + w^10*x^4 + w^12*x^2] : x in F }
>   join { Pts | [0,1,0], [0,0,1] };
> oval;
{ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ),
( 1 : w^7 : w^8 ), ( 1 : w^2 : w^9 ), ( 1 : w^11 : w^2 ),
( 1 : w^3 : w^13 ), ( 1 : 1 : 1 ), ( 1 : w^4 : w^3 ),
( 1 : w^10 : w^7 ), ( 1 : w^6 : w^14 ), ( 1 : w^14 : w ),
( 1 : w^13 : w^11 ), (1 : w^12 : w^6 ), ( 1 : w : w^4 ),
( 1 : w^8 : w^12 ), ( 1 : w^5 : w^10 ), ( 1 : w^9 : w^5 ) }

> P := Rep(oval);
> Q := Rep(Exclude(oval, P));
> P, Q;
( 1 : 0 : 0 ) ( 1 : w^13 : w^11 )
> PQ := Lns![P, Q];
> PQ;
< 0 : 1 : w^2 >

> X := Rep(oval diff {P, Q});
> X;
( 0 : 1 : 0 )
> XP := Lns![X, P];
> XQ := Lns![X, Q];
> XP, XQ;
< 0 : 0 : 1 >
< 1 : 0 : w^4 >
```

Now we construct: the group $H1$ of central collineations with axis PQ; the group $H2$ of elations with centre P and axis PQ; the group $H3$ of homologies with centre P and axis XQ; and the group $H4$ of central collineations with centre P and axis through Q:

```
> H1 := Stabilizer(G, gspt, Setseq(Set(PQ)));
> H2 := SylowSubgroup(Stabilizer(H1, gsln, XP), 2);
> H3 := Stabilizer(Stabilizer(G, gspt, P), gspt,
>   Setseq(Set(XQ)));
> H4 := sub< G | H2, H3 >;
```

We construct the set *afflines* containing the lines of the new plane. They are: the translates of *oval* (excluding P and Q) under the central collineations: the lines of $\mathrm{PG}(2, F)$ (excluding PQ) incident with P; and the lines of $\mathrm{PG}(2, F)$ (excluding PQ) incident with Q. Then we can construct the affine plane itself.

```
> afflines := Orbit(H4, gspt, oval diff {P, Q}) join
>   { Exclude(Set(l), Y) : l in Lns, Y in {P,Q} |
>   l ne PQ and Y in l };
> #afflines;
272
> affpts := &join afflines;
> #affpts;
256
> affpl := AffinePlane< SetToIndexedSet(affpts) |
>   Setseq(afflines) >;
> affpl;
Affine Plane of order 16
```

Finally, we check that the plane is desarguesian by calculating the p-rank, which equals the dimension of the corresponding linear code:

```
> C := LinearCode(affpl, PrimeField(F));
> Dimension(C);
81
```

2.2 Constructing an Endo-trivial Module

This example, which is due to Jon Carlson, illustrates some of the module machinery in Magma. The idea is to test a technique for constructing endo-trivial modules. An endo-trivial module is one with the property that

$$Hom_k(M, M) = M \otimes Dual(M)$$

is the direct sum of a trivial module and a projective (free, in this case) module.

First we construct an extraspecial group of order 27 and exponent 3:

```
> ps := PSL(3, 3);
> ps;
Permutation group ps acting on a set of cardinality 13
    (1, 10, 4)(6, 9, 7)(8, 12, 13)
```

```
    (1, 3, 2)(4, 9, 5)(7, 8, 12)(10, 13, 11)
> g := SylowSubgroup(ps, 3);
> g;
Permutation group g acting on a set of cardinality 13
Order = 27 = 3^3
    (3, 13, 9)(5, 8, 6)(7, 11, 12)
    (2, 5, 3)(4, 8, 9)(6, 13, 10)
```

Now we create the module in question. It is the kernel δ_x in an exact sequence

$$0 \to \delta_x \to x \to k \to 0$$

where k is the trivial $f3[g]$-module and x is a permutation module whose point stabilizer is a noncentral cyclic subgroup:

```
> g.1 in Centre(g);
false
> h := sub<g | g.1>;
> h;
Permutation group h acting on a set of cardinality 13
    (2, 10, 4)(5, 8, 6)(7, 12, 11)
> F3 := GaloisField(3);
> x := PermutationModule(g, h, F3);
> hhh := GHom(x, TrivialModule(g, F3));
> hhh;
KMatrixSpace of 9 by 1 GHom matrices and dimension 1
over GF(3)

> delx := Kernel(hhh.1);
> delx;
GModule delx of dimension 8 over GF(3)
> xx := TensorProduct(delx, delx);
> xx;
GModule xx of dimension 64 over GF(3)
```

Now we want to decompose the tensor product of δ_x with itself. One of the summands should be an endo-trivial module. Note that the dimension of an endo-trivial module cannot be divisible by the prime 3, since the square of the dimension must be 1 plus a multiple of 27 (the order of the group g). The function **IsDecomposable** tests whether its argument is decomposable, and if this is the case then it also provides a decomposition as the second and third return values:

```
> a, m1, m2 := IsDecomposable(xx);
> a;
```

```
true
> m1, m2;
GModule m1 of dimension 9 over GF(3)
GModule m2 of dimension 55 over GF(3)
```

We want to check what the pieces are. We suspect that the module of dimension 9 is just a copy of our permutation module, and the check below confirms that. Then we proceed with the other piece.

```
> IsIsomorphic(m1, x);
true

> a,m3,m4 := IsDecomposable(m2);
> a, m3, m4;
true
GModule m3 of dimension 27 over GF(3)
GModule m4 of dimension 28 over GF(3)
```

We suspect this time that the module of dimension 27 is a free module. We use the theorem that the free module is the only module with the property that it is generated by a single element and has dimension equal to the order of the group. So we try a couple of times to see if it can be generated by a single element:

```
> sub< m3 | Random(m3) >;
GModule of dimension 21 over GF(3)
> sub< m3 | Random(m3) >;
GModule m3 of dimension 27 over GF(3)
```

So $m3$ is a free module. We can proceed.

```
> IsDecomposable(m4);
false
```

Now we check to see if $m4$ is endo-trivial:

```
> et := TensorProduct(m4, Dual(m4));
> et;
GModule et of dimension 784 over GF(3)
> Quotrem(Dimension(et), #g);
29 1
```

So the dimension is 1 more than a multiple (29) of the order of g (27), as expected.

We know that the tensor product of $m4$ with its dual has a direct summand isomorphic to the trivial module. If it is endo-trivial then the tensor of it with its dual must be one copy of the trivial module plus $(\mathrm{Dim}(et) - 1)/27 = 29$ copies of the free module. So the action of the group algebra must have exactly $29 + 1 = 30$ fixed points. We check:

```
> Fix(et);
GModule of dimension 30 over GF(3)
```

Actually at this point we can be certain that $m4$ is an endo-trivial module. Just to be sure, we factor out projective modules to see if we get down to the trivial module. We are using here the fact that the group ring is self-injective and hence any free submodule (module of dimension 27 generated by one element) is a direct summand.

```
> ww := et;
> Dim := Dimension; // shorthand
> repeat
>     sum := rep{s : i in [1..100] | Dim(s) eq 27
>       where s is sub< ww | Random(ww) >};
>     qq := quo< ww | sum >;
>     (Dim(et) - Dim(qq)) / #g, Dim(qq);
>     ww := qq;
> until Dim(qq) eq 1;
1 757
2 730
3 703
4 676
   [ etc ]
26 82
27 55
28 28
29 1
```

Finally we want to check that the module $m4$ is not one of the known endo-trivial modules. It is enough to see that it does not have the same restriction to all of the maximal elementary abelian subgroups. So we calculate all the maximal elementary abelian 3-subgroups. and then check the dimension of the fixed point set on each.

```
> cc := Centre(g);
> max1:= sub< g | g.1, cc >;
> max2:= sub< g | g.2, cc >;
> max3:= sub< g | g.1*g.2, cc >;
> max4:= sub< g | g.1*g.2^2, cc >;
```

```
> [ Fix(Restriction(m4, x)) :
>   x in [max1, max2, max3, max4] ];
[
    GModule of dimension 6 over GF(3),
    GModule of dimension 4 over GF(3),
    GModule of dimension 4 over GF(3),
    GModule of dimension 4 over GF(3)
]
```

Notice that the single fixed-point space of dimension 6 corresponds to the restriction of $m4$ to the maximal subgroup containing the subgroup h with which we started.

2.3 Molien Series and Primary Invariants

In this example we define a 4-dimensional reflection group G of order 92160 and find primary invariants for the ring of invariants of G. We start by constructing the Molien series of G, both as a rational function and as a power series; from this, it emerges that possible primary invariants have degrees 8, 24, 24, and 40. Next, we construct linearly independent invariants of these degrees. We then use the *Hilbert-driven Buchberger* algorithm to show that the ideal generated by these invariants is zero-dimensional, since the numerator of the Hilbert Series of the ideal is $(1-t^8)(1-t^{24})(1-t^{24})(1-t^{40})$ as expected. Since the ideal is zero-dimensional, the four invariants must be primary invariants for G. Note also that the whole example can be done automatically in one go by constructing the invariant ring R of G and then calling the function **PrimaryInvariants**(R) – this example just demonstrates one way of experimenting within MAGMA.

We create the cyclotomic field $K = \mathbb{Q}(\zeta_8)$, and a matrix group G over K generated by 4 matrices:

```
> K<zeta> := CyclotomicField(8);
> G := MatrixGroup<4, K |
>            [ h, h, h, h,
>              h,-h, h,-h,
>              h, h,-h,-h,
>              h,-h,-h, h ] where h is 1/2,
>
>            [-1, 0, 0, 0,
>              0, 1, 0, 0,
>              0, 0, 1, 0,
>              0, 0, 0, 1 ],
```

```
>
>            [ 1, 0, 0, 0,
>              0, w, 0, 0,
>              0, 0, 1, 0,
>              0, 0, 0, w ] where w is zeta^2,
>
>            [ 1, 0, 0, 0,
>              0, 1, 0, 0,
>              0, 0, 1, 0,
>              0, 0, 0, 1 ] where 1 is zeta >;
>
> Order(G);
92160
> FactoredOrder(G);
[ <2, 11>, <3, 2>, <5, 1> ]
```

The Molien series calculation needs the conjugacy classes of G. We call the **Classes** function, specifying that the classes C are to be found as orbits under conjugation action. We do this since the default method would be much slower for this group.

```
> C := Classes(G: Al := "Action");
> #C;
118
```

Once the conjugacy classes of G have been found, they will be remembered for subsequent calculations.

We are now in a position to compute the Molien series of G, initially as a rational function and then as a power series:

```
> MS<t> := MolienSeries(G);
> MS;
(t^32 + 1)/(t^96 - t^88 - 2*t^72 + 2*t^64 - t^56 + 2*t^48
    - t^40 + 2*t^32 - 2*t^24 - t^8 + 1)

> P<x> := PowerSeriesRing(IntegerRing(), 200);
> P ! MS;
1 + x^8 + x^16 + 3*x^24 + 4*x^32 + 5*x^40 + 8*x^48 +
    10*x^56 + 12*x^64 + 17*x^72 + 21*x^80 + 24*x^88 +
    31*x^96 + 37*x^104 + 42*x^112 + 52*x^120 + 60*x^128 +
    67*x^136 + 80*x^144 + 91*x^152 + 101*x^160 + 117*x^168
    + 131*x^176 + 144*x^184 + 164*x^192 + O(x^200)
```

It is known that the Molien series can be written in the form

$$\frac{1+t^m}{(1-t^{s_1})(1-t^{s_2})\cdots(1-t^{s_d})}$$

where d is the degree of the matrix representation. Therefore we undertake a partial factorization of the denominator D as the product of polynomials of the form $(1-x^k)$ for various k. We determine each k by taking the degree of the first non-constant term of D and then dividing out by $(1-x^k)$.

```
> D<t> := Denominator(MS);
> F := D div (1 - t^8);
> F;
-t^88 + 2*t^64 + t^48 - t^40 - 2*t^24 + 1
> F := F div (1 - t^24);
> F;
t^64 - t^40 - t^24 + 1
> F := F div (1 - t^24);
> F;
-t^40 + 1
> // Check the product
> D eq (1 - t^8)*(1 - t^24)*(1 - t^24)*(1 - t^40);
true
```

So the Molien series for G may be written as

$$\frac{1+t^{32}}{(1-t^8)(1-t^{24})(1-t^{24})(1-t^{40})}.$$

Therefore the degrees 8, 24, 24 and 40 should be tried to find primary invariants.

We now proceed to form linearly independent invariants with these degrees. The first invariant can be computed with the function **ReynoldsOperator**, and the others can be constructed using **InvariantsOfDegree**: which successively calls **ReynoldsOperator** efficiently to obtain linearly independent invariants of the desired degree.

```
> P<x1, x2, x3, x4> := PolynomialRing(K, 4);
> time i8 := ReynoldsOperator(x1^8, G);
Time: 69.980
> i8;
1/32*x1^8 + 7/16*x1^4*x2^4 + 7/16*x1^4*x3^4 + 7/16*x1^4*x4^4
    + 21/4*x1^2*x2^2*x3^2*x4^2 + 1/32*x2^8 + 7/16*x2^4*x3^4
    + 7/16*x2^4*x4^4 + 1/32*x3^8 + 7/16*x3^4*x4^4 +
    1/32*x4^8
> L8 := [32 * i8];
> time L24 := InvariantsOfDegree(G, P, 24, 2);
```

```
Time: 221.849
> time L40 := InvariantsOfDegree(G, P, 40, 1);
Time: 1591.239
> L := L8 cat L24 cat L40;
> L;
[
    x1^8 + 14*x1^4*x2^4 + 14*x1^4*x3^4 + 14*x1^4*x4^4 +
        168*x1^2*x2^2*x3^2*x4^2 + x2^8 + 14*x2^4*x3^4 +
        14*x2^4*x4^4 + x3^8 + 14*x3^4*x4^4 + x4^8,
    x1^24 - 35420/771*x1^18*x2^2*x3^2*x4^2 +
            ...
        + 2576*x3^12*x4^12 + 759*x3^8*x4^16 + x4^24,
    x1^20*x2^4 + x1^20*x3^4 + x1^20*x4^4 +
            ...
        + x3^4*x4^20,
    x1^40 + 38/109*x1^36*x2^4 + 38/109*x1^36*x3^4 +
            ...
        38/109*x3^4*x4^36 + x4^40
]
```

(The output above has been heavily edited, since the polynomials have many terms.)

Finally, we show that the ideal is zero-dimensional by checking that the Hilbert-driven Buchberger algorithm succeeds:

```
> time b, h := HilbertGroebnerBasis(L, D);
Time: 3979.420
> b;
true
```

This indicates that the ideal is zero-dimensional, so L must contain primary invariants for G.

2.4 Galois Group and Action

This example, due to Wieb Bosma, illustrates polynomial factorization, number fields, and the subgroup lattice, in the context of a Galois group computation.

If it is possible to obtain the full factorization of an integer polynomial over its splitting field, the Galois action of the group of the splitting field can be made entirely explicit. In this example we show how it can be done in MAGMA,

and how to find the Galois correspondence (between subgroups and subfields). Although there exists a polynomial-time algorithm for the factorization of polynomials over number fields, in practice this is the bottleneck for our approach to Galois theory. Only in small examples we will be able to construct the splitting field as below.

We begin with a cubic polynomial f and determine its Galois group using the intrinsic function **GaloisGroup**(f). This will be a degree-3 representation:

```
> R<x> := PolynomialRing(RationalField());
> f := x^3 - x - 1;
> GaloisGroup(f);
Permutation group G acting on a set of cardinality 3
Order = 6 = 2 * 3
    (1, 2)
    (1, 2, 3)
```

Next, we find the Galois group in two degree-6 representations. Firstly, we construct it bare-handed. We start by obtaining the splitting field for f as a two-step extension of the rational field $\mathbb{Q}$:

```
> N<n> := NumberField(f);
> ff := Factorization( PolynomialRing(N) ! f );
> ff;
[
    <$.1 - n, 1>,
    <$.1^2 + n*$.1 + n^2 - 1, 1>
]
> M<m> := ext< N | ff[2][1] >;
> A<a> := AbsoluteField(M);
> A;
Number Field with defining polynomial
x^6 - 6*x^4 + 9*x^2 + 23 over the Rational Field
```

We factorize f over the splitting field, and obtain all its roots from the linear factors:

```
> S<s> := PolynomialRing(A);
> factn := Factorization( S ! DefiningPolynomial(A) );
> factn;
[
    <s - a, 1>,
    <s + a, 1>,
    <s - 1/6*a^4 + 5/6*a^2 - 1/2*a - 2/3, 1>,
```

```
    <s - 1/6*a^4 + 5/6*a^2 + 1/2*a - 2/3, 1>,
    <s + 1/6*a^4 - 5/6*a^2 - 1/2*a + 2/3, 1>,
    <s + 1/6*a^4 - 5/6*a^2 + 1/2*a + 2/3, 1>
]
> C := [ -Coefficient(factr[1], 0) : factr in factn];
> C;
[
    a,
    -a,
    1/6*a^4 - 5/6*a^2 + 1/2*a + 2/3,
    1/6*a^4 - 5/6*a^2 - 1/2*a + 2/3,
    -1/6*a^4 + 5/6*a^2 + 1/2*a - 2/3,
    -1/6*a^4 + 5/6*a^2 - 1/2*a - 2/3
]
```

Each of the roots is an algebraic conjugate of the primitive element a of the field. The elements of the Galois group of A over $\mathbb{Q}$ are obtained by sending a to one of its conjugates. Thus the action on A of such an element of the Galois group is determined by the polynomial (with rational coefficients) expressing the conjugate in a. These polynomials are stored in P below. We (again) determine the Galois group: by numbering the algebraic conjugates c_i and finding all images of c_i under a given P_j, we obtain the permutation associated with P_j. The Galois group H consists of these permutations on 6 letters.

```
> P := [ R ! Eltseq(x) : x in C];
> P;
[
    x,
    -x,
    1/6*x^4 - 5/6*x^2 + 1/2*x + 2/3,
    1/6*x^4 - 5/6*x^2 - 1/2*x + 2/3,
    -1/6*x^4 + 5/6*x^2 + 1/2*x - 2/3,
    -1/6*x^4 + 5/6*x^2 - 1/2*x - 2/3
]
> I := [ [ Index(C, Evaluate(p, c)) : p in P ] : c in C];
> I;
[
    [ 1, 2, 3, 4, 5, 6 ],
    [ 2, 1, 4, 3, 6, 5 ],
    [ 3, 6, 1, 5, 4, 2 ],
    [ 4, 5, 2, 6, 3, 1 ],
    [ 5, 4, 6, 2, 1, 3 ],
    [ 6, 3, 5, 1, 2, 4 ]
]
```

```
> H := sub< Sym(6) | I >;
> H;
Permutation group H acting on a set of cardinality 6
    (1, 2)(3, 4)(5, 6)
    (1, 3)(2, 6)(4, 5)
    (1, 4, 6)(2, 5, 3)
    (1, 5)(2, 4)(3, 6)
    (1, 6, 4)(2, 3, 5)
```

Lastly, we find the Galois group G using the intrinsic function once more, but in terms of the degree-6 defining polynomial of the splitting field. It will be conjugate to H: a simple renumbering of roots makes them equal. As we see, we only need to cyclically permute three roots.

```
> G := GaloisGroup(DefiningPolynomial(A));
> fl, el := IsConjugate(Sym(6), G, H);
> fl, el;
true (3, 4, 5)
```

Since we have the explicit action of the Galois group, we can now find the quadratic subfield of A corresponding to the subgroup K of H of order 3. We create the sequence of automorphisms of A contained in K and see that the trace of a^3 generates the required quadratic field.

```
> S := Subgroups(H);
> S;
Conjugacy classes of subgroups
------------------------------

[1]     Order 1            Length 1
Permutation group acting on a set of cardinality 6
Order = 1
    Id($)
[2]     Order 2            Length 3
Permutation group acting on a set of cardinality 6
    (1, 2)(3, 4)(5, 6)
[3]     Order 3            Length 1
Permutation group acting on a set of cardinality 6
    (1, 6, 4)(2, 3, 5)
[4]     Order 6            Length 1
Permutation group acting on a set of cardinality 6
    (1, 2)(3, 4)(5, 6)
    (1, 6, 4)(2, 3, 5)

> K := S[3]`subgroup; // get subgroup from record S[3]
```

```
> J := [ hom< A -> A | C[1^k] > : k in K ];
> tra := &+[ h(C[1]^3) : h in J ];
> tra, MinimalPolynomial(tra);
3*a^3 - 9*a
x^2 + 207
> SquareFree(-207);
-23 3
```

Therefore the quadratic subfield is $\mathbb{Q}(\sqrt{-207}) = \mathbb{Q}(\sqrt{-23})$.

Part II

The Language

3. Basic Ideas

This chapter describes some elementary features of MAGMA. It explains how to produce output by evaluating expressions, how to assign a value to an identifier, how to perform function calls, and how to handle common subexpressions. In addition, it gives an overview of the operations on integers, rationals, reals, complex numbers, and Boolean values, and it shows how to include comments, recall previously printed values, and load input from a file.

3.1 Evaluating and Printing Expressions

An *expression* is a piece of MAGMA code which may be *evaluated* to return a *value*. For example, the value of the expression `2+3` is 5, and the value of the expression `Denominator(2/7)` is 7. A *literal* such as `87` is also an expression, with the obvious value.

The most direct way to determine the value of an expression is to *print* it, using a **print**-statement. The output is usually given on the computer screen, or sent to a file if this has previously been arranged; it is not printed on paper. A simple **print**-statement consists of the word **print**, followed by the *expression* whose *value* is to be printed, and finishing with a semicolon:

print expression;

More generally, the **print**-statement is able to evaluate and print several expressions, provided they are separated by commas:

print expression, expression, ..., expression;

It should be noted that the token **print** may be omitted when working interactively. More precisely, **print** is mandatory only when used within functions, procedures and intrinsics. Otherwise its use is optional. In this book, since almost all of the examples are sections of an interactive run, **print** will usually be omitted.

This section illustrates the use of the **print**-statement, while explaining certain common types of expression in MAGMA.

3.1.1 Arithmetic

The addition and subtraction operators in MAGMA are represented by the `+` and `-` symbols, as would be expected. As is the case in algebra, these operators are left associative so that an expression consisting entirely of additions and/or subtractions is evaluated in order from left to right. Parentheses may be used to force a particular order of evaluation. For example:

```
> 15 - 5 + 3;
13
> 15 - (5 + 3);
7
```

Since a **print**-statement may evaluate and print several expressions, the calculations above could be written more succinctly as:

```
> 15 - 5 + 3, 15 - (5 + 3);
13 7
```

Moreover, spaces around operators and commas are optional, so the statement above could also be typed as follows:

```
> 15-5+3,15-(5+3);
13 7
```

However, appropriately placed spaces do tend to improve legibility.

The other standard arithmetic operations are equally straightforward, but their notation may be unfamiliar. The multiplication operator is an asterisk (`*`). Exponentiation is a caret (`^`) or an arrow (↑), depending upon the keyboard being used. In the case of division, there are two kinds of operators: the slash (/) operator, which performs exact division; and **div** and **mod**, which give the quotient and remainder, respectively. The operators **div** and **mod** are defined for pairs of elements of Euclidean Rings, whereas exact division is defined for pairs of elements of any ring provided that the divisor is a unit element. As a special case, exact division when applied to the integers a and b constructs the element $\frac{a}{b}$ of the rational field. For example, the following lines illustrate the various division operators in the integer ring, the rational field, and the polynomial ring in x over the integer ring:

```
> 26 div 4, 26 mod 4;
```

```
6 2

> 26/4;
13/2

> P<x> := PolynomialRing(IntegerRing());
> (x^5 + 4) div (x^2 - 3*x + 1);
x^3 + 3*x^2 + 8*x + 21
> (x^5 + 4) mod (x^2 - 3*x + 1);
55*x - 17
```

Using these operators, one can construct arbitrarily complicated arithmetic expressions. MAGMA's operator precedence for the order of evaluation corresponds to that used in arithmetic and algebra, but parentheses may be used to override this or to clarify the expression for the human reader. Consider the arithmetic expression $7 \times ((354 - 15)^8 - 1892 \times (53 \text{ div } 2) + (24 \bmod 7))$. It is evaluated and printed by the following statement:

```
> 7*((354-15)^8 - 1892*(53 div 2) + (24 mod 7));
1220943664476136726244
```

3.1.2 Function Calls

MAGMA contains a very large number of standard pre-defined functions and procedures, known as *intrinsics*. An intrinsic function is evaluated for specific values of its arguments by typing its name followed by the list of arguments, separated by commas and enclosed within parentheses. If the intrinsic does not have any arguments, the parentheses must still appear. The names of intrinsics follow the rules for identifiers and almost all names for intrinsics begin with a capital letter. For example, the intrinsic function **Factorial**(n) returns $n!$, the factorial of an integer n. To evaluate and print 18!, the intrinsic **Factorial** is applied to the argument 18:

```
> Factorial(18);
6402373705728000
```

The intrinsic function **GCD**(a, b) returns the greatest common divisor (gcd) of a and b, where a and b are elements of an Euclidean Ring. The following statement prints the gcd of 15130 and 3162:

```
> GCD(15130, 3162);
34
```

The process of applying an intrinsic function to a particular set of values is often referred to as *invoking* or *calling* the function.

3.1.3 Printing Text

None of the MAGMA outputs above included any explanation of what the printed values meant. It is often desirable to print some explanatory text together with a value. This may also be done with the **print**-statement. The text must be enclosed between double quotation marks ("), (note this is a single character on the keyboard). For example:

```
> "Factorial of five is", Factorial(5);
Factorial of five is 120
```

The object formed by enclosing characters in double quotation marks is called a *string*. A detailed description of strings, including the formatting of output using the **printf**-statement, is given in Chapter 12.

3.2 Identifiers

3.2.1 Labelling Information

Suppose one is given several numbers, each of which has to be multiplied by some fixed number. If the multiplier is easy to type, say one or two digits, then it is easy to calculate the products directly. However, if the multiplier is large, this is tedious, and typing mistakes can easily occur. It would be much easier to type

```
m*5, m*73, m*48, m*636;
```

rather than

```
1492478*5, 1492478*73, 1492478*48, 1492478*636;
```

provided that MAGMA can be told that m stands for the multiplier.

The way to accomplish this is to type

```
> m := 1492478;
```

This instruction has the effect of storing the integer 1492478 in a 'box' in the computer's memory, and labelling it m. The technical term for a label used to identify a piece of information stored in a computer is *identifier* or *variable*, and the process of giving the identifier a value is called *assignment*.

Once the identifier m has been assigned the value 1492478, MAGMA will substitute this value wherever m appears in an expression. If m is later assigned a different value, this new value will be substituted in subsequent expressions involving m. A **print**-statement will give the current value of m:

```
> m;
1492478
```

and the following line solves the initial problem:

```
> m*5, m*73, m*48, m*636;
7462390 108950894 71638944 949216008
```

An identifier may have as its value, any object definable in MAGMA: an integer, a group element, a set, a rational number, a string or a vector space. The main role of identifiers, other than for purposes of abbreviation, is that they allow the construction of more general program statements than those that can be formed using constants alone. Because of their utility, it is usual for many identifiers to be used in a large MAGMA program.

Identifier names are not restricted to a single letter, such as m. In fact, it is a good idea to choose longer and meaningful names, so as to remind a reader (including the author!) of the value named by that identifier. Some useful names might be *len* (for 'length'), *product*, *lowlimit* or *uplimit*. An identifier may be constructed from letters, digits or the underscore character _, provided that it does not commence with a digit. Thus, *side3* and *angle_A* are examples of legal identifier names, while *3n* is not legal. There are four traps to avoid in choosing identifiers:

- A MAGMA reserved word such as **print** or **mod** may not be used as an identifier. See p. 299 for a list of the reserved words.
- Although it is possible assign a value to an identifier that is the name of an intrinsic function (e.g., **Factorial**), this is very unwise since the intrinsic will no longer be available. It may be retrieved by applying the **delete**-statement to the intrinsic identifier which has been redefined.
- MAGMA distinguishes between upper and lower case letters. For example, *rc*, *RC*, *rC* and *Rc* are distinct identifiers.
- The identifier name _ has a special use as the *throwaway identifier*, so it may not be used as an ordinary identifier name. See p. 32 of this chapter for details.

3.2.2 Showing All Identifiers and Their Values

The intrinsic procedure **ShowIdentifiers**() prints a list of all the identifiers that are currently assigned. A similar procedure, **ShowValues**(), prints all these identifiers together with their values.

In the example below, it should be assumed that no assignments have been made prior to the beginning of the example:

```
> Z := IntegerRing();
> d := 7;
> len := 82/35;
> ShowIdentifiers();
Z     d     len
> ShowValues();
Z:
    Integer Ring

d:
    7

len:
    82/35
```

3.2.3 Deletion and Unassigned Identifiers

If a MAGMA session has involved the creation of very large objects, or large numbers of smaller objects, process space (i.e., workspace) may be exhausted. Memory may be freed using the **delete**-statement to remove unwanted data:

delete identifier, identifier, ..., identifier;

This statement deletes the current values of all the listed identifiers, so that they revert to the status of identifiers that have not been assigned. The former values of those identifiers will be lost irrevocably from MAGMA's memory.

The unary operator **assigned** allows the user to test whether an identifier is currently assigned a value. It returns **true** or **false**. For example:

```
> zxcv := 47;
> assigned zxcv;
true
> delete zxcv;
> assigned zxcv;
false
```

3.3 Assignment

3.3.1 The Simple Assignment Statement

Various examples of assigning a value to an identifier have appeared above. The general form of an simple *assignment statement* is

identifier `:=` expression;

where the expression involves identifiers and/or constants. When MAGMA encounters this statement, it evaluates the expression on the right hand side, stores the resulting value, and associates the identifier with that value. If an error occurs during the evaluation of the expression, the identifier will not be assigned a value.

For example, suppose that the dimensions of a solid box are $L = 14$, $B = 12$ and $H = 6$, and the volume of the box has to be stored in V. Firstly, the identifiers L, B and H should be assigned:

```
> L := 14;
> B := 12;
> H := 6;
```

Note that as it is legal to have several statements on the same line, the assignment of L, B and H above could have been written on a single line:

```
> L := 14; B := 12; H := 6;
```

Now that L, B and H have their values, V may be calculated in terms of these identifiers:

```
> V := L * B * H;
```

Then V may be accessed on demand.

```
> V;
1008

> "Volume is", V, "cubic units.";
Volume is 1008 cubic units.
```

and it can also be used to calculate other quantities such as the mass of the box, given that the box's density is 100 units:

```
> "Mass is", V * 100, "units.";
Mass is 100800 units.
```

It is important to realize that only a *value* is stored in an identifier, not the expression used to calculate the value. For example, V will not change if L is now changed:

```
> L := 57;
> V;
1008
```

The expression on the right side of an assignment statement sometimes involves the identifier on the left side. A common example of this is

```
> m := m + 1;
```

The statement simply means that MAGMA should retrieve the current value of m, add 1 to it, and store the result as the new value of m. It is not related to the inconsistent equation $m = m + 1$. This is the reason why the MAGMA assignment symbol is `:=` rather than `:=`. The user should regard the symbol `:=` as meaning 'becomes' or 'has assigned to it', rather than 'is equal to' in the mathematical sense.

A final caution: if an identifier is assigned a value and then assigned another value, the old value is lost and cannot be recovered. The classic case occurs in trying to interchange the values of two identifiers, a and b. It seems at first that the following lines will suffice, but they do not:

```
> a := b;
> b := a;
```

(What happens instead?) The correct solution is to use a temporary identifier (called *temp*, say), as follows:

```
> temp := a;
> a := b;
> b := temp;
```

In this way a can acquire b's former value and b can acquire a's former value, without either of them being lost in the process.

3.3.2 The Mutation Assignment Statement

There is an alternative form of assignment statement available for making a simple change to an existing identifier that involves its former value. Suppose that the following assignment has been made:

```
> d := 7;
```

and then d has to be multiplied by 53. Instead of typing

```
> d := d * 53;
```

the user can type

```
> d *:= 53;
```

This is known as a *mutation assignment* because it mutates, or changes, the old value of the identifier. Its general form is

identifier ∘`:=` expression;

where ∘ is the operator, and it has the same end result as

identifier `:=` identifier ∘ expression;

Apart from being more compact than the standard assignment, a mutation assignment is often more efficient since MAGMA can operate directly on the identifier's value without first having to make a copy of it.

3.3.3 Multi-Valued Expressions and the Multiple Assignment Statement

Certain classes of expression in MAGMA may return multiple values. In particular. functions may return several values, and *constructors*, such as **sub** and **quo**, (see Chapter 4), return both a structure and an associated mapping. Such an expression will be termed a *multi-valued expression*. The manner in which these multiple values are handled depends upon the context in which the expression appears. There are four cases:

– The expression forms the right-hand side of an assignment statement in which exactly one identifier appears on the left-hand side. In this case, only the first value will be computed.

– The expression forms the right-hand side of an assignment statement in which more than one identifier appears on the left-hand side (including throwaway identifiers). All the values are created and the designated ones are assigned.

– The expression is embedded as part of a larger expression. In this case only the first value (the *principal value*) is computed and used when evaluating the outer expression.

– The expression is a term in a print list. Here the principal value is always printed while the remaining values may or may not be printed depending upon circumstances.

To access the values returned by a multi-valued assignment a *multiple assignment statement* is provided. It has the syntax:

identifier, ..., identifier `:=` multiple-value expression;

For example, the function **Quotrem**(a, b) returns the values of both a **div** b and a **mod** b:

```
> q, r := Quotrem(26, 4);
> q, r;
6 2
> 4*q + r;
26
```

If such an expression forms part of a larger expression, then only the principal value (the first return value) will be used:

```
> 5 * Quotrem(100, 13);
35
```

In the case of some multiple return value expressions, the **print**-statement will display all the return values. Whether this happens or not is decided on a case-by-case basis; as a guide, all the values will be printed if they are cheap to compute, generate little output and convey useful information.

The number m of identifiers on the left of the multiple assignment statement must be less than or equal to the total number n of return values. If m is strictly less than n, then only the first to the m^{th} return values of the expression will be returned and assigned. The special `_` character (known as the *throwaway identifier*) may be used in the identifier list to discard the corresponding return value:

```
> qq := Quotrem(100, 13);
```

```
> qq;
7

> _, rr := Quotrem(100, 13);
> rr;
9
```

It should be noted that in the case of functions for which significant additional work has to be done to compute non-principal return values, this extra computation may be avoided by requesting only the first return value (by assigning only to a single identifier). So to avoid unnecessary work, values other than the first should not be requested unless they are required. An example is the **SmithForm**(X) function, whose principal return value is the Smith Normal Form S of the matrix X. Its second and third values are unimodular matrices P and Q such that $PXQ = S$, but since these matrices take significantly longer to compute than S, they are only calculated if the user requests them specifically. The example below demonstrates the time savings for a random 50×50 integer matrix X, each of whose coefficients is -1, 0 or 1. Note that the **time** command, placed at the beginning of a statement, has the effect of printing the execution time (in seconds) for that statement:

```
> n := 70;
> M := MatrixRing(IntegerRing(), n);
> M;
Full Matrix Algebra of degree 70 over Integer Ring
> X := M ! [Random(-1, 1): i in [1 .. n^2]];
> time S := SmithForm(X);
Time: 0.770
> time S, P, Q := SmithForm(X);
Time: 16.019
```

3.4 The where-construction

Often a MAGMA expression will include the same subexpression more than once. It is faster for both the user and the system to assign the value of that expression to a temporary identifier. This may be accomplished with the **where**-construction, which has the form

expression **where** identifier **is** expression

This entire construct is understood as constituting a single expression. Its value is found by evaluating the first expression, with the given identifier

taking as its value, the value of the second expression. (For convenience, the assignment symbol := may be used instead of **is**.)

For example, let P be the polynomial ring in x over the integers:

```
> P<x> := PolynomialRing(IntegerRing());
```

Suppose that the user wants to create $f = d(d+1)(d^2+5)$ where d is x^5-4x^3+17. Without using the **where**-construction, there are two methods. Firstly, the user could type the expression for d into every part of the expression for f in which it appears. This would involve excessive typing, and the expression for d may have to be evaluated several times by the system. Secondly, the user could assign $x^5 - 4x^3 + 17$ to the identifier d, and then create f:

```
> d := x^5 - 4*x^3 + 17;
> f := d*(d + 1)*(d^2 + 5);
> f;
x^20 - 16*x^18 + 96*x^16 + 69*x^15 - 256*x^14 - 828*x^13 +
    256*x^12 + 3312*x^11 + 1790*x^10 - 4416*x^9 - 14320*x^8
    + 28640*x^6 + 20694*x^5 - 82776*x^3 + 89964
```

This approach is preferable to the first, but is not ideal for two reasons: the value of d is not otherwise required; and if d already had a value, then this value would be lost. The **where**-construction affords a superior solution:

```
> f := d*(d + 1)*(d^2 + 5) where d is x^5 - 4*x^3 + 17;
> f;
x^20 - 16*x^18 + 96*x^16 + 69*x^15 - 256*x^14 - 828*x^13 +
    256*x^12 + 3312*x^11 + 1790*x^10 - 4416*x^9 - 14320*x^8
    + 28640*x^6 + 20694*x^5 - 82776*x^3 + 89964
```

The scope of the identifier in the **where**-construction is limited to that construction, so it does not affect an identifier of the same name outside it. Consider the following:

```
> d := 1000;
> f := d*(d + 1)*(d^2 + 5) where d is x^5 - 4*x^3 + 17;
> d;
1000
```

Notice that d has kept its old value. If d had not been assigned, then after the **where**-construction it would have remained unassigned.

Several **where**-constructions may be used in succession. For example, each of the following statements assign the same value to m:

```
> m := (n-2)*Factorial(n) where n is k^2-3*k+1 where k is 4;

> m := (n-2)*Factorial(n) where n is 4^2-3*4+1;

> m := (5-2)*Factorial(5);
```

If an expression involves successive **where**-clauses, they associate to the left. As an example, compare the following three expressions. The first two evaluate to give the same result, but the third yields a different value:

```
> x := 1; y := 2;
> x + y where x is 5 where y is 7;
12
> (x + y where x is 5) where y is 7;
12
> x + y where x is (5 where y is 7);
7
```

In the case of a comma-separated expression list, the effect of a **where***-clause will extend left across the commas, but not right.* Such expression lists are used as the arguments of a function or procedure, in **print**-statements, and in other statements not yet introduced (e.g., **return** and **error**). In the following example, notice that the first two expressions in the **print**-statement are evaluated using the values for x and y given in the **where**-constructions, but the other expression is evaluated using the external values $x = 1$ and $y = 2$:

```
> x := 1; y := 2;
> x-y, x+y where x is 5 where y is 7, x*y;
-2 12 2
```

3.5 Integers, Rationals, Reals, and Complex Numbers

Arithmetic with integers has already been demonstrated. There are no size restrictions on integers in the case of arithmetic operations (except for exponentiation, in which the absolute value of the exponent must be less than a constant, dependent upon the word-length of the computer):

```
> n := 9817855152457389762535312;
> n * (n+1);
963902797946341159773679542149821095582278094726 56
```

Rational numbers in MAGMA are expressed in fractional form, and real numbers in decimal form:

```
> halfA := 1/2;
> halfB := 0.5;
```

A rational number such as $20\frac{4}{7}$ may be entered as either $20 + 4/7$ or $144/7$. A real number having very large or very small magnitude may be entered in scientific notation, using `e` to denote the base 10. For example, the input syntax for 5.2×10^{-16} is:

```
> smallnum := 5.2e-16;
```

A complex number is best created as an expression involving two real numbers and $i = \sqrt{-1}$, where i is created using the **ComplexField**() function:

```
> C<i> := ComplexField();
> z := 2.6 + 3.7*i;
```

Rational arithmetic in MAGMA is performed to arbitrary precision, since it is based internally on integer arithmetic. Real/complex arithmetic is performed to free precision or to a fixed precision, depending on the setting of the default real field; see Chapter 26.

MAGMA can usually cope easily with an expression involving both an integer and a rational, real or complex number, by coercing the integer into the appropriate field. However, for some applications MAGMA must be told explicitly that an integer is to be considered as a real, rational or complex number. The easiest way of doing this to write it as such, as shown below:

```
> rat37 := 37/1;
> real37 := 37.0;
> cmplx37 := 37.0 + 0*i;
```

These numbers will be deemed by MAGMA to be equal to the integer 37, but the structure to which they belong (their *parent magma*) will be the rational, real or complex field rather than the integer ring:

```
> Parent(rat37), Parent(real37), Parent(cmplx37);
Rational Field
Real Field
Complex Field
```

An alternative way to construct *rat37*, *real37* and *cmplx37* is to *coerce* them explicitly into the required structure, using the `!` operator:

```
> rat37 := RationalField() ! 37;
> real37 := RealField() ! 37;
> cmplx37 := C ! 37;
```

This method produces approximately the same results as above, but is slightly preferable in the real and complex case since the coerced integers will have infinite precision.

Integers, rationals, reals and complex numbers share most of the common arithmetic operators, but, in addition, each has its own special functions. For example:

```
> Numerator(r), Denominator(r) where r is 6/4;
3 2
> Sin(4.6);
-0.99369100363346445613810465991370046463O
> Round(x), Truncate(x) where x is 3.7;
4 3
> Sqrt(-3); // square root
1.732050807568877293527446341495*i
> ComplexToPolar(3 + 4*i); // modulus and argument
5.000000000000000000000000000
0.92729521800161223242851246291
```

Integers and rationals are discussed in more detail in Chapter 20, and real and complex numbers in Chapter 26.

3.6 Booleans

Table 3.1. Relational operators

Operator	Maths	Usage	Meaning
eq	$=$	x **eq** y	**true** iff x is equal to y
ne	$\neq$	x **ne** y	**true** iff x is not equal to y
lt	$<$	x **lt** y	**true** iff x is less than y
le	$\leq$	x **le** y	**true** iff x is less than or equal to y
gt	$>$	x **gt** y	**true** iff x is greater than y
ge	$\geq$	x **ge** y	**true** iff x is greater than or equal to y

One of the most basic structures available in Magma is the *Boolean* structure, so named after the logician George Boole. The Boolean structure

contains two values: **true** and **false**. Boolean values are used to describe a two-state system, such as *yes/no*, *on/off*, or *true/false*.

There are several operators that return a Boolean value. The *relational operators* test if two quantities are related in a certain way and they then return an answer of **true** or **false**. For example, x **eq** y returns **true** if and only if x equals y. The principal relational operators are described in Table 3.1. In this table, and in all other tables listing functions and operators returning Booleans, it should be noted that 'iff' is an abbreviation for 'if and only if'. Some illustrations are:

```
> 3+4 eq 7;
true
> 100 eq 1000-0;
false
> 2/3 lt 3/4;
true
```

Table 3.2. Boolean operators

Operator	Usage	Meaning
not	**not** a	**true** iff a is **false**
and	a **and** b	**true** iff both a and b are **true**
or	a **or** b	**true** iff at least one of a and b are **true**
xor	a **xor** b	**true** iff exactly one of a and b is **true**

Table 3.2 lists the Boolean operators. They operate on Boolean values and return Boolean results. For example:

```
> (3+4 gt -5) and (2 le 3);
true
```

The meanings of the Boolean operators correspond fairly closely to the ordinary meanings of the corresponding words, except that that **or** is *inclusive* (true if and only if one or both of a and b are true), whereas **xor** is *exclusive* (true if one of a and b is true but not if both are true). In other words, **or** is in accordance with the use of 'or' in 'This seat is reserved for aged or disabled persons' and **xor** corresponds to 'or' in 'Would you like coffee or tea?'.

Expressions involving the operators **and** and **or** are evaluated using *conditional* evaluation, as distinct from MAGMA's usual *call-by-value* evaluation. Conditional evaluation of an expression has the effect that arguments are evaluated only as required, from left to right, stopping as soon as the return

value is determined. If a evaluates to **false** in the expression a **and** b, then the value of the entire expression must be **false** so that b is not evaluated; and if a is **true** in the expression a **or** b, then the value of the entire expression must be **true** so that b is not evaluated. The examples below show how to exploit this property in order to avoid error messages or lengthy computation:

```
> 5/2 and false;

>> 5/2 and false;
             ^
Runtime error: Expected a logical for the 'and' operator
> false and 5/2;
false

> time IsPrime(871489712761289798134091201215907 09)
>   or IsPrime(29);
true
Time: 4.510
> time IsPrime(29)
>   or IsPrime(871489712761289798134091201215907 09);
true
Time: 0.000
```

There are many functions in MAGMA that test whether their argument(s) satisfies some property, and return a Boolean value as the result. Boolean functions typically have names beginning with **Is** (or **Has** or **Are**) followed immediately by the name of the property to be tested. For instance, **IsBipartite**(G) returns **true** if and only if the graph G is bipartite, and otherwise returns **false**:

```
> PG5 := PolygonGraph(5);
> IsBipartite(PG5);
false
```

Some Boolean functions have additional return values that provide further information. One such function is **IsSquare**(a), which, if the integer a is a perfect square, returns **true** and the positive square root of a, and otherwise returns **false** alone:

```
> sq, sqroot := IsSquare(144);
> sq;
true
> sqroot;
12
```

```
> sq, sqroot := IsSquare(143);
> sq;
false
> assigned sqroot;
false
```

3.7 Comments

It is standard practice to include *comments* in a computer program to describe its purpose and its internal logic. Such remarks are ignored by the computer.

A comment is placed between the `/*` and `*/` symbols; MAGMA will ignore anything between these symbols, and the symbols themselves. Note that comments may not be nested. Alternatively, any text following the symbols `//` on a line will be ignored (treated as a comment).

The following example is rather artificial because of its shortness, but comments may also be found in other examples in this book:

```
> /*
>  * Find the volume of a box,
>  * given its length, breadth, and height.
>  */
> // assign the dimensions
> L := 14;   // length
> B := 12;   // breadth
> H := 6;    // height
> // compute the volume
> V := L * B * H;
```

3.8 Recalling Previously Printed Values

It frequently occurs that a user requires a value printed by an earlier **print**-statement, where the value has not been assigned to an identifier. Since it would be inefficient to recompute the value, MAGMA provides a facility for recalling values that have been obtained by evaluating expression lists in **print**-statements.

The list of values displayed by the most recent **print**-statement corresponds to `$1`, the list of values in the next most recent **print**-statement corresponds to `$2`, and so on. The number after the dollar sign must be a literal integer, not a general expression evaluating to an integer. For example:

```
> z := 15;
> 3 * z;
45
> y := 13;
> $1;
45
> z * y + 1;
196
> $1 - $2; // i.e., 196 - 45
151
```

If several values were previously printed in a single statement, then the dollar construct will return all of them exactly as if it were a multi-valued expression, so the values may be assigned using the multiple assignment statement. For example:

```
> 12 * 9, 4 * 8;
108 32
> a, b := $1;
> a;
108
> b;
32

> Quotrem(5000, 183);
27 59
> c, d := $1;
> c;
27
> d;
59
```

The previous values (or more accurately, previous value lists) are stored in the *previous value buffer*. The maximum number of entries in this buffer is returned by the function **GetPreviousSize**(). The default value is 3, but it may be changed to any small non-negative integer n by means of the procedure **SetPreviousSize**(n). Note that **SetPreviousSize** simply specifies a new buffer size; increasing the maximum will not restore very old values, as demonstrated below:

```
> Factorial(4);
24
> Factorial(5);
120
> Factorial(6);
```

```
720
> Factorial(7);
5040
> $4;

>> $4;
   ^
Runtime error: Previous value number (4) should be in the range [1 .. 3]
> SetPreviousSize(6);
> Factorial(8);
40320
> $5;

>> $5;
   ^
Runtime error: Previous value number (5) should be in the range [1 .. 4]
```

As will be evident from the end of the preceding example, printing a previous value changes the previous value buffer, just as is the case with any other **print**-statement. However, the procedure **ShowPrevious** allows the user to inspect the values in the buffer without affecting the contents of the buffer. **ShowPrevious**(i) displays the value \$$i$, and **ShowPrevious**() displays the whole buffer. The next example should be understood as a continuation of the one before:

```
> ShowPrevious(3);
$3: 720
> ShowPrevious();
$4: 120
$3: 720
$2: 5040
$1: 40320
> $3;
720
```

The procedure **ClearPrevious**() deletes all values stored in the previous value buffer.

3.9 Loading Input from a File

Short bodies of code may easily be typed directly into MAGMA. For longer programs, it is usually more efficient to create blocks of code as text files external to MAGMA. The creation and editing of such files is performed using

whatever software the computer offers, and does not involve MAGMA. After the file has been created, its contents may be loaded into MAGMA by typing:

load "filename";

where 'filename' is the name of the file. MAGMA will execute each line of the file just as if it had been typed directly intoMAGMA. The **load**-statement must appear at the top level, that is, it may not be included within any other statement.

If the file does not reside in the current directory then the directory must be specified as well. Alternatively, the directory may be included in the environment variable `MAGMA_PATH`. See p. 278.

For example, suppose a file, `m11pol`, has been created containing the following statements:

```
Q    := RationalField();
R<x> := PolynomialRing(Q);
f    :=  x^11 - x^10 - 121*x^9 + 65*x^8 + 5345*x^7 -
         481*x^6 - 96739*x^5 - 23689*x^4 + 413690*x^3 -
         493810*x^2 + 26910*x - 856170;
print "The following polynomial f of degree 11 is defined:";
print f;
```

Now, from within MAGMA, the user may **load** this file as many times as desired. Each time it is loaded, the statements in it are executed:

```
> load "m11pol";
Loading "m11pol"
The following polynomial f of degree 11 is defined:
x^11 - x^10 - 121*x^9 + 65*x^8 + 5345*x^7 - 481*x^6 -
    96739*x^5 - 23689*x^4 + 413690*x^3 - 493810*x^2 +
    26910*x - 856170
> G := GaloisGroup(f);
> G;
Permutation group G acting on a set of cardinality 11
    (1, 4, 6, 8, 10)(2, 7, 9, 11, 5)
    (1, 9, 10)(3, 7, 5)(4, 6, 11)
    (1, 6, 5, 3, 10)(2, 4, 9, 7, 11)
> CompositionFactors(G);
    G
    |  M11
    1
```

Not only can the user's own files be loaded into MAGMA, but also many other files provided with the MAGMA system. There are two main kinds of these files: *library* or *database* definitions of commonly-used algebraic structures and functions, and certain labelled examples from this book and the *Handbook*. When examples from the books are loaded, (most of) the contents of the file containing the example are echoed to the terminal so that they can be seen. This does not necessarily happen when a file is loaded, but happens for these examples because the function **SetEchoInput** is being used. The interested reader is advised to look at some of the files and read p. 280 to understand how it works.

MAGMA also provides a package mechanism, which provides for the autoloading of files. Given a file containing the user's definitions of functions and procedures in a special syntax, it is able to incorporate these routines into MAGMA as *user intrinsics* (as distinct from *system intrinsics*). Chapter 10 explains the use of packages.

4. Algebraic Structures

Structures are the weapons of the mathematician.

See *Scientific American* May 1957
Nicholas Bourbaki

With almost no exceptions, any object x definable in the MAGMA language is viewed as an element of some algebraic structure A. The properties of A define the operations that may be performed on x. As a general rule, the structure A must be defined before its element x (although there are exceptions). It is therefore very important for the reader to understand how the concept of an algebraic structure is realised in MAGMA together with the mechanisms provided for constructing them.

Many different types of algebraic structures may be constructed in MAGMA, including polynomial rings, number fields, vector spaces, modules, matrix algebras, and finitely-presented groups. However, this diverse range of structures share certain common conventions concerning operators for arithmetic, creating substructures and quotient structures, and testing membership.

This chapter provides an overview of the MAGMA philosophy concerning algebraic structures and their elements, and introduces the principal syntactic constructs for their creation and manipulation. It should be skimmed on first reading, and referred to later as needed.

4.1 Magmas, Categories and Varieties

The organization and interrelationship of algebraic structures in MAGMA is based on ideas from Universal Algebra and Category Theory. Every object that is definable in MAGMA is regarded as belonging to a set M (usually the carrier set associated with an algebraic structure). Such sets are referred to as

magmas; the term being a generalization of the Bourbaki definition ([Bou70], ch. 1, p. 1) of a magma as a set with one or more laws of composition.[1]

The magma to which an object belongs is called its *parent magma*. Given an object x, the function **Parent**(x) returns its parent. For example, the parent of the real number 62.7 is the real field:

```
> Parent(62.7);
Real Field
```

Since every object that can be created in MAGMA is ultimately defined in terms of one or more magmas, it is necessary to be able to refer to any magma. For example, a function that creates a polynomial ring P in the indeterminate t over some ring, will take the coefficient ring as its argument. In the example below, the construction of the magma comprising the ring of univariate polynomials over $\mathbb{Q}$ is created using the function **RationalField**() which defines the magma $\mathbb{Q}$.

```
> Q := RationalField();
> P<t> := PolynomialRing(Q);
> P;
Univariate Polynomial Algebra in t over Rational Field

> Parent(t^2 + 5*t + 6);
Univariate Polynomial Algebra in t over Rational Field
> Parent(t^2 + 5*t + 6) eq P;
true
```

The output shows that the parent of $t^2 + 5t + 6$ is P.

Similar magmas are organized into classes known as categories. More precisely, a *category* is a class of magmas satisfying a particular set of axioms and sharing a common representation. The category to which a magma belongs determines the set of operations that may be applied to its elements. For example, if x and y are elements of a permutation group G, then this parent magma G belongs to the category of permutation groups, **GrpPerm**. It is from the membership of their parent in this category that the processor knows that $x*y$ is defined but $x + y$ is not; that is, there is a multiplication algorithm that operates on a pair of permutation group elements, but there is no addition algorithm.

The function **Category**(M), or equivalently **Type**(M), returns the category of M. Continuing the previous example, the following output shows that P belongs to **RngUPol**, the category of rings of univariate polynomials over any ring and in any indeterminate:

[1] 'Un ensemble muni d'une loi de composition est appelé un magma.'

```
> Category(P);
RngUPol
```

The procedure **ListCategories**() lists the names of all the categories in MAGMA:

```
> ListCategories();
Alg                   GrpMatElt             PowSetEnum
AlgChtr               GrpPC                 PowSetFormal
AlgChtrElt            GrpPCElt              PowSetIndx
AlgCon                GrpPCStdPresProc      PowSetMulti
    ...
```

For most elementary purposes it is not necessary to know the precise names MAGMA uses for the categories. They are required for a few intrinsics; an example is **DihedralGroup**(C, n), which takes a category C as its first argument so that MAGMA knows whether to construct the dihedral group of order $2n$ as a permutation group, a finitely-presented group, or so on. See Section 29.1.3 for details. Category names are also required for the creation of user intrinsics, as described in Chapter 10, in order to specify the categories of arguments and return values.

Note carefully that the possession of some mathematical property by a particular magma in a category does not imply that this property will be automatically exploited by the MAGMA system. In general, unless the property is common to all magmas belonging to the category, operations depending upon it will not be defined. For example, if a quotient ring of a univariate polynomial ring over $\mathbb{Q}$ (category **RngMPolRes**) happens to be a field, then operations defined exclusively for the category of number fields (**FldNum**) are not automatically available. In order to access such operations, the field must be explicitly created as a member of **FldNum**. In summary, *the operations available for a particular magma are defined by the category in which it is created.*

Finally, a collection of categories whose magmas satisfy a common set of axioms forms a *variety*. For example, **GrpAb** (finitely-generated abelian groups), **GrpMat** (matrix groups) and **GrpFP** (finitely-presented groups) are some of the categories belonging to the variety of groups. While a user seldom needs to be aware of the existence of varieties, the concept is used by the MAGMA implementers to write code that is independent of the representation (as determined by the category).

In some cases, MAGMA provides functors for moving from one category to another or from one variety to another. For instance, **AbelianGroup**(G) returns the abelian group (in category **GrpAb**) corresponding to the abelian matrix group G (in category **GrpMat**), and **VectorSpace**(A) returns the

vector space (in **ModTupRng**) underlying a finite dimensional associative algebra A (in **AlgCon**).

4.2 Creation of Magmas

The creation of a magma M in the MAGMA system requires, in principle, two steps:

1. The definition of an appropriate 'free' magma F;
2. The construction of the desired magma M from F by applying a sequence of constructions that create substructures, normal closures, ideals, quotient structures and extensions until M is reached.

For each category possessing free magmas, a function is provided that constructs a free magma F. The derivation of a magma from an existing magma via generic operations is accomplished by *constructors* of the form

constructor< magma | details of construction >

The principal constructor names, appearing to the left of the < symbol, are **sub**, **ncl**, **ideal**, **quo** and **ext**, designating submagmas, normal closures (in the case of groups), ideals, quotients and extensions.

If a magma with r generators is assigned to an identifier M, then $M.i$ denotes the i^{th} generator, where $1 \leq i \leq r$. However, it is customary to give names to the generators of M at the same time as M is assigned. This may be accomplished by means of a *generator assignment statement*, with the following syntax:

M< $x_1, \ldots, x_r$ > := expression defining M;

(Here the $x_1, \ldots, x_r$ must obey the usual rules for identifier names.) The principal effect of this statement is to assign M in the usual way. Its auxiliary effect is to create the identifiers $x_1, \ldots, x_r$ and assign to them the values of the generators. As a result, either x_i or $M.i$ may be used in an input expression, to denote the i^{th} generator of M. The third effect of the statement operates only for magmas such as free or finitely-presented structures (including polynomial rings), whose elements are printed explicitly in terms of generators: the *printname* (output form) of each generator is changed from $M.i$ to x_i, so as to increase the legibility of the output.

For example, consider the polynomial ring R over the rational field in three indeterminates. The following statement constructs R, assigns its inde-

terminates to the identifiers x, y, z, and indicates that the indeterminates are to be printed as x, y, z:

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> R;
Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
> x + 2*y*z;
x + 2*y*z
```

In the following sections, the creation of magmas and their elements will be explained more fully.

4.3 Automatically-Created Magmas

Five magmas are created automatically when a MAGMA session begins. They are the integers, the rationals, the free real field (with default precision attribute 28), the Booleans, and the monoid of character strings. Because these magmas are automatically created, it is unnecessary when referring to their elements to declare the parent explicitly. The MAGMA input representation of elements belonging to such magmas allows the language processor to deduce the appropriate parent magma. For instance, given the statements

```
> three := 3;
> NumberTheory := "queen of mathematics";
```

then MAGMA can deduce that the parents of the objects are the integer ring and the string monoid, respectively:

```
> Parent(three), Parent(NumberTheory);
Integer Ring
String structure
```

Table 4.1 lists functions each of which returns, as its value, one of the five automatically-created magmas. Although these magmas are defined when the system starts up, and therefore do not require explicit 'creation', it is sometimes necessary to refer to them when defining magmas inductively, as illustrated above in the example of the polynomial algebra over the rationals. Moreover, if several magmas are being defined in terms of the same magma, it is advisable to assign the magma to an identifier so that the function does not have to be called repeatedly:

Table 4.1. Functions for the automatically-created magmas

MAGMA	Meaning
IntegerRing()	The ring of integers
RationalField()	The field of rational numbers
RealField()	The free field of real numbers (default precision 28)
Strings()	The string monoid (over the ASCII character set)
Booleans()	The Boolean structure, containing **true** and **false**

```
> Q := RationalField();
> Q;
Rational Field
```

Each of these five magmas has its own category, a category to which no other magmas belong. (An exception is the integer ring category, to which ideals of integer rings also belong for the sake of efficiency; see Section **??**.) These categories are **RngInt**, **FldRat**, **FldPr**, **MonStg**, and **Bool**. Details of these categories are given in Chapter 20 for the integers and rationals, Chapter 26 for the reals, Section 3.6 for the Booleans, and Chapter 12 for the strings.

4.4 Creating a Free Magma

When creating a magma M, the general strategy is to first create the appropriate free magma F and then create the desired submagma or quotient magma M in terms of it. This section explains what is meant in MAGMA by the concept *free magma*, and illustrates the construction of such magmas. To define other magmas the user must also to know how to define elements of magmas; this is discussed in the following section.

Given a variety V (e.g., groups, rings), the *free magma* F in V with n generators may be informally described as the unique algebraic structure whose elements are all possible finite arithmetic combinations of the n generators. Here the meaning of 'arithmetic combination' depends on V, but will typically involve sums, products, and/or the taking of inverses. (This informal notion of 'free magma' corresponds to the concept of a term algebra.)

For example, the free group *FG3* on three generators a, b, c is the group consisting of the set of equivalence classes of finite products of a, b, c, a^{-1}, b^{-1} and c^{-1}. It may be created in MAGMA as follows:

```
> FG3<a,b,c> := FreeGroup(3);
```

```
> FG3;
Finitely presented group FG3 on 3 generators (free)
```

MAGMA uses the concept of free magma in a looser sense as well, to describe the largest possible magma of a given category with the given parameters. Such magmas are often called *generic* or *full*. For example, in the group category of permutation groups, the free magma for each degree n is $\mathrm{Sym}(n)$, the symmetric group of degree n (i.e., the group consisting of all permutations of n elements). Similarly, in the module category of vector spaces, the free magma for each field K and dimension n is the vector space $K^{(n)}$, that is, the space of all n–dimensional vectors whose entries are elements of K. For instance, the symmetric group of degree 6 may be created as shown below, using the **SymmetricGroup** function (commonly abbreviated to **Sym**):

```
> s6 := Sym(6);
> s6;
Symmetric group s6 acting on a set of cardinality 6
Order = 720 = 2^4 * 3^2 * 5
```

and the following lines create the finite field with 27 elements (assigning its generating element to w) and then the full four-dimensional vector space with coefficients taken from this field:

```
> GF27<w> := GaloisField(27);
> GF27;
GF(3^3)
> VS4 := VectorSpace(GF27, 4);
> VS4;
Full Vector space of degree 4 over GF(3^3)
```

4.5 Creating and Operating on Elements of a Magma

Objects in MAGMA may arise as the values of many kinds of expressions, including function calls. However, there are two basic ways to create a new object as an element of a magma M:

– As a literal element, using an **elt**-constructor, sequence-coercion, or some other syntax for literal elements that is unique to the category of M;

– Arithmetically, from generators or previously assigned elements of M.

This section examines the creation of literal elements, followed by an explanation of how to use arithmetic operators to build elements from other elements, and how to compare elements using relational operators. Finally, some remarks are made about the output representation of magma elements: in some categories the elements are printed showing their composition from generators; and in other categories they are given in concrete form (e.g., as permutations or matrices). In every category except a few categories of finitely-presented magmas, elements are simplified immediately into canonical form.

4.5.1 Creating Literal Elements

One method of defining an element of M is as a *literal element*, that is, in terms of a list of more primitive objects, the nature of which depends on the category to which M belongs. In general, both of the following must be specified:

– The magma M;

– The primitive components of the element.

These components of the element might be entries of a matrix, coefficients of a polynomial, and so on, depending on the magma. The 'official' method (though not the most common!) of creating a literal element is to use the **elt**-constructor

elt< M | $a_1, \ldots, a_r$ **>**

where M is the magma and $a_1, \ldots, a_r$ are the components of the desired element.

For example, suppose the vector $v = (2w \quad 8 \quad w+2 \quad 25)$ is to be created as an element of *VS4*, the vector space defined in the previous section. The appropriate usage of the **elt**-constructor is:

```
> v := elt< VS4 | w^14, 2, w^3, 1 >;
> v;
(w^14    2  w^3    1)
```

It was noted above that the **elt**-constructor is not the most common method of creating literal elements. In practice, users tend to employ a different syntax:

M ! Q (where Q is a sequence $[a_1, \ldots, a_r]$)

Here the list of components $a_1, \ldots, a_r$ is placed in a *sequence*, and then the sequence is coerced into M using the ! operator. This operator ! performs *coercion*; that is, $M!Q$ takes the sequence Q, which is not an element of M, and returns the corresponding element of M. Since we have here an expression involving an operator, this is not strictly a case of a literally-created element, but MAGMA users often find it convenient to think of it in this way. For example, the vector v could be created by sequence-coercion as follows:

```
> sq := [w^14, 2, w^3, 1];
> v := VS4 ! sq;
```

or, in one step:

```
> v := VS4 ! [w^14, 2, w^3, 1];
```

Sequence-coercion as a means of element construction has two advantages over the **elt**-constructor: if the sequence is constructed first then it can be built using the sequence machinery (see Chapter 6); and if the sequence and element are constructed in one step then fewer keystrokes are required in this method than for the **elt**-constructor. On the other hand, the **elt**-constructor has the following advantages: it reflects the characteristic MAGMA syntax of deriving one object from another using a constructor; it is more efficient for elements with a large number of primitive components, since it avoids the creation of the sequence; and it provides superior control for a few categories, such as Laurent and power series rings (p. 409) and the real field (p. 493). This book will generally use the sequence-coercion method.

In some categories of magma there is special syntax for creating literal elements. For example, if the magma is a permutation group then disjoint-cycle notation may be used for elements:

```
> s6 := Sym(6);
> g := s6 ! (2,6,4,3);
> g;
(2, 6, 4, 3)
```

Literal elements of the five standard magmas, listed in Table 4.1 (p. 50), have a special representation, and so they are *self-identifying* to the MAGMA processor. The magma need not be stated explicitly. Examples are:

```
> three := 3;
> twothirds := 2/3;
> half := 0.5;
> NumberTheory := "queen of mathematics";
> bool := true;
```

4.5.2 Arithmetic Operators

Table 4.2. Additive operators

MAGMA	Meaning
$x + y$	Sum of elements x and y
$-x$	Additive inverse of x
$x - y$	Difference of elements x and y
$n*x$	x added to itself n times
$M!0, \mathbf{Zero}(M)$	Additive identity of M
$\mathbf{IsZero}(x)$	**true** if x is additive identity of M

Table 4.3. Multiplicative operators

MAGMA	Meaning
$x*y$	Product of elements x and y
$x\hat{}-1$	Multiplicative inverse of unit x
x/y	$x*y^{-1}$, where y is a unit (except that if $x, y \in \mathbb{Z}$ then x/y will be a rational field element)
x **div** y	Quotient when x is divided by y (used in rings)
x **mod** y	Remainder when x is divided by y (used in rings)
$\mathbf{Quotrem}(x, y)$	Quotient and remainder when x is divided by y (used in rings)
$x\hat{}n$	x^n, where $n \in \mathbb{Z}$ ($n \geq 0$ unless x is a unit)
$M!1, \mathbf{Id}(M), \mathbf{Identity}(M)$	Identity of group M
$M!1, \mathbf{One}(M)$	Multiplicative identity of ring M
$\mathbf{IsId}(x), \mathbf{IsIdentity}(x)$	**true** if x is identity of group M
$\mathbf{IsOne}(x)$	**true** if x is multiplicative identity of ring M

Arithmetic on elements of magmas uses, in general, the standard symbols that are used in computer languages, and operator precedence follows mathematical conventions. Table 4.2 and Table 4.3 list the additive and multiplicative operators on elements. For any given category, only some of these operators will be relevant, and there will be category-specific operators provided as well.

The additive operators are used in those categories in which the definition of magmas involves an abelian group. This includes rings (together with fields and algebras), vector spaces, and the abelian group category. For example,

consider the K-matrix space of 2×3 matrices with coefficients from the rational field:

```
> Q := RationalField();
> m23 := KMatrixSpace(Q, 2, 3);
> m23;
Full Vector Space of 2 by 3 matrices over Rational Field
> matrix1 := m23![1, 0, 3/5, 4/7, -5, 9/4];
> matrix1;
[  1   0 3/5]
[4/7  -5 9/4]
> 15 * matrix1;
[   15     0     9]
[ 60/7   -75 135/4]
> matrix1 + m23![-13/5, 2, 0, 1/8, 0, -1/4];
[ -8/5     2   3/5]
[39/56    -5     2]
> m23 ! 0;
[0 0 0]
[0 0 0]
```

The multiplicative operators are used in those categories in which the definition of magmas involves a semigroup or non-abelian group. This includes rings and fields, and all kinds of semigroups and groups except the abelian group category. Groups use $M!1$, **Identity**(M) or **Id**(M) for the identity, and **IsIdentity**(x) to test whether x is the identity. Rings use $M!1$ or **One**(M) for the multiplicative identity, and test for it with **IsOne**(x). As a group example, consider the permutation group *s6* and its element $g = (2, 6, 4, 3)$ defined above:

```
> s6 := Sym(6);
> g := s6 ! (2,6,4,3);
> g^3;
(2, 3, 4, 6)
> g * s6!(2,5,6);
(3, 5, 6, 4)
> g^4;
Id(s6)
> IsIdentity(g^4);
true
```

As a field example, consider the rational field Q:

```
> (6/2475) * (19/2);
19/825
```

```
> (46/9)^-1;
9/46
> IsOne((46/9) * (9/46));
true
> IsZero((22/9) - (12/5));
false
```

Under certain circumstances, binary arithmetic operators defined for elements x and y belonging to the same magma may be applied successfully when x and y are elements of different magmas M_1 and M_2. The two magmas must have a *common overstructure*, that is, there must be a magma M of which they are both substructures. (It is possible for the overstructure M to be M_1 or M_2.) If such an M exists, and MAGMA is able to construct it, then x and y will be *automatically coerced* into M, and the operator will be applied to the images of x and y in M. The result will be an element of the overstructure M.

For example, suppose that the addition of an integer and a rational is attempted. The integer will be coerced into the rational field, the addition will take place as a rational-field addition, and the parent of the result will be the rational field:

```
> sum := 4 + 8/3;
> sum;
20/3
> Parent(sum);
Rational Field
```

4.5.3 Relational Operators

Virtually every magma supports the operation of testing whether two elements are equal. The **eq** operator tests for equality, and the **ne** operator tests for inequality. For instance, the statement below prints **true** because the fourth power of g is the identity permutation:

```
> g^4 eq s6!1;
true
```

In the case of certain finitely-presented magmas M, the equality operators do not test for equality in M but rather in the associated free magma; that is, they test *word equality*, without applying the relations of the presentation (see p. 593). This is because of the unsolvability of the word problem.

Certain kinds of magmas have a total ordering defined on their elements. For such magmas, elements can be compared with **lt**, **le**, **gt**, and **ge**. See Ta-

Table 4.4. Relational operators on elements of magmas

MAGMA	Meaning
x **eq** y	**true** if $x = y$
x **ne** y	**true** if $x \neq y$
x **lt** y	**true** if $x < y$ (if defined)
x **le** y	**true** if $x \leq y$ (if defined)
x **gt** y	**true** if $x > y$ (if defined)
x **ge** y	**true** if $x \geq y$ (if defined)
IsZero(x)	**true** if x is additive identity of M
IsId(x), **IsIdentity**(x)	**true** if x is identity of group M
IsOne(x)	**true** if x is multiplicative identity of ring M

ble 4.4. For instance, the following statement assigns **false** to *bigger*, because $\frac{33}{100}$ is not greater than $\frac{1}{3}$:

```
> bigger := 33/100 gt 1/3;
> bigger;
false
```

The table also lists the functions **IsZero**(x), **IsOne**(x), **IsIdentity**(x) and **IsId**(x), which were included in the arithmetic tables too. These give the same result as comparing x with the zero/one/identity element using **eq**, but may be more efficient.

If a relational operator is applied to elements x and y belonging to different magmas, the operation will only be legal if MAGMA can find a common overstructure for x and y, as explained on p. 56. In such a case, the operator will be applied to the images of x and y in the overstructure.

4.5.4 Canonical Forms of Elements

In the majority of categories in MAGMA, elements of a magma M are stored and returned in *canonical form*. That is, there is only one way in which a given $m \in M$ will be represented, and there is no concept of a 'simplified form' versus an 'unsimplified form' of m. For example:

```
> GF27<w> := GaloisField(27);
> m1 := w^6 + w^3 + 1; m1;
w^18
> m2 := 2*w^5; m2;
w^18
> m1 eq m2;
```

```
true
```

The exceptions to this rule are certain categories of finitely-presented (fp) magmas: fp semigroups, fp monoids, fp groups and fp algebras. In such structures, it is not always algorithmically possible to find canonical forms for elements. (This restriction is known as the *word problem*.) Given a word representing an element, MAGMA only applies minimal simplifications (*free reductions*) to it before storing it. For example, in the finitely-presented group below, the words assigned to *w1* and *w2* are not recognized by MAGMA as representing the same element:

```
> bt<p, q> := Group< p, q | (p*q)^2 = p^3 = q^3 >;
> w1 := p^7 * p^-2 * q^3; w1;
p^5 * q^3
> w2 := p^3 * q^5; w2;
p^3 * q^5
> w1 eq w2;
false
```

4.5.5 Generators and Element Representation

With respect to the use of generators in the input representation and output appearance of elements, it is convenient to recognize two distinct families of magma categories. The family to which a category belongs also influences the methods of creating elements of magmas in that category.

In the first family, the elements of magmas have a concrete output representation that does not explicitly display the generators. Categories such as permutation groups, matrix rings, and vector spaces belong to this family. The output representation is usually an ordered collection of elements from another magma (e.g., the coefficient ring), punctuated in a category-dependent way. The input representation for elements also tends not to require explicit reference to the generators of the magma, although this is permitted. For example:

```
> s5 := Sym(5);
> s5elt := s5 ! (1, 5, 3)(2, 4);
> s5elt2 := s5.2 * s5.1;
> s5elt, s5elt2;
(1, 5, 3)(2, 4)
(1, 3, 4, 5)

> Z13 := ResidueClassRing(13);
> Modu13 := RModule(Z13, 4);
```

```
> Modu13elt := Modu13 ! [ 3, 9, 1, 7 ];
> Modu13elt2 := 2*Modu13.3 - 5*Modu13.4;
> Modu13elt, Modu13elt2;
Modu13: (3 9 1 7)
Modu13: (0 0 2 8)
```

Naming the generators using a generator assignment statement (p. 48) makes it easier for the user to form expressions in the generators, but does not affect the appearance of output. Continuing the example:

```
> s5<a,b> := Sym(5);
> s5elt2 := b * a; s5elt2;
(1, 3, 4, 5)

> Modu13<p,q,r,s> := RModule(Z13, 4);
> Modu13elt2 := 2*r - 5*s; Modu13elt2;
Modu13: (0 0 2 8)
```

In the second family of categories, the magma M derives from an abstract free structure, in which elements are represented in output as expressions in the generators of M. This is the case particularly for *finitely-presented magmas* (including polynomial rings), which are quotients of free magmas, and whose elements are exhibited as *words* (simple arithmetic expressions) in the generators of M. To create an element, the user must either express it explicitly in terms of generators, or express it as a combination of previously-defined elements that were created directly or indirectly in terms of generators. As for output, the generators appear explicitly within each element that is printed; they are always identifiable, and so the appearance of generators in output is very important in these categories. MAGMA's default method of printing the i^{th} generator of a magma M is $M.i$ (or `$.`i if it does not know the identifier name of M in that context). However, special generator names (identifiers and printnames) may be given by the user by means of the generator assignment statement, so as to make input easier and output more readable:

```
> gf25 := GF(25);
> Generator(gf25);
GF25.1
> gf25.1 ^ 7;
gf25.1^7

> Generator(GF(81));
$.1
> Random(GF(81));
$.1^73
```

```
> GF27<w> := GaloisField(27);
> Generator(GF27);
w
> w ^ 7;
w^7
```

4.6 Submagmas, Normal Closures and Ideals

Submagmas (subgroups, subrings etc.), normal closures and ideals of a magma M are created in very similar ways, so they are discussed together in this section. In brief, a submagma, normal closure or ideal N may be built in MAGMA by means of a **sub**-constructor, **ncl**-constructor or **ideal**-constructor that specifies M and a generating set S. The relationship between N and M is given by an inclusion monomorphism $N \to M$, and this morphism is available as the second return value of the constructor.

If the new magma N is a submagma or normal closure of M, its representation will be independent of M, but if N is an ideal of M it will retain some connection to M. In any case, N will have its own generators, notated either by $N.i$ or by the names chosen in a generator assignment statement.

For example, consider the K-matrix space *m23* defined in the previous section. Let N be the subspace of *m23* generated by the matrices

$$A = \begin{pmatrix} 5 & 2 & 6 \\ \frac{4}{3} & 8 & 9 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 5 & 23 \\ 5 & 0 & \frac{43}{7} \end{pmatrix}$$

To create both N (naming the generators A and B) and the monomorphism $i : N \to m23$, the appropriate statement is:

```
> m23 := KMatrixSpace(RationalField(), 2, 3);
> N<A,B>, i:=sub<m23 | [5,2,6,4/3,8,9], [1,5,23,5,0,43/7]>;
> N;
KMatrixSpace of 2 by 3 matrices and dimension 2
over Rational Field
> i;
Mapping from: ModMatFld: N to ModMatFld: m23
> A;
[  5   2   6]
[4/3   8   9]
> B;
[   1    5   23]
[   5    0 43/7]
```

```
> (N.1 eq A) and (N.2 eq B);
true
```

These ideas will now be explained in more detail.

4.6.1 Creating Submagmas and Ideals

Let M be a magma. A submagma N of M is a magma in the same category as M such that there exists an inclusion monomorphism $i : N \to M$. Informally, we can say that the elements of N are all elements of M, provided that we remember that the parent of these elements is N and not M. This is because each magma exists in its own right (except for ideals), and the representation of the elements of N may be different from the representation of the elements of M.

Given a subset S of M, there are several different substructures that are naturally associated with M and S:

- The true *submagma* T of M generated by S, that is, the 'smallest' submagma of M containing S;
- The *normal closure* I generated by S, that is, the 'smallest' submagma of M containing S that is normal in M. This structure is defined only if M is a group.
- The *ideal* I generated by S, that is, the 'smallest' submagma of M containing S that is an ideal of M. This structure is defined only for certain categories.

Magma offers a constructor for each of these substructures:

sub< M | specification of S **>**
ncl< M | specification of S **>**
ideal< M | specification of S **>**

If M has a non-commutative multiplication and is not a group, then **ideal** denotes the two-sided ideal, and the left ideal and right ideal constructors **lideal** and **rideal** are also available. For an example, see p. 531.

In these constructors, the magma M is given on the left of the | symbol. It is not necessary to create M first and then use an identifier; any expression whose value is M is suitable. On the right of the | symbol, the set S of generators is specified, as a comma-separated list of expressions whose values are any of the following:

- An element of M;

– A sequence of primitive objects, as would be used for constructing a literal element of M – interpreted as that element of M;

– A set or sequence of elements of M;

– A submagma or ideal of M – interpreted as the list of its generators, coerced into M;

– A set or sequence of submagmas and/or ideals of M.

Any normal syntax may be used in the list of expressions. When MAGMA processes this list of generators, it removes repetitions of an element and occurrences of the identity element. (If the list is empty or contains only the identity element, the trivial substructure will be returned.) MAGMA also places an ordering upon the generators, so that one can speak of the i^{th} generator.

For example, consider the matrix subring N described above, which was created as follows:

```
> N<A,B>, i:=sub<m23 | [5,2,6,4/3,8,9], [1,5,23,5,0,43/7]>;
```

The expressions on the right side of the | are sequences of primitive objects, as would be used for constructing literal elements of *m23*. Another way to create N would be to assemble the generators in a set (or sequence) S first, and then to place S on the right side of the **sub**-constructor:

```
> S := { m23 | [5,2,6,4/3,8,9], [1,5,23,5,0,43/7] };
> S;
{
    [  5   2   6]
    [4/3   8   9],

    [   1    5   23]
    [   5    0 43/7]
}
> N<A,B>, i := sub< m23 | S >;
```

As another example, consider *s6*, the symmetric group of degree 6, and the element $g = (2, 6, 4, 3)$ of *s6*:

```
> s6 := Sym(6);
> g := s6 ! (2,6,4,3);
```

The subgroup *s6sub* generated by g and $(1, 5)$ can be built in MAGMA in the following way:

```
> s6sub := sub< s6 | g, s6!(1, 5) >;
> s6sub;
Permutation group s6sub acting on a set of cardinality 6
    (2, 6, 4, 3)
    (1, 5)
```

Here the expressions on the right side are elements of *s6*. As it happens, it is possible to put $(1, 5)$ instead of *s6*$!(1, 5)$ as one of these expressions, since it is a description of a literal element whose parent is clear from the context:

```
> s6sub := sub< s6 | g, (1, 5) >;
```

If the original magma M is not required in its own right, then there is no need to assign it to an identifier; any expression returning that magma may be placed on the left side of the constructor. However, if it is necessary to refer to $M.i$, the i^{th} generator of M, on the right side of the constructor, then a **where** clause must be used to give M a temporary name:

```
> T := sub< V | [1,3,2,0,3,1], V.2 + V.4 >
>   where V is VectorSpace(GF(5), 6);
```

For an example of the difference between the **sub**-constructor and the **ideal**-constructor, let R be the polynomial ring in x, y, z over Q, and let T and I be the subring and ideal of R generated by the set $S = \{x^2 - 2, x + y\}$:

```
> R<x,y,z> := PolynomialRing(RationalField(), 3);
> T := sub< R | x^2 - 2, x + y >;
> I := ideal< R | x^2 - 2, x + y >;
> z*(x + y) in T;
false
> z*(x + y) in I;
true
```

Note that some elements, such as $z(x + y)$, are contained in I but not in T.

4.6.2 The Embedding in the Free Magma

Given a magma T, the function **Generic**(T) returns its free or generic magma, the magma in which T is naturally embedded. If T was created as a substructure of M, **Generic**(T) will only be M if M is free itself; otherwise it will be the same as **Generic**(M).

For instance, suppose that *s6subsub* is constructed as the subgroup of *s6sub* generated by g^2. MAGMA considers *s6subsub* to be ultimately embedded in the symmetric group of the same degree:

```
> s6subsub := sub< s6sub | g^2 >;
> s6subsub;
Permutation group s6subsub acting on a set of cardinality 6
    (2, 4)(3, 6)
> s6 eq Generic(s6sub);
true
> s6 eq Generic(s6subsub);
true
```

4.6.3 Inclusion Monomorphism and Principle of Locality

An important principle to keep in mind is that no matter how a magma is created (unless it is an ideal), its representation will be independent of any magma from which it may have been created. This is the *principle of locality*. The principle of locality has the major advantage that if M is a submagma in the category C and T is a submagma of M, then T is a magma in the category C in its own right, and is not dependent on M; thus any algorithms applying to magmas in C will apply to T.

For an example of the principle of locality, suppose M is a module of dimension n over a field K. The standard representation for an n-dimensional module over a field is in terms of the standard basis for $K^{(n)}$, so all the elements will be notated with n components:

```
> M := KModule(GF27, 5);
> M;
KModule M of dimension 5 over GF(3^3)
> Random(M);
M: ( w^7 w^25 w^11  w^8    2)
```

If T is created as an m-dimensional submodule of M, then T will be presented relative to the standard basis of $K^{(m)}$, rather than that of $K^{(n)}$. Thus all the elements of T will have m components:

```
> T := sub<M | [w^8, w^13, 2, 0, 1], [1, w, 0, w^19, w^2]>;
> T;
KModule T of dimension 2 over GF(3^3)
> t := Random(T);
> t;
T: (w^18 w^14)
```

Now there is an obvious potential difficulty here. Since the elements of T are represented independently of the elements of M, how is it possible to

recognize the element of M to which a given element of T corresponds? Fortunately, it is also part of the MAGMA philosophy of *structural computation* that the morphisms connecting new structures to old ones should be accessible. Therefore the **sub**, **ncl** and **ideal** constructors (and their relatives) can return two values:

– The substructure (of whatever kind);

– The inclusion monomorphism from the substructure to the original magma.

A multiple assignment statement must be used in order to obtain both the return values of the constructor:

T, i `:=` substructure constructor;

If the constructor is used in a different way (within a larger expression or in a one-value assignment statement) then only the substructure will be returned.

For example, the following line assigns to T the same submodule as before, and also assigns to i the inclusion monomorphism from T to M:

```
> T, i := sub< M | [w^8,w^13,2,0,1], [1,w,0,w^19,w^2] >;
> i;
Mapping from: ModFld: T to ModFld: M
```

Now the morphism i can be used to translate from elements of T to elements of M:

```
> i(t);
M: (w^18 w^14 w^18    1 w^23)
```

However, in practice one rarely defines the inclusion homomorphism, because *coercion* into M has the same effect (and uses, in essence, the same homomorphism):

```
> M ! t;
M: (w^18 w^14 w^18    1 w^23)
```

Defining the inclusion homomorphism is only convenient if the user intends to map sets or sequences of elements of T into M. For instance:

```
> i( [ t, 2*t, 3*t ] );
[
    M: (w^18 w^14 w^18    1 w^23),
    M: ( w^5    w  w^5    2 w^10),
    M: (   0    0    0    0    0)
```

```
]
```

As another example, let G be the dihedral group of degree 16, represented as a finitely-presented group with generators a and b, and let H be the subgroup of G with generators $c = a^5b$ and $d = ba^{-1}$:

```
> G<a,b> := DihedralGroup(GrpFP, 16);
> G;
Finitely presented group G on 2 generators
Relations
    a^16 = Id(G)
    b^2 = Id(G)
    (a * b)^2 = Id(G)
> H<c,d> := sub< G | a^5*b, b*a^-1 >;
Finitely presented group H on 2 generators
Index in group G is 4 = 2^2
Generators as words in group G
    c = a^5 * b
    d = b * a^-1
```

Words of H are represented as products of the generators of H and their inverses. For instance:

```
> h := c^3 * d^-2;
> H eq Parent(h);
true
```

To find the word of G corresponding to h, h must be coerced into G:

```
> G ! h;
a^5 * b * a^5 * b * a^5 * b * a * b^-1 * a * b^-1
```

The result is a word of G. It can be seen to correspond to h because of the way that the generators c and d were defined.

4.7 Magma Generators

4.7.1 Generating Sets

In most algebraic categories, magmas are generally defined and represented in terms of generating sets. If M is a magma, then the function returning the set of all generators of M is **Generators**(M), and **NumberOfGenerators**(M)

or **Ngens**(M) returns how many of them there are. The elements of the set **Generators**(M) are elements of M.

Although the generating set is a set rather than a sequence, its elements may be obtained in a fixed order for convenience in computation. The element of M which is the i^{th} generator of M is given by $M.i$, where i is a positive integer. This notation may always be used for input, and is sometimes used for output. (As demonstrated above, it is also possible to choose special names for generators.) Note that the order in which the set **Generators**(M) is printed does not necessarily coresond to the $M.i$ ordering. For instance:

```
> s5 := Sym(5);
> Ngens(s5);
2
> Generators(s5);
{
    (1, 2),
    (1, 2, 3, 4, 5)
}
> s5.1, s5.2;
(1, 2, 3, 4, 5)
(1, 2)
```

There are a few exceptions to these functions, however. Firstly, if the category of M indicates that M must have exactly one generator, these functions are not used; instead, both **Generator**(M) and $M.1$ return the generator. Secondly, if the generators of M are not commonly thought of as generators within this mathematical category, then some other function name is used instead, such as **Rank**(R) for the number of indeterminates of a polynomial ring R:

```
> GF27<w> := GaloisField(27);
> Generator(GF27);
w

> R<x,y,z> := PolynomialRing(RationalField(), 3);
> Rank(R);
3
```

The generators available to the user in this way are not necessarily those used internally by Magma. In many categories, Magma distinguishes between a foreground representation (used for input/output) and a background representation (used for computation), since algorithm designers have developed special generating sets that are much more efficient than arbitrary generating sets for operations such as element membership tests. Examples

are an echelonized basis for a vector space, and a base-and-strong-generating-set (BSGS) for a permutation group or finite matrix group. User access to the internal representation depends on the category. However, the user's chosen representation is retained as far as possible for input and output.

4.7.2 Determination of the Generators

The way in which the generators of a magma M are determined and ordered by MAGMA depends on how M is constructed. There are several possibilities:

- M may be constructed as a free magma, in the strict sense of a magma whose elements are represented as appropriate arithmetic combinations of given generators;
- M may be returned by a call to a function that constructs certain standard examples of magmas in a category (e.g., **DihedralGroup**);
- M may be built by means of a **sub**-constructor, **ncl**-constructor or **ideal**-constructor;
- M may arise from some other expression – no generalizations can be made about the generators of M in this situation.

In the first case, where M is a free magma, the number of generators is given by the context. For example:

```
> FG3 := FreeGroup(3);
> FG3;
Finitely presented group FG3 on 3 generators (free)
> Generators(FG3);
{ FG3.1, FG3.2, FG3.3 }

> R := PolynomialRing(RationalField(), 3);
> R.1, R.2, R.3;
$.1
$.2
$.3

> A4 := AbelianGroup([5,3,3,7]);
> A4;
Abelian Group isomorphic to Z/3 + Z/105
Defined on 4 generators
Relations:
    5*A4.1 = 0
    3*A4.2 = 0
```

```
    3*A4.3 = 0
    7*A4.4 = 0
> Generators(A4);
{
    A4.4,
    A4.2,
    A4.1,
    A4.3
}
```

From the abelian group example, note once again that the order in which the generating set is printed does not necessarily reflect the $M.i$ ordering of the generators.

In the second case, where M is a standard magma, created by a function invocation, the generators are chosen by the algorithm designers. The user must be familiar with the function in question in order to know the number of generators and how they are chosen. For example:

```
> hamming := HammingCode(GF(2), 3);
> Ngens(hamming);
4
> [ hamming.i : i in [1..4] ];
[
    (1 0 0 0 0 1 1),
    (0 1 0 0 1 0 1),
    (0 0 1 0 1 1 0),
    (0 0 0 1 1 1 1)
]
```

In the third case, where the magma M is built by means of one of the substructure constructors, the generating set bears some relation to the elements specified in the constructor. However, the elements described on the right side of the constructor belong to the original magma, whereas the set S' given by **Generators**(M) is a set of elements of M. While S' will bear some relation to the constructor elements, it may not correspond precisely, for several reasons:

- identity elements will be removed (unless the identity is the only generator);
- repeated elements may be removed;
- other redundant elements (i.e., not independent of elements already listed) may also be removed, depending on the category;
- for ideals, extra generating elements may be required in addition to those given in the constructor;

– for polycyclic groups, if the elements given in the constructor are not of the correct form they will be replaced in S' by pc-generators.

These phenomena are category-dependent, according to the appropriate algorithms for representing magmas in diffferent categories. The ordering of these generators will be the same as the order in which the elements are listed on the right side of the constructor, once the list has been modified according to the provisos above. (If an expression in this list is a set of elements then the generators will emerge in the standard iteration order for this set, and if an expression in the list is a magma then the generators will be ordered according to their generator-order for that magma.) In summary, then, if the elements described on the right side of the constructor are independent, non-trivial and in general suitably chosen, then the generators $M.i$ will correspond to them, in the same order. Otherwise, the behaviour of MAGMA will depend on the category.

For instance, in the example of the matrix space *m23* and its subspace N, if the generators listed for the construction of N are 'sensible' then they will be taken as the generators of N (but as elements of N, rather than elements of *m23*):

```
> m23 := KMatrixSpace(RationalField(), 2, 3);
> N := sub<m23 | [5,2,6,4/3,8,9], [1,5,23,5,0,43/7]>;
> N.1, N.2;
[  5   2   6]
[4/3   8   9]

[   1    5   23]
[   5    0 43/7]
```

However, if there are unnecessary elements listed in the constructor then the correspondence may be less direct. The following pair of examples illustrates how the correspondence breaks down if the list specifying the generators includes repeated elements, non-independent elements and the identity:

```
> GF27<w> := GaloisField(27);
> VS4 := VectorSpace(GF27, 4);
> v := VS4 ! [w^14, 2, w^3, 1];
> VS4s := sub< VS4 | v, 0, 2*v, v >;
> Ngens(VS4s);
3
> VS4s.1, VS4s.2, VS4s.3;
(w^14    2  w^3    1)
(   w    1 w^16    2)
(w^14    2  w^3    1)
```

```
> Generators(VS4s);
{
    (   w    1 w^16    2),
    (w^14    2  w^3    1)
}
> Basis(VS4s);
[
    (   1 w^25 w^15 w^12)
]

> s5 := Sym(5);
> s5s := sub< s5 | (1,2,3), 1, (1,2,3)^2, (1,2,3) >;
> Ngens(s5s);
2
> s5s.1, s5s.2;
(1, 2, 3)
(1, 3, 2)
> Generators(s5s);
{
    (1, 2, 3),
    (1, 3, 2)
}
```

In both examples, the numbering of the generators does not correspond to what might immediately be expected, nor is it the same in these two categories: the fourth generator becomes the third generator in the first case and the second generator in the second case. In the vector space case, the zero is removed, and then the numbering of the generators corresponds to all the remaining generators (including the repeated generator). However, the **Generators** function returns a two-element set, since there are only two distinct generators. Note also that the basis, which is the internal representation of the generators, contains only one element. On the other hand, in the permutation group case both the identity and the repeated element are removed, whereas in the vector space example the repeated element was retained. The generator set still contains a redundancy, but after the initial construction of the group and its BSGS the efficiency of Magma's computations would not be impaired by this.

When constructing normal closures and ideals, Magma often needs to add extra elements to the given generating set. For example, consider the normal closure of the submagma of *s5* generated by $(1, 2, 3)$ and $(1, 3)(2, 4)$:

```
> s5n := ncl< s5 | (1,2,3), (1,3)(2,4) >;
> Ngens(s5n);
3
```

```
> s5n.1, s5n.2, s5n.3;
(1, 2, 3)
(1, 3)(2, 4)
(3, 5, 4)
> Generators(s5n);
{
    (1, 2, 3),
    (3, 5, 4),
    (1, 3)(2, 4)
}
```

The submagma generated by $(1, 2, 3)$ and $(1, 3)(2, 4)$ alone is not normal in *s5*, so MAGMA must enlarge that submagma with extra generators. It provides the generator $(3, 5, 4)$, and places it on the end of the generator list, so that the two generators given by the user are numbered first and second.

4.7.3 Remarks on Generator Names

Recall from p. 58 that there are two families of magma categories with respect to the input and output representation of their elements. When a magma M is assigned using a generator assignment statement, the behaviour of this statement depends on the family to which the category of M belongs.

In the first family of categories, the magma elements have a concrete representation that does not explicitly display the generators. If the category of M is in this family, a generator assignment statement will assign M itself, and assign its generators to identifiers with the given names. For example, consider the ring of complex numbers with integral coefficients. It may be represented as the subring of the 2×2 matrices in which $A + BI$ maps to $\begin{pmatrix} A & B \\ -B & A \end{pmatrix}$ and, in particular, I maps to the generator $\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$. Thus it is logical to name the generator with the identifier I:

```
> gaussians<I> :=
>   sub< MatrixRing(IntegerRing(), 2) | [0, 1, -1, 0] >;
> gaussians;
Matrix Algebra of degree 2 with 1 generator
over Integer Ring
> Generators(gaussians);
{
    [ 0  1]
    [-1  0]
}
> I^2;
```

```
[-1  0]
[ 0 -1]
> gaussians ! (-1);
[-1  0]
[ 0 -1]
```

In the second family of categories, the output representation of magma elements is always explicitly in terms of generators, and the input representation is directly or indirectly in terms of generators. If the category of M is in this family, a generator assignment statement will assign M itself, assign its generators to identifiers with the given names, and cause the generators to be printed with the given names. For example:

```
> Q := RationalField();
> R := PolynomialRing(Q, 3);
> f := R.1*R.2 + R.2*R.3 + R.1*R.3;
> f;
$.1*$.2 + $.1*$.3 + $.2*$.3

> R<x,y,z> := PolynomialRing(Q, 3);
> f := x*y + y*z + x*z;
> f;
x*y + x*z + y*z
```

If a magma M in the second family is created without naming its generators, and the user then realizes that the generators need printnames, it is possible to provide them using the procedure **AssignNames**($\tilde{}M, Q$), where Q is a sequence of the generator names given as strings. This does not create identifiers with these names; it only changes the printnames. For example:

```
> A := AbelianGroup([4,7,8]);
> A;
Abelian Group isomorphic to Z/4 + Z/56
Defined on 3 generators
Relations:
    4*A.1 = 0
    7*A.2 = 0
    8*A.3 = 0
> A.1 + 2*A.3;
A.1 + 2*A.3
> AssignNames(~A, ["k", "l", "m"]);
> A;
Abelian Group isomorphic to Z/4 + Z/56
Defined on 3 generators
Relations:
```

```
    4*k = 0
    7*l = 0
    8*m = 0
> A.1 + 2*A.3;
k + 2*m
```

Note that the generators must still be described in input using the $A.i$ syntax, since **AssignNames** does not create identifiers called k, l, m. However, the user could manually assign the generators to identifiers:

```
> k := A.1; l := A.2; m := A.3;
```

The user should take care not to reassign a generator name. Although MAGMA will permit this, and the generator will still be accessible via the $M.i$ syntax, the subsequent output may be confusing. For the second family of categories, the printname will remain the same even though the identifier no longer has its former value. For example:

```
> R<x,y,z> := PolynomialRing(Q, 3);
> 3*x*y^2 + z^3;
3*x*y^2 + z^3
> x := 17;
> x;
17
> R.1;
x
> 3*x*y^2 + z^3;
51*y^2 + z^3
> 3*R.1*y^2 + z^3;
3*x*y^2 + z^3
```

4.7.4 The Pseudo-Identifier $

MAGMA output sometimes contains the character `$`, to indicate a nameless magma. It does this when the magma has not been assigned to an identifier, or when it cannot locate the identifier to which it was assigned. For instance:

```
> Generators(FreeMonoid(6));
{ $.1, $.2, $.3, $.4, $.5, $.6 }

> FM6 := FreeMonoid(6);
> Generators(FM6);
{ FM6.1, FM6.2, FM6.3, FM6.4, FM6.5, FM6.6 }
```

4.8 Quotient Magmas

Let M be a magma, and let I be the normal subgroup or ideal of M generated by the set S of elements of M. The quotient magma $Q = M/I$ may be created with the **quo**-constructor:

quo< M | specification of S **>**

The rules for the specification of S are exactly the same as they are for the **sub**, **ncl** and **ideal** constructors. MAGMA uses these elements of M to generate I, expanding the generating set if necessary to obtain a substructure that is normal/ideal in M. Then MAGMA uses M and I to create the quotient magma Q.

For example, let *cube* be the group of symmetries of the cube, represented as a permutation group on the 8 vertices, with generators a, b, c:

```
> cube<a, b, c> := sub< Sym(8) | (1, 2, 3, 4)(5, 6, 7, 8),
>   (2, 4, 5)(3, 8, 6), (1, 5)(2, 6)(3, 7)(4, 8) >;
> cube;
Permutation group cube acting on a set of cardinality 8
    (1, 2, 3, 4)(5, 6, 7, 8)
    (2, 4, 5)(3, 8, 6)
    (1, 5)(2, 6)(3, 7)(4, 8)
```

The following statement constructs *quotgp* as the quotient of *cube* by the normal closure generated by a^2, c, c^b:

```
> quotgp := quo< cube | a^2, c, c^b >;
> quotgp;
Permutation group quotgp acting on a set of cardinality 6
Order = 6 = 2 * 3
    (1, 2)(3, 5)(4, 6)
    (1, 3, 6)(2, 4, 5)
```

Like **sub**, **ncl** and **ideal**, the **quo**-constructor returns a homomorphism as its second return value. This homomorphism is the natural homomorphism $\phi : M \to Q$. It is obtained by means of a multiple assignment statement:

Q, n **:= quo<** magma | specification of S **>**;

Continuing the example:

```
> quotgp, natl := quo< cube | a^2, c, c^b >;
> natl;
```

```
Mapping from: GrpPerm: cube to GrpPerm: quotgp
> natl(a * b);
(1, 4)(2, 3)(5, 6)
```

The natural homomorphism may also be used to map sets or sequences of elements of M, or submagmas of M, to sets or sequences of Q, or submagmas of Q. For instance:

```
> sq := [ cube | (1,6)(4,7), (1,6,8)(2,7,4), (1,8)(2,7) ];
> natl(sq);
[
    (1, 4)(2, 3)(5, 6),
    (1, 3, 6)(2, 4, 5),
    (1, 5)(2, 6)(3, 4)
]
> natl(sub< cube | a * c >);
Permutation group acting on a set of cardinality 6
    (1, 2)(3, 5)(4, 6)
```

Furthermore, it is possible to calculate *preimages* of these with the `@@` operator:

```
> ncl< quotgp | (1, 3, 6)(2, 4, 5) > @@ natl;
Permutation group acting on a set of cardinality 8
    (1, 3)(2, 4)(5, 7)(6, 8)
    (1, 5)(2, 6)(3, 7)(4, 8)
    (2, 4, 5)(3, 8, 6)
    (1, 2)(3, 4)(5, 6)(7, 8)
```

These mapping operations are explained in more detail in Chapter 7.

If the user is constructing a quotient by an ideal or normal closure I but wishes I to be assigned to an identifier, the best way to use the **quo**-constructor is to create I first, via **ideal** or **ncl**, and then to put I as the specification of S in the **quo**-constructor. The **quo**-constructor will extract the generators of I in order to construct the quotient. For example:

```
> I := ncl< cube | a^2, c, c^b >;
> quotgp eq quo< cube | I >;
true
```

As another example, consider the problem of constructing the quotient of a vector space. After the following initial assignments:

```
> V5 := VectorSpace(RationalField(), 5);
```

```
> V5;
Full Vector space of degree 5 over Rational Field
> u1 := V5 ! [23/5, -2, 8, 0, 0];
> u2 := V5 ! [-2/5, 0, 1, -4, 0];
```

these two collections of statements produce the same quotient space *Q5* and natural homomorphism n:

```
> Q5, n := quo< V5 | u1, u2 >;
> Q5;
Full Vector space of degree 3 over Rational Field

> W5 := sub< V5 | u1, u2 >;
> Q5, n := quo< V5 | W5 >;
> Q5;
Full Vector space of degree 3 over Rational Field
```

Note that since every subspace of a vector space is normal, a **sub**-constructor is used for *W5* rather than an **ideal**-constructor. The natural homomorphism n may be used to map between parts of the structures. Observe that its kernel, returned by the function **Kernel**(n), is the ideal from which MAGMA constructed the quotient:

```
> Kernel(n) eq W5;
true
```

It is important to realize that the quotient magma is not a substructure of the original M, unlike a magma returned by **sub**, **ncl** or **ideal**. In the first example above, *cube* and *quotgp* are both represented as permutation groups, but *cube* is acting on a set of cardinality 8 whereas *quotgp* is acting on a set of cardinality 6. In the second example, *V5* and *Q5* are both vector spaces, but the number of components in the vectors is different.

Moreover, for many categories of magma, the MAGMA representation of a quotient of M may not lie in the same category as M does. This occurs because the representation and/or operations on elements of the quotient magma are different. Consider, for example, the polynomial ring $P = \mathbb{Q}[x]$, and an element f of P. Elements of a quotient Pq relative to the ideal I generated by f are standard representatives of the cosets of I. Thus Pq is in a different category, the category **RngUPolRes** of *residue classes* of a univariate polynomial ring. The multiplication of elements in Pq is achieved by multiplying the polynomial-representatives and then taking the remainder upon division by f. For instance:

```
> P<x> := PolynomialRing(RationalField());
```

```
> f := x^3 + 5;
> Pq<y> := quo< P | f >;
> Pq;
Univariate Quotient Polynomial Algebra in y
over Rational Field
with modulus y^3 + 5
> Category(P), Category(Pq);
RngUPol RngUPolRes

> p1 := 5*x^9 - 6*x^5 + 2*x^3 + x^2 - 7;
> p2 := 5*x^2 + x - 1;
> q1 := Pq ! p1; q1;
31*y^2 - 642
> q2 := Pq ! p2; q2;
5*y^2 + y - 1
> (p1 * p2) mod f;
-3241*x^2 - 1417*x + 487
> q1 * q2;
-3241*y^2 - 1417*y + 487
```

4.9 Shorthand Constructors

For many categories, shorthand constructors are provided that allow any magma to be constructed in a single step. Implicitly, the construction is either as a finitely-presented magma (i.e., a quotient of a free magma), or as a submagma of a generic magma. However, the user does not have to construct the free or generic magma explicitly.

4.9.1 Finitely-Presented Magmas

Let M be a magma belonging to an algebraic variety V. A well-known theorem states that V contains free algebras and that M is isomorphic to a quotient of some free algebra F. A description of M as a quotient of a free algebra will be called a *presentation*. Informally, a presentation consists of the free algebra F and a set S of elements of F whose ideal closure I has the property that $M \cong F/I$. If S is finite, then M is said to be *finitely-presented*.

The construction of a finitely-presented magma from first principles involves constructing the corresponding free magma and then taking a suitable quotient. For instance, the group G given by the presentation $\langle c, d \mid c^2, d^3, (cd)^4 \rangle$ can be created as a quotient of the free group F on two generators:

```
> F<a,b> := FreeGroup(2);
> G<c,d> := quo< F | a^2, b^3, (a*b)^4 >;
> G;
Finitely presented group G on 2 generators
Relations
    c^2 = Id(G)
    d^3 = Id(G)
    (c * d)^4 = Id(G)
```

However, since the construction of finitely-presented magmas is so frequently performed, there are shorthand constructors provided, allowing the direct construction of the finitely-presented magma from its presentation without explicitly building the free magma first. The categories having the shorthand form are **GrpFP**, **SgpFP**, **MonFP**, **GrpPC**, and **GrpAb** (i.e., finitely-presented groups, semigroups, monoids, polycyclic groups and abelian groups). The syntax is:

constructor-word< generators | relations >

where the word on the left of the constructor may be any one of **Group**, **Semigroup**, **Monoid**, **PolycyclicGroup** or **AbelianGroup**. The generators are a comma-separated list of identifiers, which are used as the generator names in the relations appearing on the right side of the constructor; the number of generators indicates the rank of the corresponding free magma. The relations, whose syntax depends on the category, are given in a comma-separated list on the right of the constructor. A relation may be of the form w_L=w_R, where w_L and w_R are words in the generators; if the free magma has an identity, then the word w by itself is understood to mean that w equals the identity; relation lists w_1=w_2=$\cdots$=w_k may also be permitted.

Generator assignment is still necessary if the generators are to have special names; the scope of the generator names used in the constructor is restricted to the constructor. It does not matter whether or not the same generator names are used inside and outside the constructor.

Examples are given for each of the possible categories:

```
> G<c,d> := Group< a, b | a^2, b^3, (a*b)^4 >;
> G;
Finitely presented group G on 2 generators
Relations
    c^2 = Id(G)
    d^3 = Id(G)
    (c * d)^4 = Id(G)

> SG<s,t> := Semigroup< s, t | s^7 = t^4, s*t = t*s^2 >;
```

```
> SG;
Finitely presented semigroup
Relations
    s^7 = t^4
    s * t = t * s^2

> M<A,B,C> :=
>     Monoid< a, b, c | a^5, b^2*c = a^3*c*b = b*a*b*c >;
> M;
Finitely presented monoid
Relations
    A^5 = Id(M)
    B^2 * C = B * A * B * C
    A^3 * C * B = B * A * B * C

> PG<e,f> := PolycyclicGroup< e, f | e^2, f^3, f^e=f^2 >;
> PG;
GrpPC : PG of order 6 = 2 * 3
PC-Relations:
    e^2 = Id(PG),
    f^3 = Id(PG),
    f^e = f^2

> A<x,y,z> := AbelianGroup< x, y, z | 2*y=4*x, x+10*y >;
> A;
Abelian Group isomorphic to Z/42 + Z
Defined on 3 generators
Relations:
    x + 10*y = 0
    42*y = 0
```

Like the **quo**-constructor, these shorthand constructors return the natural homomorphism as their second return value. The domain of the homomorphism is always the corresponding free magma:

```
> A<x,y,z>, n := AbelianGroup< x, y, z | 2*y=4*x, x+10*y >;
> n;
Mapping from: Abelian Group isomorphic to Z + Z + Z
Defined on 3 generators (free)
to
Abelian Group isomorphic to Z/42 + Z
Defined on 3 generators
Relations:
    $.1 + 12*$.2 = 0
    42*$.2 = 0
```

4.9.2 Submagmas of Generic Magmas

A similar shorthand device applies for some categories that have generic magmas: matrix groups, matrix algebras (matrix rings), permutation groups, and linear codes. However, the magma constructed is a submagma of the generic magma, not a quotient. The syntax for these shorthand constructors is:

constructor-word< generic magma specification | generators >

The submagma generators on the right of the constructor may be expressed in the same way as for a **sub**-constructor. The generic magma specification on the left of the constructor is category-dependent, but it relates to the arguments given to the function that constructs the generic magma. If the constructor-word is **MatrixGroup**, the information required is the degree followed by the coefficient ring, as for $\mathbf{GL}(n, R)$; if it is **MatrixAlgebra** (or equivalently **MatrixRing**), the information required is the coefficient ring followed by the degree, as for $\mathbf{MatrixAlgebra}(R, d)$; if it is **PermutationGroup**, the degree is required, as for $\mathbf{Sym}(n)$; if it is **LinearCode**, the finite field and length are required, as for the function $\mathbf{VectorSpace}(K, n)$.

In most of these categories, the shorthand constructor returns the inclusion monomorphism from the new magma to the generic magma as its second return value, just as if the **sub**-constructor had been applied.

Examples are:

```
> K<w> := FiniteField(9);
> H, i := MatrixGroup< 3, K |
>     [1,w,0, 0,1,0, 1,w^2,1], [w,0,0, 0,1,0, 0,0,w] >;
> H;
MatrixGroup(3, GF(3^2))
Generators:
    [  1   w   0]
    [  0   1   0]
    [  1 w^2   1]

    [  w   0   0]
    [  0   1   0]
    [  0   0   w]
> i;
Mapping from: GrpMat: H to GL(3, GF(3, 2))

> A, i := MatrixAlgebra< RationalField(), 3 |
```

```
>     [ 1/3,0,0,  3/2,3,0,   -1/2,4,3  ],
>     [ 3,0,0,    1/2,-5,0,   8,-1/2,4 ]   >;
> A : Maximal;
Matrix Algebra of degree 3 with 2 generators
over Rational Field
Generators:
[ 1/3    0    0]
[ 3/2    3    0]
[-1/2    4    3]

[   3    0    0]
[ 1/2   -5    0]
[   8 -1/2    4]
> i;
Mapping from: AlgMat: A to
Full Matrix Algebra of degree 3 over Rational Field

> G, i := PermutationGroup< 9 | (2,4)(6,8), (1,3,5,7,9) >;
> G;
Permutation group acting on a set of cardinality 9
    (2, 4)(6, 8)
    (1, 3, 5, 7, 9)
> Domain(i), Codomain(i);
Permutation group G acting on a set of cardinality 9
    (2, 4)(6, 8)
    (1, 3, 5, 7, 9)
Symmetric group acting on a set of cardinality 9
Order = 362880 = 2^7 * 3^4 * 5 * 7

> Q := [ [1,0,0,0,0,1,1], [0,1,0,0,1,0,1],
>        [0,0,1,0,1,1,0], [0,0,0,1,1,1,1] ];
> LinearCode< GF(2), 7 | Q >;
[7, 4, 3] Hamming code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 0 1]
[0 0 1 0 1 1 0]
[0 0 0 1 1 1 1]
```

4.10 Some Other Operations on Magmas

Table 4.5 summarizes the principal operators and functions that apply to (nearly) all magmas. They are illustrated below.

Table 4.5. Operations on magmas

MAGMA	Meaning
$\#M$	Number of elements of finite magma M
L **eq** M	**true** if magmas L and M are equal
L **ne** M	**true** if L and M are not equal
L **cmpeq** M	**true** if L **eq** M is **true**; else **false**
x **in** M	**true** if x is an element of M
x **notin** M	**true** if x is not an element of M
L **subset** M	**true** if elements of L are all in M
L **notsubset** M	**true** if elements of L are not all in M
L **meet** M	Intersection of L and M
Rep(M)	An arbitrary element of M (the same each time)
Random(M)	A random element of finite magma M
Set(M)	Set of elements of finite magma M
FormalSet(M)	Formal set of elements of magma M

If an expression returning a finite magma is preceded by the operator #, MAGMA will return the cardinality of the magma, that is, the number of elements it has. (If M is a group, **Order**(M) is a synonym for $\#M$.) For instance, the following output verifies that the finite field of size 2^4 does indeed have 16 elements:

```
> #GF(2, 4);
16
```

Given two comparable magmas L and M, the expression L **meet** M returns their intersection, that is, the magma whose elements are the elements common to L and M. Normally, the intersection will satisfy the same axioms as the magmas from which it was built. For instance, when two subspaces of the vector space *V5* created above are intersected, the result is also a subspace of *V5*:

```
> V5 := VectorSpace(RationalField(), 5);
> u1 := V5 ! [23/5, -2, 8, 0, 0];
> u2 := V5 ! [-2/5, 0, 1, -4, 0];
> W5 := sub< V5 | u1, u2 >;
> X5 := sub< V5 | u1+u2, V5![6, -2, -1/5, 0, 3] >;
> W5 meet X5;
Vector space of degree 5, dimension 1 over Rational Field
Echelonized basis:
(     1 -10/21   15/7 -20/21      0)
```

The functions **Random**(M) and **Rep**(M) both return an element of the specified magma M, where M must be finite for the function **Random**(M). **Rep** simply returns an arbitrary element of M, as a representative of it. It will return the same representative each time, since it is assumed that the user does not mind which representative is taken. By contrast, when **Random** is called, each element has equal probability of being chosen, and if the function is called repeatedly it is probable that different elements will be returned.

The function **Set**(M) returns the set of elements of the (small) finite magma M, and **FormalSet**(M) returns the formal set of elements of any magma M:

```
> Set(GF27);
{ 1, w, w^2, w^3, w^4, w^5, w^6, w^7, w^8, w^9, w^10, w^11,
w^12, 2, w^14, w^15, w^16, w^17, w^18, w^19, w^20, w^21,
w^22, w^23, w^24, w^25, 0 }
> FormalSet(FreeGroup(3));
Formal set over Finitely presented group
on 3 generators (free)
```

Several relational operators may be applied to an element and a magma, or to two magmas, as shown in the table. For example, the following lines indicate that the vector $u1$ is not in the vector space $X5$:

```
> u1 in X5;
false
```

If several objects are grouped together, in a set, sequence, or submagma L, the expression L **subset** M tests whether these objects are all in the magma M. For instance, the following line returns **true**, as might be expected from the definition of intersection:

```
> (W5 meet X5) subset W5;
true
```

The operator **eq** tests two magmas to see if they are equal:

```
> (W5 meet X5) eq W5;
false
```

These relational operators are only available when the operands are *comparable* (i.e., the operands must belong to the same magma S or be sufficiently related for automatic coercion to bring them into a common overstructure S, and the relevant operator must be defined for S). If they are not comparable, then an error message will be given. For example:

```
> "w" in WittDesign(12);

>> "w" in WittDesign(12);
            ^
Runtime error in 'in': Bad argument types
> MatrixRing(GF(2), 3) eq Sym(6);

>> MatrixRing(GF(2), 3) eq Sym(6);
                                ^
Runtime error in 'eq': Bad argument types
```

There is an additional relational operator, **cmpeq**, which is designed for such purposes as testing the suitability of arguments in function/procedure invocations. It is similar to **eq**, but never gives an error message if the arguments are incomparable. The value of L **cmpeq** M is **true** whenever the value of L **eq** M is **true**; otherwise, its value is **false**, no matter whether L **eq** M would return **false** or give an error message. Another way to view this operator is that it returns **true** only if L and M have a common overstructure S, **eq** is defined for the category of S, and L **eq** M is **true**; it returns **false** otherwise. For example:

```
> W5 cmpeq W5;
true
> MatrixRing(GF(2), 3) cmpeq Sym(6);
false
```

Further operations on magmas are explained in later chapters of this book, since they are more dependent on the category or variety. However, this book cannot explain every feature; see the *Handbook* or the online help for a complete account of the system.

4.11 Warning About Large Magmas

MAGMA has been designed to perform efficient computations, even on magmas which are very large. For instance, given a permutation group, it constructs a concise representation rather than a physical list of its elements, to save speed and time. Moreover, once a piece of information has been constructed for some magma, it is usually cached automatically, for subsequent reference. Nonetheless, calculations on large magmas can often consume significant amounts of computer time and/or storage space.

When working with small magmas in MAGMA, it does not matter if unnecessary or inefficient calculations are performed, because the execution time

and storage space used will be insignificant. It is with very large magmas that care is required. Before writing a program to build a standard mathematical object, the user should check whether MAGMA already provides a function or operator that performs the desired task. The MAGMA version will probably be faster than the user's own version, since the algorithm is likely to be the product of extensive research, and since MAGMA can make the most of its information storage techniques.

5. Conditional Statements and Expressions

One of the major purposes of Boolean values is to manage the flow of control in a program. Conditional statements and expressions use the results of Boolean tests in order to decide which statement to perform next, or which expression to evaluate. The main conditional statement in MAGMA is the **if**-statement, and the main conditional expression is the **select**-expression. Also provided are a **case**-statement and a **case**-expression, which are used when there are several possible courses of action, depending on the value of some expression.

The examples given in this chapter of conditionals are rather artificial. The utility of these constructions will be seen once user-defined functions and procedures are introduced in Chapter 8.

5.1 The if-statement

5.1.1 Basic Form of the if-statement

MAGMA's **if**-statement causes MAGMA to perform different actions depending on whether a certain condition is true or false. The **if**-statement has several syntactic forms, the most basic of them being:

```
if condition then
   statements
else
   statements
end if;
```

The statements between **then** and **else** are executed if the condition is true, and the statements between **else** and **end if** are executed if the condition is false. Note that, like all other statements, the **if**-statement ends with a semicolon. The newlines and indentation are optional, and are ignored by MAGMA, but they tend to make the code easier to read.

For example, suppose that a program is being written in which the absolute value of an identifier x has to be calculated and stored in y. If $x \geq 0$, y should equal x, but otherwise y should become the negation of x. The Boolean condition x **ge** 0 is appropriate for such a situation. To test an **if**-statement performing this action, x must be assigned a value (say -17) before the **if**-statement is executed:

```
> x := -17;
> if x ge 0 then
>     y := x;
> else
>     y := -x;
> end if;
> print y;
17
```

The effect of this segment of code could also be achieved with MAGMA's intrinsic function **Abs**(r):

```
> y := Abs(x);
> print y;
17
```

In order to test the absolute-value code for other integers, it is necessary to assign another value to x and re-enter the code once more. The best way to do this would be to enclose the **if**-statement in a procedure, as explained in Chapter 8. However, there are several other workarounds: it may be possible to use the editing and copying facilities provided by the computer rather than typing the whole program again; MAGMA's history commands may be used, as detailed in Chapter 15.2.2; or the code can be placed in a file and loaded into MAGMA.

5.1.2 Special Prompt Symbols

While the **if**-statement is being typed, MAGMA may provide a *special prompt* `if>` as a reminder that the statement has not yet closed:

```
> x := -17;
> if x ge 0 then
if> y := x;
if> else
if> y := -x;
if> end if;
```

```
> print y;
17
```

(MAGMA behaves in a similar way for other incomplete statements, such as the **while** and **for** statements explained in Chapter 9.) This prompt was turned off in the program listing above, so that the program would be easier to read; this convention will be followed throughout this book. The procedure **SetPrompt**(P), where P is a string, may be used to control the style of prompt. To request the simple prompt, the user should type

```
> SetPrompt("> ");
```

and to request the prompt which gives a reminder about which statements have not yet closed, the user should type

```
> SetPrompt("%S> ");
```

5.1.3 Short Form of the if-statement

If a program requires an **if**-statement for which there is no action to be taken if the condition is false, a shorter form of the **if**-statement should be used. This shorter form simply omits the **else** clause:

if condition **then**
  statements
end if;

For example, the absolute value code above could be rewritten as follows:

```
> x := -54;

> y := x;
> if x lt 0 then
>     y := -y;
> end if;
> print y;
54
```

5.1.4 Avoiding Nested if-statements

Sometimes the **else** clause of an **if**-statement will itself be an **if**-statement. At the end of such an **if**-statement there would be matching **end if**; closures.

This type of nested **if**-statement may be avoided by making use of **elif**, which is a shorter form of **else if** that also removes the need for an **end if** to close off the inner **if**-statement:

if condition **then**
  statements
elif condition **then**
  statements
elif condition **then**
  statements
  ⋮
else
  statements
end if;

For example, an **if**-statement with **elif** clauses may be used to convert from a percentage mark into a grade. The program assumes that *mark* has been assigned an integer in the range $0, \ldots, 100$:

To perform this example online, type `load "I96c5e1";`

```
> if mark ge 85 then
>     print "High Distinction";
> elif mark ge 75 then
>     print "Distinction";
> elif mark ge 65 then
>     print "Credit";
> elif mark ge 50 then
>     print "Pass";
> else
>     print "Fail";
> end if;
```

5.1.5 Longer Example: Area of a Triangle

The next example of the **if**-statement is a procedure $TriangleArea(a, b, c)$ that, given the side-lengths a, b, c of a triangle, calculates and prints the area of the triangle. It uses the formula $Area = \sqrt{s(s-a)(s-b)(s-c)}$, where s equals $(a+b+c)/2$. To check that the triangle exists, it checks that the sum of the two smaller sides is greater than or equal to the third side. (If they are equal, the triangle is degenerate.)

The code has been written as a procedure so that it can be tested easily for several choices of side-lengths. The first and last lines handle the procedure-aspect of the code, and should not cause confusion.

The first two lines of the procedure body (i.e., the second and third lines of the program) are sequence manipulations (see Chapter 6) whose purpose is to assign A, B, C the same values as a, b, c, but sorted from lowest to highest. **Explode**(Q) is a multiple return value function that returns the first term, the second term, the third term, and so on, of a given sequence Q.

To perform this example online, type `load "I96c5e2";`

```
> TriangleArea := procedure(a, b, c)
>     Q := Sort([a, b, c]);
>     A, B, C := Explode(Q);
>     sumAB := A + B;
>     if sumAB lt C then
>         print "No triangle has these side-lengths.";
>     elif sumAB eq C then
>         print "Triangle is degenerate (straight line).";
>       print "Area is 0 square units.";
>     else
>         s := (A+B+C)/2;
>         print "Area is", Sqrt(s*(s-A)*(s-B)*(s-C)),
>           "square units.";
>     end if;
> end procedure;
```

The procedure *TriangleArea*(a, b, c) may be called by supplying suitable expressions for a, b and c. For example:

```
> TriangleArea(6, 8, 10);
Area is 24.000000000000000000000000000 square units.
> TriangleArea(10.2, 12, 14.5);
Area is 60.466393507054809296259993289470 square units.
> TriangleArea(6, 20, 3);
No triangle has these side-lengths.
> TriangleArea(13, 9, 4);
Triangle is degenerate (straight line).
Area is 0 square units.
```

The figures in the output have a rather large number of digits because the arguments are elements of the real field (or coerced into it for the square root computation), which has default precision 28. This precision could be adjusted if required.

5.2 The select-expression

The **if**-statement is a very flexible statement. It can be used whenever the action to be performed depends on the truth of some expression. However, if the only action of the **if**-statement is to assign or print the value of an expression, it is sufficient to use a *conditional expression* instead. The most common form of conditional expression in MAGMA is the **select**-expression:

condition **select** expression **else** expression

and it has the value calculated from the first expression if the condition is true, and the value from the second expression if the condition is false. Since a **select**-expression can be used as part of a longer expression, it can be very convenient for such tasks as constructing sets and sequences.

For instance, the first example in this section could also be programmed as follows:

```
> y := x ge 0 select x else -x;
```

To evaluate this statement, MAGMA first tests the x **ge** 0 condition. If it is true, then y is assigned the first value, x, and otherwise y is assigned the second value, $-x$. For instance:

```
> x := 42;
> y := x ge 0 select x else -x;
> print y;
42
> x := -9;
> y := x ge 0 select x else -x;
> print y;
9
```

Like the operators **and** and **or**, the **select**-expression uses *call-by-name* evaluation. That is, it does not evaluate both the expressions following **select**, only the one specified by the truth or falsity of the initial condition.

5.3 The case-statement

The **case**-statement is rather like the **elif** variety of the **if**-statement, but it is used when the action depends on which of several values is equal to a given value. The syntax of the **case**-statement is:

case expression:
when expression, ..., expression:
  statements
when expression, ..., expression:
  statements
⋮
else
  statements
end case;

The **else** clause is optional.

To execute this statement, MAGMA evaluates the expression given after **case**. It successively compares the value of this expression with the values of the expressions given after each **when**. If it finds a match, MAGMA executes the statements following the corresponding **when**. If none of the values match, MAGMA executes the statements after **else**, or does nothing if the **else** clause is absent.

MAGMA executes at most one of the blocks of statements in a **case**-statement. If more than one of the **when** options contains an expression whose value equals the value of the **case** expression, MAGMA will only execute the statements corresponding to the first of these **when** options, starting from the top.

For example, the *TriangleArea*(a, b, c) procedure constructed above may be modified to use a **case**-statement. The expression given after **case** is **Sign**$(A + B - C)$, since it is the sign of $A + B - C$ that governs which statements should be executed. The intrinsic function **Sign**(x) returns 1, 0 or -1 according to whether x is positive, zero or negative.

```
> TriangleArea := procedure(a, b, c)
>     q := Sort([a, b, c]);
>     A, B, C := Explode(q);
>     case Sign(A + B - C):
>     when -1:
>         print "No triangle has these side-lengths.";
>     when 0:
>         print "Triangle is degenerate (straight line).";
>       print "Area is 0 square units.";
>     else
>         s := (A+B+C)/2;
>         print "Area is", Sqrt(s*(s-A)*(s-B)*(s-C)),
>           "square units.";
>     end case;
> end procedure;
```

The output for this procedure is the same as for the earlier version.

Notice that since the **else** section of a **case**-statement is optional, the **else** and the following **print**-statement in the example above could have been written as another **when** option instead:

```
>     when 1:
>         s := (A+B+C)/2;
>         print "Area is", Sqrt(s*(s-A)*(s-B)*(s-C)),
>           "square units.";
```

5.4 The case-expression

The **case**-expression resembles the **select**-expression in that it is a conditional expression, returning a value. It also resembles the **case**-statement, because it allows multiple possibilities instead of just **true**/**false** branching. The syntax of the **case**-expression is:

case< expr | expr_{L_1}:expr_{R_1}, expr_{L_2}:expr_{R_2}, ..., **default**:$\text{expr}_{\text{dflt}}$ **>**

The **default** section is compulsory, unlike the **else** clause in the **case**-statement.

To evaluate a **case**-expression, MAGMA begins by evaluating the expression given after **case**. It then evaluates each expr_{L_i}, starting at $i = 1$, and compares the result with the value of the original expression until there is a match for some i or until all the expr_{L_i} have been tested. If a match is found, MAGMA returns the value of the corresponding expr_{R_i}; otherwise, it returns the value of $\text{expr}_{\text{dflt}}$, to the right of **default**. Note that the evaluation is *call-by-name*, as for the **select**-expression.

As a short example, the following line prints a description of the parity of a given integer:

```
> n := 56;
> print case< IsOdd(n) | true:"odd", default:"even" >;
even
> n := 17;
> print case< IsOdd(n) | true:"odd", default:"even" >;
odd
```

The **case**-expression may be used to substitute for some **if**-statements with **elif** clauses, if the initial expression is **true**. The code below illustrates this for the marks-to-grades example discussed on p. 90:

```
> mark := 60;
> grade := case< true |
>     mark ge 85 : "High Distinction",
>     mark ge 75 : "Distinction",
>     mark ge 65 : "Credit",
>     mark ge 50 : "Pass",
>     default: "Fail" >;
> print grade;
Pass
```

6. Aggregate Structures

Aggregate structures are used in MAGMA in order to gather objects together into one object so that they may be operated on as a body. There are many categories of aggregate structures, the principal ones being enumerated sets and enumerated sequences. This chapter describes the features of each kind of aggregate structure, and explains how to construct and operate upon them.

The greatest emphasis is given to the homogeneous iterable aggregate categories, that is, those categories such that the elements of each aggregate all have the same parent magma, and may be produced on demand by the system. Following these, formal sets (homogeneous non-iterable unordered aggregates without repeated elements) and lists (non-homogeneous iterable ordered aggregates) are discussed. Cartesian products receive special attention, both because they are a special kind of set created as the product of several magmas or aggregates, and because their elements, known as tuples, may be seen in their own right as aggregates (fixed-length, non-homogeneous, iterable, and ordered). Coproducts, the dual of Cartesian products, are also discussed. The final topic of the chapter is the category of records, which are aggregate objects corresponding to a pre-defined record format.

6.1 Properties of the Aggregate Categories

Table 6.1 lists all the categories of aggregates. Notationally, the aggregate categories are distinguished by different kinds of bracketing symbols surrounding the description of the aggregate's contents; the table includes these bracketing symbols for reference. Most of MAGMA's aggregate categories will be familiar from mathematical contexts. The exceptions are records and record formats, which are primarily programming constructions. For this reason, the discussion of records and record formats will be deferred until the end of this chapter.

Several attributes distinguish the other categories of aggregates. The chief features of an aggregate category C are whether, for every aggregate S in C,

Table 6.1. Categories of aggregate structures, showing bracketing symbols

Bracketing	Category	Meaning
{ ... }	**SetEnum**	Enumerated set
{* ... *}	**SetMulti**	Multiset
{@ ... @}	**SetIndx**	Indexed set
[...]	**SeqEnum**	Enumerated sequence
{! ... !}	**SetFormal**	Formal set
[* ... *]	**List**	List
car< ... >	**SetCart**	Cartesian product
< ... >	**Tup**	Tuple (element of cartesian product)
cop< ... >	**Cop**	Coproduct
recformat< ... >	**RecFrmt**	Record format
rec< ... >	**Rec**	Record of specified format

S is homogeneous, iterable, ordered, and/or capable of containing repeated elements. These features will now be briefly explained.

A *homogeneous* aggregate is a collection of objects which all have the same parent magma. Many mathematical applications of aggregate structures involve homogeneous aggregates, and if an aggregate is known to be homogeneous, it can be stored and manipulated more efficiently by the computer. For this reason, MAGMA offers categories of aggregate structures having the restriction that, for each aggregate S, all the elements of S must belong to the same magma. This common parent is called the *universe*; almost any magma is valid as a universe. (Strictly speaking, a universe U of an aggregate S may also be a subset of a magma M, so that the elements of S have M as their parent but the aggregate has U as its universe.) The categories of homogeneous aggregates in MAGMA are:

set (strict sense)	**SetEnum**	enumerated sets
set (strict sense)	**SetFormal**	formal sets
set (strict sense)	**SetCart**	Cartesian products
multiset	**SetMulti**	multisets
ordered set	**SetIndx**	indexed sets
ordered multiset	**SeqEnum**	enumerated sequences

In the summary above, a set in the 'strict sense' means a set according to the ordinary mathematical sense of an unordered collection of distinct objects.

An *iterable* aggregate is a finite aggregate all of whose elements $S_1, \ldots, S_n$ MAGMA can produce successively on demand. Programming situations requiring iteration over S include: a **for**-statement having S as its domain of iteration, as explained in Section 9.4, an element-description constructor or

related construct having S as a domain for a free identifier, as discussed in Section 6.2.5, Section 6.6, and Section 6.7, and the evaluation of a reduction operator applied to S (Section 6.3.5). In all cases, the iteration employs the same internal mechanism to generate the S_i. In order for an aggregate S to be iterable, it must not only have a finite cardinality, but it must also be stored in such a way that the system can generate the elements of S from the internal description of the aggregate. Most iterable aggregates in MAGMA are stored as a collection of elements, so that iteration over these elements is trivial. (The exceptions are enumerated sets or enumerated sequences that have been created as arithmetic progressions; for these, MAGMA stores only the initial term, the final term and the common difference, but can generate all the elements very easily from them.) All the categories of homogeneous aggregates except formal sets and infinite Cartesian products are iterable, as are lists. Tuples are also iterable, in the limited sense that the user may construct the expressions $t[1], \ldots, t[n]$, where n is the number of components of the tuple t.

An aggregate with *repeated elements* is one which does not satisfy the property that all its elements are distinct. Multisets, enumerated sequences, lists, and tuples may contain repeated elements; the other aggregate categories may not. For an aggregate with repeated elements, the elements will appear with the appropriate multiplicity during iteration, and the multiplicities are counted in the cardinality.

An *ordered* aggregate is an aggregate S supplied with an index map from $\{1, \ldots, n\}$ to S, where n is the cardinality of S. In MAGMA, the i^{th} element of an ordered aggregate S is notated $S[i]$, and whenever iteration occurs over S it will be in the order $S[1], \ldots, S[n]$, 'from left to right'. (There is a complication for enumerated sequences, since they may have undefined terms: the cardinality of an enumerated sequence is the index of its rightmost defined term, but during iteration the undefined terms are ignored.) The categories of ordered aggregates are indexed sets, enumerated sequences, lists, and tuples. Note that all of these can have repeated elements except indexed sets. The reason for this is that indexed sets are implemented as distinct-element sets stored in a hash table, with an index map attached, whereas the aggregates in the other categories are implemented as linear lists of elements, so that repeated elements can be distinguished by their index position.

The most important aggregate categories in MAGMA are enumerated sets and enumerated sequences, which are often known simply as sets and sequences. Both these categories of aggregates are highly optimized, and have many operations available. For this reason, *the user should give preference to enumerated sets and enumerated sequences when choosing an aggregate data structure.* An *enumerated set* in MAGMA corresponds to the mathematical concept of a set as an unordered collection of distinct objects, except that it must be finite and homogeneous. MAGMA uses a hash table to store the ele-

ments of an enumerated set, so that the set membership test is very fast. An *enumerated sequence* is a finite, homogeneous, ordered collection of objects, in which repetitions are permitted. It is possible to create enumerated sequences 'out of order', so that some intermediate terms are undefined. There are many operations for the manipulation of sequences, such as extracting or inserting elements and subsequences.

6.2 Construction of Iterable Homogeneous Aggregates

Enumerated sets, multisets, indexed sets and enumerated sequences (the iterable homogeneous aggregate categories) share many features with respect to construction. There are two main ways to create an aggregate structure belonging to one of these categories: from a list of expressions whose values are the elements; or from an element-description, given in set-theoretic notation, which is then evaluated by the system into a collection of elements. (It is also, of course, possible to construct an aggregate structure by modifying an existing aggregate structure or by using a function that returns such a structure.) For the sake of convenience and efficiency, there is a special construction method for enumerated sets and sequences consisting of an arithmetic progression of integers, and these structures are stored as a virtual collection of elements.

These four categories are distinguished during construction by the kind of brackets surrounding the list or description of the elements, as listed in Table 6.1 (p. 98). Notice that enumerated sets and sequences have bracketing symbols consisting of only one character (the brace { and the square bracket `[` respectively), and that the other two kinds of sets have a compound bracketing symbol that includes the set brace ({`*` for multisets and {`@` for indexed sets).

6.2.1 Construction from an Element-List

Let S be an enumerated set, multiset, indexed set or enumerated sequence. If $e_1, \ldots, e_k$ are expressions evaluating to the elements of S, then S may be constructed by means of a comma-separated list of the e_i, enclosed in the appropriate bracketing symbol for the category of S. For example:

```
> odds := {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
> print odds;
{ 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 }

> q1 := [4, 8/3, 7/10, 33, 2, 1/2];
```

```
> print q1;
[ 4, 8/3, 7/10, 33, 2, 1/2 ]

> FA3<a,b,c> := FreeAbelianGroup(3);
> ix := {@ a, b+c, 2*a-7*c @};
> print ix;
{@
    a,
    b + c,
    2*a - 7*c
@}

> ms := {* 2, 2, 3, 6, 1, 3, 3, 5, 3 *};
> print ms;
{* 1, 2^^2, 3^^4, 5, 6 *}
```

As the printed version of the multiset *ms* suggests, an element of a multiset may be supplied together with its multiplicity, using the `^^` operator. In an element list for a multiset, `e^^m` has the same effect as m repetitions of e in the list. The integer m is called the *multiplicity* of e. Given a multiset S and a suitable object x, the function **Multiplicity**(S, x) returns the multiplicity of x in S, or 0 if x is not in S. For instance:

```
> ms2 := {* (a+4*c)^^4, b+c, 2*c, 9*a+b-5*c, 7*a^^2 *};
> print ms2;
{*
    a + 4*c^^4,
    2*c,
    7*a^^2,
    b + c,
    9*a + b - 5*c
*}
> print Multiplicity(ms2, a+4*c);
4
> print Multiplicity(ms2, 1000*b);
0
```

It is never an error if repeated values emerge from the list of expressions, but the resulting structure depends on the category. For enumerated sets and indexed sets, in which all the elements are distinct, any duplicated values are stored only once, like all the other elements; for multisets, the repeated values are accumulated, to give an overall multiplicity for that element; for enumerated sequences, each value is stored at its appropriate index position, and MAGMA does not collect together the repetitions in any sense.

Both indexed sets and sequences have an *ordering* on their elements. If an indexed set or sequence S is created by means of an expression list, the resulting ordering corresponds to the ordering of the values in the list (after duplicates are removed, in the case of indexed sets). In particular, $S[i]$ is the i^{th} element of S, counting from the left. The indexing commences at 1, not at 0 as in some programming languages. For example:

```
> print ix[2];
b + c
```

6.2.2 The Universe of a Homogeneous Aggregate

The function returning the universe of a homogeneous aggregate structure S is **Universe**(S). It returns the common parent magma of the elements of S. The parent of S itself is obtained from the function **Parent** as usual – its return value is the set of aggregates (in the category of S) over the universe.

It is possible to state the universe U explicitly when defining an iterable homogeneous aggregate S. Immediately to the right of the opening bracketing symbol for S, the user should place an expression evaluating to U, and then follow it with a | symbol. For example, a set of polynomials could be constructed in this way:

```
> P<x> := PolynomialRing(IntegerRing());
> pols := { P | x^6 - 5*x^3, 2, 8*x^4 };
> print Universe(pols);
Univariate Polynomial Algebra in x over Integer Ring
```

However, it is often unnecessary to state the universe, since MAGMA is able to deduce the universe automatically from the elements; this is possible for the example above, and also occurred in all the previous examples:

```
> pols2 := { x^6 - 5*x^3, 2, 8*x^4 };
> print pols2 eq pols;
true
> print Universe(pols2) eq P;
true
```

There are two cases in which the user should explicitly declare the universe of a set or sequence under construction. Firstly, if the required universe U differs from the universe MAGMA would calculate of its own accord, then U must be given explicitly. For example, if the user is currently creating a set of integers but intends to include rational numbers in the set later, then the initial set should be created with the rational field as its universe:

```
> ratls := { 6, 3, 8, 2 };
> Include(~ratls, 7/9);

>> Include(~ratls, 7/9);
          ^
Runtime error in 'Include':
Rational argument is not a whole integer
> ratls := { RationalField() | 6, 3, 8, 2 };
> Include(~ratls, 7/9);
> print ratls;
{ 7/9, 2, 3, 6, 8 }
```

This case includes situations in which the user wants to restrict the universe to a subset of the universe that MAGMA would automatically calculate. This subset should be given as the universe, so as to guard against illegal values being included in the set later. For instance, if the sequence *binary* initially contains 1,1,1,0,0,1,0,1 and will never contain any term which is not 0 or 1, then it should be constructed in this way:

```
> binary := [ {0, 1} | 1, 1, 1, 0, 0, 1, 0, 1 ];
```

The second case in which the universe U should be declared explicitly is that in which the elements of U are specified *literally*, using sequences of primitive components (see p. 52). If U is stated on the left, then it is sufficient to describe the elements using these sequences alone, without coercing each of them into U separately. For example:

```
> V := VectorSpace(GF(3), 4);
> vecs := { V ! [2,1,0,2], V ! [0,0,1,0], V ! [1,2,0,1] };
> // or
> vecs := { V | [2,1,0,2], [0,0,1,0], [1,2,0,1] };
```

It is possible to alter the universe of a set (enumerated, multi- or indexed) or enumerated sequence after it has been constructed. Consider the following set of integers:

```
> zq := { -5, 0, 8, 2 };
> print Universe(zq);
Integer Ring
```

The statements below create a set *zqR* containing the same elements but whose universe is the rationals instead of the integers:

```
> Q := RationalField();
> zqR := ChangeUniverse(zq, Q);
```

```
> print zqR eq zq;
true
> print Universe(zqR);
Rational Field
```

The set zqR has the same elements as zq, but a different universe. The function used was **ChangeUniverse**(S, U). Given a set (enumerated, multi- or indexed) or enumerated sequence S it returns the aggregate structure (of the same category as S) with the same elements as S, but having the universe U. There is also a procedure version **ChangeUniverse**$(\tilde{}S, U)$, which changes the universe of S itself. For example:

```
> ChangeUniverse(~zq, Q);
> print Universe(zq);
Rational Field
```

The universe of zq has now been changed to the rational field.

6.2.3 Empty and Null Sets and Sequences

When an empty aggregate is used in computing, the implication is generally that it is a structure containing some particular type of object but that the number of objects in it happens to be zero. In MAGMA, an iterable homogeneous aggregate of this sort is defined by using bracketing symbols and declaring the universe, but not listing any objects. For example, an empty enumerated set of integers may be created as follows:

```
> emptyZ := { IntegerRing() | };
```

MAGMA follows common usage and calls such structures *empty*. When a set or sequence becomes empty as the result of modifications, MAGMA keeps its universe defined and deems it to be empty in the MAGMA sense. The only elements that can be put into an empty set or sequence are those that can be made to fit the universe of that set or sequence. This prevents the user from including the wrong kind of element in the set; for example, putting a permutation group element into an empty set intended for vectors.

The term MAGMA uses for an iterable homogeneous aggregate that is not only empty but has no universe is *null*. It is created with bracketing symbols, without specifiying the universe or any elements. Any kind of element can be put into a null structure, but after that the structure will have a universe, like every set or sequence that contains elements.

The Boolean functions testing whether a set or sequence S is empty or null are **IsEmpty**(S) and **IsNull**(S) respectively. Note that a null set or

sequence will return **true** when passed as an argument to **IsEmpty**, because it has no elements.

6.2.4 Construction of Arithmetic Progressions

Table 6.2. Enumerated sets and sequences from arithmetic progressions

MAGMA	Meaning
$\{i..j\}$	Set $\{i, i+1, \ldots, j\}$, where $i, j \in \mathbb{Z}$
$[i..j]$	Sequence $[i, i+1, \ldots, j]$, where $i, j \in \mathbb{Z}$
$\{i..j$ **by** $k\}$	Set $\{i, i+k, i+2k, \ldots, j\}$, where $i, j, k \in \mathbb{Z}, k \neq 0$
$[i..j$ **by** $k]$	Sequence $[i, i+k, i+2k, \ldots, j]$, where $i, j, k \in \mathbb{Z}, k \neq 0$

Table 6.2 shows MAGMA's special methods for constructing an enumerated set or enumerated sequence S whose elements are integers in arithmetic progression. If the progression has common difference 1, then the syntax $i..j$ inside the bracketing symbol means the elements $i, i+1, i+2, \ldots, j$. For example:

```
> TwentyToForty := {20..40};
> print TwentyToForty;
{ 20 .. 40 }
```

As the output implies, MAGMA stores the arithmetic progression rather than computing and storing all the elements. This is much more efficient if the cardinality of S is large, and it is still possible during subsequent computations to find elements or test whether an object is in S very quickly.

If the arithmetic progression has a common difference k other than 1, the appropriate syntax is $i..j$ **by** k within bracketing symbols, meaning the progression $i, i+k, i+2k, \ldots, j$. If the progression is decreasing, then k should be negative. (An empty structure results if $i > j$ and $k > 0$ or if $i < j$ and $k < 0$. If j is not of the form $i + mk$ for some integer m, then the progression continues as long as the range from i to j is not exceeded; that is, if $k > 0$ the last element in the progression will be the greatest integer of the form $i + nk$ that is less than j, and if $k < 0$ the last element will be the least integer of the form $i + nk$ that is greater than j.) For example:

```
> skip3 := [4..19 by 3]; seq2 := [87..43 by -2];
> print skip3, seq2;
[ 4 .. 19 by 3 ]
```

```
[ 87 .. 43 by -2 ]

> skip3a := [4..21 by 3]; seq2a := [87..42 by -2];
> print skip3 eq skip3a and seq2 eq seq2a;
true
```

In an enumerated sequence created as an arithmetic progression, the first element is i, and the order of the elements proceeds along the progression. In an enumerated set, there is no true concept of order in the elements; in practice, however, they are stored as increasing progressions (so that the direction is changed if the common difference is negative), and iteration over such a set will occur in increasing order. The following example illustrates this in the case of a **for**-statement (see Chapter 9):

```
> s := { 9..3 by -2 };
> print s;
{ 3 .. 9 by 2 }
> for x in s do
>     print x;
> end for;
3
5
7
9
```

The arithmetic progression constructors are available only for enumerated sets and enumerated sequences, not indexed sets or multisets. In the rare situation that an arithmetic progression has to be stored in one of these structures, a suitable transfer function (Table 6.9 (p. 146)) should be used:

```
> print SetToMultiset(s);
{* 3, 5, 7, 9 *}
> xs := SetToIndexedSet({4..8}); print xs;
{@ 4, 5, 6, 7, 8 @}
```

The universe of the result of an arithmetic progression constructor is always the ring of integers, but aggregates with other universes may be constructed from them, using the element-description method to be explained below. For instance, the following expression returns the arithmetic sequence of rational numbers ranging from zero to one and incrementing by tenths:

```
> print [x/10: x in [0..10]];
[ 0, 1/10, 1/5, 3/10, 2/5, 1/2, 3/5, 7/10, 4/5, 9/10, 1 ]
```

6.2.5 Construction from an Element-Description

Another method of constructing an enumerated set, multiset, indexed set or enumerated sequence is to give an *element-description*, in a notation which is remarkably close to that of conventional mathematics. MAGMA calculates all the elements of the aggregate from this description. The element-description is made in terms of a general element, which is an expression in one or more free identifiers x_i. These free identifiers have local scope, and act as 'dummy variables'. Each identifier successively takes on the values of some finite domain structure E_i, and certain combinations of the values of the free identifiers are substituted into the expression for the general element. A domain E_i may be any iterable structure, that is, any iterable aggregate or any finite magma such that MAGMA can iterate over its elements (e.g., a finite field or a linear code).

As a preliminary example, the following line calculates the set consisting of the squares of the first five positive integers:

```
> squares := { n^2 : n in [1..5] };
> print squares;
{ 1, 4, 9, 16, 25 }
```

The expression for the general element is `n^2`, and the free identifier n ranges over the values 1,2,3,4,5, which are the elements of the domain structure `[1..5]`. Since free identifiers have local scope, their use in a constructor will not interfere with an identifier of the same name outside the constructor. For instance:

```
> n := 91837410923;
> squares := { n^2 : n in [1..5] };
> print n;
91837410923
```

The element-description aggregate constructor is enclosed in the appropriate bracketing symbol for the relevant category of aggregate structure (enumerated set, multiset, indexed set or enumerated sequence). Immediately after the opening bracket is an optional specification of the universe, consisting of an expression for the universe followed by the | symbol; however, it is less common to state the universe explicitly in this kind of construction than it is for element-list constructions.

The *body of the constructor* specifies the free identifiers and domains, an optional Boolean condition indicating which elements of the domains are to be used (all fo them being used if no condition is given), and an expression in the free identifiers that is evaluated to find the elements of the aggregate.

For simplicity, the body of the constructor will first be explained in the case of a single free identifier x and domain E. It can take two forms, the simpler of these being:

expression : x **in** E

Here the expression is in the free identifier x. The elements of the resulting aggregate are the values of the expression in x, evaluated for all $x \in E$. The more general syntax is:

expression : x **in** E | condition

Here the condition is also an expression in the free identifier x, but it must evaluate to a Boolean. The elements of the aggregate are the values of the expression in x, evaluated for all $x \in E$ such that the condition evaluates to **true**.

If there are several free identifiers x_i ranging over domains E_i, then the expression on the left and the optional condition on the right should be expressions in the x_i. The syntax remains the same except for the section between the : and | symbols, which becomes:

x_1 **in** E_1, ..., x_k **in** E_k

If two or more adjacent domains are equal ($E_s = E_{s+1} = \cdots = E_t$), then the abbreviation $x_s, \ldots, x_t$ **in** E_s may be made for x_s **in** E_s, ..., x_t **in** E_t. In order to evaluate the element-description when there are several free identifiers, MAGMA considers all the k-tuples $< x_1, \ldots, x_k >$ where $x_i \in E_i$ for all i. If there is no condition on the right, MAGMA evaluates the expression for each k-tuple, to yield the elements of the aggregate under construction. If there is a condition given, MAGMA substitutes each k-tuple into the condition, and for all those k-tuples such that the condition evaluates to **true**, it evaluates the expression to find the elements of the aggregate.

For example, let *mu* be the set of five times all the units in the residue class ring $\mathbb{Z}/12\mathbb{Z}$. This set might be described more formally as 'the set of five times y, where y is in the ring $\mathbb{Z}/12\mathbb{Z}$, such that y is a unit', or as $\{5y : y \in \mathbb{Z}/12\mathbb{Z} \mid y \text{ is a unit}\}$. The MAGMA syntax is similar:

```
> mu := { 5*y : y in ResidueClassRing(12) | IsUnit(y) };
> print mu;
{ 11, 1, 5, 7 }
> print Universe(mu);
Residue class ring of integers modulo 12
```

Observe that an automatic coercion takes place in the expression `5*y`: the integer 5 is coerced into the ring $\mathbb{Z}/12\mathbb{Z}$, to which y belongs.

As another example, the following line evaluates the expression x^3-7x^2+2 for $x = -2, -1, 0, 1, 2, 3, 4$, and places the result in an enumerated sequence:

```
> vals := [ x^3 - 7*x^2 + 2: x in [-2..4] ];
> print vals;
[ -34, -6, 2, -4, -18, -34, -46 ]
> print Universe(vals);
Integer Ring
```

The universe of *vals* as created above is the integer ring, because the values calculated are integers. If the user wishes the sequence to have a different universe, such as the real field, then this universe should be placed on the left of the constructor:

```
> vals := [ RealField() | x^3 - 7*x^2 + 2: x in [-2..4] ];
> print Universe(vals);
Real Field
```

Note that if a universe is specified but there is an element that cannot be coerced into the universe, then MAGMA gives an error message. For example:

```
> seqC := [ Sqrt(t) : t in [5, 2, -4] ];
> print seqC;
[ 2.23606797749978969640917366872 2,
1.41421356237309504880168872419 8,
2.00000000000000000000000000000*i ]
> print Universe(seqC);
Complex Field
>
> seqC := [ RealField() | Sqrt(t) : t in [5, 2, -4] ];

>> seqC := [ RealField() | Sqrt(t) : t in [5, 2, -4] ];
           ^
Runtime error in sequence construction:
Can't coerce element into the universe
```

It occasionally happens in an element-description constructor that the condition and the expression used to calculate the elements contain a common sub-expression. There is a *special kind of* **where**-*construction* provided for such cases, to increase the efficiency and compactness of the code. The **where**-construction should be placed immediately after the condition, but its scope will extend to the expression for the elements as well. For example,

the set constructor below finds the values of n in the range 2 to 100 such that $\phi(n) \leq 6$, and also stores these values of $\phi(n)$ with the corresponding value of n. By employing the **where**-construction, the user can avoid recalculating $\phi(n)$ for the successful values of n:

```
> phi_le_6 := {<n, phi>: n in {2..100} |
>   phi le 6 where phi is EulerPhi(n)};
> print phi_le_6;
{ <5, 4>, <7, 6>, <10, 4>, <18, 6>, <2, 1>, <6, 2>,
<8, 4>, <4, 2>, <9, 6>, <12, 4>, <14, 6>, <3, 2> }
```

Note that each element of this sequence is a *tuple*, a kind of aggregate which is discussed in Section 6.10.

The next two examples are of constructors involving more than one free identifier. Firstly, the following **print**-statement prints the set of all multiples of the basis elements of the 4-dimensional vector space over the finite field with 3 elements:

```
> F := FiniteField(3);
> V := VectorSpace(F, 4);
> print { k * b : b in Basis(V), k in F };
{
    (0 0 0 1),
    (0 0 0 2),
    (0 0 0 0),
    (1 0 0 0),
    (0 2 0 0),
    (0 1 0 0),
    (0 0 1 0),
    (2 0 0 0),
    (0 0 2 0)
}
```

If some of the domains are equal, then it is possible to abbreviate the constructor. For instance, consider the problem of finding the set of elements of the form $4K$ where $K + 1$ is a square and $K = s^2 + t^3$ where s and t range over the finite field with 83 elements. The solution is:

```
> fs := { 4*K : s, t in FiniteField(83) | IsSquare(K+1)
>   where K is s^2 + t^3 };
> print fs;
{ 0, 44, 45, 3, 47, 5, 6, 7, 8, 12, 55, 13, 57, 59, 17, 60,
61, 19, 64, 21, 65, 22, 66, 23, 24, 25, 26, 27, 71, 29, 73,
74, 32, 33, 34, 77, 36, 79, 37, 80, 82, 40 }
```

If there are several free identifiers, the domain E_i may depend on any of the free identifiers to its right (i.e., $x_{i+1}, \ldots, x_k$). For example, suppose that the user wishes to construct the set of all the unordered triples (with repetitions permitted) of integers from 0 to 20 that sum to 20. Since each triple is unordered and may have repeated elements, a multiset is the best way to represent it. Two solutions are presented below: an intuitive approach, and a method that is more efficient because no element is generated more than once. The output from the **time** command shows the difference:

```
> time tr20A := { {*a, b, 20-a-b*} :
>   a in [0..20-b], b in [0..20] };
Time: 0.100
> time tr20B := { {*a, b, 20-a-b*} :
>   a in [b..((20-b) div 2)], b in [0..20]};
Time: 0.020
> print tr20A eq tr20B;
true
> print tr20B;
{
    {* 11, 3, 6 *}, {* 1, 3, 16 *}, {* 2, 14, 4 *},
    {* 4, 7, 9 *}, {* 12, 2, 6 *}, {* 1, 9, 10 *},
    {* 1, 4, 15 *}, {* 2, 8, 10 *}, {* 0, 3, 17 *},
    {* 0, 13, 7 *}, {* 2, 13, 5 *}, {* 4, 6, 10 *},
    {* 6, 7^^2 *}, {* 11, 1, 8 *}, {* 0, 14, 6 *},
    {* 2, 3, 15 *}, {* 0, 12, 8 *}, {* 6^^2, 8 *},
    {* 3, 8, 9 *}, {* 1, 12, 7 *}, {* 0^^2, 20 *},
    {* 0, 15, 5 *}, {* 1, 13, 6 *}, {* 0, 2, 18 *},
    {* 11, 2, 7 *}, {* 12, 4^^2 *}, {* 0, 1, 19 *},
    {* 5, 6, 9 *}, {* 3^^2, 14 *}, {* 1, 14, 5 *},
    {* 11, 4, 5 *}, {* 0, 4, 16 *}, {* 1^^2, 18 *},
    {* 0, 11, 9 *}, {* 5^^2, 10 *}, {* 2^^2, 16 *},
    {* 3, 7, 10 *}, {* 2, 9^^2 *}, {* 13, 3, 4 *},
    {* 1, 2, 17 *}, {* 12, 3, 5 *}, {* 4, 8^^2 *},
    {* 5, 7, 8 *}, {* 0, 10^^2 *}
}
```

Note, however, that it is often preferable to use a set constructor slightly inefficiently, with some needless regeneration of elements (as for the first solution to this problem), if the user's own time is being saved by describing the objects in a straightforward, naive way.

The ordering of the elements of an enumerated sequence or indexed set that is constructed from an element description corresponds to the order in which they are generated by the constructor. This order, in turn, depends on the order in which the domain structures E_i produce their elements as

values for free identifiers in the iteration process. For each domain, if it is an enumerated sequence or indexed set, then its elements are produced in the obvious order, from first to last; otherwise, the order of iteration depends on the way MAGMA represents the domain internally, and it is unwise to make assumptions about it. Considering all the domains $E_1, \ldots, E_k$, the values from the domains on the right of the list vary more frequently than those on the left, as the following example shows:

```
> print [<a,b,c> : a in [1..2], b in [3..4], c in [5..6]];
[ <1, 3, 5>, <2, 3, 5>, <1, 4, 5>, <2, 4, 5>,
<1, 3, 6>, <2, 3, 6>, <1, 4, 6>, <2, 4, 6> ]
```

6.2.6 Recursive Definition of Enumerated Sequences

Enumerated sequences may be created in a manner imitating the recursive or inductive definitions encountered in mathematics. Within the constructor, the function **Self**(i) returns the value of the i^{th} term of the sequence currently under construction; this enables terms which have already been constructed to be used in the calculation of new terms. The function **Self**() returns the current value of the sequence itself.

For example, the Fibonacci numbers may be defined as $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n > 2$. In MAGMA the corresponding sequence definition for the first 50 Fibonacci numbers is:

```
> F := [n in {1, 2} select 1 else Self(n-1) + Self(n-2) :
>   n in [1..50]];
```

Of course, in this particular example a recursive definition is not necessary, because MAGMA has a suitable intrinsic function **Fibonacci**(n):

```
> F := [Fibonacci(n): n in [1..50]];
```

The following sequence provides a trivial example of the **Self**() function:

```
> print [Self(): n in [1..3]];
[
    [],
    [
        []
    ],
    [
        [],
        [
```

```
                []
            ]
        ]
]
```

6.2.7 Enumerated Sequences with Undefined Terms

It is possible for MAGMA sequences to have undefined terms. Such sequences are called *incomplete*. Their main use is for building a sequence gradually, out of index order. For example, suppose that the user wishes to create a sequence *q3* by starting with an empty integer sequence and then defining its 7^{th} and 5^{th} terms first:

```
> q3 := [IntegerRing() | ];
> q3[7] := 45;
> q3[5] := 69;
> print q3;
[ undef, undef, undef, undef, 69, undef, 45 ]
```

Each of the undefined terms is represented in the output by the keyword **undef**. This keyword is only used in output; it cannot be included in an expression.

The Boolean function **IsDefined**(Q, i) returns **true** if and only if the i^{th} term of the sequence Q is defined, and the procedure **Undefine**$(\tilde{}Q, i)$ causes the i^{th} term of Q to become undefined.

Notice that MAGMA's incomplete sequences permit a sequence Q to be constructed without being initialized to a fixed length beforehand. The current length of the sequence, $\#Q$, is simply the index of the furthermost defined term, and can become smaller or greater as terms are defined or undefined. The only initialization necessary is to assign to the identifier a sequence with no terms. It may be defined as a null sequence rather than an empty sequence if preferred, provided that when the first term is defined MAGMA is able to compute from it the correct universe for the sequence.

Most functions that take sequences as their arguments require that the sequences be complete. The function **IsComplete**(Q) tests if this is so. The best way to make an incomplete sequence complete is to iterate over it, since MAGMA ignores undefined terms whenever iterating over a sequence. For example, the following statement assigns to *q4* a complete sequence containing the defined terms of *q3* in the same order:

```
> q4 := [n: n in q3];
> print q4;
```

```
[ 69, 45 ]
```

6.3 Operations on Iterable Homogeneous Aggregates

Several operations apply to more than one of the iterable homogeneous aggregate categories (enumerated sets, multisets, indexed sets, and enumerated sequences). These operations will be explained in this section.

6.3.1 Index Position, Cardinality and Length

It was mentioned above that if S is an indexed aggregate (indexed set or enumerated sequence), then $S[i]$ returns the element of S at the i^{th} index position. (For sequences but not indexed sets, $S[i]$ may also be used on the left side of an assignment statement, in order to change the i^{th} term of S.) Conversely, **Position**(S, x) or **Index**(S, x) returns the index of the element x in S, or zero if x is not an element of S. If S is a sequence and x occurs more than once in S, the function returns the smallest such index, that is, the position of the leftmost occurrence of x in S. For example:

```
> ii := {@ 8, 6, 6, 6, 6, 4, 6, 7, 2 @};
> print ii;
{@ 8, 6, 4, 7, 2 @}
> print ii[4];
7
> print Position(ii, 7);
4

> qq := [ 8, 6, 6, 6, 6, 4, 6, 7, 2 ];
> print qq;
[ 8, 6, 6, 6, 6, 4, 6, 7, 2 ]
> print Position(qq, 7);
8
> print Position(qq, 6);
2
```

The operator # is used throughout the MAGMA system to extract the cardinality of a magma S. If S is an object with n elements, where n is finite and can be calculated by MAGMA, then $\#S$ returns n. In the context of aggregate structures, the cardinality operator requires careful interpretation. For an enumerated set or indexed set S, in which all the elements are distinct, $\#S$ returns the number of these elements. For a multiset or an enumerated

sequence S, in which there may be more than one occurrence of the same element in S, $\#S$ returns the total number of elements, counting multiplicities. (However, for an enumerated sequence with undefined terms, it returns the highest position at which there is a defined term.) Notice that from the point of view of the indexed aggregates, this definition implies that $\#S$ is the *length* or highest index of S; the same applies to $\#S$ where S is a string. Therefore $S[\#S]$ is the 'last' or rightmost element of S. For example:

```
> m := {* wd[i]: i in [1..#wd] *} where wd is "MISSISSIPPI";
> print m;
{* M, I^^4, P^^2, S^^4 *}
> print #m;
11
```

6.3.2 Testing Equality, Membership, and Subsetness

The relational operators **eq** ($=$) and **ne** ($\neq$) may be applied to two aggregates R and S in the same category to test their equality/inequality. The universes of R and S must be *compatible*, that is, they must either be equal or have a common overstructure into which elements of each aggregate can be automatically coerced. In order for R **eq** S to return **true**, the elements of R and S must be equal, counting multiplicities where that is relevant. For instance, the following two sets are regarded as equal by MAGMA, because their elements are exactly equal after the integers in the set on the right are coerced into the finite field:

```
> print {FiniteField(7) | n: n in {2..5}} eq {2..5};
true
```

If R and S are equal then the order of their elements must be the same for enumerated sequences, but need not be the same for indexed sets. The reason for this is that order is an important aspect of the sequence concept, whereas indexed sets may be seen as modified enumerated sets with a convenient indexing facility.

The expressions x **in** S and x **notin** S are used to test whether a given object x is an element of ($\in$) an aggregate S (or a magma S). If the parent of x does not equal the universe of S, then MAGMA attempts auto-coercion of x into the universe before performing the membership test.

The expressions R **subset** S and R **notsubset** S compare two aggregates (or magmas) R and S to see whether R is a subset ($\subseteq$) of S, that is, whether every element of R is an element of S. Again, auto-coercion is attempted if the universes of R and S are not the same. The multiplicity of the elements

is not relevant unless R and S are multisets, and the order of the elements is never relevant. (If the two aggregates are both sequences, the function **IsSubsequence**, explained on p. 122, performs a test that is similar but does consider the order of the elements.)

6.3.3 Including or Excluding an Element

The intrinsics **Include** and **Exclude** are used to place an element x in an aggregate S or take it from S. The parent of x must be compatible with the universe of S. There are two versions: the functions **Include**(S, x) and **Exclude**(S, x), which return a new aggregate but do not change the given aggregate S, and the procedures **Include**$(\tilde{}S, x)$ and **Exclude**$(\tilde{}S, x)$, which modify S itself, as a reference argument. If the user wishes to change the value of an existing aggregate, stored in an identifier, then the procedure version should be used; note that a procedure call is a statement by itself, and must be followed by a semicolon. (The ~ symbol indicates that the identifier it precedes is a *variable reference argument*, that is, an identifier whose value the procedure may change. See Section 8.3 for further discussion.)

The precise effect of these intrinsics depends on the category of S. If S is an enumerated set then: the enumerated set computed by **Include** is S with x placed in it if x is not in S already, else S; the enumerated set computed by **Exclude** is S with x removed if x is in S, else S. If S is an indexed set then: the indexed set computed by **Include** is S with x placed at the end (in the index order) if x is not in S already, else S; the **Exclude** intrinsic does not exist for this category. If S is a multiset then: the multiset computed by **Include** is S with the multiplicity of x increased by 1, whether or not S initially contains x; the multiset computed by **Exclude** is S with the multiplicity of x decreased by 1, if x is in S, else S. Finally, if S is an enumerated sequence then: the enumerated sequence computed by **Include** is S with x placed at the end (in the index order) if x is not in S already, else S; the enumerated sequence computed by **Exclude** is S with the leftmost instance of x removed (and the terms to the right of this moved one place to the left) if x is in S, else S.

6.3.4 Maximum and Minimum Elements

The functions **Maximum**(S) and **Minimum**(S), which have abbreviations **Max**(S) and **Min**(S), may be applied to an aggregate S whose universe has a total ordering defined on its elements. (Examples are ordered rings, free groups, semigroups and monoids, and the string monoid.) These functions return the maximum or minimum element of S. If S is an enumerated sequence or an indexed set, the functions have a second return value, namely

the index of the (leftmost) position where the maximal or minimal element occurs. A multiple assignment statement should be used to obtain both values. (Another way to obtain both values is to invoke the function by itself as the expression in a **print**-statement.)

6.3.5 Reduction

The operation of *reduction* takes an aggregate S and returns an object with the same parent as the elements of S have. Assuming that $\circ$ is an associative binary operator suitable for the universe of S, the expression `&` $\circ S$ returns the value of $S_1 \circ S_2 \circ \cdots \circ S_n$, where the S_i are the elements of S, as described on p. 98. Note that the S_i may include repetitions, in the case of multisets and enumerated sequences. If S is an enumerated sequence or indexed set, then the elements are taken in standard index order.

For example, the following statement evaluates $\sum_{i=1}^{10}(i!)^2$, by performing reduction using the addition operator on the sequence containing each $(i!)^2$ for $i = 1, \ldots, 10$:

```
> print &+[Factorial(i)^2: i in [1..10]];
13301522971817
```

Similarly, consider the problem of finding the order of $\mathrm{GL}(n, \mathrm{GF}(q))$, that is, the number of non-singular $n \times n$ matrices over $\mathrm{GF}(q)$, where q is a prime power. It equals

$$\prod_{i=1}^{n}(q^n - q^{i-1})$$

so the MAGMA equivalent (assuming q and n have been assigned) is:

```
> ord := &*[q^n - q^(i-1) : i in [1..n]];
```

Finally, the following function, *isperfect*(n), applies reduction and sequence operations in order to test whether the integer argument n is a perfect number (i.e., whether n equals the sum of its divisors except for n itself):

```
> isperfect := func< n | n eq &+Exclude(Divisors(n), n) >;
> print isperfect(6);
true
> print isperfect(7);
false
> print isperfect(28);
true
```

Any of the operators +, *, **and**, **or**, **join** and **meet** may be used to reduce an aggregate provided that the operator is suitable for the universe. For sequences containing sequences or strings, the associative but non-commutative concatenation operator **cat** is also available.

Aggregates with a cardinality of zero or one need special attention. When reduction is applied to an aggregate with cardinality one, the value returned is the unique element of the aggregate. The special cases of reducing aggregates with no elements follow mathematical conventions for the operators. Summing or multiplying an empty aggregate returns 0 or 1 respectively, if the universe is suitable. On an empty aggregate of Booleans, or on a null aggregate, **&and** returns **true** and **&or** returns **false**.

6.4 Further Operations on Set Categories

Besides the operations given in Section 6.3, there are further operations available for the categories of enumerated sets, multisets and indexed sets.

The binary operations of union and intersection may be applied to two aggregates R and S in the same set category, provided that their universes are compatible. The aggregate returned will have the same category as the category of the arguments. The expression R **join** S returns the aggregate $R \cup S$, and R **meet** S returns $R \cap S$. If R and S are indexed sets, no generalizations can be made about the indexing of the result. If R and S are multisets, R **join** S is constructed by adding multiplicities, and R **meet** S is constructed by taking the minimum of the multiplicities for each element. For example:

```
> print { 3..9 by 2 } join { 1, 5, 11, 17 };
{ 1, 3, 5, 7, 9, 11, 17 }
> print {* 1^^4, 3^^6, 8^^2 *} join {* 1, 2^^3, 8^^5 *};
{* 1^^5, 2^^3, 3^^6, 8^^7 *}
> print {* 1^^4, 3^^6, 8^^2 *} meet {* 1, 2^^3, 8^^5 *};
{* 1, 8^^2 *}
```

There are two other binary set operators that are available only for enumerated sets and indexed sets, not for multisets. The expression R **diff** S returns the difference of R and S, that is, $\{x : x \in R \text{ and } x \notin S\}$, and the expression R **sdiff** S returns the symmetric difference of R and S, that is, $\{x : x \in R \cup S \text{ and } x \notin R \cap S\}$. For example:

```
> print { 3..9 by 2 } diff { 1, 5, 11, 17 };
{ 3, 7, 9 }
> print { 3..9 by 2 } sdiff { 1, 5, 11, 17 };
{ 1, 3, 7, 9, 11, 17 }
```

These four set operators may also be used in mutation assignments. For example:

```
> FA3<a,b,c> := FreeAbelianGroup(3);
> ix := {@ a, b+c, 2*a-7*c @};
> ix join:= {@ 5*b, c @};
> print ix;
{@
    a,
    b + c,
    2*a - 7*c,
    5*b,
    c
@}
```

For enumerated sets only, the procedure **ExtractRep**($\tilde{}R, \tilde{}x$) is provided. Given an enumerated set R and an identifier x, it modifies R by extracting a representative element from it, and it assigns this element to x. For example:

```
> J := {15, 8, 13, 42, 20};
> ExtractRep(~J, ~r);
> print J, r;
{ 8, 15, 20, 42 }
13
```

This procedure should not be confused with the function **Rep**(M), which returns a representative of the magma or aggregate M but does not change M itself.

For enumerated sets only, there are several functions provided that construct subsets, sub-multisets, permutations, and so on. See Section 35.2 for a full list. Perhaps the most generally applicable of these functions is **Subsets**(S, k). Given an enumerated set S and an integer k, it returns the set of all k-subsets of S, that is, all subsets of S having cardinality k. If $k < 0$ or $k > \#S$, the value returned is an empty set. For example:

```
> print Subsets({15, 8, 13, 42, 20}, 2);
{
    { 15, 42 },
    { 8, 15 },
    { 13, 42 },
    { 8, 20 },
    { 15, 20 },
    { 13, 15 },
```

```
      { 8, 13 },
      { 13, 20 },
      { 20, 42 },
      { 8, 42 }
  }
```

6.5 Further Operations on Enumerated Sequences

The operations in Section 6.3 apply to all the iterable homogeneous categories of aggregates, including enumerated sequences. This section discusses operations that apply to enumerated sequences only; see Table 6.3 and Table 6.4 for summaries. The operations in the first table are functions or operators that return values; those in the second table *mutate* (change) the arguments, in a manner corresponding to the non-mutation versions. For efficiency, the mutation operations should be employed where appropriate in preference to the functional versions.

Table 6.3. Enumerated sequence operations returning values

MAGMA	Meaning
Q **cat** T	Sequence formed by chaining sequence T to end of sequence Q: $[q_1, \ldots, q_n, t_1, \ldots, t_m]$
Position(Q, x), **Index**(Q, x)	Position of first occurrence of x in sequence Q, or zero if no term in Q is x
$Q[i]$	i^{th} term of sequence Q
$Q[I]$	$[q_{i_1}, \ldots, q_{i_n}]$, where I is integer sequence $[i_1, \ldots, i_n]$
$Q[i_1, i_2, \ldots, i_k]$	$Q[i_1][i_2] \ldots [i_k]$ (multi-indexing)
Explode(Q)	The terms of Q, in index order
Append(Q, x)	Q with x placed on end
Prune(Q)	Q with final term removed
Include(Q, x)	Q with x placed on end, if x not in Q already
Exclude(Q, x)	Q with first occurrence of x in Q deleted, if x in Q
Insert(Q, i, x)	Q with x inserted at position i, and following elements moved along one place: $[q_1, \ldots, q_{i-1}, x, q_i, \ldots, q_n]$
Remove(Q, i)	Q with i^{th} term removed: $[q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n]$
Insert(Q, k, m, T)	Q with sequence T inserted, replacing terms q_k through to q_m inclusive: $[q_1, \ldots, q_{k-1}, t_1, \ldots, t_m, q_{m+1}, \ldots, q_n]$
Reverse(Q)	Q with order of terms reversed
Rotate(Q, p)	Q with terms rotated cyclically p terms to the right, or $-p$ terms to the left if p negative
Sort(Q)	Q with terms sorted into increasing order

Table 6.4. Enumerated sequence mutation operations

$Q[i] := x$	Q **cat** $:= S$	
Append$(\tilde{}Q, x)$	**Prune**$(\tilde{}Q)$	
Include$(\tilde{}Q, x)$	**Exclude**$(\tilde{}Q, x)$	
Insert$(\tilde{}Q, i, x)$	**Remove**$(\tilde{}Q, i)$	**Insert**$(\tilde{}Q, k, m, T)$
Reverse$(\tilde{}Q)$	**Rotate**$(\tilde{}Q, p)$	**Sort**$(\tilde{}Q)$

6.5.1 Changing a Term of a Sequence

Let Q be an enumerated sequence. It has already been seen that the expression $Q[i]$ returns the i^{th} term of Q, and that **Position**(Q, x) or **Index**(Q, x) returns the index of the leftmost occurrence of x in Q (or 0 if x is not in Q). It is possible to change Q at the i^{th} term, by assigning a value to $Q[i]$. For example:

```
> fr := [x/10: x in [0..10]];
> fr[6] := 99997;
> print fr;
[ 0, 1/10, 1/5, 3/10, 2/5, 99997, 3/5, 7/10, 4/5, 9/10, 1 ]
```

6.5.2 Obtaining Several Terms of a Sequence

The function **Explode**(Q) is a multiple return value function, operating on a sequence Q. Its return values are the terms $Q[1], Q[2], \ldots, Q[\#Q]$, in that order. Thus the number of return values is dependent on the length of the argument. **Explode** is designed for situations where the user wishes to assign some or all of a sequence's terms to identifiers, without employing several assignment statements.

Two examples of this function are given below. In the first example, all the terms of the sequence $[8, 7, 6]$ are assigned to identifiers. In the second example, only some of the terms of the sequence *fr* created above are assigned; see Section 3.3.3 for details of the multiple assignment statement and the _ character as they are used here.

```
> z1, z2, z3 := Explode([8, 7, 6]);
> print z1;
8
> print z2;
7
> print z3;
6
```

```
> first, second, _, _, fifth := Explode(fr);
> print first;
0
> print second;
1/10
> print fifth;
2/5
```

6.5.3 Subsequences and Concatenation

It is possible to form a new sequence by selecting several of the terms of a sequence Q. The syntax for this operation is $Q[I]$, where I is an integer sequence containing the positions of the required terms. In the special case that I is an arithmetic progression, the abbreviation $Q[i..j$ **by** $k]$ may be made for the full form $Q[[i..j$ **by** $k]]$, and similarly $Q[i..j]$ is an abbreviation for $Q[[i..j]]$. For example:

```
> print fr[[1, 3, 7, 9]];
[ 0, 1/5, 3/5, 4/5 ]
> print fr[2..8 by 3];
[ 1/10, 2/5, 7/10 ]
```

However, $Q[I]$ may not be placed on the left of an assignment statement in order to reassign several terms of a sequence simultaneously.

The expression Q **cat** T returns the *concatenation* of the enumerated sequences Q and T, that is, the sequence $[Q[1], \ldots, Q[\#Q], T[1], \ldots, T[\#T]]$. For example:

```
> q1 := [4, 8/3, 7/10, 33, 2, 1/2];
> q2 := q1 cat [-3/7, 8/3, 70, -5];
> print q2;
[ 4, 8/3, 7/10, 33, 19, 1/2, -3/7, 8/3, 70, -5 ]
```

If the user wishes to modify Q by concatenating T to it, then a mutation assignment with the operator **cat:=** should be used:

```
> qq := [5..9];
> qq cat:= [100..400 by 100];
> print qq;
[ 5, 6, 7, 8, 9, 100, 200, 300, 400 ]
```

Given enumerated sequences Q and T, the function **IsSubsequence**(Q, T) returns **true** if Q is a subsequence of T. This function has three ways of in-

terpreting 'subsequence', depending on the value of the parameter **Kind**. (For an explanation of parameters, see Section 8.10.) If **Kind** is assigned the default value `"Consecutive"`, or omitted entirely, then the function tests whether the terms of Q occur consecutively in T. For example:

```
> print IsSubsequence(q1, q2);
true
> print IsSubsequence([9,6,3], [8,4,2,9,6,3,1]);
true
> print IsSubsequence([9,3,6], [8,4,2,9,6,3,1]);
false
```

However, if **Kind** is given the value `"Sequential"`, a weaker test is performed, which examines whether the terms of Q occur in T in the same order but not necessarily consecutively:

```
> print IsSubsequence([9,6,3], [3..10]: Kind:="Sequential");
false
> print IsSubsequence([9,6,3], [10..3 by -1]:
>   Kind := "Sequential");
true
```

If the parameter **Kind** is given the value `"Setwise"`, then the function tests whether all the elements in the first sequence are in the second sequence, regardless of the order in which they appear or the number of times each element appears. (This is the same as the test given by Q **subset** T.) For instance,

```
> print IsSubsequence([9,6,3], [3..10]: Kind := "Setwise");
true
```

6.5.4 Inserting and Removing Objects

Several intrinsics are provided for placing a suitable object in an enumerated sequence or removing an object from the sequence. Each of them has a function version, which returns a sequence but does not change the arguments, and a procedure version, which modifies the sequence given as a reference argument. Two of the intrinsics, **Include** and **Exclude**, were discussed in Section 6.3.3. The others are **Append**, **Prune**, **Insert**, and **Remove**.

The function **Append**(Q, x) or procedure **Append**$(\tilde{}Q, x)$ places the object x on the end of the sequence Q (i.e., following the element that was previously at the highest index position), whether or not Q currently contains x. For instance:

```
> Append(~q2, 99);
> print q2;
[ 4, 8/3, 7/10, 33, 2, 1/2, -3/7, 8/3, 70, -5, 99 ]
> Append(~q2, 70);
> print q2;
[ 4, 8/3, 7/10, 33, 2, 1/2, -3/7, 8/3, 70, -5, 99, 70 ]
```

The function **Prune**(Q) and the procedure **Prune**$(\tilde{}Q)$ are used to remove the last term from the sequence Q. For instance:

```
> print Prune(q2);
[ 4, 8/3, 7/10, 33, 2, 1/2, -3/7, 8/3, 70, -5, 99 ]
```

Another way of inserting and removing terms of a sequence is by specifying the sequence position which is to be affected. The relevant intrinsics are **Insert**(Q, i, x) or **Insert**$(\tilde{}Q, i, x)$, which inserts x at the i^{th} position of Q and moves along the rest of the sequence to make room, and **Remove**(Q, i) or **Remove**$(\tilde{}Q, i)$, which removes the i^{th} term from Q and moves back the rest of the sequence to fill the gap. For instance:

```
> q3 := Insert(q2, 4, 999);
> print q3;
[ 4, 8/3, 7/10, 999, 33, 2, 1/2, -3/7, 8/3, 70, -5, 99, 70 ]
> print Remove(q3, 4) eq q2;
true
```

There is also a function **Insert**(Q, k, m, T) or procedure **Insert**$(\tilde{}Q, k, m, T)$ which inserts a sequence T into Q, replacing terms $Q[k]$ through to $Q[m]$ inclusive. If the user does not wish to destroy any of the terms of Q, but merely to place T after the m^{th} term of Q and then continue with the rest of Q, then k should be set at $(m + 1)$.

6.5.5 Rearranging the Terms of a Sequence

There are three intrinsics that change the order of the terms of an enumerated sequence Q. Again, each has a function version and a procedure version. The enumerated sequence constructed by **Reverse**(Q) or **Reverse**$(\tilde{}Q)$ is Q with its terms in reverse order, that is, $[Q[\#Q], \ldots, Q[1]]$. The function **Rotate**(Q, p) or procedure **Rotate**$(\tilde{}Q, p)$ has the effect of rotating the terms of Q cyclically p terms to the right. If $p \geq 0$, the result is $Q[(n - p + 1)..n]$ concatenated with $Q[1..(n-p)]$, and if $p < 0$, the result is $Q[(-p+1)..n]$ concatenated with $Q[1..-p]$, where $n = \#Q$. The result of **Sort**(Q) or **Sort**$(\tilde{}Q)$ is Q with its terms sorted into increasing order; this intrinsic is only available if the universe of Q has a total ordering defined on its elements. For example:

```
> y := [10..20]; y_copy := y;
> Rotate(~y, 3);
> print y;
[ 18, 19, 20, 10, 11, 12, 13, 14, 15, 16, 17 ]
> print Reverse(y);
[ 17, 16, 15, 14, 13, 12, 11, 10, 20, 19, 18 ]
> print Sort(y) eq y_copy;
true
```

6.5.6 Multi-Dimensional Sequences

The elements of a set or sequence may themselves be sets or sequences, and the nesting may occur to arbitrary depth (within memory limits). In the case of sequences, this property permits the creation of multi-dimensional 'arrays'. MAGMA has a *multi-index* facility, whereby a comma-separated list of index positions may be placed within square brackets in order to specify the coordinates of a sequence entry. The multi-index allows the user to reach easily into inner levels of the sequence. In fact, multi-indexing may be applied to any combination of sequences, lists, and tuples.

For example, consider the following sequence Q of sequences of integers. Top-level (standard) terms of Q may be extracted in the usual manner:

```
> Q := [ [40,50], [77,66,55], [], [150,250,350,450] ];
> print Q;
[
    [ 40, 50 ],
    [ 77, 66, 55 ],
    [],
    [ 150, 250, 350, 450 ]
]

> print Q[2];
[ 77, 66, 55 ]
```

However, each term of Q is a sequence, and so terms of these sequences may also be extracted. There are two ways to perform this operation. Firstly, the inner term may be described as a term of a term of Q, using the notation $Q[i][j]$. Secondly, the indices may be abbreviated into a *multi-index*, using the notation $Q[i, j]$. For example:

```
> print Q[2][3];
55
```

```
> print Q[2, 3];
55
```

6.6 Representative and Random Elements

As is explained on p. 84, the function **Rep**(M) returns a representative element of M, for most finite magmas M. For iterable magmas E, there is an alternative which is preferable because it is more efficient in regard to computer time and storage. It is the **rep**-constructor, whose syntax is closely based on that for the homogeneous iterable aggregate constructors:

rep{ $x : x$ **in** E | condition }

where the Boolean condition is expressed in terms of x. When MAGMA evaluates this constructor, it begins to loop through the elements of E. It returns the first element x that satisfies the condition. In this way, MAGMA avoids building the whole set $\{x : x$ **in** $E \mid$ condition$\}$, which might consume a great deal of resources.

For example, consider the problem of finding an odd permutation in the symmetric group of degree 5:

```
> print rep{g: g in Sym(5) | IsOdd(g)};
(2, 3)
```

MAGMA returns the first suitable group element it finds, in this case $(2, 3)$.

Obtaining a random element follows a similar process to obtaining a representative element. Instead of employing the function **Random**(M), the user can apply the **random**-constructor:

random{ $x : x$ **in** E | condition }

Given this constructor, MAGMA will return a random element x of E satisfying the condition. For instance, the following lines construct some random elements of the permutation group on six elements which are not in the cyclic group on six elements:

```
> S6 := Sym(6);
> C6 := CyclicGroup(GrpPerm, 6);
> print random{g: g in S6 | g notin C6};
(3, 5, 4, 6)
> print random{g: g in S6 | g notin C6};
```

```
(1, 4, 6, 2, 3)
> print random{g: g in S6 | g notin C6};
(1, 4, 3, 5, 2, 6)
```

The constructors **rep** and **random** are also available in a multi-identifier version, following the syntax for element-description constructors.

If the set specified in a **rep** or **random** constructor is empty, so that there can be no representative element, then MAGMA will give an error message. In cases where the user is unsure whether there is an element satisfying the condition, the quantifier **exists** should be employed, as explained in the next section.

6.7 Existential and Universal Quantifiers

There are two MAGMA constructors that are related to the mathematical notions of 'there exists' ($\exists$) and 'for all' ($\forall$). Their syntax is similar to the syntax of the **rep** and **random** constructors:

exists{ $x : x$ **in** E | condition }
forall{ $x : x$ **in** E | condition }

where E is an iterable magma. Each of these constructors returns a boolean value: **exists** returns **true** if the set described in the constructor is non-empty, else **false**; and **forall** returns **true** if the set described in the constructor is equal to E, else **false**. However, these constructors do not always build the whole set before returning a value. Instead, they iterate over E, testing the condition for each x in E, and they stop the iteration as soon as the return value is known, which may be long before every element of E has been tested.

For example, let *S5* be the symmetric group on six elements and let *SL* be the subgroup lattice of *S5*. The following lines verify that the order of each subgroup divides the order of *S5*:

```
> S5 := Sym(5);
> ordS5 := Order(S5);
> SL := SubgroupLattice(S5);
> print forall{sg : sg in SL | IsZero(ordS5 mod Order(sg))};
true
```

Suppose now that the user wishes to know whether there are any perfect numbers in the range 100 to 10000. This may be done using the **exists**-constructor and the function *isperfect*(n) discussed on p. 117:

```
> isperfect := func< n | n eq &+Exclude(Divisors(n), n) >;
> print exists{ n : n in [100..10000] | isperfect(n) };
true
```

The output shows that there exists at least one perfect number in that range. MAGMA stops iterating over the sequence as soon as one perfect number is found, so the time taken is much less than the time taken to examine every element in the sequence:

```
> time print exists{ n : n in [100..10000] | isperfect(n) };
true
Time: 1.390
> time print { n : n in [100..10000] | isperfect(n) };
{ 496, 8128 }
Time: 66.066
```

The **exists** and **forall** constructors always return a boolean value. However, if the constructors are used in a modified syntax then they can return an element of the iterable magma E as well, under certain conditions. The syntax required is:

exists(d){ $x : x$ **in** E | condition }
forall(d){ $x : x$ **in** E | condition }

where d is an identifier. If **exists** returns **true**, then d will be assigned an element of E that satisfies the condition; if it returns **false**, then d remains unassigned (or becomes unassigned). As for the **forall**-constructor, if it returns **false**, then d will be assigned an element of E that does not satisfy the condition (i.e., a counter-example); if it returns **true**, then d remains/becomes unassigned. For example:

```
> print exists(N){ n : n in [100..10000] | isperfect(n) };
true
> print N;
496
> print forall(k){ n : n in [100..10000] | isperfect(n) };
false
> print k;
100
```

The **exists** and **forall** constructors may also be applied to situations involving more than one free identifier and domain, in a manner corresponding to that for the aggregate constructors. For example, let *D6* be the dihedral group D_6 under the permutation representation. The following **exists**-

constructor tests whether there are elements p and q of orders 3 and 2 which commute, and finds an instance of such elements:

```
> D6 := DihedralGroup(GrpPerm, 6);
> print exists(pair){<p, q>: p, q in D6 | p*q eq q*p and
>   Order(p) eq 3 and Order(q) eq 2};
true
> print pair;
<(1, 3, 5)(2, 4, 6), (1, 4)(2, 5)(3, 6)>
> print (pair[1], pair[2]); // commutator operator (a, b)
Id(D6)
```

The components of the tuple *pair* satisfy the given condition.

6.8 Creating and Operating on Formal Sets

A *formal set* F is a subset of a magma M, described as those elements of M that satisfy some boolean condition. F is represented internally in terms of M and the condition, not by storing a collection of elements. Unlike the aggregates discussed so far in this chapter, formal sets may be infinite. However, they only have a limited number of operations; for example, it is possible to test whether a given object is in a formal set F, but not to obtain elements of F from F itself. Formal sets are used in such contexts as specifying the domain or codomain of a mapping, or giving the parent for a record field.

The constructor for a formal set has the general form

{! x **in** M | condition !},

where M is any (not necessarily iterable) magma for which it is possible to test membership of objects in M, and the condition is a boolean expression in the free identifier x. The elements of this set are those elements of M that satisfy the condition, and the universe of the set is M. The version of this constructor without the condition, {! x **in** M !}, is the same as **FormalSet**(M), and may be considered as the formal carrier set of M.

For instance, the following statements create the formal set *Zplus* of positive integers and the formal set *Qd7* of rationals whose denominator (after cancellation) is 7:

```
> Zplus := {! n in IntegerRing() | n gt 0 !};
> print Zplus;
```

```
Formal set over Integer Ring

> Qd7 := {! r in RationalField() | Denominator(r) eq 7 !};
> print Qd7;
Formal set over Rational Field
> print Universe(Qd7);
Rational Field
```

The great advantage of formal sets is that they can be infinite; the disadvantage is that only a small number of operations are available for them. The binary set operations **join**, **meet**, **diff**, and **sdiff**, as explained on p. 118, may be applied if the two formal sets are compatible, and are implemented using boolean operations on the conditions. The operators **in** and **notin** test whether the given object (of a suitable kind) is an element of the given formal set. For example:

```
> print 42 in Zplus, 0 in Zplus, -3 in Zplus;
true false false
> print 8/7 in Qd7, 21/7 in Qd7;
true false
> print 21/7 in Zplus join Qd7;
true
```

Although the operations on them may seem restrictive, formal sets can be very useful in combination with other MAGMA constructions because they promote the writing of succinct and lucid code.

6.9 Lists

The purpose of the category **List** is to give the user a means to collect together several objects of assorted kinds. This would generally be done as a temporary measure during a larger task. Aggregates in this category are called *lists*, but this is a technical usage of the word, not to be confused with the informal use of 'list' elsewhere in this book.

Lists are ordered, iterable, and capable of containing repeated elements. However, their elements do not have to belong to the same magma, or even the same category. Therefore lists are like enumerated sequences, except without the restriction of homogeneity; the list bracket `[*` is intended to suggest this, by its similarity to the plain bracket `[` for enumerated sequences. However, enumerated sequences have more operations than lists and are much more efficient, so they should be preferred to lists whenever possible.

Table 6.5. Operations on lists

MAGMA	Meaning
$\#L$	Length of list L (number of elements)
IsEmpty(L)	**true** if L is empty (has zero length)
$L[i]$	i^{th} term of list L $(1 \leq i \leq \#L)$
$L[i] := x$	Changes L by assigning x to the i^{th} term $(1 \leq i \leq \#L + 1)$
$L[i_1, i_2, \ldots, i_k]$	$L[i_1][i_2] \ldots [i_k]$ (multi-indexing)
L **cat** K	List formed by chaining list K to end of list L
L **cat** $:= K$	Changes L to $(L$ **cat** $K)$
Append(L, x)	L with x placed on end
Append$(\tilde{}L, x)$	Changes L to L with x placed on end
Prune(L)	L with final term removed
Prune$(\tilde{}L)$	Changes L to L with final term removed

The only way to create a list is to 'list' its elements within the symbols `[*` and `*]`, since the more sophisticated constructors are not available. For example:

```
> L := [* GF(8), "r", 1/3 *];
> print L;
[* Finite field of size 2^3, r, 1/3 *]
> print Category(L);
List
```

The operations on lists, shown in Table 6.5, are an abridged version of the operations on enumerated sequences. The principal operations are finding the length of a list, accessing and changing the i^{th} term, concatenation, appending and pruning. For example:

```
> print #L;
3
> L[2] := KCubeGraph(2);
> Append(~L, ReedMullerCode(2, 3));

> print L;
[* Finite field of size 2^3,
Graph
Vertex  Neighbours

1       2 3 ;
2       1 4 ;
3       1 4 ;
```

```
4       2 3 ;
, 1/3, [8, 7, 2] Reed-Muller Code (r = 2, m = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 1]
[0 1 0 0 0 0 0 1]
[0 0 1 0 0 0 0 1]
[0 0 0 1 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 1 1] *]
```

6.10 Cartesian Products and Tuples

Table 6.6. Cartesian products

MAGMA	Meaning
car< $M_1, \ldots, M_k$ **>**	Cartesian product $M_1 \times \cdots \times M_k$
CartesianProduct(M, N)	$M \times N$
CartesianPower(M, k)	$M \times \cdots \times M$ (k components)
Flat(C)	Given Cartesian product C whose components may include Cartesian products, return Cartesian product of the base structures, considered in depth-first order
NumberOfComponents(C)	Number k of components of C
C **eq** D	**true** if corresponding components of C and D are equal
Rep(C)	Representative element of C
$\#C$	Cardinality of finite C
Set(C)	Set of elements of finite C
Random(C)	Random element of finite C

Given the magmas (or aggregates) $M_1, \ldots, M_k$, the *Cartesian product* $C = M_1 \times \cdots \times M_k$ of these magmas is a kind of set. Its elements or *tuples* are of the form **<** $m_1, \ldots, m_k$ **>**, where each entry m_i is an element of the corresponding magma M_i. In MAGMA, a Cartesian product is created by listing the component magmas in the product and surrounding them with the bracketing symbols **car<** and **>**:

car< $M_1, \ldots, M_k$ **>**

The category of Cartesian products is **SetCart**. Table 6.6 lists the functions for the creation and access of Cartesian products.

For instance, the following assignment creates the Cartesian product *CP1* of tuples in which the first tuple entry is an integer, the second is **true** or **false**, and the third is a member of the symmetric group $S6$ of degree 6:

```
> S6 := Sym(6);
> CP1 := car< RationalField(), Booleans(), S6 >;
> print CP1;
Cartesian Product<Rational Field, Boolean Structure,
Symmetric group S6 acting on a set of cardinality 6
Order = 720 = 2^4 * 3^2 * 5>
> print NumberOfComponents(CP1);
3
> print Rep(CP1);
<0, true, Id(S6)>
```

If some of the components of a Cartesian product C are Cartesian products themselves, the function **Flat**(C) returns the Cartesian product of the base structures of C. For example:

```
> CP2 := CartesianPower(Booleans(), 4);
> M2 := MatrixRing(GF(2), 2);
> CP3 := CartesianProduct(CP2, M2);
> print NumberOfComponents(CP3);
2
> CP4 := Flat(CP3);
> print NumberOfComponents(CP4);
5
```

Some special operations are available for finite Cartesian products. If all the components of C are known by MAGMA to be finite, then $\#C$ returns the number of elements of C, which equals the product of the cardinalities of the components. Moreover, if MAGMA is able to iterate over the components of C, then it will also be able to iterate over C. In such cases, C may be used as the domain of iteration in a **for**-statement or a set/sequence constructor, and the functions **Set**(C) and **Random**(C) are available. For example:

```
> print #CP4;
256
> S := Set(CP4);
> print Random(CP4);
<true, false, true, false,
[1 1]
[0 1]>
```

Tuples, the elements of Cartesian products, may be created in several ways. The **elt**-constructor

elt< C | $m_1, \ldots, m_k$ >

returns the tuple < $m_1, \ldots, m_k$ > as an element of the Cartesian product C. For example:

```
> t1 := elt< CP1 | 3, true, S6!(2, 4)(3, 6) >;
> print t1;
<3, true, (2, 4)(3, 6)>
> print Parent(t1) eq CP1;
true
```

Alternatively, if t already exists (as a tuple in a related Cartesian product), then it may be coerced into C by means of the coercion operation $C!t$.

It is also possible to create a tuple without specifying its parent explicitly. This is a common method in MAGMA programming; see p. 110 above for an example. The *tuple constructor*

< $m_1, \ldots, m_k$ >

returns the tuple < $m_1, \ldots, m_k$ > as an element of the Cartesian product of the parents of the m_i. In other words, MAGMA deduces the parent product from the entries of the tuple. For instance:

```
> t2 := < 3, true, S6!(2, 4)(3, 6) >;
> print Parent(t2) eq CP1;
false
> print Parent(t2);
Cartesian Product<Integer Ring, Boolean Structure,
Symmetric group S6 acting on a set of cardinality 6
Order = 720 = 2^4 * 3^2 * 5>
```

Table 6.7 lists the operations on tuples; notice that they resemble the sequence operations to some extent. The indexing operations are restricted by the fact that once a tuple t has been created, its parent C is fixed, and any subsequent mutations of t have to conform to C. Therefore the number of entries of t cannot change, and any new value for an entry must have the appropriate parent. For example, the first components of the parents of $t1$ and $t2$ are $\mathbb{Q}$ and $\mathbb{Z}$ respectively, so $t1[1]$ may be changed to a non-integral rational but $t2[1]$ may not:

```
> t1[1] := 17/23;
```

Table 6.7. Tuples

MAGMA	Meaning
elt< C | $m_1, \ldots, m_k$ >	Tuple < $m_1, \ldots, m_k$ > as element of C
< $m_1, \ldots, m_k$ >	Tuple < $m_1, \ldots, m_k$ > as element of Cartesian product implied by the parents of the m_i
C!t	Element of C corresponding to tuple t (entries of t must be coercible into components of C)
Parent(t)	Cartesian product to which tuple t belongs
t **in** C	**true** if tuple t is an element of C
t **eq** u	**true** if corresponding entries of t and u are equal
$\#t$	Number of entries of tuple t
$t[i]$	i^{th} entry of tuple t
$t[i_1, i_2, \ldots, i_k]$	$t[i_1][i_2] \ldots [i_k]$ (multi-indexing)
Explode(t)	All entries of t, in order (multiple return value)

```
> print t1;
<17/23, true, (2, 4)(3, 6)>
> t2[1] := 17/23;

>> t2[1] := 17/23;
     ^
Runtime error in :=: New component not in cartesian product
```

It is not possible to iterate directly over a tuple t, but this may be achieved indirectly by iterating over its index:

```
> for i in [1..#t1] do
>     print t1[i];
> end for;
17/23
true
(2, 4)(3, 6)
```

Given a tuple t, the function **Explode**(t) returns all its entries separately, in their index order. The purpose of this function is to provide a simple means of assigning tuple entries to identifiers, without having to use the notation $t[i]$ several times. For example:

```
> ratl, bl, g := Explode(t1);
> print ratl;
17/23
> print bl;
true
```

```
> print g;
(2, 4)(3, 6)
```

The main purpose of tuples, when considered in their own right rather than as elements of Cartesian products, is to collect together related but inhomogeneous information. Several of MAGMA's functions return tuples for this reason. For instance, if the integer polynomial $x^4 + 5x^3 + 10x^2 + 96x + 288$ is given as an argument to the function **Factorization**, the output is a *factorization sequence* of 2-tuples whose first entry is a factor and whose second entry is the multiplicity of that factor:

```
> P<x> := PolynomialAlgebra(IntegerRing());
> f := Factorization(x^4 + 5*x^3 + 10*x^2 + 96*x + 288);
> print f;
[
    <x + 4, 2>,
    <x^2 - 3*x + 18, 1>
]
> print Universe(f);
Cartesian Product<Univariate Polynomial Algebra in x
over Integer Ring, Integer Ring>
```

Individual factors may be extracted by sequence and tuple manipulation. For instance, the second factor may be obtained as follows, since $f[2]$ is the second tuple and its first entry is the factor:

```
> factor2 := f[2][1];
> print factor2;
x^2 - 3*x + 18
> // or a multi-index:
> print f[2, 1];
x^2 - 3*x + 18
```

6.11 Coproducts

Coproducts, which are the duals of Cartesian products, provide a means to draw together the elements of several magmas or aggregates $S_1, \ldots, S_n$ into a single structure C. The S_i are called the *constituents* of C. Each element x of C arises from exactly one of the constituents, and although x has C as its parent, there is a way of determining which S_i is the original parent of x. The coproduct functions include a universal map facility.

The constructor for the coproduct of $S_1, \ldots, S_n$ is

cop< $S_1, \ldots, S_n$ **>**

However, if the constituents have been placed into a sequence Q then the following alternative form may be used:

cop< Q **>**

This constructor returns the coproduct C as its principal return value, but also returns a sequence of maps $[m_1, m_2, ..., m_k]$. These maps are the injections $m_i : S_i \to C$ that define the coproduct. (This sequence of maps is also available from the function **Injections**(C).) The examples below illustrate the two versions of the coproduct constructor:

```
> F25 := FiniteField(5, 2);
> C, m := cop< F25, Strings(), IntegerRing() >;
> print C;
Coproduct<Finite field of size 5^2, String structure,
Integer Ring>
> print m;
[
    Mapping from: FldFin: F25 to Cop: C,
    Mapping from: String structure to Cop: C,
    Mapping from: Integer Ring to Cop: C
]

> q := [ FiniteField(p) : p in [1..20]  | IsPrime(p) ];
> CC, mm := cop< q >;
> print CC, mm;
Coproduct<Finite field of size 2, Finite field of size 3,
Finite field of size 5, Finite field of size 7,
Finite field of size 11, Finite field of size 13,
Finite field of size 17, Finite field of size 19>
[
    Mapping from: GF(2) to Cop: CC,
    Mapping from: GF(3) to Cop: CC,
    Mapping from: GF(5) to Cop: CC,
    Mapping from: GF(7) to Cop: CC,
    Mapping from: GF(11) to Cop: CC,
    Mapping from: GF(13) to Cop: CC,
    Mapping from: GF(17) to Cop: CC,
    Mapping from: GF(19) to Cop: CC
]
```

The function **Constituent**(C, i) returns the i^{th} constituent of the coproduct C, and $\#C$ returns the number of constituents.

Given $s \in S_i$, there is a corresponding element $x \in C$ given by the injection m_i. To create x in MAGMA, the most general method is to apply this injection, using the standard syntax for mappings, as explained in Chapter 7:

```
> s := F25 ! 3;
> print Parent(s);
Finite field of size 5^2

> x1 := m[1](s);
> print x1;
3
> print Parent(x1);
Coproduct<Finite field of size 5^2, String structure,
Integer Ring>

> xx := mm[6]( Random(Constituent(CC, 6)) );
> print xx;
11
```

However, if the *category* of S_i is unique in the coproduct, then coercion using the `!` operator is an alternative method:

```
> x2 := C ! s;
> print x1 eq x2;
true
```

Once an element x of C has been created, as the image of $s_i \in S_i$, the parent of x will be C. However, sufficient information is stored with x for the corresponding s_i and S_i to be reconstructed. The function **Index**(x) returns the constituent number i, and S_i may then be found from the function **Constituent**(C, i). Moreover, the function **Retrieve**(x) returns s_i. For example:

```
> print Index(x1);
1
> print Constituent(C, 1);
Finite field of size 5^2

> rtv := Retrieve(x1);
> print Parent(rtv) eq F25 and rtv eq s;
true

> print Parent(Retrieve(xx));
Finite field of size 13
```

If the constituents of a coproduct C include one or more structures that are themselves coproducts, then the function **Flat**(C) may be used to find the corresponding coproduct of the base structures, considered in depth-first order. For example, CCC is created below as the coproduct of C and CC, and then it is flattened so that its constituents are the constituents of C and CC:

```
> CCC, mmm := cop< C, CC >;
> print CCC, mmm;
Coproduct<Coproduct<Finite field of size 5^2,
String structure, Integer Ring>,
Coproduct<Finite field of size 2, Finite field of size 3,
Finite field of size 5, Finite field of size 7,
Finite field of size 11, Finite field of size 13,
Finite field of size 17, Finite field of size 19>>
[
    Mapping from: Cop: C to Cop: CCC,
    Mapping from: Cop: CC to Cop: CCC
]

> fCCC, fmmm := Flat(CCC);
> print fCCC, fmmm;
Coproduct<Finite field of size 5^2, String structure,
Integer Ring, Finite field of size 2,
Finite field of size 3, Finite field of size 5,
Finite field of size 7, Finite field of size 11,
Finite field of size 13, Finite field of size 17,
Finite field of size 19>
[
    Mapping from: FldFin: F25 to Cop: fCCC,
    Mapping from: String structure to Cop: fCCC,
    Mapping from: Integer Ring to Cop: fCCC,
    Mapping from: GF(2) to Cop: fCCC,
    Mapping from: GF(3) to Cop: fCCC,
    Mapping from: GF(5) to Cop: fCCC,
    Mapping from: GF(7) to Cop: fCCC,
    Mapping from: GF(11) to Cop: fCCC,
    Mapping from: GF(13) to Cop: fCCC,
    Mapping from: GF(17) to Cop: fCCC,
    Mapping from: GF(19) to Cop: fCCC
]
```

Finally, the function **UniversalMap**(C, D, Q) returns the *universal map* from the coproduct C to the structure D. The third argument of this function is a sequence Q of maps $S_i \to D$, where the S_i are the constituents of

D. MAGMA uses these maps to calculate the universal map. For example, suppose that D is the boolean structure, and that the original coproduct is the coproduct CC above, whose constituents are the finite fields $q[i]$. To keep the example simple, let each of the maps $q[i] \to D$ be of the same general kind, namely, a test of the primitivity of the given finite field element. The result is:

```
> D := Booleans();
> qq := [ map< gf -> D | f :-> IsPrimitive(f) > : gf in q ];
> um := UniversalMap(CC, D, qq);
> print um;
Mapping from: Cop: CC to Bool: B
given by function(x) ... end function
> print um(xx);
true
```

The output in the final line may be interpreted as saying that the constituent element corresponding to xx (that is, the element 11 in the finite field with 13 elements) is a primitive element of its field.

Table 6.8. Operations on coproducts

MAGMA	Meaning
$\#C$	Number of constituents of coproduct C
Injections(C)	Sequence $[m_1, \ldots, m_{\#C}]$ of injections from constituents into C (same as the second return value of the **cop**-constructor)
Constituent(C, i)	i^{th} constituent of C
Index(x)	Constituent number i of $x \in C$
Retrieve(x)	Element s_i of constituent S_i such that $m_i(s_i) = x$
Flat(C)	Flattened coproduct corresponding to C, where C is a coproduct of coproducts
UniversalMap(C, D, Q)	Given coproduct C with constituents S_i, structure D, and sequence Q of maps $S_i \to D$, returns universal map $C \to D$

Table 6.8 summarizes the operations on coproducts. Note that there is no equality test between coproducts, since this is not feasible in general.

6.12 Record Formats and Records

6.12.1 Creating Record Formats and Records

The most noticeable difference between records and tuples is that the fields of a record have names, whereas the components of a tuple are numbered. Moreover, record fields have no particular order, and it is possible for not all of the fields in a given record to be assigned.

The first step in defining a collection of records of the same kind is to define their common *record format*. A tuple can be created by itself, without its parent Cartesian product being created first, but before a record is created its record format must be stated explicitly. The syntax for defining a record format F is:

F `:=` **recformat**`<` field`:` expr ..., field`:` expr `>;`

where each field is an identifier giving the fieldname, and each expression after the colons may evaluate to either a parent magma such as **Sym**(5) or a category such as **RngIntElt**. This parent magma or category imposes restrictions on what may be stored in the corresponding record field. On the other hand, if the user wishes a record field to be able to contain data of any type, then both the colon and the expression following should be omitted.

A record r of a particular format may be defined using the syntax:

r `:=` **rec**`<` F | field`:=`expr, ..., field`:=`expr `>;`

However, it is not obligatory to assign values to all the fields of a record. Some of the fields may be assigned later, or left permanently unassigned.

For example, suppose there are some groups for which information has to be stored, in one record per group. Each record is to have up to three fields: *gens*, storing the generators of the group; *order*, storing the group's order; and *comment*, storing a comment about the group. A suitable record format may be created in this way:

```
> GroupInfo := recformat< gens,
>   order : IntegerRing(), comment : MonStgElt >;
```

In this record format, no limitations have been put on the contents of the field *gens*. It is then the user's responsibility to place sensible values in this field; Magma will not prevent a vector space being stored in it, for instance. The restriction on the *order* field is given as a parent magma, in such a way that any value stored in this field must be an integer. The *comment* field is restricted to string values.

A few record definitions are shown below. Notice that in some cases not all of the record fields are assigned:

```
> gA := rec<GroupInfo | gens := {Sym(5) | (1,2,3,4,5),
>     (1,5)}, comment := "symmetric">;
> print gA;
rec<GroupInfo | gens := {
    (1, 2, 3, 4, 5),
    (1, 5)
}, comment := symmetric>
> gB := rec<GroupInfo | gens := {Sym(7) | (1,5)(3,6)(7,2),
>     (1,2,3,4,5,6,7)},
>     order := 5040>;
> print gB;
rec<GroupInfo | gens := {
    (1, 2, 3, 4, 5, 6, 7),
    (1, 5)(2, 7)(3, 6)
}, order := 5040>
> gC := rec<GroupInfo | gens := {Sym(15) | (1,2,3,4,5,6),
>     (7,8), (9,10,11,12,13,14,15)},
>     comment := "abelian", order := 84>;
> print gC;
rec<GroupInfo | gens := {
    (1, 2, 3, 4, 5, 6),
    (9, 10, 11, 12, 13, 14, 15),
    (7, 8)
}, order := 84, comment := abelian>
```

The operator used to refer to a field of a record is the ‘ symbol. Note that this is the backquote symbol; it is not the same as ’ (the apostrophe symbol). If r is a record and f is a fieldname, then $r\text{‘}f$ is the value stored in field f of record r. This syntax may be used both in expressions and on the left side of an assignment statement. For instance:

```
> print gC‘comment;
abelian

> gA‘order := 120;
> print gA;
rec<GroupInfo | gens := {
    (1, 2, 3, 4, 5),
    (1, 5)
}, order := 120, comment := symmetric>
```

The commands for testing whether a record field is assigned and for deleting it are **assigned** and **delete**, the same as for identifiers. For instance, the following line tests whether the *comment* fields of gA and gB are assigned:

```
> print assigned gA`comment, assigned gB`comment;
true false
```

Sometimes, when referring to the contents of field f of a record r, it is not possible to give the fieldname f literally. An alternative to $r`f$ is the notation $r``s$, where s evaluates to a string that is the same as the fieldname. For example:

```
> chosen_field := "gens";
> print gC``chosen_field;
{
    (1, 2, 3, 4, 5, 6),
    (9, 10, 11, 12, 13, 14, 15),
    (7, 8)
}
```

Given a record r with record format F, the function **Format**(r) returns the value of F. The function **Names**(r) or **Names**(F) returns the fieldnames of the record r or record format F, as a sequence of strings.

The operators **eq** and **ne** (and **lt** etc.) are not available for records, because for many record formats the concept of equal records would not make sense. The user must write a suitable equality function if it is desired. This function would have to specify, for instance, whether two records that are equal in all assigned components but have different unassigned components should be considered equal.

6.12.2 Example: Using a Function to Create Records

In this extended example, one of the field values in a collection of records will be constructed from the other fields using a function. Consider a record format *irred*, containing three fields: *prime*, holding a positive prime number p; *degree*, holding a positive integer d; and *polys*, holding the set of all monic irreducible polynomials of degree d over the finite field GF(p). This record format can be defined as follows:

```
> irred := recformat<
>     prime : {! n in Z | n ge 1 and IsPrime(n) !},
>     degree : {! n in Z | n ge 1 !},
>     polys : SetEnum >
>     where Z is IntegerRing();
```

When records in this format are constructed, MAGMA will check that the contents of the fields *prime* and *degree* are exactly as they should be. However, the only test it performs on the contents of *polys* is to see that it is an enumerated set; it does not check that the set contains polynomials, and certainly does not check that the set contains all the monic irreducible polynomials with the proper degree and coefficient ring.

It is possible to construct records of this format directly or indirectly. For instance, the only monic irreducible polynomials of degree 3 over GF(2) are $x^3 + x^2 + 1$ and $x^3 + x + 1$, so the record for prime 2 and degree 3 may be created directly as follows:

```
> r := rec< irred | prime := 2, degree := 3,
>     polys:={PolynomialRing(GF(2))|[1,0,1,1],[1,1,0,1]}>;
> print r;
rec<irred | prime := 2, degree := 3, polys := {
    $.1^3 + $.1^2 + 1,
    $.1^3 + $.1 + 1
}>
```

Here the `$.1` in the output is the indeterminate of the polynomial ring; this generator has not been given a special printname.

For larger primes or degrees, it is better to use MAGMA to compute the irreducible polynomials. The function below, *irredCreate*(p, d), does this. Given a prime p and a degree d, it returns a record in the format *irred* in which *prime* is p, *degree* is d, and the record field *polys* has been assigned the correct set of monic irreducible polynomials.

```
> irredCreate := func< p, d | rec<irred |
>     prime := p,
>     degree := d,
>     polys := AllIrreduciblePolynomials(GF(p), d) > >;
```

This function may be used to confirm that the assignment to r above was correct:

```
> r1 := irredCreate(2, 3); print r1;
rec<irred | prime := 2, degree := 3, polys := {
    $.1^3 + $.1^2 + 1,
    $.1^3 + $.1 + 1
}>
> print r`polys eq r1`polys;
true
```

(Recall that it is not possible to test two records directly for equality using the **eq** operator.)

Now that the function *irredCreate* has been constructed, it is easy to create records in the *irred* format. For instance, the following statement assigns to *recs* the 18 records for primes 2, 3, 5 and degrees up to 6:

```
> recs := [irredCreate(p, d): p in {2, 3, 5}, d in {1..6} ];
```

The record *r1*, with prime 2 and degree 3, happens to be the 7^{th} entry of this sequence:

```
> print recs[7];
rec<irred | prime := 2, degree := 3, polys := {
    $.1^3 + $.1^2 + 1,
    $.1^3 + $.1 + 1
```

After this sequence of records has been created, entries may be selected that satisfy various properties. For instance, the following statement prints the set of the numbers of irreducible polynomials for each case:

```
> print {#r`polys : r in recs};
{ 1, 2, 3, 5, 6, 8, 9, 10, 18, 40, 48, 116, 150, 624, 2580 }
```

and the statement below prints the sequence of those records in *recs* which contain from 5 to 8 monic irreducible polynomials:

```
> print [r: r in recs | #r`polys in {5..8} ];
[ rec<irred | prime := 5, degree := 1, polys := {
    $.1 + 2,
    $.1 + 1,
    $.1 + 4,
    $.1,
    $.1 + 3
}>, rec<irred | prime := 3, degree := 3, polys := {
    $.1^3 + 2*$.1 + 1,
    $.1^3 + 2*$.1^2 + 2*$.1 + 2,
    $.1^3 + $.1^2 + $.1 + 2,
    $.1^3 + 2*$.1 + 2,
    $.1^3 + $.1^2 + 2,
    $.1^3 + 2*$.1^2 + $.1 + 1,
    $.1^3 + $.1^2 + 2*$.1 + 1,
    $.1^3 + 2*$.1^2 + 1
}>, rec<irred | prime := 2, degree := 5, polys := {
    $.1^5 + $.1^2 + 1,
```

```
    $.1^5 + $.1^4 + $.1^2 + $.1 + 1,
    $.1^5 + $.1^3 + $.1^2 + $.1 + 1,
    $.1^5 + $.1^3 + 1,
    $.1^5 + $.1^4 + $.1^3 + $.1 + 1,
    $.1^5 + $.1^4 + $.1^3 + $.1^2 + 1
}> ]
```

6.13 Transfer Functions Between Aggregates

Table 6.9. Transfer functions between aggregates

MAGMA	Meaning
SetToSequence(S), **Setseq**(S)	Enumerated sequence corresponding to enumerated set S
SequenceToSet(S), **Seqset**(S)	Enumerated set corresponding to enumerated sequence S
SetToMultiset(S)	Multiset corresponding to enumerated set S
MultisetToSet(S)	Enumerated set corresponding to multiset S
SetToIndexedSet(S)	Indexed set corresponding to enumerated set S
IndexedSetToSet(S), **Isetset**(S)	Enumerated set corresponding to indexed set S
IndexedSetToSequence(S), **Isetseq**(S)	Enumerated sequence corresponding to indexed set S
SequenceToMultiset(S)	Multiset corresponding to enumerated sequence S
SequenceToList(S), **Seqlist**(S)	List corresponding to enumerated sequence S
TupleToList(S), **Tuplist**(S)	List corresponding to tuple S

It occasionally happens that a single aggregate category does not offer all the operations required for certain manipulations of a collection of objects. For such cases, MAGMA has *transfer functions*, which convert an aggregate from one category to another, preserving its elements. As far as possible, given the properties of the old and new categories, the order and multiplicity of the elements are retained.

The transfer functions between aggregates are listed in Table 6.9. Notice that **Set** or **Sequence** within the name of a transfer function designate an enumerated set or enumerated sequence respectively. Several of the functions have abbreviated forms.

For example, suppose that q is an enumerated sequence of polynomials in x over $\mathbb{Z}$, and that the user wants to know how many times q contains the element $3x + 5$. Although this operation could be done using sequence constructors, it may be more convenient to convert q to a multiset, using the function **SequenceToMultiset**, and then invoke the **Multiplicity** function:

```
> P<x> := PolynomialRing(IntegerRing());
> q := [ 6*x^2, 5*x+4, 3*x+5, 6*x^2, 6*x^2, 3*x+5, x^3-1,
>   3*x+5, 17*x^2-x+2, -2*x^3, 6, x-4 ];
> mq := SequenceToMultiset(q);
> print mq;
{*
    17*x^2 - x + 2,
    x^3 - 1,
    -2*x^3,
    5*x + 4,
    6,
    x - 4,
    3*x + 5^^3,
    6*x^2^^3
*}
> print Multiplicity(mq, 3*x+5);
3
```

It may be concluded from the output that $3x + 5$ occurs three times in q.

7. Mappings and Homomorphisms

> 'I expect you will consider it imprudent, Mr Voss, if I ask whether you have studied the map?'
> ... 'The map?' said Voss. ... 'I will first make it'.
>
> *Voss*
> Patrick White

Suppose that A and B are magmas, where the term 'magma' includes sets and sequences as well as algebraic structures. One of the features of the MAGMA system is that it is possible to relate A and B together by means of mappings between them. Each mapping may be either a homomorphism (if this is meaningful for the categories involved) or some other element-image correspondence that does not preserve structural properties.

In one sense, a mapping $m : A \to B$ may be seen as a function, since it is a rule that, given an element in A, returns its image under m in B. In MAGMA a mapping can be implemented as a function, as in Chapter 8, but there is also a special mapping datatype. As this chapter shows, if a mapping is to be considered as a relationship between two structures then it is preferable in the MAGMA system to create it as a member of the **Map** category rather than as a function. One reason for this is that when m is applied to an object a, MAGMA automatically ensures that a is an element of the domain A (or coercible into it) and that $m(a)$ is an element of the codomain B (or coercible into it). By contrast, when a function is applied, no checks are made on the parents of the argument or its image; this flexibility is desirable for many programming contexts.

Homomorphisms and other mappings are returned by many MAGMA constructors and functions, either as the principal return value or as an auxiliary value that relates the magma given by the principal value to the magmas used to create it. It is also possible for the user to create a mapping or homomorphism directly, by means of the **map**-constructor or **hom**-constructor or by composing two existing mappings. For all kinds of mappings, it is possible to calculate images of domain elements and of sets or sequences of domain elements, and for some of them preimages are also available.

This chapter begins by discussing the standard mappings and mapping operations and then proceeds to the construction of mappings, since users are likely to encounter standard mappings before having to create their own. It should be noted that, in addition to the **map** and **hom** constructors, MAGMA has a **pmap**-constructor for *partial maps*; for this constructor, see the *Handbook*.

7.1 Standard Mappings

This section illustrates some of the major ways in which MAGMA returns mappings as principal or auxiliary values of constructors or functions.

As explained in Section 4.6, the **sub**, **ideal**, and **ncl** constructors, which build a submagma or ideal N from an initial magma M, have two return values: N, and the inclusion monomorphism $i : N \to M$. For instance:

```
> V := VectorSpace(GF(5), 4);
> S, i := sub< V | [0,2,1,2], [4,0,0,1] >;
> print i;
Mapping from: ModTupFld: S to ModTupFld: V
```

The **quo**-constructor, which builds a quotient magma Q from an initial magma M, also has two return values, as explained in Section 4.8: the quotient Q, and the natural epimorphism $\phi : M \to Q$. For instance:

```
> W, n := quo< V | [0,0,3,2], [2,1,4,3] >;
> print n;
Mapping from: ModTupFld: V to ModTupFld: W
```

The coercion $T!m$ of an element m of a magma M into the corresponding element of the magma T stems from a mapping from M to T. The function **Coercion**(M, T), or **Bang**(M, T), provides this mapping explicitly. For instance, the following lines construct the mapping that MAGMA uses to coerce integers into real numbers, and then compare the application of the mapping to the usual method of coercion:

```
> R := RealField();
> Z := IntegerRing();
> coerceIntToReal := Coercion(Z, R);
> print coerceIntToReal;
Mapping from: RngInt: Z to FldPr: R
> r1 := R ! 17; // standard coercion
> print Parent(r1);
```

```
Real Field
> r2 := 17 @ coerceIntToReal; // application of mapping
> print Parent(r2);
Real Field
> print r1 eq r2;
true
```

Several of the functions for direct product and direct sum return inclusion homomorphisms and projection homomorphisms as well as the product or sum itself. For instance:

```
> U := AlternatingGroup(5);
> Y := DihedralGroup(6);
> DP, m1, m2, m3, m4 := DirectProduct(U, Y);
> print m1;
Mapping from: GrpPerm: U to GrpPerm: DP
> print m2;
Mapping from: GrpPerm: Y to GrpPerm: DP
> print m3;
Mapping from: GrpPerm: DP to GrpPerm: U
> print m4;
Mapping from: GrpPerm: DP to GrpPerm: Y
```

In the output above, DP is the direct product, the first two mappings are the inclusions of U and Y into DP, and the second two mappings are the projections of DP onto U and Y.

A number of group functions whose names involve the word 'action' calculate some kind of homomorphism. They return three values: the homomorphism $h : G \to L$, the image L induced by h, and the kernel of h. For example, given a group G and a subgroup H, **CosetAction**(G, H) returns the homomorphism, image, and kernel connected with the action of G on the (right) cosets of H:

```
> G := MatrixGroup< 3, GF(3) |
    [0,2,0, 1,1,0, 0,0,1], [0,1,0, 0,0,1, 1,0,0] >;
> H := sub< G | G.1^2, G.2 >;
> A, P, K := CosetAction(G, H);
> print A;
Mapping from: GrpMat: G to GrpPerm: P
> print P;
Permutation group P acting on a set of cardinality 26
    (1, 2)(3, 4, 6, 5, 7, 9)(8, 11)(10, 13, 15, 20, 18,
        17)(12, 16, 21, 14, 19, 24)(23, 26)
    (2, 3, 5)(4, 6, 8)(7, 10, 14)(9, 12, 17)(11, 15,
```

```
          20)(13, 18, 23)(16, 22, 21)(19, 25, 24)
> print K;
MatrixGroup(3, GF(3)) of order 1
```

7.2 Operations on Mappings and Homomorphisms

7.2.1 Calculating Images

The most important operation for a mapping or homomorphism $m : A \to B$ is to calculate the image of $a \in A$ under m, as an element of B. There are two notations for this: $m(a)$ and a`@`m. The former choice relates to the function notation of analysts, and the latter choice to the mapping notation of algebraists; it makes no difference to MAGMA which is used. For instance, consider the vector space V and quotient W constructed in the previous section:

```
> v := V ! [4,1,1,0];
> w := v @ n; print w;
(4 2)
> print W eq Parent(w);
true
> print w eq n(v);
true
```

As another example, consider the direct product DP constructed above:

```
> y := Y ! (1, 5, 3)(2, 6, 4);
> print m2(y);
(6, 10, 8)(7, 11, 9)
> print y eq m4(m2(y));
true

> dp := DP ! (1, 4, 3, 2, 5)(6, 10)(7, 9);
> t := < dp @ m3, dp @ m4 >;
> print t;
<(1, 4, 3, 2, 5), (1, 5)(2, 4)>
> print (t[1] @ m1) * (t[2] @ m2);
(1, 4, 3, 2, 5)(6, 10)(7, 9)
```

Given an enumerated set or indexed set T of several elements of the domain A, the expression $m(T)$ or T`@`m returns the set of their images under m. For instance:

```
> gensV := Generators(V);
> print gensV;
{
    (0 1 0 0),
    (0 0 1 0),
    (1 0 0 0),
    (0 0 0 1)
}
> print gensV @ n;
{
    (2 4),
    (0 1),
    (1 0)
}
```

Here the set in the output contains fewer elements than the given set, because some of the elements share the same image.

Given a sequence Q of several elements of A, $m(Q)$ or Q@m returns the sequence of their images under m, where the terms of the image sequence are the images of the corresponding terms of the original sequence. For instance:

```
> gensVseq := Setseq(gensV);
> print gensVseq;
[
    (0 1 0 0),
    (0 0 1 0),
    (1 0 0 0),
    (0 0 0 1)
]
> print gensVseq @ n;
[
    (1 0),
    (0 1),
    (2 4),
    (0 1)
]
```

This output reveals that the second and fourth elements of *gensVseq* share the same image under n.

Finally, if m is a homomorphism and N is a submagma of A, then $m(N)$ or N@m returns the image of N under m as a submagma of B. This follows from the structure-preserving nature of homomorphisms. For instance:

```
> Ys := sub< Y | (1, 2)(3, 6)(4, 5), (1, 4)(2, 3)(5, 6) >;
```

```
> print m2(Ys);
Permutation group acting on a set of cardinality 11
    (6, 9)(7, 8)(10, 11)
    (6, 7)(8, 11)(9, 10)
```

7.2.2 Calculating Preimages

Given $m : A \to B$ and an element $b \in B$, a preimage of b under m is an element $a \in A$ such that $m(a) = b$. There are three possibilities concerning the preimages of b: there may be more than one preimage (a finite or infinite number); there may be exactly one preimage; or there may be no preimages (i.e., b is not in the image of m). If m is a homomorphism and at least one preimage exists, then the set of preimages will form a submagma of A.

The calculation of preimages is often non-trivial. For this reason, it is only available in MAGMA for some of the mappings returned by standard functions, and in some categories for homomorphisms defined with the **hom**-constructor (to be explained below). Otherwise, if a preimage is required for a user-defined mapping then the inverse mapping must be created explicitly by the user.

The preimage operation in MAGMA is b`@@`m, where $b \in B$. It either returns an element of the domain A that is a preimage for b, or gives an error message if no preimage exists. For example:

```
> s1 := (V ! [3,0,0,2]) @@ i;
> print s1;
(3 0 0 2)
> print Parent(s1) eq S;
true

> v2 := V ! [2,4,1,0];
> print v2 in S;
false
> s2 := v2 @@ i;

>> s2 := v2 @@ i;
           ^
Runtime error in '@@': Application of map failed
```

Even if there is more than one preimage for b, the operation b`@@`m only returns only one preimage a. In practice, this situation only arises in MAGMA for homomorphisms. If m is a homomorphism, the set of all the preimages may be found by calculating the coset of the kernel of m that contains a. For

instance, recall these definitions of $v \in V$ and $w \in W$ where n is the natural homomorphism from V to the quotient W:

```
> v := V ! [4,1,1,0];
> w := v @ n; print w;
(4 2)
```

Now, the following line calculates a preimage for w:

```
> preim := w @@ n;
> print preim;
(3 3 0 0)
```

Note that *preim* does not equal v; there are in fact 25 preimages, since 25 is the size of the kernel of n. The following lines show how to construct the set of all preimages of w:

```
> K := Kernel(n);
> print #K;
25
> allpreims := { preim + k : k in K };
> print #allpreims;
25
> print v in allpreims;
true
> print allpreims @ n;
{
    (4 2)
}
```

If m is such that preimages of elements are available, then preimages will also be available for a set or sequence of elements in the image. Once again, the set or sequence returned by MAGMA contains coset representatives of the preimages, not whole preimages. For example:

```
> print [ w, 2*w, 3*w, 4*w ] @@ n;
[
    (3 3 0 0),
    (1 1 0 0),
    (4 4 0 0),
    (2 2 0 0)
]
```

By contrast with the above cases, when MAGMA is given a homomorphism for which preimages are available, and is asked for the preimage of a

submagma or ideal of the image, it returns the whole preimage, which is a submagma or ideal of the domain and is homomorphic to the given structure. For instance:

```
> Ws := sub< W | w >;
> print Ws @@ n;
Vector space of degree 4, dimension 3 over GF(5)
Echelonized basis:
(1 0 0 2)
(0 1 0 3)
(0 0 1 4)
```

7.2.3 Functions on Mappings and Homomorphisms

The functions **Domain**(m) and **Codomain**(m) return A and B when given a mapping $m : A \to B$. The codomain B is the set used in the definition of m (by MAGMA or the user), and may be larger than the image of m, which is the actual set of images.

If m is a homomorphism, **Image**(m) returns the image of m as a submagma of the codomain, and **Kernel**(m) returns the kernel of m as an ideal of the domain. These magmas may be computed for standard homomorphisms, and in some categories for homomorphisms defined with the **hom**-constructor. For instance, using the direct product example:

```
> print Domain(m1) eq U;
true
> print Codomain(m1) eq DP;
true
> print Image(m1);
Permutation group acting on a set of cardinality 11
    (1, 2, 3)
    (3, 4, 5)
> print Kernel(m1);
Permutation group acting on a set of cardinality 5
Order = 1
    Id($)
```

7.3 Constructing Mappings and Homomorphisms

There are two methods for creating a user-defined mapping $A \to B$: the **hom**-constructor and the **map**-constructor. Their syntax is:

hom< *A* -> *B* | mapping rule >
map< *A* -> *B* | mapping rule >

Note carefully the keyboard symbols required. The mapping definition is bracketed by the keyword **hom** or **map** and angle brackets, according to the usual syntax for a constructor. The left part of the constructor specifies the domain and codomain of the mapping, and the minus sign and greater-than sign together simulate a right arrow. The right part specifies the mapping rule. There are several ways of giving this mapping rule, as will be explained below.

In general, these two constructors are used respectively for homomorphisms and for other mappings. However, MAGMA does not check whether any user-defined mapping is a homomorphism or not; the distinctions exist because these constructors possess some distinct ways of specifying the mappings, and because there are some extra operations available for mappings defined with **hom** rather than **map**. Therefore it is important for the user to use the **hom**-constructor for mappings known to be homomorphisms. Conversely, for occasional advanced purposes it may be convenient to define a non-homomorphism using **hom** so as to exploit some of the extra functionality.

7.3.1 Mapping Rule Given as Expression

If the mapping rule is given as an expression, the right side of the constructor has three parts: an identifier that serves as a local free variable, the 'maps-to' symbol `:->` (a compound symbol consisting of a colon, a minus sign, and a greater-than sign), and an expression in terms of the free variable. The expression is the image of the free variable under the mapping. To evaluate the mapping for a particular value of the domain, MAGMA substitutes that value for the free variable in the expression.

For instance, the mapping $x \mapsto x - \sin x$ over the real field may be constructed as follows:

```
> R := RealField();
> trigmap := map< R -> R | x :-> x-Sin(x) >;
> print trigmap;
Mapping from: FldPr: R to FldPr: R
given by function(x) ... end function
```

Here the right part of the **map** constructor gives the mapping rule in terms of the local identifier x, by saying that the image of x is $x - \sin x$.

As another example, consider the mapping over $\mathbb{R}$ given by $\theta \mapsto q^5 + 4q^3 - 2e^q + q\theta$ where $q = \cos\theta$. It may be elegantly defined by means of a **where**-construction:

```
> thetamap := map< R -> R |
>     theta :-> q^5 + 4*q^3 - 2*Exp(q) + q*theta
>       where q is Cos(theta) >;
> print 15.47 @ thetamap;
-20.32894612411469619708857665824 7
```

Any mapping for which the image of the generic element can be written as a single MAGMA expression can be formed by giving the mapping rule as an expression. For instance, the following mapping definition assigns the mapping of the famous $3n + 1$ problem to the identifier *three*:

```
> Zplus := {! n in IntegerRing() | n gt 0 !};
> three := map< Zplus -> Zplus |
>     n :-> IsEven(n) select n div 2 else 3*n+1 >;
```

The employment of the formal set *Zplus* as the domain and codomain of *three* ensures both that the mapping will only be evaluated for positive integers, and that MAGMA will check that the result is a positive integer. Note that if *Zplus* does not need to be assigned 'outside' the map, then *three* can be created using a **where**-construction for its domain:

```
> three := map< Zplus -> Zplus |
>    n :-> IsEven(n) select n div 2 else 3*n+1 >
>    where Zplus is {! n in IntegerRing() | n gt 0 !};
> print 5 @ three;
16
> print 0 @ three;

>> print 0 @ three;
          ^
Runtime error in map application: Element is not
    in the domain of the map
```

The mapping rule may also be given as an expression within the **hom**-constructor. For instance, consider the following homomorphism from the symmetric group on five elements to the symmetric group on two elements:

```
> S5<a, b> := Sym(5);
> S2<c> := Sym(2);
> h52:=hom<S5 -> S2 | g:-> IsEven(g) select Id(S2) else c>;
```

This is a homomorphism because the product of the parity of two permutations is the parity of their product, using rules for multiplying parity which correspond to calling an even permutation **Id**(*S2*) and an odd permutation *S2*.1. The following line tests that *h52* is indeed a homomorphism by the brute-force method of checking every case:

```
> print exists(ex){<x, y> : x, y in S5 |
>   (x * y)@h52 ne (x@h52 * y@h52) };
false
```

Of course, more subtle methods would be required for very large structures.

Another example of a homomorphism defined using an expression is an inclusion mapping from *gaussians*, a matrix ring discussed on p. 72, to the complex field:

To perform this example online, type `load "I96c7e1";`

```
> gaussians<I> :=
>     sub< MatrixRing(IntegerRing(), 2) | [0, 1, -1, 0] >;
> C<i> := ComplexField();
> gausshom:=hom<gaussians->C | m :->C![ m[1,1], m[1,2] ] >;
> print 7+3*I;
[ 7  3]
[-3  7]
> print (7+3*I) @ gausshom;
7 + 3*i
```

This homomorphism maps the matrix I to the complex number i.

When a mapping m is defined as an expression, the function underlying the mapping may be extracted using the function **Function**(m). For example:

```
> fn := Function(h52);
> print fn;
function(g) ... end function
> print fn(b);
(1, 2)
```

When this function is given arguments from the domain of *h52*, then it will behave in the same way as the homomorphism, but the function can also accept arguments with other parents:

```
> print h52(15);
```

```
>> print h52(15);
            ^
Runtime error in map application:
    Element is not in the domain of the map

> print fn(15);
(1, 2)
```

Whether this is an advantage or a disadvantage depends on the user's perspective.

7.3.2 Element-by-Element Rule for General Mappings

If $m : A \to B$ is to be created using a **map**-constructor, and A is small, it is sometimes preferable to give the mapping rule element-by-element. In this version of the mapping constructor, each element is listed with its image in a 2-tuple on the right side of the constructor. Every element of the domain must appear exactly once as the first element of a tuple in this list, otherwise the mapping definition will not make sense.

For example, consider the (rather arbitrary) mapping from GF(5) to $\mathbb{Z}$ in which 0 maps to 34, 1 and 3 map to -17, 2 maps to 42, and 4 maps to 8. It may be created as follows:

```
> m := map< GF(5) -> IntegerRing() |
>     <0, 34>, <1, -17>, <3, -17>, <2, 42>, <4, 8> >;
> print m;
Mapping from: GF(5) to RngInt: Z
    <0, 34>
    <1, -17>
    <3, -17>
    <2, 42>
    <4, 8>
> print m(3);
-17
> print m(2);
42
```

As another example, consider a set of names of colours, represented as strings. They may be classified in several ways – as warm or cool, for instance. The following lines construct a mapping to represent such a classification. Before the mapping is created, the set of the image-element pairs is formed; it is possible to place such a set on the right side of the constructor rather than listing the correspondences directly:

To perform this example online, type `load "I96c7e2";`

```
> colours := {"red", "blue", "green", "orange"};
> print colours;
{ red, blue, orange, green }
> warmcool := { <"red", "warm">, <"blue", "cool">,
>     <"green", "cool">, <"orange", "warm"> };
> temperature := map<
      colours -> {"warm", "cool"} | warmcool >;
> print temperature;
Mapping from: SetEnum: colours to { warm, cool }
    <orange, warm>
    <blue, cool>
    <green, cool>
    <red, warm>
> print temperature("blue");
cool
```

In the example above, the codomain of the mapping is chosen to be the same as the image. It is also possible to specify a larger codomain, such as the general MAGMA string structure:

```
> temperature := map< colours -> Strings() | warmcool >;
> print Codomain(temperature);
String structure
```

Finally, MAGMA has an alternative input syntax for 2–tuples which users may find convenient in this context. MAGMA understands the arrow-pair a->b (where the compound symbol is a minus sign followed by a greater-than sign) to be the same as the 2-tuple <a, b>. For instance:

```
> warmcool := { "red"->"warm", "blue"->"cool",
>     "green"->"cool", "orange"->"warm" };
> print warmcool;
{ <orange, warm>, <blue, cool>, <green, cool>, <red, warm> }
```

Notice that the output conforms to the tuple notation <a, b>, despite the use of arrow-pairs in the input.

7.3.3 Generator-Image Mapping Rule for Homomorphisms

If $m : A \to B$ is being created using a **hom**-constructor, and A is finitely generated, the mapping rule may be specified by giving the images under

m of the standard generators $A.i$ of A. From the images of the generators, MAGMA can calculate the image of any element of the domain, by assuming the homomorphism property. The images must be placed in a list on the right side of the constructor, ordered according to the standard order of the generators. For instance, in the homomorphism *h52* defined above, the generators $a = S5.1$ and $b = S5.2$ map to $\mathbf{Id}(S2)$ and c respectively, so the homomorphism can also be defined in this way:

```
> H52 := hom< S5 -> S2 | Id(S2), c >;
```

It is also permissible to give the images of the standard generators in a sequence:

```
> tq := [Id(S2), c];
> H52 := hom< S5 -> S2 | tq >;
```

In some categories, the user need not be restricted to the standard generators. If $a_1, \ldots, a_k$ are elements of A forming a generating basis for A, with images $b_1, \ldots, b_k$ under the homomorphism m, then the mapping rule may be given as a list of arrow-pairs (or 2-tuples) of the form a_i->b_i. For example, consider the vector space V and its 2-dimensional subspace S defined above. A homomorphism with domain S may be defined using any two linearly independent vectors of S as its generators:

```
> V := VectorSpace(GF(5), 4);
> S, i := sub< V | [0,2,1,2], [4,0,0,1] >;
> s1 := S ! [2,1,3,4];
> s2 := S ! [0,1,3,1];
> print IsIndependent({s1, s2});
true
> hSV := hom< S->V | s1 -> V![2,0,1,0], s2 -> V![0,0,1,0] >;
> print hSV;
Mapping from: ModTupFld: S to ModTupFld: V
> print Image(hSV);
Vector space of degree 4, dimension 2 over GF(5)
Echelonized basis:
(1 0 0 0)
(0 0 1 0)
```

In rings and fields, MAGMA homomorphisms are assumed to be unitary. In other words, it is assumed that 1 in the domain will map to 1 in the codomain. This assumption then determines the whole map. Therefore ring and field homomorphisms should be defined with no mapping rule after the | symbol. For instance:

```
> C := ComplexField();
> MC := MatrixRing(C, 2);
> h := hom< Z -> MC | >;
> print (6 @ h) ^ -1;
[1/6   0]
[  0 1/6]
```

Often such homomorphisms work like automatic or non-automatic coercion maps. Here, h maps 6 to the element $MC!6$, and then the matrix is inverted.

7.4 Composing Mappings

An important part of the concept of mappings is that they can be *composed*, that is, multiplied together to produce a new mapping. Given the mappings $m : A \to B$ and $n : B \to C$, their composition is a mapping $p : A \to C$ which is written mathematically as mn (using the algebraists' convention of writing mappings on the right). Thus the MAGMA notation for the composition of f and g is $m*n$, where `*` as usual acts as the multiplication operator. For instance, the homomorphism $hSV : S \to V$ may be composed with the natural homomorphism $n : V \to W$ to yield a homomorphism from S to W:

```
> comp := hSV * n;
> print comp;
Mapping from: ModTupFld: S to ModTupFld: W
> print comp([s1, s2, S.1, S.2]);
[
    (4 4),
    (0 1),
    (0 2),
    (3 1)
]
```

7.5 Recursive Definitions

MAGMA mappings and homomorphisms can be defined in a recursive manner. Within the right side of the constructor, the mapping or homomorphism being created is referred to with the `$` symbol.

For example, consider the problem of defining a mapping *facmap* on the non-negative integers which returns the factorial of the given number. This could be done most easily with the help of the MAGMA intrinsic **Factorial**(n),

but here a recursive method will be used. Since $n!$ can be defined with the rules that $0! = 1$ and that $n! = n(n-1)!$ for $n > 0$, *facmap* may be formed as follows:

```
> N := {! n in IntegerRing() | n ge 0 !};
> facmap := map< N -> N |
>     n :-> n eq 0 select 1 else n * $(n-1) >;
```

This mapping can be used like any other:

```
> print facmap(5), facmap(0), facmap(12);
120 1 479001600
> print facmap(-1);

>> print facmap(-1);
                ^
Runtime error in map application:
    Element is not in the domain of the map
```

Note that the domain, the formal set $\mathbb{N}$ of cardinal numbers, ensures that poorly-chosen argument values are handled appropriately.

As another example, consider the binomial coefficient $\binom{n}{r}$, where $n, r \geq 0$. It may be implemented as a function with two arguments; this is the approach taken by the intrinsic function **Binomial**(n, r), which should be employed for serious computations of binomial coefficients. However, it may also be seen as a mapping from the cartesian product $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. (In fact, **Binomial**(n, r) is defined for negative integers as well.) An arbitrary element of this cartesian product is a 2-tuple $u =$ `< `n, r` >` of non-negative integers. The mapping below uses the recurrence relation $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$:

```
> binomial := map< car< N, N > -> N |
>     u :-> n eq r select 1 else
>         n eq 0 select 0 else
>         r eq 0 select 1 else
>         ($(<n-1, r>) + $(<n-1, r-1>))
>         where n is u[1] where r is u[2] >;
> print <10, 7> @ binomial;
120
> print binomial(<10, 7>);
120
```

Strictly speaking, this mapping should be evaluated at a 2-tuple, as shown above, but in fact it can also be used with the same syntax as if it were a function with two arguments. If MAGMA encounters syntax such as $m(a_1, \ldots, a_k)$, where m is a map, it converts it to $m($`< `$a_1, \ldots, a_k$` >`$)$:

```
> print binomial(10, 7);
120
```

8. Functions and Procedures

There are three kinds of functions and procedures in MAGMA. Firstly, there are the *system intrinsics*, which are the routines supplied in MAGMA as specialist implementations of algorithms for various categories. These are the built-in functions and procedures that are explained and illustrated at many points in this book. The other two kinds are created by the user. *User-defined functions and procedures* are designed for a particular application. They are defined during a MAGMA session, either from interactive input at the keyboard or when an input file is loaded. Lastly, there are *user intrinsics*, which are user-defined functions and procedures that are so generally applicable or important to the user that they have been placed in a *package*, using special syntax, in order to be compiled and treated like system intrinsics. Chapter 10 explains how to construct user intrinsics. This chapter explains how to invoke (call) all kinds of functions and procedures, and how to create user-defined functions and procedures. See Chapter 10 for more information on system intrinsics and the creation of user instrinsics.

Functions and procedures in MAGMA are first-class objects, whether they are system intrinsics, user intrinsics in compiled packages, or ordinary user-defined functions and procedures as described in this chapter. Functions return one or more values, and procedures change the calling context. A function or procedure may have zero or more *value arguments*, and a procedure may also have zero or more *reference arguments*. Furthermore, a function or procedure may have zero or more *parameters*, with associated default values.

User-defined functions and procedures are created by means of an expression that evaluates to a function object or procedure object; this object is usually assigned to an identifier so it can be invoked easily. Recursion and forward declaration are permitted. There are two syntactic forms for user-defined functions: a **func**-constructor whose left side contains the formal arguments and whose right side contains expressions for the return values in terms of the arguments; and a more traditional form consisting of the formal arguments and then a body of statements, including a **return** statement with expressions for the return values. The syntax for user-defined procedures is similar: a **proc**-constructor whose left side contains the formal arguments and whose right side contains a procedure invocation in terms of the argu-

ments; and a more traditional form consisting of the formal arguments and then a body of statements. Non-local identifiers are visible, but cannot be reassigned. For the syntactic forms involving a body of statements, the scope of identifiers is determined by a first textual use rule, though explicit local declaration is permitted as a precaution.

System and user intrinsics have the same invocation syntax and semantics as user-defined functions and procedures, except that they perform type-checking on the arguments to category-level. For this reason, a single intrinsic name can stand for several functions or procedures with different numbers of arguments or types of arguments.

Functions and procedures serve a number of purposes. The system intrinsics provide access to efficient algorithms, and user intrinsics extend this repertoire. User-defined functions and procedures may be seen primarily as programming tools, with which to gather a sequence of statements into a discrete block of code, especially if the sequence of statements appears several times in a program, Large programs become more readable if they are presented in a modular fashion, with each major section written as a function or procedure.

8.1 Arguments and Invocations

Every function or procedure has zero or more *formal arguments*, in terms of which its action is described when it is defined (by the user or the system programmers). When the function or procedure is used by being *invoked* or *called*, *actual arguments* are supplied to be used as the values of the formal arguments. Hence the arguments are used for the transfer of data. At invocation, the actual arguments are given as expressions, in a comma-separated list enclosed in parentheses that follows the function or procedure.

The difference between functions and procedures is the effect they have when they are invoked. A *function call* returns one or more values, and therefore it is classed as an *expression*. It can be used like any other expression, in which case only the principal (first) return value will be used. It can also appear as the right side of a multiple assignment statement, if the function returns more than one value. By contrast, the effect of a procedure call is like the effect of several statements; it can change the values of identifiers, perform output and so on. For this reason, a *procedure call* in Magma forms a complete *statement* by itself.

The use of arguments and the difference between functions and procedures will be illustrated in terms of some system intrinsics. The same principles apply for user-defined functions and procedures.

As a function example, consider the system intrinsic **GCD**(a, b), which finds the greatest common divisor of integers a and b. **GCD** is the name of the function, and a and b are its formal arguments. The following are instances of function calls to **GCD**(a, b):

```
> c := GCD(49, 91);
> print c;
7

> x := 85; m := 15;
> print GCD(x, x - m);
5
```

In both cases, the function call is an expression consisting of the function followed by actual arguments in parentheses. These actual arguments are 49 and 91 in the first example, and the values of x and $x - m$ in the second example. The function call computes a *return value* that depends on the actual arguments, where the first argument corresponds to the formal argument a and the second argument corresponds to the formal argument b. When the execution of the function call has completed, the value of the function-call expression is the value returned by the function call. This is the only effect of the function call; in particular, it cannot change the values of the actual arguments.

As a procedure example, consider the system intrinsic **Rotate**$(\tilde{}Q, n)$, which rotates the sequence Q to the right by n places. The $\tilde{}$ signifies that Q is a *reference argument*, and is liable to be changed by the procedure. The other formal argument, n, is a *value argument*, like the arguments a and b of **GCD**; it cannot be changed by the procedure. When the procedure is called as shown below, the sequence *squares* of the first nine squares is rotated three places to the right:

```
> squares := [ i^2 : i in [1..9] ];
> print squares;
[ 1, 4, 9, 16, 25, 36, 49, 64, 81 ]
> Rotate(~squares, 3);
> print squares;
[ 49, 64, 81, 1, 4, 9, 16, 25, 36 ]
```

Notice that the procedure call forms a complete statement by itself. It does not return any values, but it does alter the value of the first argument.

In the case of an *actual reference argument*, the tilde must be followed by an identifier, not an arbitrary expression. It depends on the procedure as to whether the identifier needs to have been assigned a value before the procedure call; in either case, while the procedure call is being executed this

identifier corresponds to the formal reference argument in the code defining the procedure. All other procedure arguments, and all function arguments, are *actual value arguments*. Actual value arguments may be expressions, since it is their values alone which are passed to the formal arguments.

Unlike in some computer languages, functions and procedures in MAGMA are objects of *first-class* status, whether they are system intrinsics or defined by the user. This means that the system treats them like any other objects, such as integers or elements of a finite field; functions and procedures may be stored in identifiers, passed as arguments to other functions or procedures, and so on. Whenever a function (similarly a procedure) is assigned to an identifier, as normally happens when it is defined by in the user, that identifier has the function as its value. Then, when the function is called by means of that identifier and some actual arguments in parentheses, MAGMA uses the function which is the value of the identifier. However, a function (or procedure) may be called 'on the fly', without being assigned to an identifier first; that is, a function invocation can consist of any expression returning that function, followed by the actual arguments in parentheses.

It should be noted that functions bear some similarity to mappings (see Chapter 7), in that each of them operate on arguments and return values. The decision whether to implement an algorithm by means of a function or a mapping depends on the purpose of the algorithm: if it expresses a mathematical relation between two structures, a mapping should be used; if the algorithm is primarily a programming routine, a function is more appropriate. Mappings perform type-checking on the argument and the return value, to see that they conform to the domain magma and range magma, whereas type-checking is only to the category level for system and user intrinsics, and non-existent for simple user-defined functions (unless specially coded by the user). Mappings also have various mathematical operations such as composition defined for them. Note, in addition, that functions may return multiple values, whereas this is possible for mappings only in the sense of returning tuples (elements of Cartesian products).

8.2 User-Defined Functions

In MAGMA, a user-defined function is created as the value of a *function expression*, that is, an expression which evaluates not to an integer, a sequence, or so on, but to a function. There are two syntactic forms for function expressions in MAGMA.

8.2.1 Constructor Form of Function Expression

The **func**-constructor is the simplest form of function expression, and has the following syntax:

func< formal arguments | expressions for return values **>**

On the left of the | symbol are comma-separated identifiers for the formal arguments of the function, and on the right are comma-separated expressions in terms of the formal arguments for the values that the function returns.

Typically, an identifier name is chosen for the function. If so, an assignment statement is used:

identifier **:= func<** formal arguments | expressions for return values **>;**

Suppose, for instance, that a function is required to find the order of the general linear group, $\mathrm{GL}(n, \mathrm{GF}(q))$, for given values of n and q. As noted on p. 117, the order of this group is equal to $\prod_{i=1}^{n}(q^n - q^{i-1})$, so an expression for this value is `&*[q^n - q^(i-1) : i in [1..n]]`. A function allows the user to gain access to this expression easily, without typing it each time that the order is required. The following line shows how to construct a function *orderGL*(n, q) which returns the value of this expression:

```
> orderGL := func< n, q | &*[q^n - q^(i-1) : i in [1..n]] >;
```

The value of the identifier *orderGL* is the function, and the formal arguments of the function are n and q.

After *orderGL* has been defined as a function, it may be used to find the order of $\mathrm{GL}(n, \mathrm{GF}(q))$ for various n and q. For instance, the order of $\mathrm{GL}(3, \mathrm{GF}(4))$ may be printed:

```
> print orderGL(3, 4);
181440
```

and *ord32* may be assigned the order of $\mathrm{GL}(3, \mathrm{GF}(2))$:

```
> ord32 := orderGL(3, 2);
> // next line compares it with standard function Order
> print ord32 eq Order(GL(3, GF(2)));
true
```

To perform a call to this function, MAGMA evaluates each of the expressions given as actual arguments to the function, and substitutes the resulting

values for the corresponding formal arguments. Then the return value is computed and passed back. For instance, in the example below:

```
> p := 5;
> x := orderGL(3, p^2);
```

the first step is that the expressions 3 and `p^2` are evaluated as 3 and 25. Then within the function, n and q are given the values 3 and 25, so that the expression for the return value is `&*[25^3 - 25^(i-1) : i in [1..3]]`. This evaluates to 3656 016 000 000, which is assigned to x:

```
> print x;
3656016000000
```

If a function has to return more than one value, then expressions for each return value must be given on the right side of the **func**-constructor, separated by commas. For example, the following function $numden(f)$ returns the numerator and denominator of a given fraction f, where f is an element of the rational field or a function field:

```
> numden := func< f | Numerator(f), Denominator(f) >;
```

When this function is called, the values are returned in the same order in which they are given in the constructor. The return values may be printed directly, or may be assigned to two identifiers:

```
> print numden(1178612/671674);
589306 335837

> P<x> := PolynomialRing(IntegerRing());
> F<y> := FieldOfFractions(P);
> n, d := numden( (y^7 - 4*y^2 + 509) / (12*y^3 + 5*y) );
> print n;
x^7 - 4*x^2 + 509
> print d;
12*x^3 + 5*x
```

If the expressions for the return values involve a common sub-expression, it is possible to handle this using a **where**-clause that operates over an expression list. The **where** should be placed to the right of all the expressions to which it refers. For example, consider a function that returns the number of permutations and combinations of n objects taken r at a time, using the formulae

$$ {}^nP_r = \frac{n!}{(n-r)!} \qquad \text{and} \qquad {}^nC_r = \frac{n!}{(n-r)!\,r!}\,. $$

Such a function would have two formal arguments, n and r, and would return two values. The function below, *counting*(n, r), satisfies these conditions. Note that it extracts the common sub-expression $\frac{n!}{(n-r)!}$ using a **where** clause, so that it does not need to be found more than once. Note also that the divisions are calculated using the integer division operator **div**, since it is known that the divisors are factors of the dividends and the results should lie in the integer ring:

```
> counting := func< n, r | p, p div Factorial(r)
    where p is Factorial(n) div Factorial(n-r) >;
> pp, cc := counting(5, 2);
> print pp, cc;
20 10
```

The final example of a function expressed with a **func**-constructor is *ShuffleGroup*(n), which returns the shuffle group for $3n$ cards [MeM87, DGK83]. It concerns a deck of $3n$ cards which is shuffled according to certain rules. The pack is divided into three piles, each containing n cards, and then the piles are placed in a row in the order in which they were removed from the deck. Now the three piles are permuted among themselves, and finally the deck is re-assembled by taking the first card from the first pile, the second card from the second pile, and so on. The aim is to find the structure of the resulting 'shuffle group', a permutation group of degree $3n$. To find the generators p, q, s of the group, which correspond to two kinds of pile permutations and the interleaving step, *ShuffleGroup* constructs sequences. If the piles are a, b and c, then p and q correspond to permutations (a, c) and (a, b, c), and s corresponds to the interleaving step. (In suitable contexts, MAGMA can interpret sequences as permutations by considering the i^{th} term to be the image of the i^{th} element under the permutation; see Chapter 32 for details.)

To perform this example online, type `load "I96c8e1";`

```
> ShuffleGroup := func< n |
>   PermutationGroup< m |
>       &*[(i, i + 2 * n) : i in [1 .. n]],
>       &*[(i, i + n, i + 2 * n) : i in [1 .. n]],
>       [((i-1) mod 3)*n + (i-1) div 3 + 1 : i in [1..m]] >
>   where m is 3*n >;
```

For instance:

```
> print ShuffleGroup(5);
Permutation group acting on a set of cardinality 15
    (1, 11)(2, 12)(3, 13)(4, 14)(5, 15)
```

```
(1, 6, 11)(2, 7, 12)(3, 8, 13)(4, 9, 14)(5, 10, 15)
(2, 4, 10, 14, 12, 6)(3, 7, 5, 13, 9, 11)
```

The structure of shuffle groups is investigated further below.

8.2.2 Statement Form of Function Expression

In the *orderGL* example above, the return expression of the function was written directly in terms of the formal arguments. If a function is being defined for which it is impossible or awkward to do this, then a more general form of function expression should be used instead of the **func**-constructor. The statement form of function expression permits the return value(s) to be calculated using several intermediate statements. It has the syntax

function(formal arguments)
  statements
end function

The body of the function may contain any number of statements, provided that when MAGMA is executing a call to this function, it will come to a line of the form

return expression, ..., expression;

somewhere in the statement body. As soon as MAGMA encounters a **return**-statement like this, it will evaluate the expression(s) and return the result(s). Then the function call is finished, and any remaining lines in the function body will be ignored.

As simple examples, *orderGL* and *counting* may be created using this form of function expression as follows:

```
> orderGL := function(n, q)
>     return &*[q^n - q^(i-1) : i in [1..n]];
> end function;

> counting := function(n, r)
>     return p, p div Factorial(r)
>       where p is Factorial(n) div Factorial(n-r);
> end function;
```

These versions of the functions perform in the same way as those created using the **func**-constructor. However, they are longer, and do not increase the legibility of the functions. The statement form of a function expression

is more appropriately used when the algorithm requires iterative statements (see Chapter 9) or when the expressions are so complicated that it would be difficult to read the function if it did not assign some intermediate values to identifiers.

For instance, consider the problem of determining for which values of n the shuffle group (as defined above) is symmetric, alternating, or neither. If the group is symmetric, then the shuffle is 'fair' or 'perfect' for that value of n, in the sense that any resulting permutation of the cards is possible after sufficiently many shuffles. The function *FairShuffles*(L) tests each shuffle group from $n = 1$ up to $n = L$ in this way, for some given upper limit L. It uses an iterative statement (a **for** loop) to iterate over n and test the structure of each resulting shuffle group. The values of n are collected into three sets, according to whether the group is symmetric (i.e., a fair shuffle), alternating, or neither symmetric nor alternating. The function returns these sets, and its principal return value is the set of n for the fair case.

```
> FairShuffles := function(L)
>     SymShuffles := { }; AltShuffles := { };
>     NotSymAltShuffles := { };
>     for n in [1..L] do
>         sh := ShuffleGroup(n);
>         if IsSymmetric(sh) then
>             Include(~SymShuffles, n);
>         elif IsAlternating(sh) then
>             Include(~AltShuffles, n);
>         else
>             Include(~NotSymAltShuffles, n);
>         end if;
>     end for;
>     return SymShuffles, AltShuffles, NotSymAltShuffles;
> end function;
```

If the upper limit is set at 30, then the result is:

```
> fair, alt, other := FairShuffles(30);
> print fair;
{ 1, 2, 5, 6, 7, 10, 11, 13, 14, 15, 17, 18, 19, 21,
22, 23, 25, 26, 29, 30 }
> print alt, other;
{ 4, 8, 12, 16, 20, 24, 28 }
{ 3, 9, 27 }
```

From the resulting sets, it is easy to form a conjecture about how the structure of the shuffle group depends on n. MAGMA is very helpful in this way as an adjunct to research.

8.3 User-Defined Procedures

There are two ways to define a procedure. These methods correspond to the syntax for user-defined functions: the constructor form and the statement form. Both of these are expressions which evaluate to a procedure; the value of the expression is usually assigned immediately to an identifier.

8.3.1 Constructor Form of Procedure Expression

The constructor form of the procedure expression is discussed only briefly here, because it is rarely used. The **proc**-constructor has the following syntax:

proc< formal arguments | procedure call **>**

On the left of the | symbol are comma-separated identifiers for the formal arguments (value and/or reference), and on the right is an procedure invocation in terms of the formal arguments. If the procedure is being assigned to an identifier, then the assignment statement will have the following form:

identifier **:= proc<** formal arguments | procedure call **>**;

The constructor form of procedure expression is rather limited in its applicability because the right side of the constructor has to be a single procedure call.

For example, suppose that a procedure *ShiftR*($\tilde{}Q$) is required that modifies Q by shifting it one place to the right. *ShiftR* may be defined by invoking the system intrinsic procedure **Rotate**($\tilde{}Q, n$) on the right side of a **proc**-constructor:

```
> ShiftR := proc< ~Q | Rotate(~Q, 1) >;

> V := VectorSpace(GF(5), 7);
> b := Eltseq(V.2);
> print b;
[ 0, 1, 0, 0, 0, 0, 0 ]
> ShiftR(~b);
> print b;
```

```
[ 0, 0, 1, 0, 0, 0, 0 ]
> print V!b;
(0 0 1 0 0 0 0)
```

8.3.2 Statement Form of Procedure Expression

The syntax of the statement form of a procedure expression is:

procedure(formal arguments)
  statements
end procedure

where each formal argument may be a reference argument or a value argument; every reference argument must be preceded by a tilde.

For example, the procedure *ShiftR* may be created in statement form as follows:

```
> ShiftR := procedure(~Q)
>     Rotate(~Q, 1);
> end procedure;
```

A common use for a procedure is to print output. The following trivial example has no arguments, and always performs the same task, printing a line of sixty asterisks:

```
> SeparatingLine := procedure()
>     print "*"^60;
> end procedure;
> SeparatingLine();
************************************************************
```

A more flexible procedure would be one with two value arguments c and n that allow the procedure call to specify which character c is to be printed and the number n of such characters. It could be used as a separating line for part of a printing routine.

```
> SeparatingLine := procedure(c, n)
>     print c^n;
> end procedure;
> SeparatingLine("#", 35);
###################################
```

As a longer example, the procedure *EuclideanAlgorithm*(a, b) below has two formal value arguments, a and b. It prints the steps of the Euclidean

algorithm for the greatest common divisor (gcd) as applied to a and b. This procedure uses the formatted print statement, **printf**, so as to describe the appearance of the output more easily, and it performs an iteration by means of the **repeat** ... **until** statement (see Section 9.3).

```
> EuclideanAlgorithm := procedure(a, b)
>     // make copies A and B of a and b (with A ge B)
>     if a ge b then
>         A := a; B := b;
>     else
>         A := b; B := a;
>     end if;
>     repeat
>         q, r := Quotrem(A, B);
>         if r eq 0 then
>             printf "%o = %o * %o\n", A, q, B;
>         else
>             printf "%o = %o * %o + %o\n", A, q, B, r;
>             A := B; B := r;
>         end if;
>     until r eq 0;
> end procedure;
```

When this procedure is invoked in such a way that the actual value arguments corresponding to a and b are integers, it prints the Euclidean algorithm for those integers:

```
> EuclideanAlgorithm(120, 531);
531 = 4 * 120 + 51
120 = 2 * 51 + 18
51 = 2 * 18 + 15
18 = 1 * 15 + 3
15 = 5 * 3
```

8.3.3 Reference Arguments in a Procedure

Reference arguments are available in procedures but not in functions. By means of reference arguments, the procedure can pass values back to the calling context of the procedure. For this reason, the actual reference argument (preceded by the ~ symbol) must be an identifier, not an arbitrary expression as for an actual value argument.

There are two ways in which reference arguments may be used, according to the way the procedure is designed. In some cases, the actual reference

argument is expected to have a value assigned to it before the procedure call begins. This value will be used during the procedure call. By the time that the execution has finished, it is possible that the corresponding formal reference argument will have had a new value assigned to it; if so, the actual reference argument will take on this value. Otherwise, the actual reference argument will retain the value it had before. In other cases, the actual reference argument is merely an identifier name that does not have an assigned value. In this situation, the corresponding formal reference argument should not be used within the procedure body until it has had a value assigned to it. At the end of the procedure invocation, the final value of this formal reference argument will be given to the actual reference argument.

As an example of the first case, consider *ShiftR*(˜Q) defined above. In order to be executed successfully, this procedure requires the actual reference argument corresponding to Q to be an identifier that has a value (in particular, a sequence) assigned to it. After the procedure invocation, the actual reference argument will in general have a new value.

As an example of the second case, in which the actual reference argument does not have an initial value, consider a modification of *EuclideanAlgorithm*. Since the purpose of the Euclidean algorithm is to calculate the gcd, this procedure can easily be modified so that it returns the gcd via a reference argument g:

```
> EuclideanAlgorithm := procedure(a, b, ~g)
>     // make copies A and B of a and b (with A ge B)
>     if a ge b then
>         A := a; B := b;
>     else
>         A := b; B := a;
>     end if;
>     repeat
>         q, r := Quotrem(A, B);
>         if r eq 0 then
>             printf "%o = %o * %o\n", A, q, B;
>             g := B; // store the gcd in g
>         else
>             printf "%o = %o * %o + %o\n", A, q, B, r;
>             A := B; B := r;
>         end if;
>     until r eq 0;
> end procedure;
```

When the procedure *EuclideanAlgorithm*$(a, b, \tilde{}g)$ is called, the actual reference argument must be an identifier (d, say). The gcd will be stored in d at the end of the procedure invocation. Since the body of the procedure only

makes an assignment to g, and does not attempt to use its value prior to the call, it does not matter whether d has a value before the procedure call, but if it does, that value will be destroyed:

```
> d := "junk";
> EuclideanAlgorithm(531, 120, ~d);
531 = 4 * 120 + 51
120 = 2 * 51 + 18
51 = 2 * 18 + 15
18 = 1 * 15 + 3
15 = 5 * 3
> print d;
3
```

In some other computer languages, it is advisable to use reference arguments rather than value arguments when passing values requiring a great deal of storage space. However, this technique is of no benefit in MAGMA, because the system uses reference counts to avoid making unnecessary copies of value arguments.

8.3.4 Early Return from a Procedure Invocation

Since a procedure does not have return values in the sense of a function's return values, it should not contain any statements of the form:

return expression, ..., expression;

However, it may contain one or more lines of the form:

return;

The command **return**, when followed by a semicolon, instructs MAGMA to stop executing the procedure call immediately, without progressing through the rest of the procedure body down to **end procedure**.

For instance, *EuclideanAlgorithm* could be written in this way:

```
> EuclideanAlgorithm := procedure(a, b, ~g)
>     // make copies A and B of a and b (with A ge B)
>     if a ge b then
>         A := a; B := b;
>     else
>         A := b; B := a;
>     end if;
```

```
>     repeat
>         q, r := Quotrem(A, B);
>         if r eq 0 then
>             printf "%o = %o * %o\n", A, q, B;
>             g := B; // store the gcd in g
>             return;
>         end if;
>         printf "%o = %o * %o + %o\n", A, q, B, r;
>         A := B; B := r;
>     until r eq 0;
> end procedure;
```

Both versions are quite suitable in this case, but the **return** version has the advantage that less indentation is required.

8.4 Local and Non-Local Identifiers

At this point it may seem that there is potential for great confusion between identifiers of the same name residing inside and outside of a function/procedure. What would happen, for instance, if when *EuclideanAlgorithm* was called the identifiers a or A were already defined? Would it affect the outcome of the procedure invocation, and would the values of these identifiers be preserved? MAGMA has a simple principle known as the *first textual use rule* to prevent problems like this. Any identifier which is a formal argument, or whose first appearance in the text of the function expression is for the purpose of having a value assigned to it, is *local* to the function. This means that if there are identifiers with the same name existing outside the function, they are not changed by a function call, and on the other hand, after the function call has finished execution, the function's identifiers effectively disappear and those outside the function reappear.

The following lines illustrate this notion of local scope:

```
> a := 17; b := 62; c := 54;
> multip := function(a, b)
>     c := a * b;
>     return c;
> end function;
> print multip(8, 9);
72
> print a, b, c;
17 62 54
```

Although the formal identifiers a and b are given the values 8 and 9 within the scope of the function, the values of the identifiers a and b outside the function do not change. Similarly, although the identifier c inside the function is given the value 72, the value of the identifier c outside the function does not change. Therefore anyone can use the function *multip* by knowing merely that it returns the product of its two arguments. As is appropriate, the internal workings of the function are irrelevant to its use.

Identifiers in functions are not always local. If the first time that an identifier appears is when it is referred to in an expression rather than when it is assigned, then MAGMA looks for its value outside the function. For example:

```
> sarah := 42;
> silly := function(blanche)
>     diana := sarah * (3 - blanche);
>     return diana^2 + sarah;
> end function;
> print silly(6);
15918
```

The first textual use of *sarah* in the function body of *silly* is in the expression *sarah*$*(3 - $*blanche*$). Therefore *sarah* is a non-local identifier, and MAGMA will use the value which *sarah* has been assigned outside the function. The result is that *silly* is defined exactly as if it had been:

```
> silly := function(blanche)
>     diana := 42 * (3 - blanche);
>     return diana^2 + 42;
> end function;
```

Note that if the value of *sarah* is now changed, the function *silly* will remain as it was before, although its definition included the use of *sarah*. Only the *current values* of identifiers are relevant when a function is being defined:

```
> sarah := 100000;
> print silly(6);
15918
```

It is generally unwise to use non-local identifiers within a function. The danger is that the function could be defined at a time when its non-local identifiers are set incorrectly; this can easily happen, especially when a function is in the development stage. A non-local identifier should only be used in a function definition when it can be guaranteed that its value will be as expected. For instance, the identifier might have been given its value only a short while before, or it might be an identifier that the user is in the habit of using for special purposes, such as Z for **IntegerRing**() or R for **RealField**().

Non-local identifiers cannot be reassigned within a function body. In this, they resemble loop identifiers in **for**-statements. MAGMA sees both of these as members of the *value class of identifiers*, rather than as 'variable identifiers' in the sense of objects that can be varied by the user.

It should be noted that formal value arguments are local identifiers within a function body, and may be re-assigned. For example:

```
> blanche := "bob";
> silly := function(blanche)
>     blanche := 3 - blanche;
>     diana := 42 * blanche;
>     return diana^2 + 42;
> end function;
> print silly(6), blanche;
15918 bob
```

8.5 Functions and Procedures as First-Class Objects

Functions and procedures are *first-class objects*. They can be assigned to identifiers, as shown throughout this chapter, but they can also be passed into functions and procedures, or obtained as return values of functions. For instance, the following function *trapezoid* calculates an approximation to the integral of a single-argument real function f from a to b, using one application of the trapezoidal rule:

```
> trapezoid := func< f, a, b | (b-a)/2 * (f(a) + f(b)) >;
> print trapezoid(Sin, 2, 5);
-0.07444027175618516025
> myfunc := func< x | x^3 - Log(x) >;
> print trapezoid(myfunc, 1, 10);
4494.1383670815267944
```

The example shows *trapezoid* being applied to the system intrinsic function **Sin** and then to the user-defined function *myfunc*.

8.6 Recursion

A function or procedure is allowed to invoke itself within its own definition. This phenomenon is known as *recursion*. Recursion is used when it is difficult to solve a problem directly, but it is possible to reduce it to a slightly smaller

problem of the same kind. One then reduces the slightly simpler problem, and so on until arriving at a problem which is so simple that it can be solved directly.

For example, suppose Q is a non-empty sequence of integers and it is necessary to find the smallest term, m. The system intrinsic function **Minimum**(Q) performs this task exactly, but it is worth exploring how to write a recursive function for a situation such as this. The function will use a less powerful version of the system intrinsic, the **Minimum**(a, b) function, which returns the smaller of the two numbers a and b.

A recursive function *smallest*(Q) may be written for this task using either the constructor form or the statement form of a function expression. The statement form will be used here, so that the parts of the function can be displayed more easily.

The function should begin with a heading that gives Q as the formal identifier:

```
> smallest := function(Q)
```

Next, it is necessary to consider when the problem can be solved directly. In this example, finding the minimum term of Q is trivial when Q has only one term, so the body of the function can begin as follows:

```
>     if #Q eq 1 then
>         return Q[1];
```

The function body must continue with instructions in the case that Q has more than one term. The insight to be had here is that to find the minimum of a sequence of size n, one can take the last term from the sequence, find the minimum of the remaining sequence of size $(n - 1)$, and compare this minimum with the removed term by means of **Minimum**(a, b). This observation is useful because it reduces the problem to that of finding the minimum of a sequence which has one term fewer. Finding the minimum of this smaller sequence can be done by first finding the minimum of a sequence which is smaller again, and so on until the sequence is reduced to size 1, a case which has already been covered by the **if**–**then** section. The **else**-clause below encodes this insight:

```
>     else
>         finalterm := Q[#Q];
>         prunedmin := $$(Prune(Q));
>         return Minimum(finalterm, prunedmin);
>     end if;
```

The `$$` sign in the above stands for the function currently being defined, in this case *smallest*. It is not possible to use the name *smallest* at this point, because within the statement body of the function expression the identifier *smallest* is undefined (unless it happens by coincidence to have a value assigned to it already). Only after MAGMA has processed the whole function expression can it assign the function to *smallest*. (See p. 188 for an exception to this.)

The full function *smallest* is:

```
> smallest := function(Q)
>     if #Q eq 1 then
>         return Q[1];
>     else
>         finalterm := Q[#Q];
>         prunedmin := $$(Prune(Q));
>         return Minimum(finalterm, prunedmin);
>     end if;
> end function;
```

For example:

```
> print smallest([5, 2, 4, 4, 65, 31]);
2
```

The `$$` symbol is also recognized within the **func**-constructor. For example, the function *smallest*(Q) may be defined using **func**, as remarked previously:

```
> smallest := func< Q |
>      #Q eq 1 select Q[1]
>          else Minimum(Q[#Q], $$(Prune(Q))) >;
```

Recursive functions and procedures are characterized by this elegance, but they must be written with care. The user must ensure that the recursion will eventually terminate, irrespective of the values of the actual arguments.

8.7 The forward-declaration

Suppose that a function/procedure f is about to be defined which invokes a function/procedure according to the identifier g. If g has not yet been assigned a function/procedure, MAGMA will give an error message when it encounters the use of g within the definition of f. To prevent this error, it is possible to type

```
> forward g;
```

before the definition of f begins. This **forward**-declaration is an indication to MAGMA that g will be assigned a function/procedure later. Any number of functions/procedures may now be *defined* with expressions involving invocations of g. However, the system expects that before the *invocation* of f (or any other function/procedure whose definition involves g), g will have been assigned.

The **forward**-declaration must occur on the main level, not within a function/procedure definition. If several functions/procedures are to have these declarations, they may either have separate **forward**-declarations or be given in a comma-separated list following the word **forward**.

There are two major applications for the **forward**-declaration. The first usage is a matter of convenience, but the second one is a matter of necessity. Firstly, some programmers like to place their principal programs before the smaller programs that support them (in a file, for instance). If they wish to be able to order their functions/procedures in this way, they must precede the definition of the principal functions/procedures with **forward**-declarations for the other functions/procedures. Here is an example in miniature; it calculates the number of steps required in the $3n+1$ problem to reduce a positive integer n to 1, following the rule that if n is odd, then $n \mapsto 3n+1$, and if n is even, $n \mapsto \frac{n}{2}$. The principal function is *Nsteps*(n), and it is supported by the functions *oe*, *odd*, and *even*. Note also that *Nsteps* is recursive.

```
> forward oe, odd, even;
> Nsteps := func< n | n le 1 select 0 else 1 + $$(oe(n)) >;
> oe := func< n | IsOdd(n) select odd(n) else even(n) >;
> odd := func< n | 3*n + 1 >;
> even := func< n | n div 2 >;
>
> print Nsteps(34), Nsteps(871);
13 178
```

The second usage for the **forward**-declaration is in cases of *mutual recursion*. If f and g are mutually recursive functions/procedures, that is, if they invoke one another, the user must make a **forward**-declaration of g, then define f, then define g [or vice versa]. The functions/procedures f and g should be designed so that the mutual recursion terminates eventually.

For example, the following function *isPrime*(n) is a [rather poor] imitation of the system intrinsic **IsPrime**(n). For odd $n > 3$ it first checks for a few (3) random values of an integer a in the range 2 to $n-2$ that $a^{n-1} \equiv 1 \pmod{n}$. If this is false, then n cannot be prime; if it is true, then a primality prover *strongTest*$(primdiv, n)$ is applied, where *primdiv* is the sequence of

prime divisors of $n-1$. It uses the theorem that n is prime if there exists an integer x in the range 2 to $n-1$ such that $x^{n-1} \equiv 1 \pmod{n}$ and $x^{\frac{n-1}{p}} \not\equiv 1 \pmod{n}$ for all prime divisors p of $n-1$. A third function, *primeDivisors*(n), is needed to find *primdiv*. This function is both self-recursive and mutually recursive with *isPrime*. In the ordering of the functions below, there is a **forward**-declaration of *primeDivisors*, followed by the definition of *isPrime*, followed by the definition of *primeDivisors*:

```
> strongTest := func< primdiv, n |
>     exists{ x : x in [2..n-1] |
>       Modexp(x, n-1, n) eq 1 and
>       forall{ p : p in primdiv |
>         Modexp(x, (n-1) div p, n) ne 1 } } >;
> forward primeDivisors;
> isPrime := func< n |
>     n in { 2, 3 } or IsOdd(n) and
>     forall{ i : i in [1..3] |
>       Modexp(Random(2, n-2), n-1, n) eq 1 }
>     and strongTest(primeDivisors(n-1), n) >;
> primeDivisors := func< n |
>     isPrime(n) select { n } else
>       ($$(d) join $$(n div d) where d is
>         rep{ d : d in [2..Isqrt(n)] | n mod d eq 0 }) >;
>
> print isPrime(137821), isPrime(12853);
false true
```

8.8 Function Assignment and Procedure Assignment

Functions and procedures both have variants of the statement form. Instead of beginning a definition with the line

R := **function**(formal arguments)

or

R := **procedure**(formal arguments)

the user can start with

function R(formal arguments)

or

procedure R(formal arguments)

and then continue as explained previously. These construtions are the *function assignment statement* and *procedure assignment statement*. Some users may find this syntax more convenient, particularly if they are accustomed to it from other programming languages. The effect in MAGMA is the same, except for a slight difference in recursive routines. If this form is used, then within the function or procedure definition the routine may be called by its name R instead of by the pseudo-name `$$`.

In more detail, when MAGMA encounters a function assignment statement (similarly for a procedure), it internally transforms it into the assignment of the value of a function expression to the given function name. Before assigning the expression's value, it rewrites any instances of the function name within the function expression as `$$`, so that a recursive function or procedure's own name may be used during its definition by this method.

For example, the function *smallest* defined on p. 185 may be redefined using this technique as follows:

```
> function smallest(Q)
>     if #Q eq 1 then
>         return Q[1];
>     else
>         finalterm := Q[#Q];
>         prunedmin := smallest(Prune(Q));
>         return Minimum(finalterm, prunedmin);
>     end if;
> end function;
```

8.9 Error Messages

Users designing a function or procedure must take care that it can cater for all kinds of input in the actual arguments. Otherwise the behaviour will be unpredictable. The routine may stop, with a confusing or misleading error message, or (even worse) it may complete its execution but give an undesired result.

MAGMA has an **error**-statement which can be used to assist in this task. The form of the **error**-statement is:

error expression, . . . , expression;

If MAGMA encounters an **error**-statement while executing a procedure call or function call, it prints the values of the expressions, as it would for a **print**-statement, and also breaks out of the call.

A related statement is the **error if** statement, with the form:

error if condition, expression, ..., expression;

If MAGMA encounters this statement while executing a procedure call or function call, it evaluates the Boolean condition immediately after **error if**. If the condition is **false**, MAGMA ignores the rest of the statement. However, if it is **true**, then MAGMA acts as for a plain **error**-statement, by printing the values of the expressions, and breaking out of the call.

If a function invocation is being executed and MAGMA encounters either an **error**-statement or an **error if** with a **true** condition, then the function call does not return a value.

As an example of error-handling, consider *SeparatingLine*(c, n) defined on p. 177. Some improvements can be made to this procedure by testing for likely errors. As it stands at present, if the user confuses the order of the arguments and types

```
> SeparatingLine(10, "!");

>> print c^n;
          ^
Runtime error in '^': Bad argument types
```

then the error message is very unhelpful for a user who has never seen the procedure definition body. It would be better to check the categories of the arguments as soon as execution of the procedure call commences. This also applies to the following call, where the ^ operator happens to work but does not perform the intended string concatenation:

```
> S6<a, b> := Sym(6);
> SeparatingLine(b, a^3);
(4, 5)
```

Another possible error that should be detected is the entry of an unsuitable integer for n. The procedure should check that n is non-negative, and less than some reasonable line-length bound such as 80.

With these ideas in mind, the revised procedure can be either of these (demonstrating both kinds of error statements):

```
> SepLine := procedure(c, n)
```

```
>     if Type(c) ne MonStgElt or Type(n) ne RngIntElt then
>         error "Error in 'SepLine': Bad argument types";
>     elif n notin [0..80] then
>         error "Error in 'SepLine': Argument 2 (", n,
>           ") not in range [0..80]";
>     else
>         print c^n;
>     end if;
> end procedure;

> SepLine := procedure(c, n)
>     error if Type(c) ne MonStgElt or Type(n) ne RngIntElt,
>         "Error in 'SepLine': Bad argument types";
>     error if n notin [0..80],
>         "Error in 'SepLine': Argument 2 (", n,
>           ") not in range [0..80]";
>     print c^n;
> end procedure;
```

These two procedures have the same effect, but **error if** is preferable since this kind of error situation can be detected with a simple test of a condition.

SepLine (either version) works in the same way as *SeparatingLine* if the input is legal, but otherwise it produces error messages such as the following:

```
> SepLine(10, "!");
Error in 'SepLine': Bad argument types
> SepLine("$", 2000);
Error in 'SepLine': Argument 2 ( 2000 ) not in range [0..80]
```

Further error checks might also be desirable, such as ensuring that c has only one character.

Not all runtime errors can be found so easily. The ones shown above could be detected at the beginning of the call, but sometimes errors can only be found after several of the computations in the routine have taken place. Functions and procedures should be robust enough to deal intelligently with errors such as these as well.

If a function or procedure is sufficiently useful to be converted into a user intrinsic, then certain aspects of error-checking become easier. Category-checking of arguments becomes automatic, and there are additional statements for error-handling. See Section 10.8.

8.10 Parameters

Many of MAGMA's system intrinsics that encode the more complicated algorithms have *parameters* available. User-defined functions and procedures and user intrinsics may also have parameters. Parameters are like optional arguments, whose purpose is to control some aspect of the algorithm's execution. This kind of control could be maintained with ordinary value identifiers, but the advantage of a parameter is that it need only be referred to in a function or procedure call if the user wishes its value to be other than the standard or *default* value. For every parameter in an algorithm, the algorithm designer must decide what the most often desired value of the parameter will be, and specify it as the default value. If the function or procedure call includes an assignment to the parameter, the given value will be used for it; otherwise, the default value will be used.

Parameter names follow the rules of identifier names. They are specific to the function or procedure, and have local scope. Thus if a function/procedure has a parameter with the same name as a current identifier, there will be no confusion between their values; nor will there be confusion with another function/procedure with the same parameter name.

For each system intrinsic, the online help system includes the name and default value of every parameter, a list or description of the parameter's possible values, and a description of how the values of the parameters affect the behaviour of the function or procedure.

8.10.1 Parameters in Function or Procedure Calls

The syntax for a call to a function or procedure R that has parameters is:

R`(`arguments `:` parameter `:=` expr`, ...,` parameter `:=` expr`)`

Observe that all the actual value and reference arguments are given, then a colon, then a comma-separated list of the parameters and their values. It is only necessary to list those parameters which are being given non-default values (though it is permissible to list them even if the values are default). The order in which the parameters and their values are listed is not important.

For instance, the system intrinsic function **IsSubsequence**(Q, T) returns **true** if Q is a subsequence of T, else **false**. However, since there is some flexibility in the definition of a subsequence, the user may specify what kind of subsequence is intended using the parameter **Kind**. The default value of **Kind** is `"Consecutive"`, meaning that the elements of Q must occur consecutively in T. Therefore the function **IsSubsequence** may be used

without reference to the parameter **Kind** when a consecutive subsequence is intended:

```
> F3<a, b, c> := FreeGroup(3);
> T := [ a*b, b^2, a*c^-5, c*a*b, Id(F3), b^-2 ];
> print IsSubsequence([ b^2, a*c^-5 ], T);
true
> Q := [ a*b, a*c^-5 ];
> print IsSubsequence(Q, T);
false
```

The other possible values for the parameter **Kind** are `"Sequential"`, meaning that the elements of Q must occur in T in the same indexing order, and `"Setwise"`, meaning that the elements of Q must occur in T but that the indexing order is irrelevant. If one of these parameter values is required, the actual arguments in the function call must be followed by a colon, then the parameter name, an assignment symbol (`:=`), and the value of the parameter. For instance:

```
> print IsSubsequence(Q, T : Kind := "Sequential");
true
```

As an example of a function with several parameters, consider **Factor**(n), which attempts to find a proper factor q of n. (See Section **??** for a full explanation of this system intrinsic.) **Factor** has several parameters, the most important being **Al**, which specifies the particular factorization method or algorithm to be used within the overall algorithm. One of the possible values of **Al** is `"Division"`, the trial division method:

```
> print Factor(143 : Al := "Division");
11 13
```

If **Al** is assigned this value, there are two other parameters used by **Factor** as well, **Lower** and **Upper**, giving the lower and upper bounds for the search for divisors. (**Factor** has additional parameters, suitable for different choices of **Al**, but their values will be ignored in this case.) The default value of **Lower** is 2, and the default value of **Upper** is $\min(\mathbf{Lower}+1000, \sqrt{n})$. These default bounds are chosen so that every moderately small factor will be found and the execution time will not be large. However, for a large n the function may fail to find a proper factor q; this is indicated by returning 1 and n instead of q and $\frac{n}{q}$. If this occurs, the user may wish to increase **Upper**. For instance:

```
> n := NextPrime(1000) * NextPrime(1020);
> print Factor(n : Al := "Division");
1 1030189
```

```
> // increase Upper
> print Factor(n : Al := "Division", Upper := 1100);
1009 1021
```

8.10.2 Parameters in Function or Procedure Definitions

All the methods of construction for user-defined functions and procedures may be adapted to cater for parameters. The syntax is similar to that used in the function/procedure call: following the formal arguments comes a colon, and then a comma-separated list of parameter assignments. The values assigned to the parameters are the default values.

A simple example of parameters in a user-defined procedure may be given by a modification of the earliest version of *SeparatingLine* (p. 177). The function below prints k lines of output, consisting of n copies of the character c. The identifier k is a value argument, but c and n are parameters, with default values `"*"` and 60:

```
> SeparatingBlock := procedure(k : c := "*", n := 60)
>     for i in [1..k] do
>         print c^n;
>     end for;
> end procedure;

> SeparatingBlock(3);
************************************************************
************************************************************
************************************************************
> SeparatingBlock(2 : n := 13,  c := "&");
&&&&&&&&&&&&&
&&&&&&&&&&&&&
> SeparatingBlock(4 : c := "#");
############################################################
############################################################
############################################################
############################################################
```

As an example of parameters in a **func**-constructor, consider the function *trapezoid*(f, a, b) given on p. 183, which returns an approximation to $\int_a^b f(x)\,dx$ using one application of the trapezoidal rule. It may be easily be extended to use n applications of the rule, where n is a positive integer. There are two ways to program this modification. Either the function can be created with another value argument n, or it can be created with n as a parameter.

Both options are shown below; for the parameter version, the default value of n is 10:

To perform this example online, type `load "I96c8e2";`

```
trapezoidN1 := func< f, a, b, n |
    h/2 * ( f(a) + f(b) +
      2 * &+[ RealField() | f(a + k*h): k in [1..(n-1)]] )
        where h is (b - a) / n >;

trapezoidN2 := func< f, a, b : n := 10 |
    h/2 * ( f(a) + f(b) +
      2 * &+[ RealField() | f(a + k*h): k in [1..(n-1)]] )
        where h is (b - a) / n >;

> print trapezoidN1(Sin, 2, 5, 10);
-0.69455256458535234641769462139071206013 0
> print trapezoidN1(Sin, 2, 5, 3);
-0.64049591116678941602046156189469259028 4

> print trapezoidN2(Sin, 2, 5);
-0.69455256458535234641769462139071206013 0
> print trapezoidN2(Sin, 2, 5 : n := 3);
-0.64049591116678941602046156189469259028 4
```

The decision as to whether n should be a parameter or a value argument depends on whether a default value can be found for n that would be the desired value most of the time.

Now, another common method of quadrature (approximate integration) is Simpson's rule. The function *quadrature* below provides access to both the trapezoid rule and Simpson's rule. It has two parameters: Al, a string which takes the value `"Trapezoid"` (the default) or `"Simpson"`; and n, a positive integer (default 10) giving the number of applications of the rule. The function includes some tests to see that the values given to the parameters in the function call are legal; for brevity, the legality of the actual arguments is assumed.

To perform this example online, type `load "I96c8e3";`

```
> quadrature := function( f, a, b : Al := "Trapezoid", n := 10)
>     // Check that parameter values are legal
>     error if Category(n) ne RngIntElt or n lt 1,
>         "Error: n not a positive integer";
>     error if Category(Al) ne MonStgElt
```

```
>       or Al notin {"Trapezoid", "Simpson"},
>         "Error: Al must be \"Trapezoid\" or \"Simpson\"";
>
>     R := RealField();
>     h := (b - a) / n;
>     if Al eq "Trapezoid" then
>         return h/2 * ( f(a) + f(b) +
>           2* &+[ R | f(a + k*h): k in [1..(n-1)]] );
>     else // "Simpson" method
>         return h/3 * ( f(a) + f(b) +
>           4 * &+[ R | f(a + k*h): k in [1..(n-1) by 2]] +
>           2 * &+[ R | f(a + k*h): k in [2..(n-2) by 2]] );
>     end if;
> end function;
```

This function may be used as follows:

```
> print quadrature(Sin, 2, 5);
-0.69455256458535234641769462139071206013 0
> print quadrature(Sin, 2, 5 : Al := "misprint");
Error: Al must be "Trapezoid" or "Simpson"
> print quadrature(Sin, 2, 5 : Al := "Simpson");
-0.69984085404238731047751892806132063282 3
> print quadrature(Sin, 2, 5 : n := 100);
-0.69975653554641585327802399258554846861 3
> print quadrature(Sin, 2, 5 : Al := "Simpson", n := 100);
-0.69980902515984669032017474218789230626 3
> // compare with use of exact integral
> print -Cos(5) + Cos(2);
-0.69980902201036865146420740109
```

Note that MAGMA has an intrinsic **Integral**(f, a, b) which performs a similar task. However, the first argument in **Integral** is a map, not a function.

8.11 The assert-statement

The **assert**-statement has the following syntax:

assert condition;

where the condition is an expression that evaluates to a Boolean. The effect of this statement depends on the value of the condition when the statement is executed: if it is **true**, MAGMA does nothing; if it is **false**, MAGMA stops execution and gives an error message, stating that the assertion failed.

This statement is provided as a preventative against errors in programming. If it is included in the statement body of a function or procedure definition, then the assertion made in the condition will be checked each time it is encountered during an invocation of that function/procedure. The assertion should express a fact that must be true at that point in the statement body.

For example, in the *EuclideanAlgorithm* procedure discussed in earlier sections of this chapter, the value of $\gcd(a, b)$ is calculated by means of the Euclidean algorithm and assigned to g. It would be wise to check that the implementation of the algorithm is correct by calculating the greatest common divisor another way, with the kernel intrinsic function **GCD**(a, b), and asserting that this value equals g:

```
> EuclideanAlgorithm := procedure(a, b, ~g)
>     // make copies A and B of a and b (with A ge B)
>     if a ge b then
>         A := a; B := b;
>     else
>         A := b; B := a;
>     end if;
>     repeat
>         q, r := Quotrem(A, B);
>         if r eq 0 then
>             printf "%o = %o * %o\n", A, q, B;
>             g := B; // store the gcd in g
>             assert g eq GCD(a, b);
>             return;
>         end if;
>         printf "%o = %o * %o + %o\n", A, q, B, r;
>         A := B; B := r;
>     until r eq 0;
> end procedure;
```

Since the implementation of the algorithm is indeed correct, the **assert**-statement will have no effect on invocations of the procedure from the caller's point of view.

It is possible for the user to control whether assertions are checked, according to the value of **GetAssertions**(). By default, this function returns **true** (i.e., check the assertions). However, the procedure call **SetAssertions**(b), where b is a Boolean, may be used to change this. If b is set to **false**, then MAGMA will ignore **assert**-statements entirely, thus reducing the execution time slightly.

9. Iterative Statements

Iteration or *looping* is the repeated performance of a task that is syntactically constant but for which the data may change. The terms *iteration* or *loop* may also be applied to instances of the task's performance. The classic method of specifying iteration in computer languages is by means of *iterative statements*. These are compound statements, since they surround other statements providing the description of the task which is to be performed repeatedly. As this chapter explains, MAGMA has three forms of iterative statements: the **while**-statement, the **repeat**-statement, and the **for**-statement.

It should be noted at the outset that iterative statements are not as crucial for programming in the MAGMA language as they are in many other computer languages. MAGMA's set and sequence constructors allow many iteration processes to be expressed *statically*, as a description of the outcome rather than as an explicit iteration using an iterative statement. By the nature of elementary examples, many of the iteration tasks in this chapter are also programmable using set/sequence constructors. Nonetheless, there are applications for which iterative statements remain the best or only choice of programming tool in the MAGMA language.

9.1 Methods of Iteration

In most cases of iterative processes, the programmer intends the whole iteration to stop at the end of one of the loops. There are two ways of stating when this is to happen: either the iteration is over a predetermined finite domain D, and continues until some identifier has assumed each value of D exactly once in successive loops; or, less predictably, a Boolean expression is evaluated at the beginning or end of each loop, and the iteration process concludes when the value of this expression changes. Occasionally, it is more convenient for the iteration to conclude in the midst of one of the loops; this may be achieved by means of an interrupt that follows upon the change in value of some intermediate condition.

MAGMA's iterative statements provide for all these circumstances. In one of them, the **for**-statement, the iteration is over a domain, according to the first kind of termination condition. In the others, the **while**-statement and **repeat**-statement, termination is dependent on the value of a Boolean expression that is evaluated respectively at the beginning and end of each loop. MAGMA also provides statements for interrupting the usual course of the iterative statement: the **break**-statement for immediate termination of the whole iterative statement, and the **continue**-statement for termination of the current loop of the iteration.

MAGMA's set and sequence constructors, and related syntactic constructs, allow many kinds of iteration to be performed without the need for iterative statements. For example, the following set constructor creates the set of integers of the form p^3, where p is a prime in the range $1 \leq p \leq 20$:

```
> PrimesCubed := { p^3 : p in [1..20] | IsPrime(p) };
> print PrimesCubed;
{ 8, 27, 125, 343, 1331, 2197, 4913, 6859 }
```

Although this set is described statically, iteration takes place internally. MAGMA must iterate over the domain sequence [1**..**20], and for each element p of the domain it must test its primality and then, for the primes only, include the value of p^3 in a set. In many programming languages, this task would have to be expressed as an explicit iteration. The MAGMA version of such an iteration would use a **for**-statement, because it is an iteration over a predetermined domain:

```
> PrimesCubed := { };
> for p in [1..20] do
>     if IsPrime(p) then
>         Include(~PrimesCubed, p^3);
>     end if;
> end for;
> print PrimesCubed;
{ 8, 27, 125, 343, 1331, 2197, 4913, 6859 }
```

Note that this method of constructing the set is more verbose, and is also more difficult to write and understand. In general, MAGMA style favours constructors over iterative statements when the task to be performed is relatively simple. However, if the task contains many auxiliary identifiers, many operations, or many **print**-statements, then an iterative statement should be chosen, since a constructor will be impossible or cumbersome.

It should be remembered that in most cases the choice of a method of iteration for an algorithm affects the time taken to develop the code much more than the execution time. Convenience for the programmer and legibility

of the program are the most important factors in the decision, except when the computations are going to consume a significant amount of CPU time.

9.2 The while-statement

The syntax of the **while**-statement is

while Boolean expression **do**
  statements
end while;

where there must be at least one statement between **do** and **end while**; these statements constitute the task to be iterated. The termination condition for the iteration process is given by the Boolean expression following **while**. This expression is evaluated at the beginning of each loop, including the first loop. If it is **true**, then the loop is executed; otherwise, the execution of the whole **while**-statement finishes. Note that if the value of the Boolean expression is already **false** before the looping has started, then the statement body is never executed.

For example, consider the problem of calculating the factorial of n. (MAGMA has an intrinsic function **Factorial**(n) that should be used in practice to find this value.) Initially, the value of n will be fixed at 20. The following lines of code, incorporating a **while**-statement, cause the value of 20! to be stored in the identifier *answer*:

```
> n := 20;
> c := n;
> answer := 1;
> while c gt 0 do
>     answer *:= c;
>     c -:= 1;
> end while;
```

When this program is typed into MAGMA, the computer will probably provide a special prompt during the time that the statement body of the **while**-statement is being entered, as a reminder that **end while** has not yet been typed:

```
> n := 20;
> c := n;
> answer := 1;
> while c gt 0 do
```

```
while> answer *:= c;
while> c -:= 1;
while> end while;
```

The special prompt will usually be omitted in the examples in this chapter, for the sake of clarity.

When an iterative statement is typed into MAGMA, if the system detects an error before the end of the whole statement has been reached, then the part of the statement already typed is lost. It may be recovered with MAGMA's history-editing facilities, but for longer iterative statements it is helpful to develop them in an external file and **load** the file into MAGMA. See Section 14.2.

After the iterative statement has terminated, the number 20! will be stored in *answer*. It may be compared with the value returned by **Factorial**(n):

```
> print answer;
2432902008176640000
> print Factorial(n);
2432902008176640000
```

The workings of the code above will now be explained in detail. The first statements this program executes are the assignments to n, c and *answer*. The reason for creating c as a copy of n is that the algorithm requires the value of n to be gradually decremented, but it may be undesirable to lose the original value of n. Next comes the **while**-statement. The first step in executing a **while**-statement is to evaluate the condition given between **while** and **do**. Since c is indeed greater than 0, MAGMA starts executing the statements between **do** and **end while**. On this first pass through the loop, *answer* is 1 and c is 20, so the line

```
>     answer *:= c;
```

multiplies *answer* by c, making *answer* become 20. The other statement in the loop reduces c by 1. MAGMA now evaluates the **while**-condition again, and finds that it is still **true**, since $19 > 0$. Thus the loop statements are executed again. This time *answer* becomes 20×19 and c becomes 18. After several more iterations, execution of the loop reaches the stage when the value of c is 1. This time *answer* becomes $(20 \times 19 \times \cdots \times 2) \times 1 = 20!$ and c becomes 0. MAGMA evaluates the **while**-condition yet again, but it is no longer true, because 0 **gt** 0 is **false**. Now execution of the program falls through to the line following **end while**, and *answer* has its final value of 20!.

From the trace-through of the iteration, some ways to improve this algorithm should be readily apparent. The last pass through the loop serves no

purpose, because *answer* already equals 20! and is merely being multiplied by 1. Moreover, the very first pass through the loop could be eliminated by a change in the initial assignments to c and *answer*. A version incorporating these refinements appears below:

```
> c := n - 1;
> answer := n;
> while c gt 1 do
>     answer *:= c;
>     c -:= 1;
> end while;
```

It is now appropriate to generalize this code for all integers $n \geq 0$. The code will be enclosed in a function, for ease of use:

To perform this example online, type `load "I96c9e1";`

```
> factl := function(n)
>     error if n lt 0,
>         "Error in 'factl': Argument should be >= 0";
>     if n lt 2 then
>         return 1;
>     end if;
>     c := n - 1;
>     answer := n;
>     while c gt 1 do
>         answer *:= c;
>         c -:= 1;
>     end while;
>     return answer;
> end function;
```

To obtain the value of $n!$ for various n, the function *factl* may be called as follows:

```
> print factl(5);
120
> print factl(-241);
Error in 'factl': Argument should be >= 0
> print factl(0);
1
> print factl(17);
355687428096000
```

As another example of the **while**-statement, consider the problem of calculating $\sum_{i=1}^{t} i^3$, where t is a given positive integer. It may be found with an iterative statement, such that in each iteration the index i is incremented and i^3 is added to an identifier *sum*. This summation also has a well known closed-form expression:

$$\sum_{i=1}^{t} i^3 = \left(\frac{t(t+1)}{2}\right)^2 .$$

The procedure *testsum*(t) below verifies this identity for the given t. It evaluates the left and right sides of the equation, using a **while**-statement for the left side, and prints a sentence concerning whether the equation holds for t:

To perform this example online, type `load "I96c9e2";`

```
> testsum := procedure(t)
>     i := 1;
>     sum := 0;
>     while i le t do
>         sum +:= i^3;
>         i +:= 1;
>     end while;
>     print "Equation is", sum eq (t*(t+1) div 2)^2,
>       "for t =", t;
> end procedure;
```

Notice that in the above **print**-statement the integer division operator is suitable, since it is certain that $t(t+1)$ is even and so the division will be exact.

Since the equation is an identity, the output should verify the conjecture for all integers $t \geq 1$:

```
> testsum(2);
Conjecture is true for t = 2
> testsum(160);
Conjecture is true for t = 160
> testsum(1867);
Conjecture is true for t = 1867
```

9.3 The repeat-statement

The syntax of the **repeat**-statement is:

repeat
  statements
until Boolean expression;

It is similar to the **while**-statement, except that the termination condition is tested at the end, not the beginning, of each loop. The iteration continues until the Boolean expression evaluates to **true**. Therefore the statement body is always executed at least once in this kind of iterative statement.

An example of a **repeat**-statement that tests a conjecture will now be given. In 1640, Pierre Fermat made the conjecture that $2^{2^n} + 1$ is a prime for all positive integers n. Numbers of this form are now known as Fermat numbers. The program below investigates the primality of the first six Fermat numbers, and stops if it encounters one that is not prime. It uses the intrinsic function **IsPrime**, thus making the task much easier than in Fermat's day. A **repeat**-statement is a slightly better choice than a **while**-statement for this situation because at least one pass through the loop must be made, for the calculation and testing of the first Fermat number. Because it is not clear whether the iterative process might finish when all six numbers have been tested or when a composite Fermat number has been found, the **until**-condition must consist of two conditions, joined by **or**. The program to test Fermat's conjecture follows:

```
> n := 1;
> repeat
>     composite := not IsPrime(2^2^n + 1);
>     if composite then
>         print "Conjecture is false, since";
>         print "(2^2^", n, "+ 1) is composite.";
>     else
>         n +:= 1;
>     end if;
> until composite or n eq 7;
Conjecture is false, since
(2^2^ 5 + 1) is composite.
```

The output shows what was discovered by Euler almost a hundred years after Fermat's conjecture: the fifth Fermat number is not a prime. Fermat numbers are still of interest to mathematicians, however; computational number theorists continue to investigate primality and factorization issues connected with them.

As another example of the **repeat**-statement, cconsider the calculation of Fibonacci numbers according to the recurrence relation $F_n = F_{n-1} + F_{n-2}$, for $n > 2$, with $F_1 = 1, F_2 = 1$. Two functions, employing different algorithms, will be discussed for constructing the sequence of all the Fibonacci numbers

less than some given large limit L. (Note that MAGMA's intrinsic function **Fibonacci**(n) returns individual Fibonacci numbers F_n.)

The first algorithm involves three identifiers, called *old*, *middle* and *new*. At each stage, the two Fibonacci numbers most recently stored in the sequence are in *old* and *middle*, and their sum is assigned to *new* and put on the end of the sequence. Then *middle* is shifted to *old*, and *new* to *middle*, so that the process can be repeated. This algorithm can be encoded as follows:

To perform this example online, type `load "I96c9e3";`

```
> fiboA := function(L)
>     middle := 1;
>     new := 1;
>     Q := [1];
>     repeat
>         Append(~Q, new);
>         old := middle;
>         middle := new;
>         new := old + middle;
>     until new ge L;
>     return Q;
> end function;
```

The second approach to the Fibonacci problem comes from realizing that the numbers only have to be held in their identifiers to calculate the next two terms of the sequence. Hence it might be more efficient to do calculations in pairs, rather than shifting the numbers from one identifier to another. This idea is incorporated in the following function:

To perform this example online, type `load "I96c9e4";`

```
> fiboB := function(L)
>     f := 1;
>     g := 1;
>     Q := [1];
>     repeat
>         Append(~Q, g);
>         f +:= g;
>         Append(~Q, f);
>         g +:= f;
>     until g ge L;
>     if f ge L then
>         Prune(~Q);
```

```
>     end if;
>     return Q;
> end function;
```

Notice that the final **if**-statement checks to see whether the final addition to the sequence was greater than L. It is not possible to avoid the risk of overshooting the limit L in this algorithm, without making the code inefficient. The size of g is always checked by the **until** condition before it is put on the end of the sequence, but if f were also checked before being appended to the sequence there would be another statement in the loop body, to be executed on every iteration.

MAGMA's **time** command may be used to compare these algorithms for speed. When **time** is placed at the beginning of a statement, the statement is executed as usual, but the number of seconds of CPU time taken to execute the statement is also printed. The following example demonstrates how to time both functions, with a limit L of 10^{9000}:

```
> L := 10 ^ 9000;
> time fibonacciA := fiboA(L);
Time: 2.919
> time fibonacciB := fiboB(L);
Time: 2.669
> print fibonacciA eq fibonacciB;
true
> print #fibonacciA;
43066
```

From the time comparison, *fiboB* is slightly faster than *fiboA* for this value of L.

9.4 The for-statement

The general structure of the **for**-statement is

for identifier **in** domain **do**
  statements
end for;

The loop identifier, following the word **for**, takes the elements in the domain as its successive values, and for each value of the identifier, the statements are executed. Typically they would include some reference to the loop identifier. The domain of the loop identifier should be an expression that evaluates

to some iterable structure, that is, an iterable aggregate structure or a finite magma whose elements MAGMA is able to list. This domain is calculated when the execution of the **for**-statement begins, and cannot be changed during the iteration.

For example, consider the residue class ring $\mathbb{Z}_{18}$ of integers modulo 18. Some of the elements n of this ring are squares, in the sense that there exists an element m of the ring such that $m^2 = n$. One way to identify these squares is to test each element of the ring in turn, using the function **IsSquare**(n). The **for**-statement below does this, using the residue class ring as its domain:

```
> for n in ResidueClassRing(18) do
>     if IsSquare(n) then
>         print n;
>     end if;
> end for;
0
1
4
7
9
10
13
16
```

To execute the **for**-statement above, MAGMA initially assigns to n the first element of $\mathbb{Z}_{18}$ delivered to it by the iteration mechanism for this ring. Then it performs everything between the **do** and the **end for**, using this value of n. Next, n is assigned the second element of the ring and the body of the **for**-loop is executed again, with the new value of n. The process continues until the loop body has been executed for each element of the domain.

As another example, let G be a permutation group. The elements of G of order k may be collected into a set by means of a **for**-statement. The function below returns this set for a given group G and integer k:

To perform this example online, type `load "I96c9e5";`

```
> order_elts := function(G, k)
>     eltset := { G | };
>     for g in G do
>         if Order(g) eq k then
>             Include(~eltset, g);
>         end if;
>     end for;
```

```
>     return eltset;
> end function;

> print order_elts(Sym(5), 6);
{
    (1, 5, 3)(2, 4),
    (1, 5)(2, 4, 3),
    (1, 4)(2, 5, 3),
    (1, 3, 5)(2, 4),
    (1, 2)(3, 4, 5),
    (1, 5, 2)(3, 4),
    (1, 5, 4)(2, 3),
    (1, 3)(2, 4, 5),
    (1, 3)(2, 5, 4),
    (1, 4)(2, 3, 5),
    (1, 2)(3, 5, 4),
    (1, 3, 2)(4, 5),
    (1, 4, 3)(2, 5),
    (1, 3, 4)(2, 5),
    (1, 4, 2)(3, 5),
    (1, 2, 3)(4, 5),
    (1, 4, 5)(2, 3),
    (1, 2, 4)(3, 5),
    (1, 2, 5)(3, 4),
    (1, 5)(2, 3, 4)
}
```

9.4.1 Properties of Loop Identifiers

The **for**-construct is designed with a safeguard: the loop identifier is *local* to the loop. Effectively, it has no existence outside the loop. For instance, after the loop over $\mathbb{Z}_{18}$ has finished, it is not possible to obtain the final value that n had. Indeed, if there were an identifier called n that was assigned before the **for**-statement started, then when the loop had finished n would have its old value, not its final value in the loop; on the other hand, if there were no identifier n prior to the loop, then this would still be the case after the execution of the **for**-statement had finished. This scoping rule is designed to protect the user from destroying important data existing outside the loop. For instance:

```
> a := 134237402;
> A := AbelianGroup([2, 3]);
> for a in A do
>     print Order(a);
```

```
> end for;
1
6
3
2
3
6
> print a;
134237402
```

Similarly, after the loop over G has finished in the permutation group example above, it is not possible to obtain the final value that g had, and any identifier g outside the loop is independent of it. This is the case even within the function *order_elts*, but the function itself provides another level of scope, such that not only g but also *eltset* have no existence outside the function.

Another property of loop identifiers of **for**-statements is that they may not be reassigned in the statement body. They exist to be inspected for their current value rather than to be changed by the user. For this reason, loop identifiers are said to belong to the *value class of identifiers*.

9.4.2 Using Sequences as Domains of Iteration

If the domain of a **for**-loop is a sequence (or a list or indexed set), the loop identifier will run through the elements of the domain in the standard indexing order. If it is a sequence (or a list or multiset) with repeated elements, then those elements will be assigned to the loop identifier the corresponding number of times. For example, the following statement prints each term r of a sequence of rational numbers, preceded by the floor $\lfloor r \rfloor$ of r (the greatest integer less than or equal to r):

```
> for j in [9/2, 1, 3/4, 2, 3/4, 3/4, 54/11] do
>     print Floor(j), j;
> end for;
4 9/2
1 1
0 3/4
2 2
0 3/4
0 3/4
4 54/11
```

The output shows that the terms of the sequence are used in order, including the repeated terms.

If the domain of a **for**-statement is a sequence Q with repeated terms and it is undesirable to repeat the iteration for the same value of the loop identifier, then the **for**-statement should be given the enumerated set **Seqset**(Q) as its domain. This change can be important when the execution time for each iteration is non-trivial and the proportion of repeated terms is high.

It often happens that the user wishes the loop identifier to take on the values of a range of integers in arithmetic progression: $i, i+k, i+2k, \ldots, j$. This may be accomplished in MAGMA by constructing the domain as an arithmetic-progression sequence, using the syntax $[i..j$ **by** $k]$, or $[i..j]$ if $k = 1$. The procedure on p. 202, which verifies a summation formula, may be encoded with a **for**-loop instead of a **while**-loop using this technique:

```
> testsum := procedure(t)
>     sum := 0;
>     for i in [1..t] do
>         sum +:= i^3;
>     end for;
>     print "Equation is", sum eq (t*(t+1) div 2)^2,
>       "for t =", t;
> end procedure;
```

Notice that the statement `i +:= 1;` is no longer required, since the **for**-loop performs the incrementing automatically. The **for**-statement is the best iterative statement for this task.

Similarly, the factorial problem on p. 201 can be implemented using a **for**-statement with an arithmetic-progression sequence as its domain, since c changes by the constant value -1 on each pass through the loop:

```
> factl2 := function(n)
>     error if n lt 0,
>         "Error in 'factl': Argument should be >= 0";
>     answer := 1;
>     for c in [n..2 by -1] do
>         answer *:= c;
>     end for;
>     return answer;
> end function;
```

The loop identifier c in the **for**-statement above runs through the integers $n, n-1, \ldots$ down to 2. If n is less than 2 (i.e., $n = 1$ or $n = 0$, since $n \geq 0$) then the domain sequence is empty. In this case, the statements in the body of the **for**-loop are never executed, so *answer* retains its initial value of 1, which is the correct value of both 0! and 1!.

9.5 Exiting an Iteration or Loop Quickly

9.5.1 Exiting the Current Iteration Quickly

It sometimes happens within the body of a loop that it is unnecessary to execute the rest of the statements in the loop body for the current iteration. This often happens when an iterative statement is searching for objects satisfying certain conditions; once it becomes established that a particular object does not satisfy these conditions, it should be possible to proceed immediately to the next object, in the next iteration. The MAGMA statement for this situation is

continue;

When MAGMA is executing a loop body and encounters this statement, it omits the rest of the statements in the loop body. Execution will proceed to the next iteration or, if all the iterations have been done, to the statement following the whole loop. For example, the program on p. 206 that prints the squares in $\mathbb{Z}_{18}$ may be modified as follows to use a **continue**-statement:

```
> for n in ResidueClassRing(18) do
>     if not IsSquare(n) then
>         continue;
>     end if;
>     print n;
> end for;
0
1
4
7
9
10
13
16
```

In this case, there is only one statement left in the loop body after the **continue**-statement, so there is no real advantage to the modification. For larger loops, **continue** is useful because it helps avoid a tangle of incomplete **if**-statements.

9.5.2 Exiting the Whole Loop Quickly

MAGMA also has a statement

break;

which causes immediate exit from the entire iterative statement. This statement is useful in dealing with special cases, or in algorithms that are searching for a single value. The following modification of the Fermat conjecture program (p. 203) illustrates it:

```
> n := 1;
> repeat
>     composite := not IsPrime(2^2^n + 1);
>     if composite then
>         break;
>     end if;
>     n +:= 1;
> until n eq 7;
> if composite then
>     print "Conjecture is false, since";
>     print "(2^2^", n, "+ 1) is composite.";
> else
>     print "Conjecture is true for 1 <= n <= 6.";
> end if;
Conjecture is false, since
(2^2^ 5 + 1) is composite.
```

It is very appropriate to use the **break**-statement here because it permits a single termination condition for the **repeat**-loop. The previous version had a double exit condition, which is more difficult to handle logically. However, since the iteration operates on the basis of a counter n that is incremented by 1, a **for**-statement with a **break**-statement is an even better choice:

```
> for n in [1..6] do
>     composite := not IsPrime(2^2^n + 1);
>     if composite then
>         break;
>     end if;
> end for;
> if composite then
>     print "Conjecture is false, since";
>     print "(2^2^", n, "+ 1) is composite.";
> else
>     print "Conjecture is true for 1 <= n <= 6.";
> end if;
Conjecture is false, since
(2^2^ 5 + 1) is composite.
```

Now the program is more elegant: it is easier to write, easier to read, and displays the logic of the underlying algorithm more clearly. Moreover, the final **if**-statement guarantees that there will be some intelligible output, no matter what value *composite* has.

9.5.3 Exits in Nested Loops

If one loop resides within another, it is said to be *nested* within the larger loop. When MAGMA encounters

```
continue;
```

or

```
break;
```

within a nested loop, it assumes that these statements refer to the innermost, smallest loop.

In the case of **for**-loops, it is possible to override this assumption by specifying the loop identifier of the relevant loop, after the word **continue** or **break**. For example, in the following nested **for**-loops:

```
for x in xdomain do
  ⋮
  for y in ydomain do
    ⋮
      break x;
    ⋮
  end for;
  ⋮
end for;
```

the statement

```
break x;
```

refers to the outer loop, so the outer loop will be terminated. If the statement were

```
break;
```

then the inner loop would be terminated.

9.6 Iterations Without Iterative Statements

As was stated at the beginning of this chapter, many iterative tasks that have to be performed with iterative statements in other computer languages may also be performed in MAGMA with set/sequence constructors and associated operations. Some of the examples in this chapter will now be implemented without using iterative statements, for the sake of comparison.

The identification of the square elements of $\mathbb{Z}_{18}$ on p. 206 can be made shorter and easier to read as follows:

```
> print [n: n in ResidueClassRing(18) | IsSquare(n)];
[ 0, 1, 4, 7, 9, 10, 13, 16 ]
```

The output has changed from a succession of single integers to a single sequence, but this may well be preferable anyway. An improvement to the underlying algorithm may also be made by constructing the squares directly:

```
> print {n^2: n in ResidueClassRing(18)};
{ 0, 1, 13, 4, 16, 7, 9, 10 }
```

The function returning the set of order–k elements of the permutation group G (p. 206) may also be created with a set constructor. By doing this, the return value can be described in a single expression, so the function itself can be built with the **func**-constructor:

```
> order_elts := func< G, k | {g: g in G | Order(g) eq k} >;
```

The operators on sets and sequences can be used in combination with constructors to produce similarly succinct code. For instance, the verification of the summation identity on p. 202 may also be done in this way:

```
> testsum := procedure(t)
>     sum := &+[ i^3 : i in [1..t] ];
>     print "Equation is", sum eq (t*(t+1) div 2)^2,
>       "for t =", t;
> end procedure;
```

The code increases in clarity because the details of the summation process do not need to be given explicitly by the user.

The factorial program of p. 209 can be encoded similarly, by performing multiplication on all the elements of a sequence:

```
> fact13 := function(n)
```

```
>     error if n lt 0,
>         "Error in 'factl': Argument should be >= 0";
>     return &*[n..2 by -1];
> end function;
```

The above statement works for the $n = 0$ and $n = 1$ cases because of a property of MAGMA's reduction operator. When $n = 0$ or $n = 1$, the sequence whose elements are being multiplied together is empty. Since MAGMA follows convention by evaluating an empty product as 1, the result is the correct one.

The testing of a conjecture, such as Fermat's conjecture (p. 203), can be performed with **forall**:

```
> if forall(ex){n: n in [1..6] | IsPrime(2^2^n + 1) } then
>     print "Conjecture is true for 1 <= n <= 6.";
> else
>     print "Conjecture is false, since";
>     print "(2^2^", ex, "+ 1) is composite.";
> end if;
Conjecture is false, since
(2^2^ 5 + 1) is composite.
```

It can be seen that many of the examples in this chapter can be programmed without iterative statements. This does not mean that iterative statements are always a poor choice, merely that loop examples sufficiently elementary for illustrative purposes are quite likely to be most clearly encoded in a compact static form.

10. Intrinsics, Signatures and Packages

The functions and procedures which form part of the MAGMA system are known as *intrinsics*. Most of them are *system intrinsics*, that is, functions and procedures programmed by the MAGMA developers and their associates. However, it is also possible for users of MAGMA to create their own *user intrinsics* in *packages*, and attach them to the system in such a way that they are virtually indistinguishable from system intrinsics. Indeed, it is possible that the reader's version of MAGMA has been customized in this way, perhaps in a teaching or research context.

Both system intrinsics and user intrinsics are compiled into the MAGMA internal pseudo-code, in order to save time when the definition of the intrinsic has to be loaded. When either kind of intrinsic is invoked, type-checking of its actual arguments is performed, to see that they belong to the correct category. System intrinsics reside permanently in MAGMA, but user intrinsics can easily be attached to and detached from the system during a MAGMA session. All user intrinsics are written in the MAGMA language, in a modified form of the syntax for functions and procedures; this syntax is assumed knowledge for this chapter, and is explained in Chapter 8. By contrast, most of the system intrinsics are implemented in the computer language C, but a growing number are written in MAGMA and then permanently attached to the system by means of the package mechanism described in this chapter.

A moderate quantity of documentation is available for all intrinsics. The *signature* of an intrinsic may be printed, to show the categories of its arguments and return values, as well as a brief description. It is possible to list the signatures of all intrinsics having an argument of category C. The tab-completion facility, which is discussed in Section 15.1, provides for shorthand typing of all intrinsics. However, only the system intrinsics are explained in the online help system (see Chapter 11), since that is a form of official documentation provided with MAGMA that cannot be altered by the inclusion of user intrinsics.

10.1 Signatures of Intrinsics

The *signature* of an intrinsic is a short description of the intrinsic, in a fixed format. First comes the list of its arguments and their categories, then (if it is a function) the list of the categories of its return values, then details of any parameters it has, and finally a line or two of text describing what the intrinsic does.

The way to obtain the signature of an intrinsic is to print the intrinsic. For example, the following line shows how to obtain information about **Factorial**:

```
> print Factorial;
Intrinsic 'Factorial'

Signatures:

    (<RngIntElt> n) -> RngIntElt

        The factorial n! for small non-negative n
```

The signature above states that **Factorial** has one argument n, in the **RngIntElt** category (that is, an integer), and that the function returns the factorial of its argument as an integer.

Some intrinsics can take several kinds of arguments, with respect to their categories or the number of arguments, so they have several signatures. For instance:

```
> print Rank;
Intrinsic 'Rank'

Signatures:

    (<AlgMatElt> X) -> RngIntElt
    (<ModMatRngElt> X) -> RngIntElt

        The rank of X

    (<ModTupRng> M) -> RngIntElt
    (<ModMatRng> M) -> RngIntElt

        The rank of the free module M
        over its base coefficient ring

    (<RngUPol> R) -> RngIntElt
```

```
(<RngMPol> R) -> RngIntElt
(<RngMPolRes> R) -> RngIntElt

    The number of indeterminates of R
    over its coefficient ring

(<ModLat> L) -> RngIntElt

    The rank of the lattice L

(<CurveEll> E) -> RngIntElt

    The (lower bound on the) rank of the
    Mordell-Weil group of E;
    E must be defined over Z
```

The signatures of **Rank** show that this function always returns an integer, but that its argument can be: an element of a matrix algebra; an element of a homomorphism module; a tuple module; a homomorphism module; a univariate or multivariate polynomial ring or a quotient of such a ring; a lattice; or an elliptic curve.

10.2 Operators Regarded as Intrinsics

Operators such as `+`, `#` or `@` are actually system intrinsics with special syntax. If the operator is enclosed in apostrophe signs, MAGMA will treat it as if it were a standard system intrinsic:

```
> print '+'(2, 3);
5
```

This technique may be used for obtaining the signature of an operator:

```
> print 'cat';
Intrinsic 'cat'

Signatures:

    (<SeqEnum> s, <SeqEnum> t) -> SeqEnum

        Concatenation of the enumerated sequences s and t
```

```
(<List> L1, <List> L2) -> List

    Concatenation of the lists L1 and L2

(<MonStgElt> s, <MonStgElt> t) -> MonStgElt

    Concatenation of the strings s and t
```

10.3 Listing Signatures

The procedure **ListSignatures**(C) lists the signatures of all intrinsics which have an argument or return value belonging to the category C. (Only the category information is shown, not the explanatory comment.) This procedure is useful when the user has forgotten the name of an intrinsic. For instance, the output from the following line would display all the signatures for intrinsics and operators involving a finite field:

```
> ListSignatures(FldFin);
```

The list of signatures will usually include many signatures that apply to all categories in the variety of C, or to all structures or objects in general. If only the signatures that are explicitly for the category C are desired, the parameter **Isa** should be set to **false**. For example:

```
> ListSignatures(FldFin: Isa := false);
```

By default, **ListSignatures**(C) finds both the intrinsics having C as an argument and those having C as a return value. The search may be restricted to one or the other of these by assigning `"Arguments"` or `"ReturnValues"` (respectively) to the parameter **Search**. The default value of **Search** is `"Both"`.

Recall that the procedure **ListCategories**() provides a list of all the categories in MAGMA, so that category names can be checked.

10.4 Creating User Intrinsics

To be incorporated into the MAGMA system, the definitions of user intrinsics must be placed in package files and attached to MAGMA. This section only explains how each definition of an intrinsic is structured; the file-handling aspects will be described later.

User intrinsic functions and user intrinsic procedures are defined in similar ways. The syntax for defining a user intrinsic that is a function is:

intrinsic name(argument list) -> return list
{comment text}
  statements
end intrinsic;

and for a user intrinsic procedure it is:

intrinsic name(argument list)
{comment text}
  statements
end intrinsic;

The name of the intrinsic may be any identifier. (It may also be a non-alphanumeric collection of characters enclosed in apostrophe signs; this is only of practical use for extending the use of an existing operator such as + so that it works for some other categories.) If the intrinsic name is already the name of a system or user intrinsic, but the types of the arguments are different, the usages are combined to give multiple signatures.

A function is distinguished from a procedure by having a *return list*, giving information about the return values. The return list is a list of comma-separated simple types, where a *simple type* is a category name such as **ModTupFld** (a vector space) or **GrphUnd** (an undirected graph), a variety name such as **Grp** (any group category), a full stop (.) indicating 'any type', or a pair of brackets indicating a kind of set or sequence ([], {}, {@@}, {**}, or <>). The return list is preceded by the characters ->, representing an arrow.

The argument list is a list of comma-separated items that specify the arguments of the intrinsic. For value arguments, the most common kind of argument specification is name::simple type where the name is an identifier that serves as a formal value argument. Alternatively, the double colon may be followed by a more general type specification, consisting of bracketing symbols enclosing a simple type, to indicate the category of the elements of the set or sequence: [simple type], {simple type}, {@simple type@}, or {*simple type*}. For a reference argument of an intrinsic procedure, the syntax depends on whether the identifier serving as the actual reference argument has to be assigned a value before the procedure is called. If the argument is expected to have a value already, then ~name::simple type is the appropriate syntax. If the argument is not expected to have a value (but will be assigned within the procedure body) then it should be specified as ~name with no type being given.

The comment text, enclosed in braces, is compulsory. It is the source for the textual description of an intrinsic. The statements in the body of the definition are normal MAGMA statements, as would be found in the statement form of a function expression or procedure expression; in the function case, they must include a **return**-statement with expressions for the return values.

Parameters are permitted in user intrinsics. The syntax is the same as for user-defined functions/procedures: after the argument list comes a colon and then a list of items of the form parameter := default , giving the parameter name and its default value. There is no category-checking for parameters; it must be done within the body of the intrinsic. The user intrinsic *Quadrature* given on p. 228 provides an illustration of parameters.

As a simple example of a function intrinsic, consider a conversion of the *ShuffleGroup*(n) function on p. 173. This function takes as its argument an integer (category **RngIntElt**) and returns a permutation group (category **GrpPerm**). Therefore the intrinsic is:

```
intrinsic ShuffleGroup(n::RngIntElt) -> GrpPerm
{The shuffle group for a deck of 3n cards}
    m := 3 * n;
    return PermutationGroup< m |
       &*[(i, i + 2 * n) : i in [1 .. n]],
       &*[(i, i + n, i + 2 * n) : i in [1 .. n]],
       [((i-1) mod 3)*n + (i-1) div 3 + 1 : i in [1..m]] >;
end intrinsic;
```

As an example of a procedure intrinsic, the version of *EuclideanAlgorithm* defined on p. 178 may be converted as follows:

```
intrinsic EuclideanAlgorithm(a::RngIntElt, b::RngIntElt)
{Displays working of Euclidean Algorithm
for gcd of a and b}
    if a ge b then
        A := a; B := b;
    else
        A := b; B := a;
    end if;
    repeat
        q, r := Quotrem(A, B);
        if r eq 0 then
            printf "%o = %o * %o\n", A, q, B;
        else
            printf "%o = %o * %o + %o\n", A, q, B, r;
            A := B; B := r;
        end if;
```

```
    until r eq 0;
end intrinsic;
```

These intrinsics *cannot* be typed into MAGMA directly. To be incorporated into the system, they must be placed in a package file, and the package file must be *attached* to MAGMA (this has a different effect from that achieved by the **load**-statement).

10.5 Package Files

A package file is a text file created outside MAGMA, containing definitions of user intrinsics. It may also contain assignment statements, so that constants (including ordinary user-defined functions/procedures) can be defined for use inside the definitions of intrinsics. These constants will not be loaded into the MAGMA session; their scope is limited to the package file unless they are explicitly imported (see Section 10.10). MAGMA comments are permitted in the file too.

For example, if a file called `mypck1` is created containing the two user intrinsics above, *ShuffleGroup* and *EuclideanAlgorithm*, it will be a valid package file.

As another example, let the following intrinsic functions for string-handling be the contents of the file `mypck2`. This is another valid package file. Notice that it defines the constants STC and R. Constants are useful when programming several related functions or procedures, not only because they reduce the amount of typing, but because changes and corrections can be made more easily.

```
intrinsic StringToSequence(s::MonStgElt) -> []
{Sequence of the characters of string s}
    return [ s[n] : n in [1..#s] ];
end intrinsic;

intrinsic SequenceToString(q::[MonStgElt]) -> MonStgElt
{Concatenation of strings in sequence q}
    return &* q;
end intrinsic;

STC := StringToCode; // shorthand
R := STC("A") - STC("a");

intrinsic LowerToUpperCase(s::MonStgElt) -> MonStgElt
```

```
{Convert lower-case alphabetic string s to upper-case}
    q := [ STC(c) : c in StringToSequence(s) ];
    return SequenceToString([CodeToString(c + R) : c in q]);
end intrinsic;

intrinsic UpperToLowerCase(s::MonStgElt) -> MonStgElt
{Convert upper-case alphabetic string s to lower-case}
    q := [ STC(c) : c in StringToSequence(s) ];
    return SequenceToString([CodeToString(c - R) : c in q]);
end intrinsic;
```

10.6 Attaching a Package File

There are several ways of incorporating user intrinsics into MAGMA. While a package file is being developed, the best method is to *attach* the package during the MAGMA session. After a package has been brought to its final form and tested, it can be specified along with the user's other completed package files in a *spec file* (short for 'specification file'). The spec file may be handled in various ways, to be described later.

The procedure for attaching a package file is **Attach**(F), where the filename is stored in the string F. This string may be a literal string, enclosed in " symbols, or an expression returning a string. If the file is not in the current directory, its absolute or relative path must be given in F. The procedure for detaching an attached package file during the MAGMA session is **Detach**(F).

For example, suppose that a MAGMA session is in progress and that the file `mypck1` described above is in the current directory. The package may be attached, and then the intrinsics defined in it become indistinguishable from system intrinsics:

```
> Attach("mypck1");
> print EuclideanAlgorithm;
Intrinsic 'EuclideanAlgorithm'

Signatures:

    (<RngIntElt> a, <RngIntElt> b)

        Displays working of Euclidean Algorithm for
        gcd of a and b
```

```
> EuclideanAlgorithm(133, 24);
133 = 5 * 24 + 13
24 = 1 * 13 + 11
13 = 1 * 11 + 2
11 = 5 * 2 + 1
2 = 2 * 1
> print ShuffleGroup;
Intrinsic 'ShuffleGroup'

Signatures:

    (<RngIntElt> n) -> GrpPerm

        The shuffle group for a deck of 3n cards


> print ShuffleGroup(5);
Permutation group acting on a set of cardinality 15
    (1, 11)(2, 12)(3, 13)(4, 14)(5, 15)
    (1, 6, 11)(2, 7, 12)(3, 8, 13)(4, 9, 14)(5, 10, 15)
    (2, 6, 12, 14, 10, 4)(3, 11, 9, 13, 5, 7)
```

When MAGMA detects an error while attaching a package, it gives an error message that includes the abbreviation `[PC]`, for 'package compiler'. For example, suppose that the compulsory comment at the top of the intrinsic *ShuffleGroup* had been omitted. When the file was attached, the following message would have been printed:

```
> Attach("mypck1");
[PC]
[PC] In file "mypck1", line 2, column 5:
[PC] >>     m := 3 * n;
[PC]        ^
[PC] User error: bad syntax

>> Attach("mypck1");
         ^
Runtime error in 'Attach': Can't attach intrinsics
of "mypck1"
```

When such an error occurs, the file must be corrected and saved. The file must then be attached, because the previous attempt was unsuccessful.

10.7 Changing and Updating a Package File

One of the advantages of user intrinsics over ordinary user-defined functions is that once a package has been successfully attached to MAGMA (by any method, whether singly or as part of a spec file), MAGMA automatically updates the intrinsics in the package after a file has been changed and saved. The automatic updating applies both to alterations in existing intrinsics and to new or removed intrinsics. It is not necessary for the user to re-attach the altered file.

In more detail, just before each top-level statement is executed, MAGMA compares the time when the file was last saved with the time when the file was last processed by the system. If the file has been saved more recently, then the system recompiles the package. This process of time comparison and recompilation if necessary is called *updating* a package. Note that the time comparisons between statements make the operation of the system slightly slower. To prevent them in cases where the package is stable, the user can place the command **freeze**; at the top of the package file, to make a *frozen package*.

Moreover, every time a package (frozen or unfrozen) is attached, MAGMA updates it. The attachment may be by means of **Attach**, **AttachSpec**, or the automatic attachment of spec files at start-up as discussed later in this chapter. Since package attachment normally only occurs once per MAGMA session, the time overhead is minimal.

For example, suppose that the file `mypck1` is altered so that the procedure *EuclideanAlgorithm*(a, b) prints the gcd of a and b at the end of the list of equations. This would require the line

```
printf "Gcd of %o and %o is %o.\n", a, b, B;
```

to be placed after the first **printf**-statement. The following extract of a MAGMA session indicates how the change would be perceived by the user:

```
> // file not changed yet
> Attach("mypck1");
> EuclideanAlgorithm(50, 14);
50 = 3 * 14 + 8
14 = 1 * 8 + 6
8 = 1 * 6 + 2
6 = 3 * 2
> // file is now edited and saved
> EuclideanAlgorithm(50, 14);
50 = 3 * 14 + 8
14 = 1 * 8 + 6
```

```
8 = 1 * 6 + 2
6 = 3 * 2
Gcd of 50 and 14 is 2.
```

Notice that the update of the intrinsic is automatic. It is not necessary to re-attach the altered package. However, if the package had been frozen, then this update would not have occurred.

10.8 Testing for Errors in Intrinsics

Recall that the **error** and **error if** statements, discussed in Section 8.9, provide a means for a user-defined function or procedure to detect runtime errors and stop execution gracefully with an appropriate error message. These statements are also valid in user intrinsics. However, error-checking is slightly easier within user intrinsics, for two reasons.

Firstly, since argument types are given in the signatures, arguments of the wrong category are detected immediately. For example, in the package file `mypck2`, the intrinsic *LowerToUpperCase*(s) takes a string as its argument. An error message will be given if the argument is not a string:

```
> Attach("mypck2");
> print LowerToUpperCase;
Intrinsic 'LowerToUpperCase'

Signatures:

    (<MonStgElt> s) -> MonStgElt

        Convert lower-case alphabetic string s to upper-case


> print LowerToUpperCase("small");
SMALL
> print LowerToUpperCase(4/7);

>> print LowerToUpperCase(4/7);
                       ^
Runtime error in 'LowerToUpperCase': Bad argument types
```

If an argument is an aggregate type, such as a set or sequence, then the category of the elements may also be tested. For instance, the intrinsic

SequenceToString(q), in the same package file, takes as its argument a sequence q of strings. It is an error if q is not a sequence or if its elements are not strings:

```
> print SequenceToString;
Intrinsic 'SequenceToString'

Signatures:

    (<SeqEnum[MonStgElt]> q) -> MonStgElt

        Concatenation of strings in sequence q


> print SequenceToString(["small", "SMALL", "MeDiUm"]);
smallSMALLMeDiUm
> print SequenceToString(1234);

>> print SequenceToString(1234);
                          ^
Runtime error in 'SequenceToString': Bad argument types
> print SequenceToString([1, 2, 3, 4]);

>> print SequenceToString([1, 2, 3, 4]);
                          ^
Runtime error in 'SequenceToString': Bad argument types
```

The second reason that error-checking is easier in user intrinsics is that some special statements are provided for user intrinsics only: **require**, **requirerange**, and **requirege**. They are intended for checking that required conditions on the arguments and parameters hold.

The **require**-statement is the most general, since the user can specify any condition whose truth is to be tested. The syntax of this statement is:

require condition: expression list;

where the expression list is a list of comma-separated expressions that are to be evaluated and printed (often the expression list is simply a literal string). If the condition evaluates to **true**, no action is taken; if it evaluates to **false**, then execution stops, and MAGMA prints a message that advises the user of the runtime error in the intrinsic and displays the values of the expressions.

The other two statements are only used for an argument or parameter v which takes integer values. The **requirerange**-statement is used to verify that v falls within a certain range. It has the syntax:

requirerange v, L, U;

where L and U are integer expressions. If $L \leq v \leq U$, this statement has no effect, but otherwise it prints an error message stating the problem. The **requirege**-statement is similar:

requirege v, L;

This statement has no effect if $v \geq L$, but otherwise it prints an error message stating the problem.

As an example of the **require**-statement, observe that in its current form the intrinsic *LowerToUpperCase*(s) does not check whether all the characters of the string s are lower-case:

```
> print LowerToUpperCase("silly|{}||STRING");
SILLY\[]\\342).'
```

If *LowerToUpperCase* is changed to the following intrinsic:

```
intrinsic LowerToUpperCase(s::MonStgElt) -> MonStgElt
{Convert lower-case alphabetic string s to upper-case}
    q := [ STC(c) : c in StringToSequence(s) ];
    require q subset [STC("a")..STC("z")]:
      "String characters must be a, ..., z";
    return SequenceToString([CodeToString(c + R) : c in q]);
end intrinsic;
```

then this kind of error will be detected:

```
> print LowerToUpperCase("silly|{}||STRING");

>> print LowerToUpperCase("silly|{}||STRING");
                          ^
Runtime error in 'LowerToUpperCase': String characters
must be a, ..., z
```

A similar change should be made to *UpperToLowerCase*(s).

The intrinsic *EuclideanAlgorithm*(a, b) would also benefit from a **require**-statement, to ensure that $b \neq 0$, since otherwise division by zero would be attempted. The line

```
require b ne 0:
  "Argument 2 cannot be 0";
```

should be added after the comment.

The last two examples of intrinsics containing error-detection statements are modifications of *SepLine* and *quadrature* from p. 189 and p. 194 respectively. Collectively, they should be understood to be the contents of another package file, `mypck3`.

```
intrinsic SepLine(c::MonStgElt, n::RngIntElt)
{Prints a line of n copies of string character c}
    require #c eq 1: "Argument 1 (", c,
      ") should be a single string character";
    requirerange n, 0, 80;
    print c^n;
end intrinsic;

intrinsic Quadrature(f::Program, a::FldPrElt, b::FldPrElt :
  Al := "Trapezoid", n := 10) -> FldPrElt
{Approximate integral of f from a to b}
    require Category(Al) eq MonStgElt:
      "Parameter Al not a string";
    require Al in {"Trapezoid", "Simpson"}:
      "Parameter Al must be \"Trapezoid\" or \"Simpson\"";
    require Category(n) eq RngIntElt:
      "Parameter n not an integer";
    requirege n, 1;

    R := RealField();
    h := (b - a) / n;
    if Al eq "Trapezoid" then
        return h/2 * ( f(a) + f(b) +
          2* &+[ R | f(a + k*h): k in [1..(n-1)]] );
    else // "Simpson" method
        return h/3 * ( f(a) + f(b) +
          4 * &+[ R | f(a + k*h): k in [1..(n-1) by 2]] +
          2 * &+[ R | f(a + k*h): k in [2..(n-2) by 2]] );
    end if;
end intrinsic;
```

In the first argument of *Quadrature*, the simple type **Program** encompasses the categories **Intrinsic** (system and user intrinsics) and **UserProgram** (user-defined functions and procedures), just as the simple types **Grp** and **Rng** encompass several categories. Observe that parameters do not have types given in the signature, so type-checking for them must be done by hand.

Examples of how these intrinsics handle correct and incorrect invocations are given below:

```
> Attach("mypck3");
> print SepLine;
Intrinsic 'SepLine'

Signatures:

    (<MonStgElt> c, <RngIntElt> n)

        Prints a line of n copies of string character c


> SepLine("#", 20);
####################
> SepLine("#", -20);

>> SepLine("#", -20);
          ^
Runtime error in 'SepLine': Argument 2 (-20)
should be in the range [0 .. 80]
> SepLine("#$", 20);

>> SepLine("#$", 20);
          ^
Runtime error in 'SepLine': Argument 1 ( #$ )
should be a single string character


> print Quadrature;
Intrinsic 'Quadrature'

Signatures:

    (<Program> f, <FldPrElt> a, <FldPrElt> b) -> FldPrElt
    [
        Al,
        n
    ]
        Approximate integral of f from a to b


> fn := func< x | x^2 + 4/x >;
> print Quadrature(fn, 1.0, 10.0);
```

```
343.67585772176995519638651092653
> print Quadrature(fn, 1.0, 10.0 : Al := "Simpson");
342.24791204795303221761295937441 6
> print Quadrature(Sin, 2.0, 5.3 : Al := "Simpson");
-0.97058595358028565931450853487
> print Quadrature(Sin, 2.0, 5.3 : n := 3, Al := "Simpson");
0.089229299256145711157092870945
> print Quadrature(Sin, 2.0, 5.3 : Al := 33/4);

>> print Quadrature(Sin, 2.0, 5.3 : Al := 33/4);
                    ^
Runtime error in 'Quadrature': Parameter Al not a string
> print Quadrature(Sin, 2.0, 5.3 : Al := "junk");

>> print Quadrature(Sin, 2.0, 5.3 : Al := "junk");
                    ^
Runtime error in 'Quadrature':
Parameter Al must be "Trapezoid" or "Simpson"
> print Quadrature(Sin, 2.0, 5.3 : n := 3.4);

>> print Quadrature(Sin, 2.0, 5.3 : n := 3.4);
                    ^
Runtime error in 'Quadrature': Parameter n not an integer
> print Quadrature(Sin, 2.0, 5.3 : n := -300);

>> print Quadrature(Sin, 2.0, 5.3 : n := -300);
                    ^
Runtime error in 'Quadrature': Parameter n should be >= 1
```

Note also that the arguments that are elements of the real field must be explicitly supplied by the user as such; integers and rational numbers will not be coerced automatically into the real field:

```
> print Quadrature(Sin, 2, 5.3);

>> print Quadrature(Sin, 2, 5.3);
                    ^
Runtime error in 'Quadrature': Bad argument types
```

10.9 Package Spec Files

When a package file or several package files have been tested thoroughly, they should be gathered together in a *package spec file* ('specification file'), so that they can be incorporated as a whole into MAGMA.

10.9.1 Structure of Spec Files

A package spec file is basically a list of filenames, given in a tree format that indicates the directory of each file. All the files and directories are described relative to the directory in which the spec file is stored.

The simplest form of spec file arises from the situation in which the spec file and all the package files reside in the same directory. In that case, the spec file is a list of all the filenames. The filenames must be separated from one another by 'whitespace' characters (space, newline, tab); it is conventional to put them on separate lines. For example, suppose that the package files are `mypck1`, `mypck2`, and `mypck3`, all in the same directory. If the spec file `myspec` is also to be in that directory, then its contents should be:

```
mypck1
mypck2
mypck3
```

In other situations, it is necessary to list directories as well, stating the required files within the directory. The general rule is that each item in a spec file may be: a filename (as above); or a directory name followed by the character { then a whitespace-separated list of spec file items [recursive definition] and then } to close off the directory listing.

For example, suppose that there are some additional package files: `genlpck` in the subdirectory `Grp` of the directory containing `myspec`; `ord1` and `ord2` in the subdirectory `FP` of the subdirectory `Grp`; and `other` in the subdirectory `Misc` of the directory containing `myspec`. The spec file `myspec` should now be:

```
mypck1
mypck2
mypck3
Grp
{
    genlpck
    FP
    {
```

```
        ord1
        ord2
    }
}
Misc
{
    other
}
```

The indentation is not essential, but it indicates the tree-structure of the spec file to the reader.

10.9.2 Using Spec Files

There are two ways to incorporate the packages in a spec file into MAGMA: within a session by attaching the spec file, or (semi-)permanently by means of an environment variable. The first method is for testing the spec file or for occasional usage (just as one attaches an individual package file), whereas the second method is employed in order to make the intrinsics in the spec file into part of the user's standard MAGMA tools.

The procedure for attaching a spec file is **AttachSpec**(F), where the name of the spec file is stored in the string F; it may be detached during the MAGMA session with the procedure **DetachSpec**(F). The effect of attaching a spec file F is the same as that of attaching all the package files specified in F. For example:

```
> AttachSpec("myspec");
> G := ShuffleGroup(3);
> print LowerToUpperCase("tiny");
TINY
```

If the package files are changed, then the package mechanism automatically updates the intrinsics. However, if the spec file is changed there is no updating, so it will need to be detached and re-attached.

There is an environment variable `MAGMA_USER_SPEC` which enables the user to attach one or more spec files automatically each time a MAGMA session is commenced. This variable should be set to a colon-separated list of the full pathnames of the spec files. The normal way to do this is to include a line such as the following in the user's `.cshrc` file:

```
setenv MAGMA_USER_SPEC "$HOME/Magma/spec:/home/friend/specp"
```

where the two spec files are `spec` in the user's `Magma` directory, and `specp` in the top directory belonging to the `friend`. This technique is particularly useful when customizing MAGMA for student purposes, since the modification is transparent to the user.

Packages that have reached the stage of inclusion in a start-up spec file should be made into frozen packages, as explained on p. 224, so as to decrease the execution time slightly.

10.10 Sharing Constants Among Package Files

Sometimes the same constants (integers, magmas, functions/procedures, etc.) are required by several package files. However, since the scope of objects assigned in a package file is limited to that file, MAGMA provides an **import**-statement so that identifiers can be explicitly referenced from outside the package. This technique is designed for the sharing of constants among several packages within a spec file, and is preferable to keeping a copy of the constants in each package.

The syntax of the **import**-statement, as placed in a package file P, is:

import "filename" : identifier list;

where the identifiers are given in a comma-separated list. The filename is either absolute or relative to P, and this file must be included in the spec file along with P, so that it is already attached when the import is performed. The effect of this statement is to create identifiers in the calling context P with the same names and values as they have in the named file. If the file where the identifiers are assigned is changed later, then automatic updating takes place in the context of P.

For example, suppose that the file `mydefs` is a legal package file, included in the spec file `myspec2`. It does not matter for the purposes of this illustration whether `mydefs` contains definitions of intrinsics, but suppose that it includes the following assignment statements:

```
Z := IntegerRing();
Q := RationalField();
R := RealField();
```

Suppose also that the package file `mypck4` is in the same directory as `mydefs` and is also included in `myspec2`. If the user wishes to use Z and R but not Q with these meanings within `mypck4`, then the following statement must be included at the top of `mypck4`:

```
import "mydefs" : Z, R;
```

After this statement, Z will mean the integer ring and R will mean the real field when referred to in `mypck4`. If the values of these identifiers are ever changed in `mydefs`, this change will flow on to `mypck4`.

Part III

The User Interface

11. Online Help

MAGMA is documented in several ways: online help, signatures of intrinsics, hypertext help, and books. This chapter explains how to use MAGMA's *online help system*, a mechanism for obtaining help during a MAGMA session. The online help system includes specially-written material, suitable for near-beginners, and complete access to the descriptions in the *Handbook*. Most of this chapter is devoted to the *online help browser*, a full subsystem with its own syntax. However, the browser section is preceded by an overview of how to gain access to the online help system without entering the browser; this style of help request is reviewed later, in the final section.

Section 10.1 and the following sections demonstrated how to find basic information about intrinsics, such as the categories of their arguments. Since this assistance is obtained during a MAGMA session, it could be described as online help, but it is distinct from the online help system proper.

The online help system has a hypertext version called the *Magma HTML Help Document*, which can be read using an Internet browser. The Magma HTML Help Document is accessed *from the shell (the standard interface to the computer's operating system), not from within* MAGMA. To read it, the user should move to a shell prompt and type

```
magmahelp
```

If the Magma HTML Help Document can be supported by the computer and has been installed correctly, an Internet browser will appear, showing the top node. Since the hypertext help system is self-documenting, it will not be discussed further here.

The major books on the system are this text, the *Handbook* [BoC96], and a book entitled *Solving Problems with Magma* [BCP96] containing longer examples of MAGMA code.

11.1 Single Online Help Requests

It is possible to gain access to the online help system without entering the browser. This technique is suitable for making one or two simple online requests in the midst of a MAGMA task, whereas the browser is appropriate for more extensive enquiries.

Each single online help request begins with a `?` character. If the `?` is followed by a word, the help system looks for help system nodes referenced by that word. (The referencing is not affected by lower/upper case.) If only one node is found satisfying the reference, the online help system shows that node:

```
> ?changedirectory
======================================================================
PATH: /system/detail/system/ChangeDirectory
KIND: Intrinsic
======================================================================
ChangeDirectory(s) : MonStgElt ->

    Procedure.  Change to the directory specified by the string s.
    'Tilde expansion' is allowed.
======================================================================
```

Otherwise, if several nodes satisfy the reference given after the `?`, they are listed with numbers. To see one of these nodes, the user should type `?` followed by its number:

```
> ?permutation
7 matches:
     1   O   /system/database/perfgps/permutation-representation
     2   O   /system/database/pergps
     3   S   /magma/group/character/Boolean/permutation
     4   S   /magma/group/permutation
     5   S   /magma/group/permutation/Boolean/permutation-group
     6   S   /magma/ring-field-algebra/\
             distributive-multivariate-polynomial/group-action/\
             permutation
     7   I   /magma/group/permutation/\
             BSGS-base-strong-generator/operation/Permutation
To view an entry, type ? followed by the number next to it
> ?6
======================================================================
PATH: /magma/ring-field-algebra/polynomial/group-action/permutation
KIND: Section
======================================================================
Permutation Group Actions.

If R is a polynomial ring in n indeterminates x_1, ..., x_n, over
any coefficient ring, Sym(n) acts on R by permuting the indices of
```

```
the indeterminates.  Thus, the polynomial f(x_1, ..., x_n) is
mapped into the polynomial f(x_(g(1)), ..., x_(g(n))).

Intrinsics:

    [+1] ^
======================================================================
```

These commands are the basic ones for single help requests. Observe that help requests do not end with a semicolon, since they are not MAGMA statements.

If `?` is typed alone on a line, information about the help system itself will be given.

11.2 The Online Help Browser System

The *online help browser system* permits more extensive and sophisticated searches for help than are available through single help requests. To enter the online help system, the user should type `??` from within MAGMA:

```
% magma
Magma V1.10-2          Fri Jul  7 1995 17:03:39     [Seed = 1]
Type ? for help.  Type <Ctrl>-D to quit.
> ??
Magma Help Browser
Type help for more information
??>
```

Notice that `??>` is the special prompt symbol for the browser.

For help about the browser, the user should type **help** (while in the browser). This command, like all other commands in the browser, should not be followed by a semicolon, since it is not a MAGMA statement.

The command to leave the browser is **quit**, **exit** or control-D.

The browser commands will now be explained. A summary is given in Table 11.1.

11.2.1 The Online Help Nodes

The online help system consists of many *help nodes*, each of which explains a feature of the language, the system, or one of the categories. Many of the nodes give details of a particular intrinsic as it applies to a magma, or give an example of such an intrinsic in action. The online help system is

structured as a tree, in which each node (except the root) has a parent node relating to a more general aspect of the subject. For instance, the parent of the node for soluble groups is the node for groups, whose parent is the node for magmas, whose parent is the root node. The tree design is similar to the file hierarchies in many operating systems, except that there is no difference between nodes with no children (leaf nodes) and nodes with children, because every node contains text. The command **ls** (for 'list') causes the browser to print the nodenames of the children of the current node. When the browser is first entered, the current node is the root node, so the output should be as follows:

```
??> ls
browser    example    language   magma      system
```

The command **show** followed by a nodename displays the contents of that node, and **show** by itself shows the current node. For instance:

```
??> show system
======================================================================
PATH: /system
KIND: Overview
======================================================================

The Magma system offers several features designed to assist
the user to perform Magma tasks as easily as possible.
These include commands for storing Magma information in
files and retrieving it from them, for editing input lines,
and for timing computations.

Subnodes (if in Browser, type "w" repeatedly to scan them):

    [+1] Magma
    [+2] control-C-key
    [+3] database
    [+4] documentation
    [+5] history
    [+6] intrinsic
    [+7] library
    [+8] load
    [+9] prompt
    [+10] quit
    [+11] save-restore
    [+12] shell-escape
    [+13] signature
    [+14] time
    [+15] detail
======================================================================
??> show magma
======================================================================
PATH: /magma
```

```
KIND: Overview
======================================================================

The primary concept in the design of the Magma system is that of
a 'magma'.  Following Bourbaki, a magma can be defined as
a set with a law of composition.

Thus, types correspond to magmas; a collection of magmas sharing a
common representation forms a category (e.g., the category of
permutation groups); a collection of categories satisfying the same
set of identical relations forms a variety (e.g., the variety of
groups).  Functors may be used to move between categories, and the
variety operations (substructure, homomorphic image, and Cartesian
product) are available as uniform constructors across all
categories.

    ...

Subnodes (if in Browser, type "w" repeatedly to scan them):

    [+1] Boolean
    [+2] aggregate
    [+3] combinatorial
    [+4] group
    [+5] mapping
    [+6] module
    [+7] ring-field-algebra
    [+8] semigroup
    [+9] string

SEE ALSO:
       +10    /system/Magma
======================================================================
```

The command **where** or **pwd** prints the current node. To change the current node, the user should type **goto** or **cd** followed by the desired node. For instance:

```
??> where
Current node: /
??> goto language
??> ls
category                     operator
comment                      statement
expression                   detail
function-procedure-mapping   semantics
identifier
??> goto statement
??> where
Current node: /language/statement
??> show
======================================================================
```

```
PATH: /language/statement
KIND: Overview
=======================================================================

A statement is a complete command to Magma.
Every statement must end with a semicolon ( ; ).

If you press the <return> key before typing ; then Magma
may give you a special prompt symbol such as

print>

or

if>

to remind you that the statement is not yet finished.

EXAMPLE:

> print 5+8;

> DivBy3 := func< n | IsZero(n mod 3) >;

Subnodes (if in Browser, type "w" repeatedly to scan them):

    [+1] assignment
    [+2] case
    [+3] if
    [+4] iteration
    [+5] print
=======================================================================
```

The pathname of each node is given at the top of the output when the node is shown. Every pathname begins with the `/` character. The root node has `/` as its whole pathname, but every other node has a longer path, indicating its place in the tree. Each part of the path is separated from the next by a `/` symbol. For instance, the node in the previous example, `/language/statement`, is the statement node whose parent is the language node whose parent is the root node. The command **up** causes the current node to be changed to its parent node.

11.2.2 Finding a Node

There are a number of ways to seek help on a particular topic within the browser. Although it is possible to perform a series of **goto** and **ls** commands in the hope of finding the goal, this is not the best method unless the user

is literally browsing and has no idea of the name of the topic. The **find** command is generally preferable:

find word

where the word should be the name of the topic or intrinsic.

The topics in the online help tree have been named with nouns in the singular, and so **find** commands should be issued likewise. For example, information about polynomials should be sought using the word `polynomial`, not `polynomials`, and information about dividing should be sought using `division` rather than `dividing` or `divide`. However, the online help system is entirely case-insensitive, so capitals and lower-case letters may be used with identical results; this is advantageous since the same **find** command can find both nodes on topics (section nodes, usually named with lower-case letters) and nodes for intrinsics (usually named with an initial capital). For example:

```
??> find image
8 matches:
     1   S   /magma/group/matrix/image-orbit-stabilizer
     2   S   /magma/group/permutation/G-set/image-orbit-stabilizer
     3   S   /magma/mapping/detail/image-preimage
     4   I   /magma/group/permutation/G-set/\
             image-orbit-stabilizer/Image
     5   I   /magma/mapping/detail/operation/\
             domain-kernel/Image
     6   I   /magma/module/homomorphism/operation-element/Image
     7   I   /magma/module/vector-space-linear-transformation/\
             linear-transformation/Image
     8   I   /magma/ring-field-algebra/matrix/\
             homomorphism-element/Image
To view an entry, simply type the corresponding number
```

The output of a **find** is a numbered list of pathnames. The pathnames suggest the contents of the node, and therefore indicate which node or nodes probably apply to the intended help request. For instance, if the command above had been issued in order to research the image of a mapping in MAGMA, the user would inspect 3 and 5. Only the numbers themselves need to be typed, since the command **show** may be omitted for a node specified by a number:

```
??> 3
======================================================================
PATH: /magma/mapping/detail/image-preimage
KIND: Section
======================================================================
Calculating Images and Preimages.

The standard mathematical notation is used to denote the
calculation of a map image.  Some mappings defined by certain
```

```
system intrinsics and constructors permit the taking of preimages.
However, preimages are not available for any mapping defined
by means of the mapping constructor.

Intrinsics:

    [+1] @
    [+2] @@
=======================================================================
??> 5
=======================================================================
PATH: /magma/mapping/detail/operation/Image
KIND: Intrinsic
=======================================================================
Image(f) : Map -> Elt

    Given a mapping f with domain A and codomain B, return the
    image of A in B as a substructure of B.  This function is
    currently available only for certain intrinsic maps.
=======================================================================
```

The **find** command treats the given word as a *reference* to the node, not as a necessary part of the end of the pathname. For instance, the functions **PrimeBasis** and **PrimeDivisors** are documented in the same node because they are identical, and both names act as references to that node:

```
??> find PrimeDivisors
1 match:
     1   I   /magma/ring-field-algebra/integer/factor/PrimeBasis
To view an entry, simply type the corresponding number
```

The online help system has references for all the alternative names of intrinsics, as in the example above, and also has some special references to help users not familiar with MAGMA's name for, say, a standard programming concept. Additionally, all the words in a multi-word part of a pathname (such as `field` above) are valid references. For instance, in the output below, the sixth node corresponds directly to the requested word, the fifth node is referenced by `orbit`, and the other four nodes contain `orbit` as a word separated by `-` characters in the final part of their pathname:

```
??> find orbit
6 matches:
     1   S   /magma/group/matrix/image-orbit-stabilizer
     2   S   /magma/group/matrix/image-orbit-stabilizer/\
             orbit-action
     3   S   /magma/group/permutation/G-set/\
             image-orbit-stabilizer
     4   S   /magma/group/permutation/G-set/orbit-action
```

```
     5   I   /magma/group/matrix/image-orbit-stabilizer/^
     6   I   /magma/group/permutation/G-set/\
             image-orbit-stabilizer/Orbit
To view an entry, simply type the corresponding number
```

The **find** command has a variant called **delve**. It only looks for nodes referenced by the given word that are at or below the current node. This command is useful when it is expected that the word will reference many nodes. The output below shows how to use **delve** to look for those nodes referenced by `sub` that are concerned with rings rather than groups, modules and so on:

```
??> goto /magma/ring-field-algebra
??> delve sub
8 matches:
     1   I   /magma/ring-field-algebra/finite-Galois-field/\
             creation/structure/sub
     2   I   /magma/ring-field-algebra/finitely-presented/\
             subalgebra/sub
     3   I   /magma/ring-field-algebra/integer/operation/related/\
             sub
     4   I   /magma/ring-field-algebra/matrix/\
             subring-ideal-quotient/sub
     5   I   /magma/ring-field-algebra/number-field/creation/\
             magma/sub
     6   I   /magma/ring-field-algebra/number-field/creation/\
             order/sub
     7   I   /magma/ring-field-algebra/quadratic/creation/\
             structure/sub
     8   I   /magma/ring-field-algebra/residue-class/operation/\
             related/sub
```

The **search** command is provided for exhaustive searching. Given some characters, it looks for every reference having these characters as a substring. The **search** command is useful when the user is having difficulty identifying the required nodes using **find**. For example, compare the following **search** on the substring `orbit` with the output (listed above) of **find** applied to the same word:

```
??> search orbit
36 matches:
     1   S   /magma/group/matrix/image-orbit-stabilizer
     2   S   /magma/group/matrix/image-orbit-stabilizer/\
             orbit-action
     3   S   /magma/group/permutation/G-set/image-orbit-stabilizer
     4   S   /magma/group/permutation/G-set/orbit-action
     5   I   /magma/combinatorial-geometrical-incidence/graph/\
             group-code-design/group/OrbitalDigraph
```

```
 6   I   /magma/combinatorial-geometrical-incidence/graph/\
         group-code-design/group/OrbitalGraph
 7   I   /magma/combinatorial-geometrical-incidence/graph/\
         symmetry-regularity/OrbitsPartition
 8   I   /magma/group/character/Boolean/conjugate/GaloisOrbit
 9   I   /magma/group/matrix/BSGS-base-strong-generator/\
         access/BasicOrbit
10   I   /magma/group/matrix/BSGS-base-strong-generator/\
         access/BasicOrbitLength
11   I   /magma/group/matrix/BSGS-base-strong-generator/\
         access/BasicOrbitLengths
12   I   /magma/group/matrix/image-orbit-stabilizer/^
13   I   /magma/group/matrix/image-orbit-stabilizer/LineOrbits
14   I   /magma/group/matrix/image-orbit-stabilizer/\
         orbit-action/OrbitAction
15   I   /magma/group/matrix/image-orbit-stabilizer/\
         orbit-action/OrbitActionBounded
16   I   /magma/group/matrix/image-orbit-stabilizer/\
         orbit-action/OrbitImage
17   I   /magma/group/matrix/image-orbit-stabilizer/\
         orbit-action/OrbitImageBounded
18   I   /magma/group/matrix/image-orbit-stabilizer/\
         orbit-action/OrbitKernel
19   I   /magma/group/matrix/image-orbit-stabilizer/\
         orbit-action/OrbitKernelBounded
20   I   /magma/group/matrix/image-orbit-stabilizer/\
         OrbitBounded
21   I   /magma/group/matrix/image-orbit-stabilizer/\
         OrbitClosure
22   I   /magma/group/matrix/image-orbit-stabilizer/Orbits
23   I   /magma/group/permutation/BSGS-base-strong-generator/\
         access/BasicOrbit
24   I   /magma/group/permutation/BSGS-base-strong-generator/\
         access/BasicOrbitLength
25   I   /magma/group/permutation/BSGS-base-strong-generator/\
         access/BasicOrbitLengths
26   I   /magma/group/permutation/BSGS-base-strong-generator/\
         access/BasicOrbits
27   I   /magma/group/permutation/BSGS-base-strong-generator/\
         access/IsMemberBasicOrbit
28   I   /magma/group/permutation/G-set/\
         image-orbit-stabilizer/Orbit
29   I   /magma/group/permutation/G-set/\
         image-orbit-stabilizer/OrbitClosure
30   I   /magma/group/permutation/G-set/\
         image-orbit-stabilizer/Orbits
31   I   /magma/group/permutation/G-set/orbit-action/IsOrbit
32   I   /magma/group/permutation/G-set/orbit-action/\
         OrbitAction
33   I   /magma/group/permutation/G-set/orbit-action/OrbitImage
34   I   /magma/group/permutation/G-set/orbit-action/\
         OrbitKernel
35   E   /magma/group/matrix/image-orbit-stabilizer/\
```

```
          Example-Orbits
   36   E  /magma/group/permutation/G-set/orbit-action/\
          Example-OrbitActions
```

11.2.3 Walking the Online Help Tree

It is possible to traverse the online help tree step by step. The easiest method is to **goto** the node from which the walk will begin, and then type **walk** (to go forward) or **back** (to go backward) as many times as required. The following example shows how to read about the statements connected with iteration:

```
??> goto /language/statement/iteration
??> ls
break      continue   for        repeat     while
??> walk
======================================================================
PATH: /language/statement/iteration
KIND: Overview
======================================================================

Magma's statements  while  and  repeat
perform iteration controlled by a Boolean condition.

Another control statement,  for , iterates over
each element of some enumerable domain.
    ...

??> walk
======================================================================
PATH: /language/statement/iteration/break
KIND: Overview
======================================================================

The statement

> break;

inside an iterative statement (loop) causes immediate exit from the
    ...

??> walk
======================================================================
PATH: /language/statement/iteration/continue
KIND: Overview
======================================================================

The statement

> continue;
```

```
inside a while-statement or repeat-statement causes Magma to omit
    ...

??> walk
======================================================================
PATH: /language/statement/iteration/for
KIND: Overview
======================================================================

The for-statement is one of Magma's iterative statements.
It has the syntax:
    ...

??> back
======================================================================
PATH: /language/statement/iteration/continue
KIND: Overview
======================================================================

The statement

> continue;

inside a while-statement or repeat-statement causes Magma to omit
    ...
```

The current node does not change when a **walk** or **back** is performed. The online help system has a separate *walk pointer* that changes instead.

11.2.4 The Kinds of Help Nodes

There are four kinds of help nodes: Overview (O), Section (S), Intrinsic (I) and Example (E). The online help system states the kind of each node when it shows its contents, and when it prints it in a list arising from a statement such as **find**. The S, I and E kinds are taken from the *Handbook*, though some S nodes come from subsections or chapters rather than true chapter-sections. The O nodes are purpose-written for the online help system, so if there is an O node about the user's topic, it should be read before other nodes on the topic. Although the O nodes are designed for relative beginners to MAGMA, they are more terse than the explanations in this book.

It is possible to restrict **find**, **delve**, **walk** and **back** (as well as **search**, explained in Table 11.1 (p. 255)) so that they only select one kind of node. The relevant commands are **ofind**, **sfind**, **ifind**, **efind**, **odelve**, and so on, with the obvious meanings. For instance, if information is wanted on the intrinsic function **Field**, then an **ifind** (a **find** on I nodes only) is sufficient:

```
??> ifind field
```

```
2 matches:
     1   I   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/access/Alphabet
     2   I   /magma/combinatorial-geometrical-incidence/plane/\
             access/Field
To view an entry, simply type the corresponding number
```

The nodes which have been found are the function **Field**(C) (also called **Alphabet**(C)) for a code C, and the function **Field**(P) for a classical plane P.

11.2.5 Cross-References

Some nodes give cross-references, indicating nodes directly under the node being read, or whose subject matter is related to this node. They have special numbers, beginning with a + symbol. Most cross-references are listed at the end of the contents of the node. For example:

```
??> show /language/function-procedure-mapping

    ...

Subnodes (if in Browser, type "w" repeatedly to scan them):

    [+1] error
    [+2] forward
    [+3] function
    [+4] procedure
    [+5] return

SEE ALSO:
        +6    /language/function-procedure-mapping/function
        +7    /language/function-procedure-mapping/procedure
        +8    /magma/mapping
        +9    /system/intrinsic
======================================================================
??> +8
======================================================================
PATH: /magma/mapping
KIND: Overview
======================================================================

This section provides an overview of the ideas
behind mappings, including homomorphisms and
partial mappings.
    ...
```

A few cross-references are given in the body of the text. These can be read in the same way, by typing the + symbol and the appropriate number. For instance, the following lines are contained in the node /language/operator:

```
All Magma operators use call-by-value evaluation, like the
other intrinsics, except the short-circuit operators
'and', 'or', and 'select', which use call-by-name evaluation.
See [+1].
```

The reference above could be read by typing +1 at the browser prompt.

11.2.6 Tab Completion in the Browser

On most operating systems, the *tab completion* feature may be used; compare Section 15.1, which explains tab completion for standard MAGMA statements. (If tab completion is not available, the best substitute is the **search** command.) When the tab key is pressed, the browser looks at the word which the cursor is on or just after, and considers all the commands, references or pathnames (according to the syntax of the command) of which the word could be the beginning. If there are no possibilities, there will be a beep. If there is only one possibility, the word will be completed. If there are several possibilities, there will be a beep and the word will be expanded as far as possible, to the end of the common prefix of all the possibilities. In this last case, if the tab key is pressed again, the browser will list all of the possible completions, and repeat the input line so that the user can finish it.

For example, the following lines demonstrate how to see the references concerned with prime numbers. The user can then choose some of these nodes to investigate.

```
??> find Prime  [user types 2 tabs]
Prime               PrimeDivisors       PrimeRing
PrimeBasis          PrimeField
PrimeCertificate    PrimeForm
??> find Prime
```

When employing the tab completion feature, the user should type as little as possible of the beginning of the word, so as not to miss anything relevant. For instance, the reference Primality might also be of interest in the example above, but it was not found because it did not begin with prime (case-insensitive). The following command would have found it:

```
??> find Prim  [user types 2 tabs]
Primality                   PrimeDivisors
primality                   PrimeField
```

```
Primary                    PrimeForm
PrimaryDecomposition       PrimeRing
PrimaryInvariantFactors    primitive
PrimaryInvariants          PrimitiveElement
PrimaryRationalForm        PrimitivePart
Prime                      PrimitivePolynomial
PrimeBasis                 PrimitiveRoot
PrimeCertificate           PrimitiveStructure
??> find Prim
```

However, some of these references are not associated with prime numbers. There is a trade-off between obtaining all the relevant references and not obtaining an excessive number of irrelevant references.

A series of tab completions on a pathname may be used in a kind of inline browsing, with very few keystrokes required. For example:

```
??> show /  [user types 2 tabs]
browser     example     language    magma       system
??> show /m  [user types tab]
    ... above line becomes:
??> show /magma  [user types tab]
    ... above line becomes:
??> show /magma/  [user types tab]
    ... above line becomes:
??> show /magma/  [user types 2 tabs]
Boolean
aggregate
combinatorial-geometrical-incidence
group
mapping
module
ring-field-algebra
semigroup
string
??> show /magma/g  [user types 4 tabs]
    ... above line becomes:
??> show /magma/group/
group-overview        soluble               character
abelian               permutation
finitely-presented    matrix
??> show /magma/group/m  [user types 4 tabs]
introduction
creation-general-linear-group
operation-element
creation-general-matrix-group
extension-standard-group
change-ring
operation
element
image-orbit-stabilizer
operation-subgroup
```

```
Boolean
characteristic-subgroup-normal-structure
conjugate
soluble-matrix-group
character-representation
BSGS-base-strong-generator
??> show /magma/group/matrix/b  [user types 2 tabs]
    ... above line becomes:
??> show /magma/group/matrix/B
Boolean                       BSGS-base-strong-generator
??> show /magma/group/matrix/Bo  [user types 4 tabs]
    ... above line becomes:
??> show /magma/group/matrix/Boolean/
matrix-group      abstract-group
??> show /magma/group/matrix/Boolean/a  [user types 4 tabs]
    ... above line becomes:
??> show /magma/group/matrix/Boolean/abstract-group/Is
IsAbelian              IsNormal
IsCyclic               IsPerfect
IsElementaryAbelian    IsSelfNormalizing
IsCentral              IsSimple
IsConjugate            IsSoluble
IsMaximal              IsSubnormal
IsNilpotent
??> show /magma/group/matrix/Boolean/abstract-group/Is
   ...
```

11.2.7 Recalling Nodes

The command **numbers** reprints the list of nodes from the most recent **find**, **delve** or **search** command. It is helpful when the user has looked at the contents of some nodes in such a list and has forgotten the numbers of the other nodes.

Another way of keeping track of useful nodes is to **mark** them for later reading. The following example shows how to mark the nodes connected with cyclic linear codes:

```
??> search cyclic
18 matches:
     1   S   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/creation/cyclic
     2   S   /magma/group/soluble/introduction/\
             polycyclic-power-conjugate
     3   I   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/Boolean/IsCyclic
     4   I   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/creation/cyclic/\
             CyclicCode
```

```
     5   I   /magma/group/abelian/group-Boolean/IsCyclic
     6   I   /magma/group/finitely-presented/\
             extension-standard-group/standard/CyclicGroup
     7   I   /magma/group/group-overview/creation-general/magma/\
             Group
     8   I   /magma/group/group-overview/extension-standard-group/\
             standard/CyclicGroup
     9   I   /magma/group/group-overview/operation-subgroup/\
             abstract-group/IsCyclic
    10   I   /magma/group/matrix/Boolean/abstract-group/IsCyclic
    11   I   /magma/group/matrix/soluble-matrix-group/conversion/\
             PolycyclicGenerators
    12   I   /magma/group/permutation/Boolean/abstract-group/\
             IsCyclic
    13   I   /magma/group/permutation/extension-standard-group/\
             standard/CyclicGroup
    14   I   /magma/group/soluble/creation-element/magma/\
             PolycyclicGroup
    15   I   /magma/group/soluble/group-Boolean/IsCyclic
    16   I   /magma/group/soluble/subgroup-quotient-extension/\
             standard-group/CyclicGroup
    17   E   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/creation/cyclic/\
             Example-CyclicCode
    18   E   /magma/group/soluble/creation-element/magma/\
             Example-PolycyclicGroup
??> mark 1
['1] S /magma/combinatorial-geometrical-incidence/\
       error-correcting-linear-code/creation/cyclic
??> mark 3
['2] I /magma/combinatorial-geometrical-incidence/\
       error-correcting-linear-code/Boolean/IsCyclic
??> mark 4
['3] I /magma/combinatorial-geometrical-incidence/\
       error-correcting-linear-code/creation/cyclic/CyclicCode
??> mark 17
['4] E /magma/combinatorial-geometrical-incidence/\
       error-correcting-linear-code/creation/cyclic/\
       Example-CyclicCode
```

Now these nodes will have the special node specifications `'1`, `'2`, `'3` and `'4`. For example:

```
??> '3
======================================================================
PATH: /magma/combinatorial-geometrical-incidence/\
      error-correcting-linear-code/creation/cyclic/CyclicCode
KIND: Intrinsic
======================================================================
CyclicCode(u) : ModTupFldElt -> Code

    Given a vector u belonging to the K-space K^((n)), construct
```

```
    the [n, k] cyclic code generated by the right cyclic shifts of
    the vector u.

CyclicCode(n, g) : RngIntElt, RngUPolElt -> Code

    Let K be a finite field.  Given a positive integer n and a
    univariate polynomial g(x) in K[x] of degree n - k such that
    g(x) | x^n - 1, construct the [n, k] cyclic code generated
    by g(x).
======================================================================
```

The command **marks** lists the marked nodes:

```
??> marks
    '1   S   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/creation/cyclic
    '2   I   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/Boolean/IsCyclic
    '3   I   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/creation/cyclic/\
             CyclicCode
    '4   E   /magma/combinatorial-geometrical-incidence/\
             error-correcting-linear-code/creation/cyclic/\
             Example-CyclicCode
```

The browser system will remember these marks for the rest of the MAGMA session, even if the user quits the browser and does some MAGMA computations before re-entering the browser.

The command **unmark** followed a mark number removes that mark from the list of marks.

11.2.8 Summary of Browser Commands

Table 11.1 summarizes the online help system browser commands, but omits the prefixed versions of the commands, such as **efind** and **odelve**. In the table, n denotes a node specification, which may be any of: an absolute pathname; a pathname relative to the current node (e.g., `iteration/while` when the current node is `/language/statement`); an integer in the current **numbers** range; the character `+` and a integer within the current cross-reference range; the character `'` and a integer within the current marks range; or the character `"`, indicating the last node to which the user referred.

Every browser command may be abbreviated to a substring with which it begins. If more than one command begins with the given substring, the browser interprets it as the command that comes first alphabetically. (Exceptions are **s** for **show** and **e** for **exit**.) For example, some command abbre-

Table 11.1. Online help system browser commands

MAGMA	Meaning
`??`	Enter the online help system browser
quit, **exit**, control-D	Leave the browser
help	Help about browser (node `/browser`)
?	Help about help system (root node `/`)
show, **cat**, **more**	Display text of current node
show n, **cat** n, **more** n	Display text of help node specified by n
where, **pwd**	State current node
ls	List children of current node
ls n	List children of n
goto, **cd**	Change current node to root node `/`
goto n, **cd** n	Change current node to n
up	Change current node to parent of current node
up n	Change current node to parent of n
walk, **back**	Go forward/backward with respect to walk pointer
walk n, **back** n	**show** n, and set walk pointer to it
find w	List nodes referenced by word w
delve w	List nodes referenced by word w that are beneath current node in the help tree
search s	List nodes whose references contain substring s
numbers	Reprint list from most recent **find**, **delve** or **search**
i	**show** i, where $i \in \mathbb{Z}$ is a node specification
mark n	Mark n (or current node, if n omitted) for future reference; n will get a node specification $'i$, where $i \in \mathbb{Z}$
unmark m	Remove mark m from marks list
marks m	List the marks list

viations are **d** (**delve**), **f** (**find**), **iw** (**walk** along the intrinsics) and **sb** (**back** along the sections).

11.3 Other Features of Single Online Help Requests

A single help request, that is, one beginning with a `?` character, is rather like **find** or **show** in the browser. If the `?` is followed by a reference word, then the system performs a **find**. In the case that there is exactly one match, a **show** is performed as well. The `?` may also be followed by a full pathname, a number, or a cross-reference. In these cases, it performs a **show**. For example:

```
> ?/system/time
```

```
======================================================================
PATH: /system/time
KIND: Overview
======================================================================


Magma has a mechanism for measuring how long it
takes to execute a statement or statements.

    ...

======================================================================
>
```

Single help requests (i.e., those starting with `?`) may be made *within* the browser, but they operate somewhat independently from the browser. For instance, they do not change the value of `"`, the last-node character.

Tab completion is available following `?`, although only outside the browser. This facility greatly enhances the usefulness of single online help requests from the MAGMA level, especially for users who are accustomed to the structure of the online help system tree.

12. Strings

Strings are strongly associated with MAGMA input and output, but they also belong to a magma in their own right, the string structure. The category of this structure is the character string monoid. The fundamental string operations are concatenation and indexing, and thus they share some operations with other monoids and with sequences. Sophisticated operations for the analysis of strings are provided, including the matching of regular expressions.

12.1 Creating a String

It has already been seen in this book that MAGMA allows text – letters and any other characters – to be part of the output of a **print**-statement. For example:

```
> print "The result of 6 times 7 is", 6 * 7;
The result of 6 times 7 is 42
```

Text like this is called a *string*, and it is composed of *characters* (`T`, `h`, `e`, the space character, `r`, etc.). When a literal string is typed into MAGMA, the characters must be enclosed in `"` symbols (double quotation marks).

Like any other values, strings can be assigned to identifiers. For instance, if the above line is rewritten as

```
> ResultStr := "The result of 6 times 7 is";
> print ResultStr, 6 * 7;
The result of 6 times 7 is 42
```

then the output is the same.

Every character in a string is significant, even the spaces. For instance:

```
> AreEqualStrings := ResultStr eq "Theresultof6times7is";
```

```
> print AreEqualStrings;
false
```

All strings have the same parent, the *string structure*; the function **Strings**() returns this magma. The category of the string structure is the *character string monoid*, **MonStg**, and the category of each string is **MonStgElt**.

12.2 Operations on Strings

Table 12.1. Operations on strings

MAGMA	Meaning
`#`s	Length of string s
s`*`t, s **cat** t	String formed by concatenating s and t
s`^`n	s`*`$\cdots$`*`s (n copies concatenated)
Position(s,t), **Index**(s,t)	First character position of s at which a copy of t starts, or zero if t not a substring of s; if t is empty, always returns 1
$s[i]$	The length-1 string which is the i^{th} character of string s
Substring(s,n,k)	Substring of s starting at position n with length k (or extending to end of string, if $k > \#s-n+1$)
s **eq** t	**true** if string s is equal to string t
s **ne** t	**true** if s is not equal to t
s **in** t	**true** if s is a contiguous substring of t
s **notin** t	**true** if s not a contiguous substring of t
s **gt** t	**true** if s comes later than t (with respect to the standard lexicographic ordering)
s **ge** t	**true** if s equals t or comes later than t
s **lt** t	**true** if s comes earlier than t
s **le** t	**true** if s equals t or comes earlier than t

The character string monoid is so named because it is a monoid in the sense of semigroup theory. The main operation for strings is concatenation, and the identity of the string structure is the empty string (the string with zero characters). The concatenation operator may be typed as the `*` symbol, and `^` has a corresponding function as an exponentiation operator.

On the other hand, because concatenation and the position of characters in strings are important operations for strings, several of the string operators

and functions are the same as those for sequences. Thus **cat** is another way of expressing the concatenation operator, and the indexing facilities are similar.

Table 12.1 summarizes the basic operations on strings, incorporating both the monoid-style operations and the sequence-style operations.

The length of a string is defined as the number of characters it has. Given a string s, `#`s returns its length. Thus the statement

```
> print #ResultStr;
26
```

shows that *ResultStr* is 26 characters long, and the following lines verify that the identity of the string structure has zero length:

```
> print #Identity(Strings());
0
```

A particular character in a string is obtained by giving its position in square brackets, so the following statement prints the seventh character of *ResultStr*:

```
> print ResultStr[7];
s
```

(Unlike the situation for sequences, it is not possible to mutate a string by making an assignment to the i^{th} place. Such manipulations are best performed using sequence operations.)

Conversely, the function **Position**(s, t) or **Index**(s, t) returns the position at which a substring t first appears in a string s. If t is not a *contiguous substring* of s, that is, if there are not adjacent characters in s that match t, then the function returns 0. (As a special case, if t is empty then the value returned is 1, even if s is also empty.) For example, the following output shows that `"es"` first occurs in *ResultStr* starting at the sixth position:

```
> print Position(ResultStr, "es");
6
```

but *ResultStr* does not contain the string `"aardvark"`:

```
> print Position(ResultStr, "aardvark");
0
```

The relational operators **eq** and **ne** have the usual meaning of equality-testing when applied to two strings. The operators for testing whether a string is a contiguous substring of another string are **in** and **notin**. For instance:

```
> print "om" in "computational";
true
> print "magma" notin "volcano";
true
```

Strings may be composed from smaller strings by concatenation, using the * operator (or the operator **cat**, which behaves in the same way). For instance:

```
> L := "left";
> R := "right";
> LR := L cat " " cat R;
> print LR;
left right
> print LR eq (L * " " * R);
true
```

The mutation assignment operators **cat**:= and *:= are available, as are the reduction operators &**cat** and &*.

The ^ operator relates to * in the same way as it does for integers or monoid elements. Given a string s and a non-negative integer n, s^n returns the concatenation of n copies of s. For instance, the following lines construct four cycles of a marching call:

```
> LRsp := LR * " ";
> marching := LRsp ^ 4;
> print marching;
left right left right left right left right
```

The operators **lt**, **le**, **gt**, and **ge** compare two strings with respect to alphabetical order. (More exactly, these operators refer to *lexicographic ordering*, according to the character code used by the computer system on which MAGMA is running; for most systems it will be the ASCII character set.) For instance, the statement below prints the value **false**, because 'xylophone' comes ahead of 'yacht' in dictionary order:

```
> print "xylophone" gt "yacht";
false
```

As a longer illustration of the string operations, suppose that a sentence such as 'There is a chair.' or 'There are some books.' is to be printed, based on the value of a noun such as 'chair' or 'books'. The following procedure will perform this task:

```
> Sentence := procedure(Noun)
```

```
>     VerbArticle := Noun[#Noun] eq "s" select
>         " are some " else " is a ";
>     print "There" * VerbArticle * Noun * ".";
> end procedure;
```

The procedure works well for most sensible input, but fails for singular nouns that begin with a vowel:

```
> Sentence("chair");
There is a chair.
> Sentence("books");
There are some books.

> Sentence("egg");
There is a egg.
```

A modification can be made to cater for this case:

```
> Sentence := procedure(Noun)
>     if Noun[#Noun] eq "s" then
>         Verb := " are ";
>         Article := "some ";
>     else
>         Verb := " is ";
>         if Noun[1] in "aeiouAEIOU" then
>             Article := "an ";
>         else
>             Article := "a ";
>         end if;
>     end if;
>     print "There" * Verb * Article * Noun * ".";
> end procedure;

> Sentence("egg");
There is an egg.
```

While this modification is simple, it is harder to write a procedure which will cope with nouns such as ‘boss’ and ‘women’, for which having a final ‘s’ is not equivalent to being plural. Regrettably, it is not so easy to teach human languages to computers.

Table 12.2. Special characters in strings

MAGMA	Meaning
`\t`	Tab character
`\n`, `\r`	New-line or return character
`\"`	Double quotation mark character
`\\`	Backslash character
`\<return>`	Line continuation (ignored)

12.3 Special Characters in Strings

There are some characters that it is not possible to obtain straightforwardly at the keyboard. These characters can be included in strings by using a special symbol beginning with a backslash. Table 12.2 lists them.

Tab characters and newlines are useful for manipulating the presentation of output. For instance:

```
> print "Tab is soon\tThere it was\nsecond line now";
Tab is soon     There it was
second line now
```

To include the " character in a string without having MAGMA think that the string has finished, it must be preceded by a backslash. To obtain the backslash character itself, it must also be typed with an extra backslash:

```
> print "a double quotation mark \" and\ra backslash \\";
a double quotation mark " and
a backslash \
```

There is a related device called *line continuation* that is useful for long input lines. If a backslash is followed immediately by the 'return' key, MAGMA will provide a newline on the keyboard but will ignore both the backslash and the 'return' internally. For example:

```
> print "This could well be described as an extreme\
> ly long line";
This could well be described as an extremely long line
```

Line continuation works outside strings as well, and so it can be used for long numerical input or for presentations of large magmas:

```
> print 12345678901234567890123456789012345\
```

```
> 1234567890;
12345678901234567890123456789012345678901234567890
```

12.4 Analysis of Strings

12.4.1 String Conversions

Sophisticated operations on a string, such as changing parts of it, reordering it, or seeing how often the same character appears in it, are best done by converting the string into a sequence first. Sequence operations are explained in Chapter 6. The following lines show some MAGMA functions for converting between a string and the sequence of its characters in the same order. They use a sequence constructor, string indexing, and the reduction operator for concatenation:

```
> StringToSequence := func< s | [ s[n] : n in [1..#s] ] >;
> SequenceToString := func< q | &* q >;
> // define abbreviations as well
> Strseq := StringToSequence;
> Seqstr := SequenceToString;
```

For example:

```
> Str := "asdliua0987tas";
> Sq := StringToSequence(Str);
> print Sq;
[ a, s, d, l, i, u, a, 0, 9, 8, 7, t, a, s ]
> print SequenceToString(Sq);
asdliua0987tas
```

There are several intrinsics for more specialized conversions. The functions **IntegerToString**(n) and **StringToInteger**(s) convert an integer into a string, and an integer-string into an integer. Here an *integer-string* is a string that consists entirely of digits, except that the first character may be a `+` or `-` character. For example:

```
> s := "-05663"; print StringToInteger(s);
-5663
```

A related function is **StringToIntegerSequence**(s), which takes a string s consisting of integer-strings, and returns a sequence of integers corresponding to these integer-strings. The integer-strings must be separated from one another by one or more spaces, or by an initial `+` or `-` sign:

```
> print StringToIntegerSequence("1234 -58     +04-283");
[ 1234, -58, 4, -283 ]
```

It is possible to convert between a one-character string and the character code that it has in the computer's operating system, using **CodeToString**(n) and **StringToCode**(s). For most systems on which MAGMA is running, this will be the ASCII character code. The following example, which assumes an ASCII context, converts a string of lower-case letters into the corresponding string of upper-case (capital) letters:

```
> LowerToUpperCase := function(s)
>     STC := StringToCode; // shorthand
>     q := [ STC(c) : c in Strseq(s) ];
>     error if q notsubset [STC("a")..STC("z")],
>       "Runtime error in LowerToUpperCase:",
>       "String characters must be a, ..., z";
>     R := STC("A") - STC("a");
>     return Seqstr([ CodeToString(c + R) : c in q ]);
> end function;

> print LowerToUpperCase("MAGMA");
Runtime error in LowerToUpperCase: String characters
must be a, ..., z
> print LowerToUpperCase("magma");
MAGMA
```

See p. 221 and p. 227 for the implementation of these functions as user intrinsics.

12.4.2 Splitting into Substrings

Given strings s and d, the function **Split**(s, d) splits s into a sequence of substrings, where the separation points are the places in s at which one or more of the characters in d occur one or more times. The characters in the delimiter string d (such as a space, a newline, a tab, a comma, or a colon) are not included in the substrings in the resulting sequence. More precisely, if s is of the form $s_1 d_1 s_2 d_2 \cdots d_{k-1} s_k$ or $s_1 d_1 s_2 d_2 \cdots d_{k-1} s_k d_k$ where the strings s_i and d_i are all non-empty (except possibly s_1) such that all the characters of each d_i are in d and none of the characters of any s_i are in d, then the function returns the sequence $[s_1, \ldots, s_k]$. As an exception, if s is empty then the function returns an empty sequence.

For example, the following invocation of **Split** takes a string *str* consisting of letters and digits, and constructs the letter-only substrings of *str* that are separated by digits:

```
> str := "qj23jk259xvnsd9832h34r98723un5kj";
> print Split(str, "1234567890");
[ qj, jk, xvnsd, h, r, un, kj ]
```

The next example illustrates how to use *whitespace* (spaces, tabs, and newlines) as the delimiter:

```
> str2 := "The quick brown fox\tjumps
> over a    lazy\t\t\tdog";
> print str2;
The quick brown fox     jumps
over a    lazy                          dog
> print Split(str2, " \t\n");
[ The, quick, brown, fox, jumps, over, a, lazy, dog ]
```

There is an alternative form of this function, **Split**(s), which splits the string s into its distinct lines. That is, it uses the newline character as the delimiter d. For instance, *str2* above would be split into two lines:

```
> print Split(str2);
[ The quick brown fox   jumps,
over a    lazy                 dog ]
```

12.4.3 Regular Expressions

The function **Regexp**(r, s) analyzes the string s according to the regular expression given by the string r. Its principal return value is **true**, if s matches r, else **false**. If the first return value is **true**, it returns two additional values: the substring of s that was found to match r, and the sequence of matched substrings of s corresponding to the parenthesized expressions of r. The meaning of these return values is explained below.

The regular expression r is used to describe a pattern that may be satisfied by a substring of s. Apart from literal characters, it may contain any of the special symbols listed in Table 12.3. For example:

```
> str3 := "quick brown fox";
> m, ss := Regexp("ui.*o", str3);
> print m;
true
> print ss;
uick brown fo

> m, ss := Regexp("[ e-z]*$", str3);
```

Table 12.3. Special characters in regular expressions

MAGMA	Meaning
a|b	String a or string b
a*	Zero or more copies of a
a+	One or more copies of a
a?	Zero or one copies of a
^a	String a at the beginning of the expression
$a$$	String a at the end of the expression
.	Any character
[$c_1 \cdots c_k$]	Any single character c_i
[^$c_1 \cdots c_k$]	Any single character except one of the c_i
[c_1-c_k]	Any single character in the range c_1 to c_k, according to the machine character code (e.g., consecutive letters or digits)
[^c_1-c_k]	Any single character not in the range c_1 to c_k
\c	Character c as a literal, without its special meaning (used for characters in this table)
(a)	Grouping symbol to restrict scope; a will be a term of the sequence given as the third return value of **Regexp**

```
> print m;
true
> print ss;
rown fox
```

If some parts of r are enclosed in parentheses, there will be two effects. Firstly, it will restrict the scope of that part of the expression. Secondly, each section of the matched substring corresponding to a parenthesized portion of r will be returned in a sequence, in order from left to right, as the third return value of **Regexp**. For instance:

```
> print Regexp("u(i.)*o", str3);
false

> print Regexp("[a-z]([a-z]*)[a-z] (.)[a-z]*(.) ", str3);
true quick brown  [ uic, b, n ]
```

The final example uses **Regexp** and **Pipe** (see Section 14.8.2) to extract information about the date from the output of the UNIX command `date`:

```
> date := Pipe("date\n", "");
> print date;
Tue Jul 23 14:31:14 EST 1996
```

```
> _, ss, q :=
>     Regexp(" (.*) ([0-9][0-9]?) .*EST (....)", date);
> print ss;
 Jul 23 14:51:53 EST 1996
> print q;
[ Jul, 23, 1996 ]
> month, day, year := Explode(q);
> printf "Today is %o %o %o.", day, month, year;
Today is 23 Jul 1996.
```

For more details about **Regexp**, refer to documentation for the freely distributable reimplementation of the V8 `regexp` package by Henry Spencer, on which **Regexp** is based.

13. Printing and User Input

13.1 Levels of Printing in the print-statement

The most common syntax for the **print**-statement in MAGMA is:

print expression, ..., expression;

where the word **print** may be omitted if the statement is not within a function or procedure definition. It prints the values of the expressions, in order. If there is more than one expression, the values may be printed on new lines, depending on the categories of the values.

There is also a more general form of the **print**-statement, that specifies the kind of output required:

print expression, ..., expression : level;

There are four possibilities for the level: **Minimal**, **Default**, **Maximal** and **Magma**. The first three of these options specify the quantity of the output, from the least to the greatest (though for many categories, two or three of the levels produce the same quantity of output). If the colon and the level are omitted entirely, as shown in the first version of the syntax above, then the **Default** level is used. The level **Magma** specifies that the output is to be in MAGMA-readable format; that is, it should be possible to copy the output and paste it into MAGMA as an expression that evaluates to the same value.

For example:

```
> F3 := GF(3);
> MR := MatrixRing< F3, 2 | [2,1,1,0] >;
> print MR;
Matrix Algebra of degree 2 with 1 generator over GF(3)
> print MR : Magma;
MatrixAlgebra<GF(3, 1), 2 | [ 2, 1, 1, 0 ]>
```

```
> A4 := Alt(4);
> print A4;
Permutation group A4 acting on a set of cardinality 4
Order = 12 = 2^2 * 3
    (1, 2)(3, 4)
    (1, 2, 3)
> print A4 : Minimal;
GrpPerm: A4

> D := Design< 3, 5 | Setseq(Subsets({1..5}, 3)) >;
> print D;
3-(5, 3, 1) Design with 10 blocks
> print D : Maximal;
3-(5, 3, 1) Design with 10 blocks
Points: {@ 1, 2, 3, 4, 5 @}
Blocks:
    {1, 3, 5},
    {2, 3, 4},
    {1, 4, 5},
    {2, 4, 5},
    {1, 2, 5},
    {1, 2, 3},
    {1, 2, 4},
    {3, 4, 5},
    {2, 3, 5},
    {1, 3, 4}
> print D : Magma;
Design< 3, {@ IntegerRing() | 1, 2, 3, 4, 5 @} |
KMatrixSpace(GF(2), 5, 10) ! [ 1, 0, 1, 0, 1, 1, 1, 0, 0,
1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0,
1, 1, 0 ] : Check := false >
```

13.2 Printing an Object to a String

The output obtained when printing an expression x may be obtained as a string by means of the **Sprint** function. Any of the levels in the **print**-statement may be specified.

For the default level of printing, **Sprint**(x) returns a string whose value is the output that would be given by a **print** x; statement. For example:

```
> str := Sprint(Alt(4));
```

```
> print #str;
94
> print str;
Permutation group acting on a set of cardinality 4
Order = 12 = 2^2 * 3
(1, 2)(3, 4)
(1, 2, 3)
> _, _, ordq := Regexp("Order = (.*) =", str);
> ord := Rep(ordq); print ord;
12
```

For the other levels of printing, suppose that L is any of the strings `"Minimal"`, `"Default"`, `"Maximal"`, or `"Magma"`. Then **Sprint**(x, L) returns a string whose value is the output that would be given by printing x with the corresponding level. For example, parts of the MAGMA description of a design may be extracted (as strings):

```
> D := Design< 3, 5 | Setseq(Subsets({1..5}, 3)) >;
> str2 := Sprint(D, "Magma");
> print str2;
Design< 3, {@ IntegerRing() | 1, 2, 3, 4, 5 @} |
KMatrixSpace(GF(2), 5, 10) ! [ 1, 0, 1, 0, 1, 1, 1, 0, 0,
1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0,
1, 1, 0 ] : Check := false >
> _, _, q := Regexp(
>    "([0-9]*), ({@.*@}).*(KMatrixSpace.*) : Check", str2);
> t, pts, blks := Explode(q);
> print t;
3
> print pts;
{@ IntegerRing() | 1, 2, 3, 4, 5 @}
> print blks;
KMatrixSpace(GF(2), 5, 10) ! [ 1, 0, 1, 0, 1, 1, 1, 0, 0,
1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,
1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0,
1, 1, 0 ]
```

13.3 Verbose Printing

For some of the major algorithms in MAGMA, it is possible to request that information on the progress of the algorithm be provided during execution. This additional information is called *verbose printing*. The *Handbook* states

whether a given intrinsic offers verbose printing. By default, all the flags for verbose printing are turned off.

The setting for a verbosity flag may be altered to the value x by means of the procedure **SetVerbose**(s, x); similarly, the function **GetVerbose**(s) returns the current setting. Here s is a string giving the name of the flag, examples being `"OnanScott"` and `"Conj"`. For flags which have only two settings ('off' and 'on'), x should be given as 0 or **false** to mean 'off', but as 1 or **true** to mean 'on'. Some flags have several levels of verbosity, in which case the values 0 and **false** denote 'off', small positive integers denote the successive levels of verbosity, and **true** denotes the minimum level of verbosity.

There are also five spare flags available for use in user-defined intrinsics or functions/procedures. Their names are `"User1"`, ..., `"User5"`, and their possible settings are 0 or **false** ('off', the default) and 1 or **true** ('on').

An example of the use of a verbosity flag is given on p. 685.

13.4 Formatted Printing

MAGMA is not designed to cope with large word-processing tasks, but it has enough text-handling ability to provide well-presented output, containing words as well as computational results. The facilities provided for this include the tab and newline (return) characters, string concatenation, string conversion functions, and string analysis in general. Another major tool for formatted printing is the **printf**-statement, a variant of the **print**-statement.

The syntax of the **printf**-statement is:

printf format string, expression,...,expression;

where the format string describes how the values of the expressions are to be printed. The format string is a normal string, except that it must contain instances of the characters `%o` (standing for 'object') and/or `%m` (standing for 'magma') the same number of times as there are expressions following the format string. (The format string does not have to be a literal string; it may be any expression that returns a string.) The effect of the **printf**-statement is to print the format string, substituting the values of the expressions in the places marked by the `%o` or `%m` symbols. Values of expressions corresponding to `%o` symbols are printed in the `"Default"` printing mode, whereas values of expressions corresponding to `%m` symbols are printed in the `"Magma"` printing mode. All kinds of values are permissible. This technique allows greater formatting control than printing a comma-separated list of expressions in an ordinary **print**-statement.

If a newline is desired at the end of the output of the **printf**, the newline character `\n` must be included in the format string. Otherwise, no newline will take place. The reason for this is to allow an output line to be built up gradually from several **printf**-statements. For example:

```
> demo := procedure(a, b, c)
>     printf "first value %o ", a;
>     printf "and second value %o\n", b;
>     printf "and third value %o\n", c;
> end procedure;
> demo(4+5, 2/3, 1000);
first value 9 and second value 2/3
and third value 1000
```

There is an exception if MAGMA is about to give a prompt symbol and there is output waiting to be printed. In that case, MAGMA *flushes the buffer* (prints the text) and performs a newline before giving the prompt. For example:

```
> printf "fourth value %o", 1/2;
fourth value 1/2
> printf "and fifth value %o", 3628;
and fifth value 3628
```

Nonetheless, it is good practice to supply the newline character explicitly, even in this situation.

The next example compares the effect of the `%o` and `%m` symbols:

```
> G := Sym(3);
> printf "Default style first\n%o\nthen Mag\
> ma style\n%m\n", G, G;
Default style first
Symmetric group G acting on a set of cardinality 3
Order = 6 = 2 * 3
then Magma style
Sym(3)  /* order = 6 = 2 * 3 */
```

The *Sentence* procedure on p. 261 may be written more elegantly using a **printf**-statement. Notice that the spaces around *Verb* and *Article* are no longer required, since all the spacing is handled by the format string:

```
> Sentence := procedure(Noun)
>     if Noun[#Noun] eq "s" then
>         Verb := "are";
>         Article := "some";
```

```
>     else
>         Verb := "is";
>         if Noun[1] in "aeiouAEIOU" then
>             Article := "an";
>         else
>             Article := "a";
>         end if;
>     end if;
>     printf "There %o %o %o.\n", Verb, Article, Noun;
> end procedure;
```

The output is the same as before.

The **printf**-statement is often used in combination with tab characters and newlines, for tabulated output. See p. 502 for an example.

13.5 Reading Input from the User

There are two commands which make it possible to get input from the user during a MAGMA session: **read**, which obtains string input, and **readi**, which obtains integer input. Each of these has a plain form and a form that also prints some explanatory text for the user. The input commands are most useful within a function or procedure, since they allow the person who calls the routine to control its progress.

The plain form of the **read**-statement is:

read identifier;

This statement causes MAGMA to pause, waiting for the user to type some input and then the 'return' key. After this, the identifier will be assigned the string consisting of all the characters typed before the 'return' key. (If no characters are typed before 'return', the result will be the empty string.)

In the following example, the second line is typed by the user, and the fourth line is MAGMA output. As for the first and third lines, they are keyboard input as well, but it would be more usual for them to come from an input file or from the body of a function or procedure.

```
> read jane;
austen
> print jane;
austen
```

It is advisable in the context of a **read**-statement to tell the user that input is required, and to indicate the kind of input that is appropriate. The best way to do this in MAGMA is to use another kind of **read**-statement:

read identifier, string;

where the string is an expression returning a string that is printed before the reading takes place. This string should be a prompting phrase, worded so as to make the intention clear to the user. For example:

```
> read name, "Type your name then press ‘return’";
Type your name then press ‘return’
Algernon
> printf "Hello, %o!\n", name;
Hello, Algernon!
```

The **readi**-statement has the same syntax as the **read**-statement, but it is designed for integer input. The input line typed by the user must be a representation of a literal integer; that is, the characters must form an integer-string as defined on p. 263. For example:

```
> readi age, "What is your age (in years)?";
What is your age (in years)?
79
> printf "Next year you will be %o.\n", age + 1;
Next year you will be 80.
```

14. Files and External Processes

There are several ways in which MAGMA can interact with files: loading input files (during the session or at start-up); saving in a file the output or the whole log of a session; reading and writing to data files; saving the workspace of a session for subsequent restoration; and attaching package files to MAGMA to create user intrinsics. This chapter explains all of these facilities except for package files, which are covered in Chapter 10. It also explains the mechanisms for system calls and pipes.

14.1 The Current Directory

Within MAGMA, a file is usually described by means of a string. The contents of this string may be the *full pathname* of the file (tilde expansion is allowed), or a *pathname relative to the current directory*. (As discussed below, there is a third possibility, involving the `MAGMA_PATH` environment variable, for files of MAGMA code that are loaded at start-up or by means of the **load**-statement.) Therefore it is important for the user to know what the current directory is before attempting to interact with files.

At the beginning of a MAGMA session, the current directory is the directory from which the `magma` command was issued. **GetCurrentDirectory**() returns the current directory, and the procedure **ChangeDirectory**(D) changes it to the directory given in the string D.

14.2 Loading Files During a Session

The **load**-statement may be used both to load pre-prepared MAGMA code and to load objects defined in the MAGMA libraries and databases. Suppose that F is a string representing a file whose contents are legal MAGMA input. During a MAGMA session, when the time has come for MAGMA to take its input from the file, the user should type:

load F;

At this point, all the contents of the file represented by F will be read into MAGMA, as if entered directly at the keyboard.

The string F may be either the full pathname of the file or a relative pathname (e.g., the filename only). If it is a full pathname, MAGMA attempts to find the corresponding file. If it is a relative pathname, then MAGMA first tries to find a file with this pathname relative to the current directory. If no such file exists, it attempts to interpret the pathname relative to one of the directories in the `MAGMA_PATH` environment variable. Finally, it attempts to find a library or database file with the given name. (If MAGMA has been installed correctly, it is sufficient to specify a library or database file by its filename alone, without a directory.) An error message is given if all of these attempts fail.

`MAGMA_PATH` is an *environment variable* defined outside MAGMA as a colon-separated list of directories. (Issues associated with environment variables are operating system tasks, not MAGMA tasks; seek local assistance for this if necessary.) The function **GetPath**() returns the value of `MAGMA_PATH` as a string, and the procedure **SetPath**(P) changes it to the path given by the string P, for the duration of the current MAGMA session.

The **load**-statement must appear at the top level, not within other constructions such as a **for**-statement or a function definition. This is because **load** is a parser instruction, not part of the MAGMA language proper. It informs the MAGMA parser that the next collection of statements is to be read from the file rather than the keyboard.

14.3 Files and Identifiers at Start-Up

14.3.1 Loading Files at Start-Up

A special form of file loading exists for the commencement of MAGMA sessions. Suppose that $F_1, F_2, \ldots, F_k$ are strings *(but without " marks)*, each representing a MAGMA input file in one of the ways described in Section 14.2. In order to load these files at the beginning of a MAGMA session, the user should enter MAGMA by typing:

`magma` F_1 F_2 $\ldots$ F_k

where the F_i are separated by spaces. The F_i do not require " marks, since this is a command issued outside MAGMA. As soon as MAGMA starts, it will read the files in order as MAGMA input and execute them, just as if the

contents of the files had been typed directly into MAGMA from the keyboard. When all of the files have been executed, the interactive use of MAGMA may proceed as usual.

14.3.2 Assigning Identifiers at Start-Up

Sometimes the user may wish to assign identifiers at start-up. This technique is particularly beneficial when several files are being loaded and some of them require extra values to be set.

The only kind of value that can be assigned to an identifier at start-up is a literal string. (This value may be converted into another category later, using functions such as **StringToInteger**.) The assignment syntax is

identifier:=string

and such assignments should be placed after the `magma` command, interspersed with the filenames. Note that there can be no spaces around the `:=` symbol, and the assignment must not finish with a semicolon. The string does not have to be surrounded by `"` marks, but special operating system characters (e.g., a backslash) will be required if the string includes spaces.

For example, suppose that `fred`, `pat` and `jane` are files of MAGMA input, and that the file `jane` refers to an string identifier d which should have the value `"qwert"`. In order to load these files and this identifier at the beginning of a MAGMA session, the user should type:

```
magma fred pat d:=qwert jane
```

After this command, MAGMA will be started. It will load and execute the files `fred` and `pat`, assign `"qwert"` to d, load and execute the file `jane`, and finally print a prompt symbol.

14.3.3 Using a Standard Start-Up File

A *standard start-up file* is a file that is loaded automatically at start-up. If the user wishes to have a file as a permanent start-up file, its full pathname should be assigned to the environment variable `MAGMA_STARTUP_FILE`. Then the command `magma` by itself will automatically load this file as a start-up file, and the command `magma` followed by a list of files and/or identifiers will load this file first and then load the other files and assign the identifiers.

Typically, a start-up file would contain abbreviations for frequently-used intrinsics, assignments of magmas to standard names, and settings of the

MAGMA environment, so as to customize the system. For example, assume that the following lines are the contents of a file called `.magstart` whose full pathname has been assigned to `MAGMA_STARTUP_FILE`:

```
Poly := PolynomialAlgebra;
Z := IntegerRing();
Q := RationalField();
SetPrompt("ready! ");
```

Then the command `magma` will load `.magstart` at start-up, so that a MAGMA session could begin as follows:

```
% magma
Magma V2.00-1    Wed May 29 1996 13:25:10    [Seed = 1]
Type ? for help.  Type <Ctrl>-D to quit.

Loading startup file ".magstart"

ready! print Z;
Integer Ring
ready! P<x> := Poly(Q); print P;
Univariate Polynomial Algebra in x over Rational Field
ready!
```

Occasionally the user may wish to commence a MAGMA session without loading the usual start-up file. This may be done with the command-line option `-n`. The command `magma -n` (followed optionally by a list of files and/or identifiers) will cause the file in `MAGMA_STARTUP_FILE` to be ignored for that session.

14.4 Echoing the Input from Files

The procedure **SetEchoInput**(v), where v is a boolean, is used to request that the contents of files loaded to MAGMA (at start-up or using **load**) be echoed on the screen (or to the output file or log file, as explained below). By default, no echoing takes place. However, if v is set to **true**, then any file loaded into MAGMA later in that session will have its contents echoed. The function **GetEchoInput**() returns the current value of the setting.

14.5 The Output File and Log File

MAGMA's output may be sent to an *output file* instead of to the computer terminal, so that the output can be examined easily after the MAGMA session. The procedure required is **SetOutputFile**(F), where F is a string specifying the output file, either with a full pathname or with a pathname relative to the current directory. The procedure **UnsetOutputFile**() causes the output to go to the terminal once more, rather than to the file.

A *log file* is a record of both the input and the output of a session. From the time when the user types **SetLogFile**(F), MAGMA will send a complete record of the session to the file specified by the string F, as well as displaying it in the normal fashion on the terminal. The procedure **UnsetLogFile**() causes MAGMA to stop making the log file.

It is not possible to have more than one output or log file in operation simultaneously. If a second file is set to start receiving data, the first file will no longer have data sent to it.

If the filename specified for the output file or log file already exists, then MAGMA will append the new session information to the end of the file, so that the former contents of the file are not destroyed accidentally. This is particularly useful when the same log file or output file is set or unset several times within the same session. If the user wishes to delete the old contents of the file and write to it afresh, then the parameter **Overwrite**, which is available for both **SetOutputFile** and **SetLogFile**, should be set to **true**.

The function **HasOutputFile**() returns **true** if MAGMA currently has an output file or logfile defined. If there is a file defined, the function also returns the name of the file as a string.

14.6 Reading and Writing to Data Files

It is sometimes the case that a user needs to have MAGMA read and process a file of data that has been generated by some other program. Alternatively, it may be necessary to write information generated by MAGMA to a file so that it can be processed further by some other application. There are two ways of approaching these tasks in MAGMA: either by performing operations on the whole file at once; or by reading or writing to the file in smaller portions.

14.6.1 Read/Write Operations on a Whole File

Let F be a string specifying a file, either as a full pathname or as a pathname relative to the current directory. The function **Read**(F) returns the contents

of F as a string. Standard string operations may then be used to analyze this string.

If x is any expression and L is a string which represents one of the possible levels in a **print**-statement, then the procedure **Write**(F, x, L) or **PrintFile**(F, x, L) constructs the string whose value is the output that would be given by printing x with this level, and writes this string to the file F. (Compare the function **Sprint**(x, L), explained in Section 13.2.) If the file already exists, the string will be appended to it; if the user wishes to overwrite the file instead, then the parameter **Overwrite** should be set to **true**.

There are two special versions of **Write**: if the `"Default"` level of printing is desired, then it is sufficient to type **Write**(F, x) or **PrintFile**(F, x); and if the `"Magma"` level is desired, then **PrintFileMagma**(F, x) may be used. For these procedures, the parameter **Overwrite** operates in the same way as explained above.

14.6.2 Read/Write Operations on Parts of a File

Table 14.1. Reading and writing to data files

MAGMA	Meaning
Open(F, t)	Open file given by string F with type indicator t, and return an associated MAGMA file object
POpen(s, t)	Open a pipe, with type indicator t, between MAGMA and the shell command given in string s (ending with newline), and return an associated MAGMA file object
Put(f, s)	Put (write) string s to file f
Puts(f, s)	Put (write) string s and newline character to file f
Gets(f)	Next line (not including newline character) from f, as a string (may be EOF)
Getc(f)	Next character from f, as a string (may be EOF)
Ungetc(f, c)	Push character c back into input buffer of f
IsEof(s)	**true** if string s is the EOF marker
Rewind(f)	Move file pointer of f to beginning
Tell(f)	Position of current offset (in bytes) within file f
Seek(f, o, p)	Move file pointer of f to o bytes more than position p, where offset o is an integer, and p is 0 for beginning of file, 1 for current position, or 2 for end of file

In MAGMA, the reading and writing of data files in sections is accomplished with a special *file object* category, **File**. The functions for handling these files are described in Table 14.1. Since the functions are derived from

standard C library functions, only a few comments need be made about them here.

In order to read or write to parts of a file described by a string F (full pathname or pathname relative to the current directory), the user must first create a file object f, say, by calling the function **Open**(F, t) and assigning the result to f. The file may then be read and/or written, depending on the type indicator t. When reading a file, it is important to check whether the end of the file has been reached; this is signified by means of the special EOF marker. Therefore, each time after applying **Gets** or **Getc**, the user should test the result with the function **IsEof**, and proceed to use the string or character only if it is not the EOF marker. When the file is no longer required, the identifier f should be deleted (using the **delete**-statement) or reassigned; MAGMA will then close the file, unless there are multiple references to it.

The type indicator t in the functions **Open** and **POpen** should be a string. The possible values for this string are the same as those allowed for the corresponding C functions `fopen()` and `popen()` in the current operating system, and have the same interpretation. The most important values are: `"r"` (open for reading), `"w"` (truncate or create for writing), `"a"` (append or create for writing), `"r+"` (open for update – reading and writing), `"w+"` (truncate or create for update), and `"a+"` (append – open or create for update at EOF). On a PC, the character `"b"` should also be included in t if the file is to be opened in binary mode. (The possible type indicators for **POpen** are `"r"` and `"w"`; this function is discussed in Section 14.8.2.)

As an example of reading a file, suppose that the file `my_spelling_list` is in the current directory, and contains a list of words, one per line. The following code prints the words in the list which begin with `M` or `m`, and either include an `o` or end with `t`, `a`, or `e`:

```
> f := Open("my_spelling_list", "r");
> s := Gets(f);
> while not IsEof(s) do
>     if Regexp("^[Mm].*(o|[tae]$)", s) then
>         print s;
>     end if;
>     s := Gets(f);
> end while;
Macquarie
Magma
meataxe
mindset
Molien
monoids
monomial
```

```
Mopsuestia
morphisms
Mosman
multiset
```

As an example of writing to a file, the program below constructs the Conway polynomial for GF(p^n), for certain small values of p and n, and writes the values of p and n and the coefficients of the polynomials to the file `conway`, in the current directory:

```
> f2 := Open("conway", "w");
> for p in [2, 3, 5, 7, 11] do
>     Puts(f2, "p = " cat Sprint(p));
>     for n in [1..4] do
>         cp := ConwayPolynomial(p, n);
>         Put(f2, Sprint(n) cat " ");
>         Puts(f2, Sprint(Coefficients(cp)));
>     end for;
> end for;
> delete f2;
```

After these lines have been executed, the contents of `conway` will be:

```
p = 2
1 [ 1, 1 ]
2 [ 1, 1, 1 ]
3 [ 1, 1, 0, 1 ]
4 [ 1, 1, 0, 0, 1 ]
p = 3
1 [ 1, 1 ]
2 [ 2, 2, 1 ]
3 [ 1, 2, 0, 1 ]
4 [ 2, 0, 0, 2, 1 ]
p = 5
1 [ 3, 1 ]
2 [ 2, 4, 1 ]
3 [ 3, 3, 0, 1 ]
4 [ 2, 4, 4, 0, 1 ]
p = 7
1 [ 4, 1 ]
2 [ 3, 6, 1 ]
3 [ 4, 0, 6, 1 ]
4 [ 3, 4, 5, 0, 1 ]
p = 11
1 [ 9, 1 ]
```

```
2 [ 2, 7, 1 ]
3 [ 9, 2, 0, 1 ]
4 [ 2, 10, 8, 0, 1 ]
```

14.7 Saving the Workspace of a Session

There are occasions when a user has to quit a MAGMA session before finishing the task. In fact, it is common when studying a complicated algebraic structure to do so in a number of separate MAGMA sessions, so that some theoretical work may be done in order to decide on the next stages in the computation.

MAGMA provides a facility for the *workspace* (the identifiers, structural information, and environment) to be saved in a file before the user quits MAGMA, and then restored later at the beginning of another session. In this way, the user can resume the session as if it had never stopped, rather than having to repeat the computations. Suppose that F is a string containing either a full pathname or a pathname relative to the current directory. Then the appropriate commands to finish the session are:

save F;
quit;

and the commands for starting a new session with the former workspace are:

`magma`
restore F;

(It is unwise to **restore** a workspace in the midst of a session, since the current workspace will be destroyed.)

The user may wish to **delete** irrelevant values stored in identifiers before saving the workspace, in order to save memory.

14.8 External Processes

14.8.1 System Calls

During a MAGMA session, the user may call a process (e.g., an operating-system task or another program) by typing the process as a string argument s to the procedure **System**(s). For instance, to execute `who` to see who is logged on, the user should type

```
> System("who");
```

The output of this function will be sent to the standard output. **System** may also be used as a function, in which case it returns the system command's return value as an integer, for use in error-checking.

The shell-escape `%!`c causes the command c to be executed in the shell, outside MAGMA. For instance:

```
> %!date
Thu Jul 25 18:01:04 EST 1996
```

The effect of the shell-escape resembles that of **System**, except that it is not recorded in the history for the MAGMA session.

14.8.2 Pipes

There are two ways to establish a *pipe* for a shell command. Firstly, given the strings C and s, where C is a shell command and s is an input string to this command, the function **Pipe**(C, s) creates a pipe to C, sends s into the standard input of C, and returns the output of C as a string. For many commands, s should finish with a new line character if it consists of only one line. For example:

```
> wds := "moyt truth beauty bewdy charm\n";
> wrong_wds := Pipe("spell", wds);
> print wrong_wds;
bewdy
moyt

> MailMe := procedure(msg)
>     _ := Pipe("mail -s 'Magma message' $USER", msg);
> end procedure;
> //then some function might include these lines:
> MailMe("Finished stage 2B");
> // (e-mail now received)
> MailMe("Finished stage 2C");
> // (e-mail now received)
```

As a further example, the following invocation of **Pipe** involves a call to the GP/Pari calculator [BBC93]. This system returns its values on lines beginning with `%1 =` , `%2 =` , and so on. Suppose that the user wishes to call GP/Pari from MAGMA in order to compute the product of 123 and 456. Then the appropriate commands are:

```
> s := Pipe("gp", "123*456");
> _, _, q := Regexp("%1 = ([^\n]*)", s);
> ans := StringToInteger(q[1]);
> print ans;
56088
> print ans eq 123*456;
true
```

The second way to establish a pipe is to use the function **POpen**(C, t). It opens a pipe between MAGMA and the shell command given in the string C (ending with a newline character), and returns an associated MAGMA file object. The kind of pipe depends on the type indicator t: either `"r"` for 'read' or `"w"` for 'write'. The file object may be manipulated with the commands described in Section 14.6.2. For example, the following statements compute the **Maximal** output for the Witt designs on 12 points and 24 points, and send to `outfile` the lines of the output which include the string `2, 4, 6, 8`:

```
> f := POpen("grep '2, 4, 6, 8' > outfile", "w");
> W12 := WittDesign(12);
> Puts(f, Sprint(W12, "Maximal"));
> W24 := WittDesign(24);
> Puts(f, Sprint(W24, "Maximal"));
> delete f;
```

After the execution of the statements above, the contents of `outfile` will be:

```
{1, 2, 4, 6, 8, 12},
{2, 4, 6, 8, 9, 11, 17, 18},
{1, 2, 4, 6, 8, 14, 20, 23},
```

14.8.3 Temporary Names and MAGMA Processes

The function **Getpid**() returns the process ID of the current MAGMA process.

Given a string s, the function **Tempname**(s) returns a unique string beginning with s, by use of the C library function `mktemp()`. This function may be used to construct a unique temporary name for a file, so that if several MAGMA processes are constructing files their names will not interfere with one another.

15. The User Environment

15.1 Tab Completion

On most operating systems, *tab completion* may be used to minimize the amount of typing required during an interactive MAGMA session. (Compare Section 11.2.6, which describes tab completion in the browser.) When the user presses the tab key in the course of entering a statement, MAGMA looks at the word currently being typed and considers possible completions of it, among intrinsics (including operators), identifiers (including user-defined functions and procedures), and reserved words (listed on p. 299). Knowledge of this facility is best acquired experimentally; in brief, the tab key completes the word as far as possible, and if it is pressed again then all the possible completions are listed. The beep sound is used as a warning.

For example, suppose that the quadratic field $\mathbb{Q}(\sqrt{5})$ is being assigned to the identifier Q. The following lines demonstrate how to do this with the minimum number of keystrokes:

```
> Q := Qua  [user types tab]
    ... beep, and above line becomes:
> Q := Quadratic  [user types   Fi   then tab]
    ... above line becomes:
> Q := QuadraticField  [user types   (5);]
    ... above line becomes:
> Q := QuadraticField(5);
```

As another example, the next lines show how to obtain the signature for the function **AbsolutePrecision** efficiently:

```
> print Ab  [user types tab]
    ... beep, and above line is unchanged:
> print Ab  [user types tab again]
AbelianBasis                    AbsoluteLogarithmicHeight
AbelianGroup                    AbsoluteMinimalPolynomial
AbelianInvariants               AbsoluteNorm
```

```
AbelianQuotient                 AbsoluteOrder
AbelianQuotientInvariants       AbsolutePrecision
AbelianSubgroups                AbsoluteRepresentationMatrix
Abs                             AbsoluteTrace
AbsoluteDegree                  AbsoluteValue
AbsoluteField                   AbsoluteValues
> print Ab  [user types   so   then tab]
    ...  beep, and above line becomes:
> print Absolute  [user types   P   then tab]
    ... above line becomes:
> print AbsolutePrecision   [user types   ;   then ‘return’]
> print AbsolutePrecision;
Intrinsic ’AbsolutePrecision’

Signatures:

    (<RngPadElt> x) -> RngIntElt
    (<FldPadElt> x) -> RngIntElt

        The absolute precision of p-adic x
        with finite relative precision

    (<FldPowElt> x) -> RngIntElt

        The absolute precision of series x
        with finite relative precision
```

15.2 Editing Lines of Input

15.2.1 Editing the Current Input Line

Corrections may easily be made to the *current input line*, that is, the line of input at the prompt, before the return key is pressed. The editing facility involved in making these corrections is known as the *line editor*. The line editor keys include the delete and/or backspace keys on the keyboard. Depending on the computer terminal, arrow keys or other special keys may also be of assistance. In addition, there are several control characters which can help the user edit the current line. Table 15.1 lists them, together with a few control characters that perform other kinds of tasks.

MAGMA also provides access to Emacs or VI style key bindings for use in the line editor. See the *Handbook* for details of these. The default mode

Table 15.1. Control characters

MAGMA	Meaning
control-a	Move to beginning of line (cf. 'a' at beginning of alphabet)
control-b	Move back a character ('back')
control-c	Abort current line and start a new line; see also Section 15.3
control-d	Quit MAGMA, if on an empty line; otherwise delete character under cursor ('delete')
control-e	Move to end of line ('end')
control-f	Move forward a character ('forward')
control-h	Same as backspace key
control-i	Same as tab key
control-j	Same as return key
control-k	Delete all characters from cursor to end of line ('kill')
control-l	Redraw the line on a new line
control-m	Same as return key
control-n	If at beginning of line, go forward to the next history line ('next'); otherwise, go to next history line that commences in the same way as current line (ignoring spaces)
control-p	If at beginning of line, go back to the previous history line ('previous'); otherwise, go to previous history line that commences in the same way as current line (ignoring spaces)
control-u	Clear whole of current line
control-v	Insert the following character literally (used to insert special keyboard characters in the line)
control-w	Delete previous word ('word')
control-x	Clear whole of current line
control-\	Immediately quit MAGMA

is Emacs; to change to the VI mode, type **SetViMode**(**true**). The control characters in Table 15.1 work in both modes.

15.2.2 Editing a Previous Input Line

If the user wishes to enter a line that is similar to or the same as a line which has already been completed and executed, it may be convenient to recall and/or edit previous lines of input. This is made possible because MAGMA stores the last 50 lines of input as *history*. (The procedure **SetHistorySize**(n) allows this default number of history lines to be changed, and the function **GetHistorySize**() returns the current value.)

There are two ways of using the history so as to recall and edit old lines. In the first method, a previous line is converted to the current line, and then

edited using the line editor. In the second method, a line or group of lines is edited using an editing program of the user's choice.

For the first method, two special keys are available for gaining access to previous lines: control-p, which goes back a line in the history, and its opposite, control-n. (The computer terminal may also have arrow keys which are more convenient for this purpose.) These keys search for lines that start in the same way as the current line, and so the user may wish to type the first few characters of the desired line before starting to search the history. Once the desired previous input line is found, it can be edited using the line editor.

The other method of using the history involves certain commands commencing with the `%` character. It is suitable for more extensive editing tasks, especially those involving more than one previous line. Firstly, the command for displaying all the history lines is `%p` (followed by the return key):

```
> print 2 + 4;
6
> m := 1492478;
> real37 := 37.0;
> marching := "left right " ^ 4;
> print marching;
left right left right left right left right
> %p
/*  36 */   S5 := Sym(5);
/*  37 */   print Random(S5);
/*  38 */   print Order(S5);

    ⋮

/*  81 */   print 2 + 4;
/*  82 */   m := 1492478;
/*  83 */   real37 := 37.0;
/*  84 */   marching := "left right " ^ 4;
/*  85 */   print marching;
```

Previous input lines may be repeated by typing a `%` mark followed by their history number. A `%` mark by itself causes the most recent line to be repeated, and the command `%`n_1 n_2 causes lines n_1 to n_2 to be repeated. For example:

```
> %
>> print marching;
left right left right left right left right
```

```
> %37
>> print Random(S5);
(2, 4, 3)
> %36 38
>> S5 := Sym(5);
>> print Random(S5);
(2, 4, 5, 3)
>> print Order(S5);
120
```

The command to edit a line is `%e` followed by the line number. If the line to be edited is the most recent line, then `%e` is sufficient, and if lines n_1 to n_2 are to be edited, then the appropriate command is `%e`n_1 n_2. The editor used is not the same as the line editor for the current line, but the preferred editor that the user has stored in the `EDITOR` environment variable, or `/bin/ed` if this variable is not set. The following example shows how to edit line 84 so that the exponent is 5 instead of 4:

```
> %e84
```

[then the change is made and saved, and the editor is exited]

```
>> marching := "left right " ^ 5;
> %85
>> print marching;
left right left right left right left right left right
```

15.3 Interrupting a Computation or Forcing a Stop

If the user types control-c (i.e., holds down the control key and types the letter 'c') while MAGMA is executing a command, then MAGMA will perform an *interrupt* as soon as possible. This technique should be employed if the system is taking too long to perform a computation, in the user's estimation, so that the user can seek an alternative approach.

MAGMA will be forced to stop if control-c is typed twice quickly, within half a second. This should be done only as a last resort, since it will cause an exit from the whole MAGMA system.

15.4 Timing MAGMA Operations

MAGMA keeps a record of how much computation time the given statements take the CPU (central processing unit) to perform. For example, at the end of each MAGMA session, the total CPU time for the session is printed.

The command **time**, placed before any statement, causes MAGMA not only to execute the statement as usual but also to print the number of seconds taken by the CPU to do so. Note that this is only the CPU time; since the computer would be performing other tasks as well, the user would be waiting longer than this for execution to complete. For example:

```
> time n := 123456^987;
Time 0.270
```

On the machine used for the example, it has taken MAGMA about 0.270 seconds to calculate 123456^{987}.

In order to time several statements collectively, the first statement should be preceded by

```
> t := Cputime();
```

and the last statement should be followed by

```
> print Cputime(t);
```

With no arguments, the function **Cputime** returns the time in seconds since the beginning of the MAGMA session, and with an argument, it returns the time since the time given by the argument.

15.5 Scrolling Output

If the computer system does not allow the user to scroll backwards through output, then MAGMA's own scrolling system may be employed. The procedure **SetRows**(n) restricts the output to a maximum of n rows at a time. Whenever MAGMA's output from one occasion exceeds n rows, the user will be given a chance to read the first n rows before continuing on to read the next n rows. For example:

```
> SetRows(20);
> for x in [1..30] do
for> print x^2;
```

```
for> end for;
1
4
9
16
25
36
49
64
81
100
121
144
169
196
225
256
289
324
361
Press return for more:   [‘return’ key was pressed here]
400
441
484
529
576
625
676
729
784
841
900
```

The command **SetRows**(0) turns off this output filter.

15.6 Memory Limit

There is, of course, a hard limit imposed by the user's operating system on the amount of memory available to a MAGMA process (session). However, it is also possible to impose a soft limit on the memory which the memory manager will allocate to MAGMA. The user might do this for a large task, in order to avoid using a vast amount of swap space accidentally and thus disturbing the work of others on the same machine. By default, there is no soft limit, but it may be

set (in bytes) by means of the procedure **SetMemoryLimit**(n) or the environment variable `MAGMA_MEMORY_LIMIT`. The function **GetMemoryLimit**() returns the current value of the limit, and a value of zero denotes no soft limit.

Part IV

Appendices

16. Reserved Words

_	do	hom	notin	save
adj	elif	if	notsubset	sdiff
and	else	import	or	select
assert	end	in	print	subset
assigned	eq	intrinsic	printf	then
break	error	is	procedure	time
by	exists	join	quit	to
case	exit	le	random	true
cat	false	load	read	until
clear	for	local	readi	when
cmpeq	forall	lt	repeat	where
continue	forward	meet	require	while
default	freeze	mod	requirege	xor
delete	function	ne	requirerange	
diff	ge	not	restore	
div	gt	notadj	return	

17. Precedence of Operators

‘ ‘‘	*left*
(	*left*
[	*left*
assigned	*right*
~	*non-associative*
#	*non-associative*
&* &+ &and &cat &join &meet &or	*non-associative*
$ $$	*non-associative*
.	*left*
@ @@	*left*
! !!	*right*
^	*right*
unary -	*right*
cat	*left*
*** / div mod**	*left*
+ -	*left*
meet	*left*
sdiff	*left*
diff	*left*
join	*left*
adj in notadj notin notsubset subset	*non-associative*
cmpeq eq ge gt le lt ne	*left*
not	*right*
and	*left*
or xor	*left*
^^	*non-associative*
? else select	*right*
->	*left*
=	*left*
:= is where	*left*

18. Summary of the Grammar

program →
    ϵ
  | statement
  | directive
  | intrinsic

directive →
    clear ;
  | **load** directive_string ;
  | **save** directive_string ;
  | **restore** directive_string ;
  | **freeze** ;

directive_string →
    StringConstant
  | Identifier

statements →
    statements statement
  | statement

opt_statements →
    ϵ
  | statements

statement →
    normal_statement
  | **time** normal_statement

normal_statement →
    ;
  | lval_list := expr ;
  | expr MutationAssignment expr ;
  | lval_list opt_print_level ;
  | **if** expr **then** opt_statements else_part **end if** ;
  | **while** expr **do** statements **end while** ;
  | **repeat** statements **until** expr ;
  | **print** expr_list opt_print_level ;
  | **printf** expr_list ;
  | **error** opt_error_if expr_list ;

| **read** ident opt_comma_expr ;
| **readi** ident opt_comma_expr ;
| **return** opt_expr_list ;
| for_push ident := expr **to** expr by_part **do** statements **end for** ;
| for_push designator_list_lr **do** statements **end for** ;
| for_push **random** designator_list_lr **do** statements **end for** ;
| **delete** expr_list ;
| **break** opt_break_spec ;
| **continue** opt_break_spec ;
| **assert** expr ;
| func_statement
| proc_statement
| **forward** ident_list ;
| **import** package_ref_list ;
| **case** expr : when_list **end case**
| **quit** ;
| intrinsic_assert

opt_comma_expr →
    ϵ
  | , expr

opt_error_if →
    ϵ
  | **if** expr ,

for_push →
    for

else_part →
    else opt_statements
  | **elif** expr **then** opt_statements else_part
  | ϵ

by_part →
    ϵ
  | **by** expr

opt_break_spec →
    ident
  | ϵ

opt_print_level →
    ϵ
  | : ident

package_ref_list →
    StringConstant : ident_list

when_list →
    ϵ
  | when_list_non_empty

when_list_non_empty →

when_clause
| when_list_non_empty when_clause
when_clause →
when expr_list **:** statements
| **else** statements
| **else :** statements
func_statement →
function ident prog_decl_args variable_declarations
opt_statements **end function ;**
proc_statement →
procedure ident prog_decl_args variable_declarations
opt_statements **end procedure ;**
opt_expr_list →
ϵ
| expr_list
expr_list →
expr_list **,** expr
| expr
expr →
std_expr
| product_cycles_or_commutators
std_expr →
`$`
| `$$`
| PREVIOUS
| **assigned** expr
| expr actual_func_args
| expr `@` expr
| expr `!` expr
| `+` expr
| **not** expr
| `#` expr
| `&+` expr
| `&*` expr
| **&and** expr
| **&or** expr
| **&meet** expr
| **&join** expr
| **&cat** expr
| expr `+` expr
| expr `-` expr
| expr `*` expr
| expr **div** expr
| expr `^` expr

| expr `^^` expr
| expr **mod** expr
| expr **and** expr
| expr **or** expr
| expr **xor** expr
| expr **meet** expr
| expr **diff** expr
| expr **sdiff** expr
| expr **join** expr
| expr **eq** expr
| expr **cmpeq** expr
| expr **ne** expr
| expr **gt** expr
| expr **ge** expr
| expr **lt** expr
| expr **le** expr
| expr **in** expr
| expr **notin** expr
| expr **subset** expr
| expr **notsubset** expr
| expr **cat** expr
| expr `.` expr
| expr `@@` expr
| expr `!!` expr
| expr **adj** expr
| expr **notadj** expr
| `-` expr
| expr `/` expr
| expr `->` expr
| expr `[` index_list `]`
| expr **select** expr **else** expr
| expr `=` expr
| `(` expr `)`
| expr ‘ ident
| expr ‘‘ expr
| expr **where** ident_underscore_list tok_is expr
| literal
| elt_constr
| rec_constr
| rec_format_constr
| str_constr
| pred_constr
| ident
| `~` expr

| **case** < expr | case_expr_list >
tok_is →
is
| :=
case_expr_list →
ϵ
| case_expr_list_non_empty
case_expr_list_non_empty →
case_expr_when
| case_expr_list_non_empty , case_expr_when
case_expr_when →
expr : expr
| **default** : expr
index_list →
expr_list
| range
range →
expr .. expr
| expr .. expr **by** expr
lval_list →
lval_list , lval
| lval
lval →
lval_pure
| expr
lval_pure →
lval_gen
| lval_underscore
lval_gen →
ident < ident_list >
lval_underscore →
_
literal →
simple_literal
| special_literal
simple_literal →
IntegerConstant
| StringConstant
| BooleanConstant
| RealConstant
| LiteralSequenceConstant
| LiteralCycleConstant
special_literal →
prog_literal

prog_literal →
    func_literal
  | proc_literal
func_literal →
    function func_body
  | **func** < opt_formal_in_args opt_var_args | expr_list >
proc_literal →
    procedure proc_body
  | **proc** < opt_formal_in_args opt_var_args | expr >
func_body →
    prog_decl_args variable_declarations opt_statements **end function**
proc_body →
    prog_decl_args variable_declarations opt_statements **end procedure**
elt_constr →
    elt < expr bar_expr_list constr_varargs >
  | **Character** < expr bar_expr_list constr_varargs >
  | < expr_list >
  | [* opt_expr_list *]
  | map_constr
rec_constr →
    rec < expr | rec_list >
rec_list →
    ϵ
  | rec_field
  | rec_list , rec_field
rec_field →
    ident := expr
rec_format_constr →
    recformat < rec_format_field_list >
rec_format_field_list →
    ϵ
  | rec_format_field
  | rec_format_field_list , rec_format_field
rec_format_field →
    ident
  | ident : expr
product_cycles_or_commutators →
    product_cycles_or_commutators cycle_or_commutator
  | cycle_or_commutator
cycle_or_commutator →
    (expr , expr_list)
map_constr →
    map < map_lhs | map_rhs constr_varargs >
  | **PartialMap** < map_lhs | map_rhs constr_varargs >

| **hom** < map_lhs | map_rhs constr_varargs >
map_lhs →
expr -> expr
map_rhs →
opt_expr_list
| map_rule
map_rule →
ident :-> expr
str_constr →
standard_constr
| special_constr
| seq_constr
| set_constr
| iset_constr
| mset_constr
| fset_constr
| ring_constr
standard_constr →
car < expr_list >
| **cop** < expr_list >
| constr_head < expr bar_expr_list constr_varargs >
| constr_head < expr : expr bar_expr_list constr_varargs >
| constr_head < expr , expr | >
constr_head →
ext
| **sub**
| **quo**
| **ideal**
| **lideal**
| **rideal**
| **ncl**
constr_varargs →
: actual_var_args
| ϵ
special_constr →
fp_constr
| **PermutationGroup** < expr bar_expr_list constr_varargs >
| **LinearCode** < expr , expr bar_expr_list >
| **MatrixAlgebra** < expr , expr bar_expr_list >
| **ExtensionField** < expr , ident bar_expr_list >
| **Graph** < expr bar_expr_list constr_varargs >
| **Digraph** < expr bar_expr_list constr_varargs >
| **IncidenceStructure** < expr bar_expr_list constr_varargs >
| **AffinePlane** < expr bar_expr_list constr_varargs >

| **ProjectivePlane** < expr bar_expr_list constr_varargs >
| **NearLinearSpace** < expr bar_expr_list constr_varargs >
| **LinearSpace** < expr bar_expr_list constr_varargs >
| **Design** < expr , expr bar_expr_list constr_varargs >
| **MatrixGroup** < expr , expr bar_expr_list constr_varargs >

ring_constr →
ring_constr_tok < expr bar_expr_list constr_varargs >

ring_constr_tok →
frac
| **loc**
| **comp**

fp_constr →
fp_constr_tok < ident_list bar_opt_expr_list_opt_varargs >

fp_constr_tok →
Group
| **Semigroup**
| **Monoid**
| **Algebra**
| **PolycyclicGroup**
| **AbelianGroup**

set_constr →
{ range }
| { expr | range }
| { opt_expr_list }
| { expr | opt_expr_list }
| { expr : designator_list_rl setq_pred }
| { expr | expr : designator_list_rl setq_pred }

seq_constr →
[range]
| [expr | range]
| seq_constr_list
| [expr | opt_expr_list]
| [expr : designator_list_rl setq_pred]
| [expr | expr : designator_list_rl setq_pred]

seq_constr_list →
[ ]
| [seq_list]

seq_list →
expr
| seq_list , expr

iset_constr →
{@ opt_expr_list @}
| {@ expr | opt_expr_list @}
| {@ expr : designator_list_rl setq_pred @}

```
    | {@ expr | expr : designator_list_rl setq_pred @}
mset_constr →
      {* opt_expr_list *}
    | {* expr | opt_expr_list *}
    | {* expr : designator_list_rl setq_pred *}
    | {* expr | expr : designator_list_rl setq_pred *}
fset_constr →
      {! ident in expr setq_pred !}
setq_pred →
      ϵ
    | | expr
designator_list_lr →
      designator_list_lr , designator
    | designator
designator_list_rl →
      designator_list_rl , designator
    | designator
designator →
      ident_list in expr
bar_expr_list →
      | opt_expr_list
bar_opt_expr_list_opt_varargs →
      | opt_expr_list constr_varargs
pred_constr →
      exists pred_result { expr : designator_list_rl setq_pred }
    | forall pred_result { expr : designator_list_rl setq_pred }
    | random { expr : designator_list_rl setq_pred }
    | rep { expr : designator_list_rl setq_pred }
pred_result →
      ϵ
    | ( expr_list )
prog_decl_args →
      ( formal_in_args formal_prog_in_args_end
    | ( formal_prog_in_args_end
formal_prog_in_args_end →
      )
    | : actual_var_args )
formal_in_args →
      formal_in_args , formal_in_arg
    | formal_in_arg
formal_in_arg →
      ident
    | ~ ident
opt_formal_in_args →
```

```
      ϵ
    | formal_in_args
actual_func_args →
      ( expr_list actual_func_in_args_end
    | ( actual_func_in_args_end
actual_func_in_args_end →
      )
    | : actual_var_args )
actual_var_arg →
      ident := expr
    | ident
opt_var_args →
      ϵ
    | : actual_var_args
actual_var_args →
      actual_var_args , actual_var_arg
    | actual_var_arg
ident →
      IDENTIFIER
ident_list →
      ident_list , ident
    | ident
ident_underscore_list →
      ident_underscore_list , ident_underscore
    | ident_underscore
ident_underscore →
      ident
    | _
variable_declarations →
      ϵ
    | variable_declarations local ident_list ;
intrinsic →
      intrinsic ident ( intrinsic_args actual_func_in_args_end intrinsic_opt
            intrinsic_return_types intrinsic_comment variable_declarations
            opt_statements end intrinsic
intrinsic_args →
      ϵ
    | intrinsic_args_list
intrinsic_args_list →
      intrinsic_arg
    | intrinsic_args_list , intrinsic_arg
intrinsic_arg →
      ident
    | ident :: intrinsic_type
```

| ~ ident :: intrinsic_type
| ~ ident
| ‘ ident
| ident :: [intrinsic_type]
| ident :: { intrinsic_type }
| ident :: {@ intrinsic_type @}
| ident :: {* intrinsic_type *}

intrinsic_type →
ident
| .
| < >
| [ ]
| { }
| {@ @}
| {* *}

intrinsic_opt →
ϵ
| [~]
| [<]
| [>]

intrinsic_return_types →
ϵ
| -> intrinsic_type_list

intrinsic_type_list →
intrinsic_type
| intrinsic_type_list , intrinsic_type

intrinsic_comment →
{ CommentStringConstant }

intrinsic_assert →
require expr : expr_list ;
| **requirerange** ident , expr , expr ;
| **requirege** ident , expr ;

Part V

Rings and Fields

19. Overview of Rings and Fields

This chapter consists of an overview of the main categories of commutative rings in MAGMA, and the operations that may be performed on them. Fields are included in the discussion, as special cases of rings.

The various categories of commutative rings cover a very wide spectrum, and some of the most important objects in computer algebra are found here. Subsequent chapters explain in detail how to use the most important particular types of commutative rings, such as the ring of integers $\mathbb{Z}$, the field of rational numbers $\mathbb{Q}$, residue class rings $\mathbb{Z}/m\mathbb{Z}$ (see Chapter 20), real and complex fields (see Chapter **??**), univariate and multivariate polynomial rings (see Chapter 21 and Chapter 22), power and Laurent series rings (see Chapter **??**), finite fields (see Chapter **??**), number fields (see Chapter 25), function fields (see Chapter **??**), p-adic and local rings and fields (Chapter **??**). Some important types of non-commutative rings (such as matrix rings) appear in the part on algebras. Their category names (like **AlgMat**) start with **Alg**, whereas the names of ring categories usually start with **Rng**, or with **Fld** if only fields are involved.

In this Chapter we describe some of the general principles for the construction and use of rings and fields, and some principles underlying the design and organization of this diverse area.

19.1 Creating Rings and Fields

Two different families of ring creation functions can be distinguished. The first family is that of standard functions to create rings (and fields) directly. The second family consists of constructors and functions that create new rings from existing ones.

19.1.1 Standard creation functions

Many rings and fields in MAGMA may be created directly using standard functions. Thus $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, finite fields GF_q and residue class rings $\mathbb{Z}/m\mathbb{Z}$

may be created using an intrinsic function with the appropriate arguments. Here are some examples:

```
> Q := RationalField();
> Q;
Rational Field

> Z12 := ResidueClassRing(12);
> Z12;
Residue class ring of integers modulo 12

> G := FiniteField(5, 2);
> G;
Finite field of size 5^2

> Category(Q), Category(Z12), Category(G);
FldRat RngIntRes FldFin
```

The meaning of the arguments in the above will be clear. In certain cases, such as the finite field example, different numbers of arguments are allowed: **FiniteField**(25) is synonymous to **FiniteField**(5, 2).

Table 19.1. Functions for standard rings

IntegerRing	**PolynomialRing**	**FiniteField**
RationalField	**PowerSeriesRing**	**FunctionField**
RealField	**LaurentSeriesRing**	**NumberField**
ComplexField	**pAdicRing**	**pAdicField**

Arguments are not always just integer parameters; here is the construction of the number field $\mathbb{Q}(\sqrt[3]{2})$.

```
R<x> := PolynomialRing(Integers());
K<k> := NumberField(x^3-2);
```

We will comment on the use of angle brackets further on.

As we saw already, in cases like the definition of the ring of integers and the rational field, sometimes no argument is needed at all; this is also true for the construction of the real (and complex) fields. The rings $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ have another feature in common: they are already present when MAGMA is started up. This makes it possible to define integers, rational numbers and real numbers without having to create their parents first.

```
> 91/21;
13/3
> 3.142857;
3.14285699999999999999999999985
```

19.1.2 Recursive ring constructions

The construction of new rings from old ones occurs in various guises. Firstly it may occur that the ring construction involves a 'coefficient ring' as parameter. We saw an example of that already in the polynomial ring definition above. Here are two more examples, of $\mathbb{Q}[x, y]$ and $\mathrm{GF}_5[[x]]$ respectively.

```
> R<x, y> := PolynomialRing(RationalField(), 2);
> P<t> := PowerSeries(FiniteField(5));
```

The user should beware of possible problems in using recursively defined rings and algebras over 'approximate' ground rings, such as the real numbers. Numerical instabilities can lead to unexpected problems, for example in dealing with matrices over real or complex numbers.

19.1.3 Ring constructors

A second form of constructing rings from existing ones is given by the constructors **sub**, **ext**, and **quo**, for subrings, extensions, and quotient rings, and the constructors **ideal**, **comp**, and **loc**, for ideals, completions and localizations. These constructors are not available for all rings, since in some cases there are no algorithms to compute the new structures effectively. Subrings, extensions, quotients, completions and localizations are rings in their own right, and are the parents of the elements that belong to them. By contrast, ideals retain some connection with the ring from which they come.

Although the syntax for these constructors is fairly uniform, they return objects of different kinds. As slight variations (depending on the category) do occur, and since not all of the constructors apply to every category, we refer to the next chapters for details. We will give a few examples here.

The **sub** constructor for a ring R has the form

sub< R | list of ring generators **>**

It returns two values: the subring S of R generated by the elements listed, and the embedding ring homomorphism from S to R. The ring S will be in the same category as R. For example, the following line constructs the subring of a finite field of 625 elements, generated by the element g^{26}:

```
> G<g> := FiniteField(625);
> F<f>, m := sub< G | g^26 >;
> m(f^2);
g^52
```

The syntax of the **ideal** constructor is similar to that of the **sub** constructor. For some categories of rings the ideals form a separate category in MAGMA.

Here is an ideal in a univariate polynomial ring:

```
> R<x> := PolynomialRing(RationalField());
> f := x^3 + 1/2*x^2 - 3*x - 3/2;
> I := ideal< R | f >;
> I;
Ideal of Univariate Polynomial Algebra in x
over Rational Field generated by x^3 + 1/2*x^2 - 3*x - 3/2
```

The **quo** constructor creates the quotient of a ring R by an ideal of R. It returns two values: the quotient, and the natural homomorphism from the ring to the quotient. Continuing the previous example,

```
> Q<y>, h := quo< R | I >;
> Q, h;
Univariate Quotient Polynomial Algebra in y
over Rational Field with modulus y^3 + 1/2*y^2 - 3*y - 3/2
Mapping from: RngUPol: R to RngUPolRes: Q
> h(x), h(f);
y
0
```

The ideal may be specified explicitly as an ideal, as above, or in terms of its generators:

```
> Q2<z> := quo< R | f >;
> Q2;
Univariate Quotient Polynomial Algebra in z
over Rational Field with modulus z^3 + 1/2*z^2 - 3*z - 3/2
```

The quotient is generally an object of another category than that of the ring; which category depends on R.

The **ext** constructor has a limited applicability in categories of fields and for transcendental extensions: it may be used to extend finite fields or number fields, and to create polynomial rings.

For transcendental extensions there are two versions:

ext< R | **>**

is the same as **PolynomialRing**(R), returning a univariate polynomial ring over R, and:

ext< R, n | **>**

is the same as **PolynomialRing**(R, n), returning a multivariate polynomial ring of rank n over R.

```
> P<q,r,s,t> := ext< Z, 7 | >;
> P;
Polynomial ring of rank 7 over Integer Ring
Lexicographical Order
Variables: q, r, s, t, $.5, $.6, $.7
```

The main constructor for algebraic extensions has the form

ext< F | f **>**

where f is a monic irreducible polynomial over F. The constructor returns an extension E of F by a root of f. For example:

```
> K<w> := GF(8);
> PK<X> := PolynomialRing(K);
> f := X^2 + w*X + 1;
> IsIrreducible(f);
true
> E<alpha> := ext< K | f >;
> E;
Finite field of size 2^6
> IsZero(Evaluate(f, alpha));
true
```

The completion constructor

comp< F | list of ideal generators **>**

is equivalent to **Completion**(F, I) Both return the completion of F at I, if I is the zero ideal or a prime ideal of a the ring of integers R of a field F, as well as a mapping from F to C.

As the examples indicate, there are also intrinsic MAGMA functions to construct certain kinds of completions (Laurent series rings and p-adic fields):

```
> Q := RationalField();
> F := FunctionField(Q);
> comp< F | 0 > eq LaurentSeriesRing(Q);
true

> Z := IntegerRing();
> comp< Q | 3 > eq pAdicField(3);
true
```

If R is a ring and P is a prime ideal of R, the *localization* of R at P may be created with the function **Localization**(R, P) or the constructor

loc< R | list of ideal generators **>**

The function and constructor each return the localization L and a mapping from R to L.

For example, the localization of $\mathbb{Z}$ at 5 is the valuation ring of $\mathbb{Q}_5$ (which may also be created using the function **ValuationRing**):

```
> locn, m := loc< Z | 5 >;
> locn, m;
Valuation ring of Rational Field
Mapping from: RngInt: Z to RngVal: locn
> ValuationRing(RationalField(), 5);
Valuation ring of Rational Field
```

19.1.4 Other associated ring constructions

There are more ways of creating new rings out of existing ones; usually these are category-specific. Some are merely access functions (like **CoefficientRing**, **PrimeField**, or **GroundField**), others construct entirely new objects. We mention two important field constructions: that of fields of fractions, and that of residue class fields.

The set of all elements of the form a/b, where a and b are elements of an integral domain D with $b \neq 0$, forms the *field of fractions* of D, when two such elements a_1/b_1 and a_2/b_2 are considered equal when $a_1 \cdot b_2 = a_2 \cdot b_1$. The MAGMA function which creates this field is **FieldOfFractions**(D). It is available for several categories of domain. Conversely, **IntegerRing**(K) returns the ring of integers of a field K.

The simplest example of this kind of field is the rational field, which is the field of fractions of the integer ring:

```
> FieldOfFractions(IntegerRing());
Rational Field
```

If the domain D is a polynomial ring over a ring S, then the field of fractions of D is a rational function field, consisting of fractions whose numerator and denominator are polynomials in D. This structure may also be created with the function **FunctionField**(S) or **FunctionField**(S, n). See Chapter **??** for details.

When I is a maximal ideal of the ring R, the quotient structure R/I forms a field, known as a *residue class field*. If this structure is created using MAGMA's **quo**-constructor, the magma returned will be in a (non-field) ring category, and not all field operations will be available for it, In order to create a residue class field in a *field* category, the user must apply a special construction, namely, the function **ResidueClassField**(R, I), or equivalently the constructor with a category specification

quo< Cat : R | list of ideal generators **>**

where the category up front must be an appropriate category of fields, such as **FldFin** or **FldNum**. The function and constructor each return the residue class field K and a mapping from R to K.

For example, let R be a univariate polynomial ring in x over the rationals, as before. In this ring, the polynomial $x^2 + 7$ is irreducible, so it generates a maximal ideal from which a residue class field may be constructed. The residue class field is constructed as a number field, whereas the corresponding quotient ring is constructed simply as a quotient of R:

```
> R<x> := PolynomialRing(RationalField());
> resfld, m := quo< FldNum : R | x^2 + 7 >;
> resfld, m;
Number Field with defining polynomial x^2 + 7
over the Rational Field
Mapping from: RngUPol: R to FldNum: resfld

> // compare quo-constructor
> quo< R | x^2 + 7 >;
Univariate Quotient Polynomial Algebra in $.1
over Rational Field
with modulus $.1^2 + 7
```

Similarly, the structure $\mathbb{Z}/5\mathbb{Z}$ may be created in MAGMA as the finite field GF(5) rather than as a residue class ring of the integers:

```
> Z := IntegerRing();
```

```
> I5 := ideal< Z | 5 >;
> I5;
Ideal of Integer Ring generated by 5
> ResidueClassField(Z, I5);
Finite field of size 5
Mapping from: RngInt: Z to GF(5)
> quo< Z | I5 >;
Residue class ring of integers modulo 5
```

Finally, if the initial ring is a polynomial ring over a finite field then an extension of this field may be produced:

```
> Rgf7<y> := PolynomialRing(GF(7));
> quo< FldFin : Rgf7 | y^5 + 3*y + 1 >;
Finite field of size 7^5
Mapping from: RngUPol: Rgf7 to GF(7^5)
```

19.2 Operations on Ring Elements

Methods for creating elements of a ring R depend on the category of R, but it generally involves one of the following constructions: the coercion into R of an element or a sequence of elements of a ring related to R; an expression in the generators of the ring, using the arithmetic operations shown in Table 19.3 (p. 327); or the construction of some special element of R.

```
> Z12 := Integers(12);
> Z12 ! 57;
9

> R<x> := PolynomialRing(RationalField());
> p := R ! [5/4, 0, 13/7, -1, 2]; p;
2*x^4 - x^3 + 13/7*x^2 + 5/4

> R<s, t> := PolynomialRing(FiniteField(5), 2);
> s*t^5 + 4*t + 7;
s*t^5 + 4*t + 7
```

We will briefly discuss the use of the angle brackets on the left hand side of an assignment here; see also Chapter 21. Angle brackets are used to give names to 'indeterminates' (as in the above example), or elements that are 'generators' in a more general sense, for example in number fields or finite fields. ('Generator' here means generator of the new ring that is being

Table 19.2. Ring element creation

MAGMA	Meaning
R**.i**	i-th generator of ring R
Zero(R)	Additive identity of ring R
One(R)	Multiplicative identity of unitary ring R
Representative(R), **Rep**(R)	A representative element of ring R
Random(R)	A random element of finite ring R
Parent(x)	Parent ring of ring element x
Category(x), **Type**(x)	Category to which ring element x belongs

constructed as an algebra over the ring it is defined over.) It is very important to know that the effect of:

```
> R<s, t> := PolynomialRing(FiniteField(5), 2);
```

instead of

```
> R := PolynomialRing(FiniteField(5), 2);
```

is twofold. In the first place the 'names' placed between angle brackets (separated by comma's) are used as strings in MAGMA output: indeed, elements of R will be printed as polynomials in which 's' and 't' are the indeterminates. Simultaneously, however, (two) identifiers (s and t) will be assigned values, being the first indeterminates or generators in the ring that is being constructed. This makes it possible from here on to define elements (of R in the above example) using the identifiers (s and t).

That the two operations are indeed distinct can be seen from the following example, where s and t as identifiers are reassigned, but the output of polynomials is not affected:

```
> R<s, t> := PolynomialRing(FiniteField(5), 2);
> f := s^3+t^2;
> f;
s^3 + t^2
> s := -2; t := 1;
> s^3 + t^2;
-7
> f;
s^3 + t^2
```

If identifiers have been reassigned, or if the angle brackets were not used in the first place, it is still possible to do the assignment afterwards. For this the

'dot' notation is used. This makes it possible to get hold of the i-th generator of a ring, via by $R.i$. Continuing the above example

```
> u := R.1; v := R.2;
> u^2*v;
s^2*t
```

Indeed, the output has not changed! It is also possible to change the string used in printing, afterwards, with the **AssignNames** procedure.

```
> AssignNames(~R, ["you", "vee"]);
> u + v;
you + vee
```

The second argument of **AssignNames** is a sequence of strings. Like the number of names in angle brackets the length of that sequence is allowed to be smaller but not larger than the number of generators. If the not all generators are given new names, MAGMA resorts to the default option, which is $.i for the i-th generator:

```
> AssignNames(~R, ["X"]);
> u + v;
X + $.2
```

Given a ring or algebra R, **Zero**(R) is the additive identity and **One**(R) is the multiplicative identity. These elements can also be obtained by coercion of the integers 0 and 1. For example:

```
> M := MatrixRing(Integers(12), 3);
> One(M), M ! 0;
[1 0 0]
[0 1 0]
[0 0 1]

[0 0 0]
[0 0 0]
[0 0 0]
```

Once some ring elements have been created, others can be constructed from these using the arithmetic operations in the ring. For convenience we list the standard operators in Table 19.3. Of course not every element has an inverse, and therefore there are also restrictions on the second argument for `/`. Note that we already saw how `/` is sometimes used to create elements of a field of fractions (for example in the case of integers); in such circumstances it is

useful also to have exact division, where the result is in the ring itself. The operator **div** is provided for that:

```
> q := 6/3;
> q, Parent(q);
2 Rational Field
> z := 6 div 3;
> z, Parent(z);
2 Integer Ring
```

Table 19.3. Ring element arithmetic

MAGMA	Meaning
$x + y$	Sum of elements x and y
$-x$	Additive inverse of x
$x - y$	Difference of elements x and y
n*x	x added to itself $\|n\|$ times, multiplied by the sign of n
$x * y$	Product of elements x and y
x^−1	Multiplicative inverse of unit x
x^n	x^n, where n is an integer (n may be negative only if x is a unit)
x/y	xy^{-1}, where y is a unit

There are many Boolean operators and functions for ring elements. Firstly, there are the standard equality-testing operators, **eq** and **ne**. Next, there are the order operators such as **gt**. They are only available for rings which have a total ordering defined on their elements (see Section 19.4.2); these rings are the integer ring, the rational field and the real field. The other Boolean operations on ring elements are the functions beginning with **Is**, as listed in Table 19.4. For instance:

```
> a := Integers(12) ! 4;
> IsZeroDivisor(a);
true
> IsZero(3*a); // check
true
```

Table 19.4. Tests on elements of rings

MAGMA	Meaning
IsZero(x)	**true** if x is the additive identity of its parent ring
IsOne(x)	**true** if x is the multiplicative identity of its parent ring
IsMinusOne(x)	**true** if x is the -1 of its parent ring
IsUnit(x)	**true** if $xy = 1$ for some y in the parent ring
IsNilpotent(x)	**true** if $x^q = 0$ for some $q \in \mathbb{Z}^+$; if **true**, also returns the smallest such q (the index of nilpotence)
IsZeroDivisor(x)	**true** if $x \neq 0$ and $xy = 0$ for some $y \neq 0$
IsRegular(x)	**true** if x is not a zero divisor
IsIdempotent(x)	**true** if $x^2 = x$
IsIrreducible(x)	**true** if x (in an integral domain) is not a unit, and if whenever $x = ab$, then a or b is a unit
IsPrime(x)	**true** if x (in an integral domain) is not 0 nor a unit, and if whenever $x\|ab$, then $x\|a$ or $x\|b$

19.3 Testing Properties of a Ring

There are many Boolean functions which test properties of a ring (or field) R. Table 19.5 (p. 329) lists them. *The return value depends on the mathematical properties of R, not the category to which R belongs.* For instance:

```
> Z9 := ResidueClassRing(9);
> IsField(Z9);
false
> Z7 := ResidueClassRing(7);
> IsField(Z7);
true
> IsPrincipalIdealDomain(Z7);
true
```

This illustrates that if R is known to satisfy the given property then the result is **true**, even if a stronger property also holds. If there is no appropriate algorithm (implemented) for determining the answer, MAGMA will give an error message.

Even if a ring R returns **true** *to one of these Boolean functions, operations employing this property may not be available if the property does not hold for all rings in the category of R.* This follows from the MAGMA-wide principle that the operations that apply to an object are given by the category of definition, as explained on p. 47. For instance, *Z7* is not a member of a field category, so it is not possible to construct a vector space over it, even though MAGMA can tell that *27* satisfies the axioms of a field. However, it is possible

Table 19.5. Some tests on properties of a ring

MAGMA	Meaning
IsField(R)	**true** if R is a commutative division ring
IsOrdered(R)	**true** if 'greater than' is defined in R
IsEuclideanDomain(R)	**true** if R is a domain and has a known Euclidean norm function
IsEuclideanRing(R)	**true** if R has a known Euclidean norm function
IsPrincipalIdealDomain(R), **IsPID**(R)	**true** if R is a domain and every ideal of R has the form aR for some $a \in R$
IsPrincipalIdealRing(R), **IsPIR**(R)	**true** if every ideal of R has the form aR for some $a \in R$
IsUniqueFactorizationDomain(R), **IsUFD**(R)	**true** if R is a domain and factorization is unique
IsIntegralDomain(R), **IsDomain**(R)	**true** if R is commutative, has a $1 \neq 0$, and has no zero divisors
IsCommutative(R)	**true** if R is commutative
IsDivisionRing(R)	**true** if R has a $1 \neq 0$, and every $a \in R$ except 0 has an inverse
IsUnitary(R)	**true** if R has a 1
IsFinite(R)	**true** if R has a finite number of elements; if **true**, also returns the cardinality $\#R$ of R

to construct a vector space over **FiniteField**(7), which is isomorphic to $Z7$. The user must explicitly create the structure as a field if MAGMA is expected to regard it as a field.

An important ring property is whether it is *unitary*, that is, whether it has a multiplicative identity. This is tested by **IsUnitary**(R). Many MAGMA operations involving rings are only permitted if the ring is unitary. There is a certain subtlety involved here: a subring is regarded as unitary by MAGMA only if it contains the multiplicative identity of its generic ring. For example, in the ideal I of $\mathbb{Z}/12\mathbb{Z}$ generated by 5, the element 25 is a multiplicative identity, but since I does not contain the one of $\mathbb{Z}/12\mathbb{Z}$, it is not regarded as unitary:

```
> I := ideal< ResidueClassRing(30) | 5 >;
> I;
Ideal of residue class ring of integers modulo 30
generated by 5
> forall{ a : a in I | 25*a eq a };
true
> IsUnitary(I);
```

```
false
```

19.4 Operations on Various Kinds of Rings

Chapter 4 explains the operations on magmas in general, especially in relation to generators and the set of elements. Table 19.6 lists some of the additional operations on rings. These operations are in theory available for all kinds of rings. In practice, however, there are categories of rings, or certain rings within a category, for which certain operations are well-defined but cannot be easily computed. MAGMA will give an error message if such operations are attempted on these rings.

Table 19.6. Operations on rings

MAGMA	Meaning
Characteristic(R)	Smallest positive integer m such that $mr = 0$ for every $r \in R$, or zero if such an m does not exist
Centre(R)	Subring of R consisting of the elements that commute with all elements of R
UnitGroup(R)	Group of units of R
PrimeField(F)	For a field F, returns GF(p) if the characteristic n of F is positive, else the rational field
PrimeRing(R)	For a unitary ring R, returns $\mathbb{Z}/n\mathbb{Z}$ if the characteristic n of R is positive, else the integer ring
$R + S$	Given subrings R and S of the same ring, return ring $\{r+s : r \in R, s \in S\}$; generators will be the sums of the generators of R and S
$R * S$	Given subrings R and S of the same ring, return ring $\{rs : r \in R, s \in S\}$; generators will be the products of the generators of R and S
R **meet** S	Given subrings R and S of the same ring, return ring $R \cap S$

The remainder of this section explains operations that are only applicable to rings with certain properties.

19.4.1 Finite Rings

The function **IsFinite**(R) returns **true** if MAGMA knows that the ring R is finite. If the return value is **true**, then the cardinality or size of R is

also returned. The cardinality of R may also be obtained directly from the expression $\#R$.

If MAGMA knows that R is finite then it can usually calculate the elements of R. If so, then the set of these elements is $\mathbf{Set}(R)$. It will also be possible to iterate over R, that is, to type x **in** R within a set/sequence constructor or at the top of a **for**-loop.

19.4.2 Ordered Rings

Table 19.7. Operations for elements of totally ordered rings

MAGMA	Meaning
x **lt** y	**true** if $x < y$
x **le** y	**true** if $x \leq y$
x **gt** y	**true** if $x > y$
x **ge** y	**true** if $x \geq y$
$\mathbf{Maximum}(x, y), \mathbf{Max}(x, y)$	The larger of the elements x and y
$\mathbf{Maximum}(Q), \mathbf{Max}(Q)$	Returns (i) the largest of the elements in the sequence Q (ii) the position of this element
$\mathbf{Minimum}(x, y), \mathbf{Min}(x, y)$	The smaller of the elements x and y
$\mathbf{Minimum}(Q), \mathbf{Min}(Q)$	Returns (i) the largest of the elements in the sequence Q (ii) the position of this element

Table 19.7 lists the operations for elements of a totally ordered ring R, that is, a ring for which $\mathbf{IsOrdered}(R)$ returns **true**. If a ring does not have a total ordering defined on its elements then it makes no sense to compare the elements for size. The integer ring, rational field and real field are all totally ordered.

19.4.3 Euclidean Rings

A Euclidean ring R is a ring which has a known Euclidean norm function, that is, a function ϕ from the set S of regular elements of R to the non-negative integers such that (a) $x|y$ implies $\phi(x) \leq \phi(y)$ and (b) if $x \in R$ and $y \in S$ then there exist $q, r \in R$ such that $x = yq + r$ and either $r = 0$ or $\phi(r) < \phi(y)$. Important examples of Euclidean rings are the integers, in which ϕ is the absolute value, and the univariate polynomials over a field, in which ϕ is the degree of the polynomial.

The function **IsEuclideanRing**(R) returns **true** if R is known to have a Euclidean norm. Most commonly-investigated Euclidean rings are also integral domains; this is tested by **IsEuclideanDomain**(R). Given an element x of a Euclidean ring R, **EuclideanNorm**(x) returns $\phi(x)$. It is not practical for MAGMA to attempt to find a Euclidean norm for any given ring, so its knowledge of whether a norm exists is rather restricted, and it always uses the standard mathematical choice for ϕ as the norm.

The purpose of a Euclidean norm is to allow division-with-remainder to take place in the ring. In the situation described above, with the equation $x = yq+r$, the expression x **div** y returns the quotient q, and x **mod** y returns the remainder r. The function **Quotrem**(x, y) returns both q and r, in that order.

19.4.4 Unique Factorization Domains

Table 19.8. Operations on elements of unique factorization domains

MAGMA	Meaning
GCD(x, y)	Greatest common divisor of x and y
GCD(Q)	Greatest common divisor of terms in sequence Q
LCM(x, y)	Least common multiple of x and y
LCM(Q)	Least common multiple of terms in sequence Q
XGCD(x, y)	Extended gcd: returns ring elements g, p, q such that g is gcd of x and y and $g = px + qy$
XGCD(Q)	Given $Q = [x_1, \ldots x_r]$, returns gcd g of Q and sequence $M = [m_1, \ldots, m_r]$ such that $g = \sum_{i=1}^{r} m_i x_i$
IsDivisible(x, y)	**true** if x is divisible by y
IsAssociate(x, y)	**true** if $x = uy$ for some unit u
Factorization(x)	Returns (i) sequence $[< f_1, k_1 >, \ldots, < f_r, k_r >]$ where the f_i are irreducible and the k_i are non-negative integers, and (ii) unit u, such that $x = u f_1^{k_1} \cdots f_r^{k_r}$

A unique factorization domain (UFD) is a domain R such that every non-zero element factorizes uniquely (up to order and associates) into a product of irreducibles. **IsUFD**(R) returns **true** if R is known by MAGMA to be a UFD. Table 19.8 (p. 332) shows the operations for UFDs.

In the integer ring and in polynomial rings over the integer ring or a field, which are the major examples of UFDs, factorization of a ring element x is an important and non-trivial task. The function **Factorization**(x) returns the factorization of x in the form of a factorization sequence and a unit. For instance:

```
> a, b := Factorization(-3798386541);
> a, b;
[ <3, 4>, <181, 1>, <509, 2> ]
-1
```

The output indicates that $-3798386541 = -1 \cdot 3^4 \cdot 181^1 \cdot 509^2$.

Several algorithms have been developed to perform factorization, and MAGMA supports a number of these algorithms, for each of integer factorization and polynomial factorization. To specify the desired method, the user should employ the function **Factorization**(x) with one or more *parameters*; see Section 20.4.2 for an explanation of their use. For instance, if f is a polynomial then the function **Factorization**(f) has a parameter **Al**, which can take any of three string values. The following lines demonstrate how to factorize a polynomial using the `"BerlekampSmall"` algorithm:

```
> GF503polys<r> := PolynomialAlgebra(GF(503));
> a, b:=Factorization(3*r^250+500: Al := "BerlekampSmall");
> a;
[
    <r + 1, 1>,
    <r + 502, 1>,
    <r^4 + r^3 + r^2 + r + 1, 1>,
    <r^4 + 502*r^3 + r^2 + 502*r + 1, 1>,
    <r^20 + r^15 + r^10 + r^5 + 1, 1>,
    <r^20 + 502*r^15 + r^10 + 502*r^5 + 1, 1>,
    <r^100 + r^75 + r^50 + r^25 + 1, 1>,
    <r^100 + 502*r^75 + r^50 + 502*r^25 + 1, 1>
]
> b;
3
```

For more information, see Section 21.6. In the case of integer factorization, the function **Factorization**(n) uses a combination of algorithms, but the parameters may be set to specify the extent to which each algorithm is employed, and the details of its use. For instance, the following call to the function alters the default values of the number of curves used in the first elliptic curve attack, and the upper bound for the trial divisors (these changes are not necessarily for the better!):

```
> Factorization(112377669312366219120309:
>     PollardRhoLimit := 100, MPQSLimit := 0);
[ <3, 8>, <181, 2>, <509, 4>, <7789, 1> ]
```

The next chapter describes these parameters, (see p. 356).

20. Ring of Integers and Field of Rationals

Reflecting the fundamental role played by the ring of rational integers in mathematics, MAGMA contains highly optimized code for performing arithmetic with arbitrary precision integers. This is augmented with advanced tools for factoring integers (elliptic curve and quadratic sieve methods) and for proving primality (an elliptic curve primality prover).

This chapter describes the many operators and functions that are defined for the ring of integers $\mathbb{Z}$ and for the rational field $\mathbb{Q}$. Also included is a discussion of residue class rings $\mathbb{Z}/n\mathbb{Z}$ and their arithmetic. A number of integer functions relating to combinatorics are described elsewhere (see Chapter 35).

20.1 Creating Integers and Rationals

No action is required on the part of the user to create the ring of integers $\mathbb{Z}$ or the field of rational numbers $\mathbb{Q}$, since both are automatically defined whenever MAGMA is started. In particular, integers and rationals may be created without reference to their parent since they are recognized from their syntax:

```
> 42;
42
> 5/7;
5/7
```

Integers of any length may be defined in this manner. However, it is recommended that the *line continuation character* \ be placed at the end of each line when the number of digits extends over more than one line. MAGMA also uses the backslash to break long lines of output:

```
> m := 12345678901234567890123456789012345678\
> 90123456789012345678901234567890;
> m^2;
```

```
152415787532388367504953515625666819450083828733760097555\
225118122311263526910001219327312604785942508763915375705\
492365005334557625361987875019051998750190521000
```

(Use **SetColumns** to change the number of characters printed on a single line.)

A rational number may be created by typing the expression p`/`q (where p and q evaluate to integers and q does not equal zero). MAGMA stores rational numbers in *reduced form*, where the numerator and denominator are coprime and the denominator is positive:

```
> r := 68/-72;
> r;
-17/18
```

The standard constructors also apply to the construction of rationals; the **elt**-constructor **elt**`<` Q | p, q `>` and the coercion Q ! t, where t is a sequence $[p, q]$ both require expressions evaluating to two integers p, q, and create the rational p/q.

MAGMA draws a distinction between the rational number $\frac{p}{1}$ and the integer p, though automatic coercion makes this distinction transparent to the user in many cases. If the user creates a rational number r whose denominator (in reduced form) equals 1, the number will be stored as an element of $\mathbb{Q}$, rather than as an element of $\mathbb{Z}$, although it will look like an integer when printed. The function **IsIntegral**(r) returns **true** for such numbers.

The function **IntegerRing**(), or **Integers**(), returns the magma corresponding to $\mathbb{Z}$ for use in situations where reference needs to be made to it. Similarly, **RationalField**() and **Rationals**() return the field $\mathbb{Q}$. For example, when using the function **MatrixRing** to define a matrix ring over the integers, $\mathbb{Z}$ must be given as one of the arguments:

```
> Z := IntegerRing();
> Z;
Integer Ring
> M := MatrixRing(Z, 5); M;
Full Matrix Algebra of degree 5 over Integer Ring
```

Another common case occurs when applying a strictly integer function to a rational argument; for example, the integer 374 is clearly divisible by 2, but omitting the coercion in the example below will lead to a 'type' error.

```
> Factorization( Integers() ! (374/2));
[ <11, 1>, <17, 1> ]
```

As will be seen in the next section, this 'type' problem would not arise if **div** was used rather than / for division.

The category containing the ring of rational integers is **RngInt**, that containing $\mathbb{Q}$ is called **FldRat**.

20.2 Manipulating Integers and Rationals

20.2.1 Arithmetic

All of the standard arithmetic operations for integers and rationals are available in MAGMA. The operators denoting addition, subtraction, multiplication, and division are +, −, *, and /. When applied to two members of the **FldRatElt** category, the result will also be a MAGMA rational; with the exception of division, which always results in a rational, the binary operations on elements of **RngIntElt** result in integers again. When one operand is an integer and the other a rational number the result is always rational.

Addition, subtraction, multiplication, and division may be performed on arbitrarily large integers, the size being restricted only by the amount of computer memory available. For example, the following computation returns the exact value of the result, with no approximation or rounding:

```
> n := 12345678901234567890123456789O;
> n*(n-1/7);
152415787532388367504953515625185621033572843886034141138З0
```

The last result, although integral, is an element of $\mathbb{Q}$. The operator / constructs the rational number $\frac{a}{b}$, and consequently always returns an element of the rational field, even if the division is exact. If this occurs, it is of course possible to coerce to **RngIntElt** using !. Alternatively, one could use **div**, which returns the integral quotient when applied to two integers; together with the **mod** operator this returns the quotient and remainder pair of the division of two integers a and b. Both quotient q and remainder r (in that order) are also returned by the single function **Quotrem**(a, b). The quoitient and remainder are determined by the rules that $a = qb + r$, and r has the same sign as b while satisfying $0 \leq |r| < |b|$.

```
> q, r := Quotrem(123871, 4249);
> q, r;
29 650
> 123871 div 4249;
29
```

```
> 123871 mod 4249;
650
> 374 div 2;
187
> Type($1);
RngIntElt
```

As a longer example, consider the problem of writing a positive integer N in the form $N-1=2^s d$, where s and d are positive integers. The following function $EvenexpOdd(N)$ has s and d as its return values:

To perform this example online, type `load "I96c20e1";`

```
> EvenexpOdd := function(N)
>     s := 0; d := N - 1;
>     q, r := Quotrem(d, 2);
>     while r eq 0 do
>         s +:= 1;
>         d := q;
>         q, r := Quotrem(d, 2);
>     end while;
>     return s, d;
> end function;
```

For instance:

```
> N := 158561449985;
> EvenexpOdd(N);
11 77422583
> N - 1 eq 2^11 * 77422583;
true
```

The expression $m\hat{}k$ denotes the k-th power m^k of m. If the exponent k is a rational or real number the result will be a real number but in this chapter k will be assumed to be an integer. Provided that k is a non-negative integer and m is an integer, then the result will be an integer, but in all other cases (m rational or k negative) the result is a rational number. (Note that $0\hat{}0$ is defined to be 1, following convention.) The result may be a rational number even if it happens to be integral:

```
> 3^5, Parent(3^5);
243 Integer Ring
> 3^(-5), Parent(3^(-5));
1/243 Rational Field
```

```
> 1^(-5), Parent(1^(-5));
1 Rational Field
> (1/2)^-1, Parent((1/2)^-1);
2 Rational Field
```

The exponent k must be a small integer.

Table 20.1. Divisibility functions

MAGMA	Meaning
Quotrem(a, n)	Given a and n with $n \neq 0$, returns (i) quotient q, (ii) remainder r, such that $a = qn + r$ and $\|r\| < n$ ($0 < r < n$ if $n > 0$; $n < r < 0$ if $n < 0$)
a **div** n	Quotient q when a is divided by n, as in **Quotrem**
a **mod** n	Remainder r when a is divided by n, as in **Quotrem**
GCD(a, b)	Greatest common divisor of a and b
GCD(Q)	Greatest common divisor of the sequence of integers Q
XGCD(a, b)	Greatest common divisor d of a and b together with cofactors p, q such that $d = pa + qb$
XGCD(Q)	Greatest common divisor of the sequence of integers Q together with cofactors
LCM(a, b)	Lowest common multiple of a and b
LCM(Q)	Lowest common multiple of the sequence of integers Q
IsDivisible(a, b)	**true** if a is divisible by b; if **true**, also returns the quotient a/b
IsEven(a)	**true** if and only if integer a is even
IsOdd(a)	**true** if and only if integer a is odd

20.2.2 Elementary Functions

The function **GCD** computes the greatest common divisor of integers a and b, while **LCM** returns their least common multiple. The use of **GCD** is illustrated by looking for a factor of $n = 557200336747259$. If p is a prime, by Fermat's Theorem, $a^{p-1} = 1 \bmod p$, for any integer a coprime to p. This implies that p divides $a^m - 1$, for any multiple m of $p-1$. So if n has a prime divisor p such that $p - 1$ divides m, the factor p is easily found:

```
> n :=  67394403;
> [ Gcd(2^m-1, n) : m in [2..18] ];
[ 3, 1, 3, 31, 9, 1, 3, 73, 93, 1, 9, 1, 3, 31, 3, 1, 1971]
> Factorization(n);
[ <3, 3>, <31, 1>, <73, 1>, <1103, 1> ]
```

An extended version of the greatest common divisor function, **XGCD**, returns the greatest common divisor d of a and b, together with integers p and q such that $d = pa + qb$. To get all three values it is necessary to use a multiple assignment:

```
> d, p, q := XGCD(21, 27);
> d, p, q;
3 4 -3
```

Now d will equal $21p + 27q$.

Versions of each of the functions **GCD**, **LCM** and **XGCD** are provided which allow a sequence of integers as their argument. For example, the smallest positive integer divisible by all of $7, 12, 36, 8, 2$ and 10 is given by

```
> LCM([7, 12, 36, 8, 2, 10]);
2520
```

Table 20.2. Elementary functions

MAGMA	Meaning
Abs(a)	Absolute value of a
Sign(a)	Returns $-1, 0$, or 1 depending on whether a is negative, zero, or positive respectively
Max(a, b),**Min**(a, b)	The maximum/minimum of a and b
Ilog2(a)	Integral part (floor) of base-2 logarithm of a
Isqrt(a)	Integral part (floor) of square root of a
Iroot(a, n)	Integral part (floor) of n^{th} root of a
Intseq(a, b)	Sequence $Q = [a_0, a_1, \ldots, a_n]$ which is the base b representation of $a = a_0b^0 + a_1b^1 + \cdots + a_nb^n$
Seqint(Q, b)	Integer $a = a_0b^0 + a_1b^1 + \cdots + a_nb^n$ for which $Q = [a_0, a_1, \ldots, a_n]$ is the base b representation
Random(a, b)	A random integer n such that $a \leq n \leq b$
Random(a)	A random integer n such that $0 \leq n \leq a$
RandomBits(a)	A random integer n such that $0 \leq n < 2^a$

Table 20.2 lists some of MAGMA's other elementary functions on integers and rationals. Only the functions **Abs**, **Sign**, **Max**, and **Min**. apply to rational numbers as well as integers. As an illustration, the line below finds the maximum of the two numbers 1234 and 3701/3:

```
> Max(1234,  3701/3);
1234
```

Functions **Max** and **Min** may also be used to find maxima or minima of a sequence of numbers.

Several of these intrinsics are illustrated in the following example, a function for extracting square roots of squares:

To perform this example online, type `load "I96c20e2";`

```
> SafeSqrt := function(x)
>     if Category(x) ne RngIntElt then
>         error "Error in SafeSqrt: Argument of wrong type";
>     elif Sign(x) eq -1 then
>         error "Error in SafeSqrt: Argument negative";
>     elif IsSquare(x) then
>         return Isqrt(x);
>     else
>         error "Error in SafeSqrt: Argument not a square";
>     end if;
> end function;
```

Intseq and **Seqint** may be used to compute the representation of an integer with respect to different bases. For example, the number 25 in decimal notation may be converted to binary notation as follows:

```
> Intseq(101, 3);
[ 2, 0, 2, 0, 1 ]
```

The output sequence should be interpreted to mean that $101 = 2 \times 3^0 + 2 \times 3^2 + 1 \times 3^4$.

Besides the usual operators **eq** and **ne** for testing equality and inequality, the magnitude of integers, rational numbers and real numbers can be compared by use of **lt** (for $<$), **le** ($\leq$), **gt** ($>$), **ge** ($\geq$). These may be used with arguments of different types:

```
> 3 lt Log(10);
false
> (10/2) eq Ilog2(33);
true
```

Table 20.3 (p. 342) lists functions for rational numbers (the usual generic functions for fields and their elements are also available). Except for the Boolean-valued **IsIntegral** and the function **RationalReconstruction**(s), explained below, these functions return integers.

Table 20.3. Operations on rational numbers

MAGMA	Meaning
Numerator(r)	Numerator of r (in reduced form)
Denominator(r)	Denominator of r (in reduced form)
IsIntegral(r)	**true** if **Denominator**(r) is one
Abs(r)	Absolute value $\lvert r\rvert$ of r
Sign(r)	Returns $-1, 0$, or 1 depending on whether r is negative, zero, or positive respectively
Floor(r)	Greatest integer $\lfloor r \rfloor$ less than or equal to r
Ceiling(r)	Smallest integer $\lceil r \rceil$ greater than or equal to r
Round(r)	Integer nearest r ($n + \frac{1}{2}$ is rounded to $n + 1$, for $n \in \mathbb{Z}$)
Truncate(r)	Integer part of r, or r rounded towards zero, i.e., **Sign**(r)*$\lfloor \lvert r\rvert \rfloor$
RationalReconstruction(s)	Given s in $\mathbb{Z}/m\mathbb{Z}$ (or GF(m), where m is prime), return **true** if s has a rational reconstruction r; if **true**, also return r

When applied to an element s of any residue class ring $\mathbb{Z}/m\mathbb{Z}$ or any prime finite field, **RationalReconstruction**(s) returns a Boolean value, with a rational number if the Boolean is **true**. The purpose of this function is to provide a partial inverse of the function $\psi_m \colon \mathbb{Q} \to \mathbb{Z}/m\mathbb{Z}$ of taking residues modulo m, where the obvious value of ψ_m is only defined for rational numbers with denominator coprime to m. The value of $\psi^{-1}(s)$ is the rational number $r = \frac{p}{q}$ (in reduced form) for which $\psi_m(r) = s$ (i.e., $pq^{-1} \equiv s \bmod m$) and, in addition, $|p|, q \leq \sqrt{m/2}$. Such an r does not always exist, but if it exists it is unique. For example:

```
> F := GF(29);
> RationalReconstruction(F ! 22);
false
> RationalReconstruction(F ! 20);
true 2/3
> F ! (2/3);
20

> Z45 := ResidueClassRing(45);
> RationalReconstruction(Z45 ! 33);
true -3/4
> (Z45 ! -3) / 4;
33
```

20.2.3 Arithmetical Functions

Table 20.4 lists some standard functions on integers that are available in MAGMA. These functions may be applied to integers of arbitrary size. The functions listed that take one argument are *multiplicative* arithmetical functions of the argument n, which means that for coprime n_1, n_2 the value $f(n_1 \cdot n_2)$ equals $f(n_1) \cdot f(n_2)$. In particular, the value on any n can be computed from the prime factorization of n. As this is essentially the only way known to compute the values of these functions, witht the exception of the Jacobi and Legendre symbols, the time to compute their values for very large arguments is dominated by the time to factor the argument! Therefore versions are provided which take the factorization of n as argument as well (see Section 20.2.4).

Table 20.4. Arithmetical functions

MAGMA	Meaning
CarmichaelLambda(n)	Carmichael function $\lambda(n)$
DivisorSigma(i, n)	Divisor function $\sigma_i(n) = \sum_{k\|n} k^i$
EulerPhi(n)	Euler totient function $\phi(n)$
MoebiusMu(n)	Möbius function $\mu(n)$
LegendreSymbol(n, m)	Legendre symbol $\left(\frac{n}{m}\right)$
JacobiSymbol(n, m)	Jacobi symbol $\left(\frac{n}{m}\right)$

Euler's totient function, $\phi(n)$, gives the number of positive integers less than n that are coprime to n; in other words, it is the order of $(\mathbb{Z}/n\mathbb{Z})^*$, the unit group of the residue class ring $\mathbb{Z}/n\mathbb{Z}$. The MAGMA function **EulerPhi** computes $\phi(n)$ by evaluating the formula

$$\phi(n) = \prod_{p^k \| n} (p-1)p^{k-1}$$

where the product ranges over the prime powers p^k exactly dividing n. As the prime factorization of n is required, the function may be slow for very large n.

Related to Euler's totient function is Carmichael's λ-function which is the exponent of the group $(\mathbb{Z}/n\mathbb{Z})^*$. The function **CarmichaelLambda**(n) computes Carmichael's λ-function for integers $n > 1$. The value of $\lambda(n)$ is computed using the formula

$$\lambda(n) = \mathrm{lcm}_{p^k \| n} \lambda(p^k)$$

where the least common multiple is taken over the prime powers p^k exactly dividing n. For prime powers p^k, the value of λ is determined as follows: if p

is odd, then $\lambda(p^k) = (p-1)p^{k-1}$, while $\lambda(2) = 1$, $\lambda(4) = 2$ and $\lambda(2^k) = 2^{k-2}$ for $k \geq 3$. The prime factorization of n is required.

Möbius' μ-function also requires the prime factorization of n. For positive integers n, the value of $\mu(n)$ is 0 if n is divisible by the square of a prime number, and equals $(-1)^k$ if n is square-free and divisible by exactly k different primes. The MAGMA name for this function is **MoebiusMu**(n). (Notice the 'oe' in the spelling, to substitute for the 'ö'.) For example, the output

```
> MoebiusMu(24), MoebiusMu(70);
0 -1
```

indicates that $24 (= 2^3 \times 3)$ is divisible by a square, and that $70 (= 2 \times 3 \times 5)$ is the product of an odd number of distinct primes.

The family of divisor functions σ_i is available in MAGMA under the name **DivisorSigma**. The i^{th} member of this family is defined by

$$\sigma_i(n) = \sum_{d|n} d^i$$

Thus, $\sigma_0(n)$ is the number of divisors of n, $\sigma_1(n)$ is the sum of the divisors of n, and so on. The function **DivisorSigma**(i, n) returns $\sigma_i(n)$. The following statement, prints the sequence containing $\sigma_0(4), \sigma_1(4), \ldots, \sigma_5(4)$:

```
> [DivisorSigma(i, 4): i in [0..5]];
[ 3, 7, 21, 73, 273, 1057 ]
```

As a more advanced example, we define in a single statement a function that returns for positive n and non-negative k the value of

$$\sum_{d|n} \sigma_k(d) \cdot \mu(\frac{n}{d}),$$

summing over all divisors of n:

```
> conv := func< n, k |
        &+[  DivisorSigma(k, n) * MoebiusMu(n div d) :
             d in Divisors(n) ] >;
> conv(5, 3);
125
> conv(8, 2);
64
```

Some experimentation with this function, which uses the **Divisors** function described in Section 20.4.3) will easily lead to the hypothesis that its value

equals n^k. This is true indeed, and forms a particular instance of a general principle called Möbius inversion.

The value of the Legendre symbol $\left(\frac{n}{m}\right)$ for odd primes m and arbitrary integers n is computed by the function **LegendreSymbol**(n, m). For primes m, the value of $\left(\frac{n}{m}\right)$ is 0 if n is divisible by m, and otherwise it equals 1 or -1 depending upon whether or not n is a quadratic residue modulo m. For example, since 11 is congruent to 7^2 modulo 19, $\left(\frac{11}{19}\right)$ equals 1:

```
> LegendreSymbol(11, 19);
1
```

The function **JacobiSymbol**(n, m) extends the Legendre symbol to all positive odd integers m by multiplicativity. These two functions are very fast, since they are easily computed without a knowledge of the prime factorization of n or m, using the property of quadratic reciprocity. The recursive function below, which uses *EvenexpOdd* as defined on p. 338, is a MAGMA language version of the algorithm:

```
function jac(n, m)
    if n ne n mod m then
        return jac(n mod m, m);
    elif n in {0, 1} then
        return n;
    elif n eq m-1 then // i.e., congruent to -1
        return m mod 4 eq 3 select -1 else 1;
    elif IsEven(n) then
        s, d := EvenexpOdd(n);
        return (m mod 8 in {3, 5} and IsOdd(s) select
          -1 else 1) * jac(d, m);
    else
        return (n mod 4 eq 3 and m mod 4 eq 3 select
          -1 else 1) * jac(m mod n, n);
    end if;
end function;
```

20.2.4 Factorizations

Primes are the multiplicative building blocks of the integers. Therefore the algorithms for decomposing integers into primes and for testing primality are of major importance. MAGMA contains advanced functions with many options to call these very complicated algorithms; a description will be found in the final section of this Chapter.

For the frequent use on integers of moderate size the functions **IsPrime** and **Factorize** will suffice. The function **IsPrime**(n), when invoked on an integer n, returns true if and only if $|n|$ is a prime number. The test may take a considerable amount of time for integers of hundreds of decimal digits (for an explanation of the algorithms used and for the faster 'probabilistic' test used by **IsProbablyPrime** see Section 20.4). The function **Factorization**(n) returns the prime decomposition of $|n|$ in the form of a *factorization sequence*, as well as an integer ± 1 that is equal to the sign of n. A factorization sequence for a positive integer n is a sequence of the form $[< p_1, k_1 >, \ldots, < p_r, k_r >]$ where the p_i are primes such that $p_1 < \cdots < p_r$ and the e_i are positive integers, such that $n = \prod_{i=1}^{r} p_i^{e_i}$. Each of the p_i has been subjected to the primality proving algorithm used by **IsPrime**.

```
> f := Factorization(123456789012345678901);
> f;
[ <11, 1>, <5471, 1>, <10000799, 1>, <205126079, 1> ]
> IsPrime(f[3][1]);
true
```

Although factorization sequences are printed in the same way as ordinary sequences (of tuples), they belong to a special category **RngIntEltFact**. This makes it possible to allow them as input to certain functions (such as the arithmetical functions in a previous section) for which (re)computing the factorization takes most of the time.

For example, it would be very inefficient to factorize the product of two integers that have both been factored already. MAGMA allows them to be multiplied in their factored form instead:

```
> a := Factorization(9014637873208);
> b := Factorization(81290847134);
> b;
[ <2, 1>, <7, 1>, <23, 1>, <461, 1>, <547627, 1> ]
> a * b;
[ <2, 4>, <7, 1>, <11, 2>, <23, 3>, <97, 2>, <461, 1>,
<1871, 1>, <547627, 1> ]
```

It is also possible to divide one factorization sequence by another using the / operator, provided that the quotient is exact.

```
> a / [<11, 2>, <97, 1>];
[ <2, 3>, <23, 2>, <97, 1>, <1871, 1> ]
```

Addition and subtraction may also be performed. However, in the case of subtraction, the operands must represent distinct integers, and the result returned will correspond to their difference (as a positive number):

```
> a + b;
[ <2, 1>, <3, 1>, <23, 1>, <251, 1>, <262599709, 1> ]
> a - b;
[ <2, 1>, <23, 1>, <194203196219, 1> ]
> b - a;
[ <2, 1>, <23, 1>, <194203196219, 1> ]
```

Exponentiation is available for positive powers:

```
> a ^ 14;
[ <2, 42>, <11, 28>, <23, 28>, <97, 28>, <1871, 14> ]
```

Table 20.5. Functions for factorization sequences

GCD(s,t)	**IsOdd**(s)	**MoebiusMu**(s)
LCM(s,t)	**IsEven**(s)	**EulerPhi**(s)
PrimeBasis(s)	**IsPrime**(s)	**NumberOfDivisors**(s)
SquareFree(s)	**IsPrimePower**(s)	**SumOfDivisors**(s)
IsSquare(s)	**IsPower**(s)	**Divisors**(s)

Most of the other integer functions that may also be applied to factorization sequences are listed in Table 20.5.

To build a factorization sequence directly, or when 'copying and pasting' previous output, the use of **SequenceToFactorization**, or **SeqFact** for short, is recommended (but the sequence has to have the right format!):

```
> f := SeqFact([<3, 11>, <19,2>]);
> f, Facint(f);
[ <3, 11>, <19, 2> ]
63950067
> g := Seqfact([<3, 11>, <2, 2>]);

>> g := SeqFact([<3, 11>, <2, 2>]);
       ^
Runtime error in 'SeqFact': Sequence not a factorization
list (must be a sequence of ordered <prime, exponent>
tuples)
```

The function **Facint**(Q) or **FactorizationToInteger**(Q) is the inverse of **Factorization**(n), and converts a factorization sequence Q to the corresponding integer n. For example:

```
> a := Factorization(9014637873208);
```

```
> a;
[ <2, 3>, <11, 2>, <23, 2>, <97, 2>, <1871, 1> ]
> Facint(a);
9014637873208
```

20.2.5 Modular Arithmetic

There are two ways of performing modular arithmetic in MAGMA. If several operations modulo the same n are required, then the user should define the ring **ResidueClassRing**(n) and proceed as described in Section 20.3. For modular arithmetic in the context of ordinary integer computation, the fundamental MAGMA operations are **Quotrem**, **div**, and **mod**. These and the other operations are listed in Table 20.6. Many number-theoretic computations may be performed much more efficiently using these functions rather than integer arithmetic followed by a final modular reduction, since the intermediate values in the computation are never allowed to become very large.

For example, in situations where an integer a has to be raised to some very large power, k say, it is important to note if it is sufficient to know the result modulo some integer n. If so, then the function **Modexp**(a, k, n) should be used. Given integers a, k, n such that $k \geq 0$ and $n > 1$, **Modexp** returns $a^k \bmod n$. While this could be computed using the expression $a\hat{}k$ **mod** n, if k is sufficiently small, **Modexp** performs reduction modulo n as the powers of a are computed, so that the creation of intermediate integers that are large relative to n is avoided. As a consequence, **Modexp**(a, k, n) is far more efficient:

```
> N := 508213
> time 101^(N-1) mod N;
1
Time: 27.080
> time Modexp(101, N - 1, N);
1
Time: 0.000
```

Note that, unlike in the case of $a\hat{}k$ **mod** n, there is no restriction on the size of the exponent k when using the function **Modexp**(a, k, n).

As an example of the function **Solution**, consider the linear congruence $8x \equiv 7 \pmod{11}$. The appropriate function call for finding x is:

```
> Solution(8, 7, 11);
5
```

Table 20.6. Modular arithmetic

MAGMA	Meaning
Quotrem(a, n)	Given a and n with $n \neq 0$, returns (i) quotient q, (ii) remainder r, such that $a = qn + r$ and $\|r\| < n$ ($0 < r < n$ if $n > 0$; $n < r < 0$ if $n < 0$)
a **div** n	Quotient q when a is divided by n, as in **Quotrem**
a **mod** n	Remainder r when a is divided by n, as in **Quotrem**
Modexp(a, k, n)	Modular power $a^k \bmod n$
Modsqrt(a, n)	b such that $b^2 \equiv a \pmod{n}$ and $0 \leq b < n$
Modinv(a, n)	b such that $ab \equiv 1 \pmod{n}$ and $1 \leq b < n$ ($\gcd(a, n) = 1$)
Order(a, n)	Order of a modulo n, i.e., smallest integer $k \geq 1$ such that $a^k \equiv 1 \pmod{n}$, or 0 if $\gcd(a, n) \neq 1$.
IsPrimitive(a, n)	**true** if a is a primitive root for n
PrimitiveRoot(n)	Seeks a primitive root for n: returns primitive root if function finds it, or 0 if none exists or can be found
NormEquation(d, m)	If a solution to $x^2 + dy^2 = m$ exists ($0 \leq m < d$), returns **true** and values for x and y; else returns **false**
Solution(a, b, n)	Returns solution to $ax \equiv b \pmod{n}$
Solution(A, B, N)	Given integer sequences A, B, N of same length, with terms of N being pairwise coprime, returns solution to simultaneous system $A[i]x \equiv B[i] \pmod{N[i]}$. Solution will satisfy $0 \leq x < N_1 N_2 \ldots N_k$.

The output shows that $x \equiv 5 \pmod{11}$.

More generally, a system of simultaneous linear congruences may be solved using the with Chinese Remainder Theorem, which is implemented as the function **ChineseRemainderTheorem** (which is synonymous to **Solution**). Given the system

$$\begin{aligned} a_1 x &\equiv b_1 \pmod{n_1} \\ a_2 x &\equiv b_2 \pmod{n_2} \\ &\vdots \\ a_k x &\equiv b_k \pmod{n_k} \end{aligned}$$

then x is returned by **Solution**(A, B, N), where A is the sequence $[a_1, \ldots, a_k]$, B is $[b_1, \ldots, b_k]$, and N is $[n_1, \ldots, n_k]$. The moduli n_i must be pairwise coprime. For example, consider this simultaneous system:

$$\begin{aligned} 5x &\equiv 4 \pmod{13} \\ -7x &\equiv 1 \pmod{24} \\ x &\equiv 3 \pmod{7} \end{aligned}$$

The sequences could be assigned and the function call made as follows:

```
> A := [5, -7, 1]; B := [4, 1, 3]; N := [13, 24, 7];
> Solution(A, B, N);
1865
```

Therefore 1865, satisfies all the congruences, and is the least positive solution.

20.3 Residue Class Rings

In the integer ring $\mathbb{Z}$, all subrings are ideals. Therefore the two constructors **sub** and **ideal** produce the same result when applied to $\mathbb{Z}$ and a list of generators. To construct the ideal generated by 12 and 18, type:

```
> I := ideal< Z | 12, 18 >; I;
Ideal of Integer Ring generated by 6
```

The category of an ideal of $\mathbb{Z}$ is **RngInt**, the same as the category of $\mathbb{Z}$.

Since $\mathbb{Z}$ is a principal ideal domain, for every ideal there exists a single element which generates it. MAGMA automatically finds a generator, which is returned by **Generator**(I):

```
> Generator(I);
6
```

This generator, 6, is the greatest common divisor of 12 and 18.

Ideals of $\mathbb{Z}$ are infinite (except the zero ideal), so it is impossible to iterate over their elements. However, the sum, product and intersection of two ideals can be found, using the operators +, * and **meet**:

```
> I + ideal< Z | 15 >;
Ideal of Integer Ring generated by 3
```

MAGMA computes the result by doing greatest common divisor calculations and other simple operations.

Note that the parent of the elements of an ideal of $\mathbb{Z}$ is $\mathbb{Z}$ itself rather than the ideal:

```
> Z eq Parent(I!42);
true
```

A quotient of the integer ring $\mathbb{Z}$ modulo the ideal of $\mathbb{Z}$ generated by n is the same as the residue class ring $\mathbb{Z}/n\mathbb{Z}$ consisting of the equivalence classes of remainders upon division by n. There are three ways of creating such a quotient in MAGMA: as a quotient of $\mathbb{Z}$ by the ideal, as a quotient of $\mathbb{Z}$ by n, or directly using the function **ResidueClassRing**(n):

```
> Z6 := ResidueClassRing(6);
> Z6;
Residue class ring of integers modulo 6
> Z6 eq quo< Z | I >, Z6 eq quo< Z | 6 >;
true true
```

When used to create an integer residue class ring, the **quo** constructor only returns one value, the quotient itself. It does not return a homomorphism from the ideal to the quotient, since (as explained above) ideals of the integer ring are not implemented in the usual way.

It is possible to create ideals in residue class rings:

```
> Z6I := ideal< Z6 | 2 >;
> Z6I;
Ideal of residue class ring of integers modulo 6
generated by 2
> Generator(Z6I);
2
> Set(Z6I);
{ 0, 2, 4 }
```

All subrings of residue class rings are ideals, and all these ideals are principal. The category of all residue class rings and their ideals is **RngIntRes**.

Table 20.7 (p. 352) lists some of the functions on residue class rings and their elements. A few of them are shared with finite field operations, since if p is prime then **ResidueClassRing**(p) is isomorphic to **FiniteField**(p). Some others are related to the modular arithmetic functions shown in Table 20.6 (p. 349).

Given a residue class ring R, the function **Modulus**(R) returns the generator of the ideal used to construct it as a quotient. The ideals of R have the same modulus as R:

```
> Modulus(Z6), Modulus(Z6I);
6 6
```

The function **PrimitiveRoot**(R) or **PrimitiveElement**(R) returns an element of R that generates all the units multiplicatively:

Table 20.7. Functions for residue class rings

MAGMA	Meaning
Generator(R)	Generator of residue class ring R
Modulus(R)	Modulus of residue class ring R
AdditiveGroup(R)	Returns (i) Abelian group A isomorphic to R as an additive group, (ii) bijection from R to A
MultiplicativeGroup(R), **UnitGroup**(R)	Returns (i) Abelian group A isomorphic to group of units of R as an additive group, (ii) map from units of R to A
Order(a)	Multiplicative order of a, or zero if the order is infinite
IsPrimitive(a)	**true** if a is primitive
PrimitiveElement(R)	A primitive element of R, or 0 if no primitive element exists or can be found
IsSquare(a)	**true** if a is square
SquareRoot(a), **Sqrt**(a)	A square root of a
AllSquareRoots(a), **AllSqrts**(a)	Sequence of all square roots of a
Solution(a, b)	A solution to $ax \equiv b \pmod{n}$ where n is the modulus of the parent of a and b

```
> Z38 := ResidueClassRing(38);
> prim := PrimitiveElement(Z38);
> prim;
3
```

The function returns the zero of R if R does not have a primitive element or if it has one that cannot be found by MAGMA. This value is chosen since zero is never a primitive element.

The set of all the units of R may be formed by forming the powers of a primitive element of R. The cardinality of this set is given either by applying the function **Order** to the primitive element or by evaluating $\phi(n)$ (using **EulerPhi**) where n is the size of R:

```
> Order(prim);
18
> EulerPhi(38);
18
> units := {prim ^ i: i in [1..18]}; units;
{ 17, 35, 1, 3, 37, 21, 5, 23, 7, 25, 9, 27, 11, 29, 13,
31, 15, 33 }
> #{ u : u in units | IsPrimitive(u) };
6
```

```
> EulerPhi(18);
6
```

As the last two lines above suggest, the number of units of R which are primitive elements is $\phi(\phi(n))$.

The set of all the units of R forms a cyclic multiplicative group. The function **UnitGroup**(R) or **MultiplicativeGroup**(R) returns the abelian group G corresponding to this set, as a member of the **GrpAb** category, together with the map from the units of R to G. For instance:

```
> G, h := UnitGroup(Z38);
> G, h;
AbelianGroup isomorphic to Z/18
Defined on 1 generator
Relations:
    18*G.1 = 0
Mapping from: RngIntRes: Z38 to GrpAb: G
```

The elements of G are group elements, not elements of R. However, they may be converted to units in R by means of the preimage operator `@@` and the mapping given by the second return value of **UnitGroup**:

```
> G.1 @@ h;
21
> IsPrimitive(Z38 ! 21);
true
> units eq (Set(G) @@ h);
true
```

Section 20.2.5 deals with modular arithmetic on integers. The operations explained there are suitable if the main objects under investigation are integers. If a residue class ring is being used as a structure in its own right (e.g., as the coefficient ring for a polynomial ring) then it should be constructed with the operations outlined above.

20.4 Primes and Factorization

A number of modern applications of number theory, such as cryptography, require the factorization of large integers. Despite great progress over the past few decades, finding a factor of a large number may take a considerable amount of time (and memory). On the other hand, it is much easier to determine whether a large number is prime or not, without explicitly finding any

factors. Such an algorithm should be employed before a factorization attempt is started. In this section, MAGMA's algorithms for testing primality and for factorization are discussed.

20.4.1 Primality versus Compositeness

MAGMA's most straightforward function for testing primality is **IsPrime**(n). It returns **true** if and only if the positive integer n is prime:

```
> IsPrime(51);
false
```

If $n < 25 \times 10^9$, a strong pseudoprime test is used to determine the primality. For larger n, a version of the Atkin-Morain elliptic curve primality proving algorithm is employed. Both are rigorous tests.

Although very powerful, these rigorous primality proofs may take some time for very large input. A probabilistic test, often referred to as the 'Miller-Rabin' algorithm, is provided as an alternative. This works very fast, but has the disadvantage that there is a very small probability that a composite number is declared prime. There are two ways of invoking this algorithm, both designed to ensure that the user is aware that the method is non-rigorous:

```
> n := 4507445537641;
> IsProbablePrime(n);
false
> IsPrime(n : Proof := false);
false
```

The output **false** is always correct, so a prime number will never be declared composite. (Therefore this test is more aptly referred to as a *compositeness test* rather than a primality test.)

The probability that a composite is declared prime is so small, that it hardly ever occurs in practice. Therefore this 'probabilistic' algorithm is often used to distinguish prime numbers from composites. Since this is so frequently used, we will present the idea behind the test.

The Miller-Rabin algorithm is based (again) on Fermat's little theorem, or rather a kind of converse: if $a^{n-1} \not\equiv 1 \bmod n$, then n cannot be prime. Unfortunately, there exist integers that are composite, but for which $a^{n-1} \equiv 1 \bmod n$ for every a coprime to n.

```
> n := 9091*18181*27271;
> Modexp(3, n-1, n);
```

```
1
> Modexp(Random(n), n-1, n);
1
```

Using the **EvenexpOdd** function given earlier, the situation may be examined in more detail.

```
> s, d := EvenexpOdd(n);
> s, d;
3 563430692205
> r := Random(n);
> m := Modexp(r, d, n); m;
3295609346672
```

This shows that $r^d \equiv 3295609346672 \bmod n$, where d is the odd part of $n-1$ (and the even part is 2^3).

```
> m := Modexp(m, 2, n);
> m;
1983365289
m := Modexp(m, 2, n);
> m;
1
```

Thus,

$$r^{2\cdot d} \equiv 1983365289 \bmod n, \qquad r^{2^2\cdot d} \equiv 1 \bmod n,$$

(and therefore also $r^{n-1} \equiv 1 \bmod n$). Hence

$$1983365289^2 \equiv 1 \bmod n;$$

so n cannot be a prime: if it were, the integers modulo n would form a field, and in a field 1 and -1 are the only square roots of 1.

It has been proven that for every composite n there exists integers r that will 'witness' the compositeness of n as above. Not every randomly chosen r between 1 and n is a witness, but at most a quarter of them (and usually much less) will fail to be witness. Repeating this procedure with several, k say, random choices for r will usually suffice to expose composites; if each fails as a witness one could say that with error probability less than 4^{-k} n is a prime.

The number of choices for r may be modified by the user in the case of **IsProbablePrime** by setting the **Bases** parameter:

```
> IsProbablePrime(n : Bases := 3);
false
```

The default, both for `IsProbablePrime` and also for the non-rigorous version of **IsPrime**, is 20. Thus, the probability that `IsPrime(n : Proof := false)` declares a composite number prime is less than 2^{-40}. This is why it never occurs in practice.

Note that the fact that the execution time for this algorithm tends to be much shorter than that for a rigorous test, often outweighs the minor risk of an incorrect return value. However, MAGMA's default algorithm for **IsPrime** is rigorous, following the MAGMA philosophy that results should be exact unless this condition is relaxed by the user.

If the user wishes to verify MAGMA's proof for the primality of a number, the function **PrimeCertificate** may be used. It returns sufficient information to enable the user to independently verify primality. For details we refer to the *Handbook*.

20.4.2 Factorization

MAGMA offers several algorithms for finding factors of an integer. The standard function **Factorization** will attempt to find the complete prime factorization, using a range of methods. Each of these methods may also be used independently by the user; table Table 20.8 lists the options.

The function **Factorization**(n) is designed to factorize n completely. It returns a sequence listing all the prime factors of n, together with their multiplicities, in the form of a MAGMA factorization sequence (see Section 20.2.4). For instance:

```
> Factorization(1959703885480);
[ <2, 3>, <5, 1>, <11, 2>, <23, 1>, <97, 2>, <1871, 1> ]
```

Therefore $1959703885480 = 2^3 \times 5^1 \times 11^2 \times 23^1 \times 97^2 \times 1871^1$.

Factorization(n), for a non-zero integer n, actually has up to three return values, though in practice usually only the first value is significant. The first return value is, strictly speaking, a factorization of $|n|$, and the second value is 1 or -1, depending on the sign of n. For example:

```
> a, b := Factorization(-1959703885480);
> a, b;
[ <2, 3>, <5, 1>, <11, 2>, <23, 1>, <97, 2>, <1871, 1> ]
-1
```

In the unlikely event that the function is unable to factorize n completely, it returns a third value, a sequence containing factors of n which are known to

be composite, but could not be decomposed. However, unless any of the parameters limiting the search for factors has been set (see below), the function will continue trying to factor its argument, and there will never be unfactored composites (but the computation may not finish at all!). In such a case there will be no third return value, and any third variable will be left unassigned. The correct way to deal with such case is as follows:

```
> a, b, c := Factorization(-1959703885480);
> if assigned c then "remaining composites:", c; end if;
```

To completely factorize a large integer n, the best approach is to use a combination of methods. The function **Factorization** applies a fairly complicated strategy. Firstly a compositeness test is applied. Then a test is applied to determine whether n is a power of an integer; if so, factorization is much easier whereas general factorization problems have trouble detecting it. Next a test is made to determine if n is of the special form $b^m \pm 1$; if that is the case, an intelligent data base search is performed (see also the **Cunningham** intrinsic below). At this point several factorization methods are deployed in succession: trial division to extract small factors, Pollard's ρ method, the elliptic curve method, and finally a quadratic sieve. The parameters for these algorithms are set to values that are thought to be well chosen when applied to a 'general' n, about which nothing is known.

The primality of each prospective prime factor is established using a rigorous primality prover. It is possible to suppress the rigorous test in favour of a probabilistic compositeness test, by setting the parameter **Proof** to **false**. It is also possible to specify the number of random bases used by the Miller-Rabin compositeness test, by setting the **Bases** parameter:

```
> n := 113572619;
>  Factorization(n : Proof := false, Bases := 4);
[ <10337, 1>, <10987, 1> ]
```

Finally, each of the four algorithms used by the general factorization function may be limited by a parameter. The algorithms are more fully discussed below, but here we briefly indicate the meaning of the parameters. Setting **TrialDivisionLimit** to B limits the search for factors by trial division to primes not exceeding B; **PollardRhoLimit** bounds the number of iterations for the ρ method, **ECMLimit** bounds the number of elliptic curves tried; and **MPQSLimit** restricts the application of the quadratic sieve to integers having at most this many digits. — a lower bound (of around 25 digits applies automatically: the sieve can not be applied to small numbers).

Setting the verbose flag for factorization will illustrate the effect:

```
> SetVerbose("Factorization", true);
```

```
> n := 113572619;
> Factorization(n : PollardRhoLimit := 10, ECMLimit := 0);
Integer main factorization (primality of factors will be
proved)
    x: 113572619

Trial Division
    x: 113572619
    min: 3
    max: 10000
    0 factors found
    Time: 0.009

Pollard Rho
    Trials: 10
    x: 113572619
    No factor found
    Time: 0.019

1 composite number remaining

Too small or too big for MPQS

Total time: 0.040
[]
1 [ 113572619 ]
```

Hence the limits imposed caused the factorization attempt to fail.

In general, it is not necessary to change the default values for the **Factorization** parameters.

This concludes the description of the general hybrid factorization function; we continue with an explanation of how individual factorization algorithms may be invoked.

The **Cunningham** function uses an intelligent database search to factor integers of the form $b^e \pm 1$. The database contains many key factors for integers of these form, without being excessively large. For moderate values of b and e this may yield spectacularly fast factorizations:

```
> Cunningham(17, 93,-1);
[ <2, 4>, <307, 1>, <4093, 1>, <6123493, 1>, <23125381, 1>,
<32914090064285401, 1>, <4083338668402072819, 1>,
<347340647626008901939025023, 1>,
<203333341969890208758135465337 51, 1> ]
```

Table 20.8. Factorization functions

MAGMA	Meaning
Factorization(n)	Full factorization using various methods
Cunningham(b, e, c)	Factorization of $b^e + c$, $c = \pm 1$
TrialDivision(n)	Trial division (up to 10000)
TrialDivision(n, B)	Trial division up to B
PollardRho(n)	Pollard's ρ method (8191 iterations)
PollardRho(n, c, s, k)	ρ method, using $x^2 + c$, seed s, k steps
pMinus1(n)	Pollard's $p - 1$ method, smoothness 100000
pMinus1(n, B, s, w)	$p - 1$ method, smoothness B, seed s
SQUOFOF(n)	Shanks' square form factorization, 200000 iterations
SQUOFOF(n, B)	Shanks' square form factorization, B iterations
ECM(n)	Lenstra's elliptic curve method, smoothness 500, 10 curves
ECM(n, s, r, c)	ECM, smoothness s (up by r), c curves
MPQS(n)	(double prime) multiple polynomial quadratic sieve

```
Time: 0.149
```

The **TrialDivision** function attempts to find factors by performing trial division with (prime) numbers up to a given bound. If a bound is not supplied by the user, a default bound of 10000 is used.

```
> n := 17^93-1;
> TrialDivision(n);
[ <2, 4>, <307, 1>, <4093, 1> ]
[ 13441557227706275246837019714402875334208629544585947240 5\
43457627951138685515775079359882091961892138542137 1 ]
```

In general, trial division, in common with most of the other algorithms, will not succeed in finding the complete factorization. Therefore, a sequence of unfactored composite factors is returned as a second value.

Pollard's ρ method uses iteration of a (quadratic) function of the form $x^2 + c$, starting at some value s, in the hope that eventually a cycle modulo one of the factors of n is obtained. The default values $c = 1$, $s = 1$, and $k = 8191$ may be changed by the user, as follows (continuing the previous example):

```
> c := 1; s := 2; k := 21000;
> PollardRho(n, c, s, k);
[ <2, 4>, <307, 1>, <4093, 1>, <6123493, 1>, <23125381, 1> ]
[ 94920815999855427820607040128793738158309205740121616465 4\
```

```
911229181091566480170006540911378387 ]
```

As can be seen here, this method is very good at finding small prime factors.

To a certain extent this is also true of Pollard's $p-1$ method, if one the factors p of the number to be factored has the property that $p-1$ is *smooth*, which is to say that all of its prime factors are 'small'. How small they need be for the method to succeed is controlled by the smoothness bound, which is 100000 by default. The user can also change the seed (2) used as well as the interval after which greatest common divisors are taken to detect factors (100 by default); we refer to the *Handbook* for more details.

```
> pMinus1(n);
[ <6123493, 1>, <23125381, 1> ]
[ 1256551, 151873305599768684512971264206069981053294729184\
19458634478579666897465063682720104654582054192 ]
```

The most powerful factorization methods are the elliptic curve method 'ECM' and the multiple polynomial quadratic sieve 'MPQS'. Usually ECM is employed to find prime factors of up to 25 digits (reamining after trial division and Pollard ρ have been used), and MPQS is used to crack products of two primes of up to 45 digits or so. Options on ECM are to change its initial smoothness bound (default 500), the proportion by which this bound is incremented at every new curve (default factor 1.2), and the total number of curves that is tried (10). For example

```
> ECM(n, 1000, 1.1, 5);
[ <2, 4>, <307, 1>, <4093, 1>, <6123493, 1>, <23125381, 1> ]
[ 94920815999855427820607040128793738158309205740121616465 4\
911229181091566480170006540911378387 ]
```

MPQS should only be applied to numbers of more than 25 digits having no small prime factors. For numbers of more than 80 digits substantial amounts of time and space may be required. A user wishing to factor large integers is urged to read the relevant *Handbook* sections.

20.4.3 Other Functions Related to Primes and Factorization

MAGMA provides two functions which use primality tests to locate primes in the neighbourhood of a given non-negative integer n. **NextPrime**(n) returns the least prime number greater than n and **PreviousPrime**(n) returns the greatest prime number less than n:

```
> NextPrime(100000000), PreviousPrime(100000000);
100000007 99999989
```

When speed is required and n is large, the **Proof** parameter can be set to **false**. In this case the Miller-Rabin test is used with 20 iterations, and so the previous or next probable prime is returned. The **Proof** parameter works in the same way for these functions as it does with the function **IsPrime**.

The function **PrimeBasis** or **PrimeDivisors** returns a sequence containing the distinct prime divisors of a positive integer. For example, $44442 = 2 \times 3^3 \times 823$, so its distinct prime divisors are 2, 3 and 823:

```
> PrimeBasis(44442);
[ 2, 3, 823 ]
```

There is also a function **Divisors**(n) that returns a sequence containing all the divisors of n, not just the prime divisors. Continuing the example,

```
> Divisors(44442);
[ 1, 2, 3, 6, 9, 18, 27, 54, 823, 1646, 2469, 4938, 7407,
14814, 22221, 44442 ]
```

produces a sequence of all the divisors of 44442, including 1 and 44442 itself.

Divisors may also be applied to a factorization sequence.

20.5 Notes.

For fast arithmetic on large integers MAGMA employs both the method attributed to Karatsuba, and the fast Fourier transform. (The best general reference is Knuth's second volume in the series *The art of computer programming*, called *Seminumerical algorithms*). For primality proving of large integers (exceeding $25 \cdot 10^9$ in particular) a version of François Morain's *elliptic curve primality prover* is employed. For the factorization of integers implementations of the *elliptic curve method* by Arjen Lenstra and Allan Steel are used, as well as an iplementation by Arjen Lenstra of the double prime variant of the multiple-polynomial quadratic sieve.

21. Univariate Polynomial Rings

Polynomials in MAGMA may have one or several indeterminates, and may be defined over any ring, even another polynomial ring. MAGMA has two types of polynomial rings: *univariate* and *multivariate*. Univariate polynomial rings always have one indeterminate, while multivariate polynomial rings may have more than one indeterminate. The two types of polynomial rings have quite different properties and features. Consequently, the two types of polynomial rings are implemented with different categories and although they share many simple operations, the major algorithms are quite distinct. Most functions in other areas of MAGMA which happen to deal with polynomials take and return univariate polynomials. For example, univariate polynomials are universal in computations with finite fields, as they arise in computations of minimal polynomials, field extensions, and embeddings. In linear algebra and module theory, functions like **CharacteristicPolynomial** return univariate polynomials. In contrast, multivariate polynomials usually only arise as elements of ideals of multivariate polynomial rings in the area of Commutative Algebra.

Note that a multivariate polynomial ring with one indeterminate is not the same as a univariate polynomial ring over the same coefficient ring. The term "univariate" does not just mean "one variable" but implies a type of polynomial ring with quite different properties (and representation). It is almost always best to use the univariate and multivariate facilities in the obvious way, rather than to mimic a multivariate ring by creating univariate polynomial rings with univariate polynomial rings as their coefficient rings, or to construct single-indeterminate multivariate polynomial rings.

This chapter describes univariate polynomial rings and their elements, while Chapter 22 describes multivariate polynomial rings and their elements.

A univariate polynomial ring $P = R[x]$ over the coefficient ring R always has one indeterminate x and a univariate polynomial f of P is a sum of the form $\sum_{i=0}^{d} c_i x^i$, with $c_i \in R$ for each i. Such a polynomial f is represented internally as the coefficient vector $[c_0, \ldots, c_d]$. This is the most efficient way in general for representing univariate polynomials.

The operations available on univariate polynomials include arithmetic, computation of derivatives, evaluation for a particular value of the indeter-

minate, and division to find quotient and remainder. For certain coefficient rings there are functions for greatest common divisors, factorization, and resultants.

The category of univariate polynomial rings is **RngUPol**.

21.1 Constructing Polynomial Rings

The function to create a univariate polynomial ring is **PolynomialRing**(R) or **PolynomialAlgebra**(R), where R is the ring from which the coefficients are taken. For example, the following assignment defines the polynomial ring P over the integers in the indeterminate x:

```
> P<x> := PolynomialAlgebra(IntegerRing());
> print P;
Univariate Polynomial Algebra in x over Integer Ring
```

It is customary to supply the indeterminate's name by means of generator assignment (within angle brackets on the left of the assignment statement), so that it becomes an identifier and a printname. Otherwise, the indeterminate is referred to as $P.1$, since it is the only (first) indeterminate of P.

The functions **CoefficientRing**(P) and **Rank**(P) return the coefficient ring R of a polynomial ring P, and the number of indeterminates which P has (always 1 for univariate polynomial rings).

21.2 Creating Polynomials

There are several ways of creating elements of an univariate polynomial ring. The most straightforward is to follow the mathematical representation involving powers of the indeterminate. For instance, the next line shows how to create the polynomial $f = x^4 - 5x^2 + 3x + 6$ as a member of P:

```
> f := x^4 - 5*x^2 + 3*x + 6;
> print f;
x^4 - 5*x^2 + 3*x + 6
```

For constant polynomials, there is a slight complication, because the element of the coefficient ring has to be coerced into the polynomial ring. For example, the following assignment creates the polynomial $g = 7$:

```
> g := P!7;
```

There is another way of creating univariate polynomials, in which every coefficient is listed in order from the constant term to the leading coefficient. The coefficients may either be placed on the right side of an **elt** constructor, or in a sequence which is coerced into the ring. For example, the polynomial $h = 8x^7 + 5x^6 + 9x^5 - 2x^4 + x^3 - x + 3$ in P can be created as follows:

```
> h := P![3, -1, 0, 1, -2, 9, 5, 8];
> print h;
8*x^7 + 5*x^6 + 9*x^5 - 2*x^4 + x^3 - x + 3
```

or alternatively as

```
> h := elt< P | 3, -1, 0, 1, -2, 9, 5, 8 >;
```

The coefficient-list method may be preferable when most of the coefficients are non-zero, whereas the method of expressing the polynomial in the indeterminates is preferable when the degree is large but only a few coefficients are non-zero.

21.3 Univariate Rational Function Fields

A *univariate rational function field* F of over the ring R is the field of fractions of a univariate polynomial ring P over the ring R. That is, it consists of fractions whose numerator and denominator are univariate polynomials over R. The polynomial ring P is called the *ring of integers* of F. There are two ways to create the field F. Given P, **FieldOfFractions**(P) returns F. Alternatively, **FunctionField**(R) returns F. The function **IntegerRing**(F) returns P.

The example below demonstrates how to compute in the field of rational expressions in the indeterminate y over the field GF(27):

```
> GF27<w> := GF(27);
> P<x> := PolynomialRing(GF27);
> FF<y> := FieldOfFractions(P); print FF;
Rational function field of rank 1 over GF(3^3)
Variables: y
> print P eq IntegerRing(FF);
true
> t := (1+y) / (1-y^2); print t;
2/(y + 2)
> print t^-1;
2*y + 1
```

```
> print Numerator(t), Denominator(t);
2
x + 2
```

Notice from the example that if $t = \frac{n}{d}$ is an element of a function field, where a and b are expressed in lowest terms, then **Numerator**(t) and **Denominator**(t) return n and d as elements of the corresponding ring of integers (i.e., the univariate polynomial ring). These functions behave similarly for elements of the rational field.

21.4 Operations on Univariate Polynomials

Table 21.1. Degree, terms and coefficients of univariate polynomials

MAGMA	Meaning
Coefficients(f)	Sequence of coefficients of f in ascending order
Coefficient(f, k)	Coefficient of k^{th} power of indeterminate of f
MonomialCoefficient(f, m)	Coefficient that the monomial x^m has as a term of f
Degree(f)	Degree of f
Terms(f)	The non-zero terms of f as a sequence
LeadingTerm(f)	Term with highest occurring power of indeterminate
LeadingCoefficient(f)	Coefficient of leading term of f
TrailingTerm(f)	Term with lowest occuring power of indeterminate
TrailingCoefficient(f)	Coefficient of trailing term
Reductum(f)	f minus the leading term

Table 21.2 and Table 21.1 (p. 366) list many operations on univariate polynomials. For example:

```
> P<x> := PolynomialAlgebra(IntegerRing());
> f := x^4 - 5*x^2 + 3*x + 6;
> print Derivative(f);
4*x^3 - 10*x + 3
> print Evaluate(f, 5);
521
> print Coefficient(f, 1);
3
```

Table 21.2. Some operations on univariate polynomials

MAGMA	Meaning
Derivative(f)	Derivative of univariate f
Derivative(f, n)	n^{th} derivative of univariate f
Evaluate(f, r)	Value of univariate f when r is substituted for indeterminate; if r is in coefficient ring R, result will be in R, else (if possible) result will be in parent of r
Quotrem(f, g)	Given polynomials f and g, return polynomials q and r such that $f = qg + r$ and the degree of r is strictly less than the degree of g
f **div** g	q as described in **Quotrem**
f **mod** g	r as described in **Quotrem**

Notice that **Evaluate** for univariate polynomials is able to accept some substitution values that do not belong to the coefficient ring:

```
> MR := MatrixRing(IntegerRing(), 3);
> m := MR ! [3, 6, 1, 34, 1, 5, 2, 0, 3];
> print Evaluate(f, m);
[49101 10410  9255]
[61340 45161 16665]
[ 4410  2820  1161]
```

The zero polynomial is a special case. Mathematically, its degree is $-\infty$, but MAGMA deems its degree to be -1. Since it has no coefficients and no terms, the user should take care when writing routines that call these functions, including special treatment for the zero polynomial case as appropriate.

21.5 Functions for Special Types of Polynomial Rings

The functions **GCD** and **LCM**, which return the greatest common divisor and least common multiple (see Table 19.8 (p. 332)), are available for elements of a univariate polynomial ring whose coefficient ring is the integer ring $\mathbb{Z}$, a field, a residue class ring $\mathbb{Z}/p\mathbb{Z}$ where p is prime, a valuation ring, or a polynomial ring over any of these. For example:

```
> P<u> := PolynomialRing(IntegerRing());
> f := (2*u+1)^10 - 1;
> g := (2*u-1)^10 - 1;
> print GCD(f, g);
4*u
```

In addition, over a field or $\mathbb{Z}/p\mathbb{Z}$ only, the function **XGCD**(f, g) calculates the extended greatest common divisor. It returns three polynomials d, u and v such that d is the greatest common divisor of f and g, and $tf + ug = d$:

```
> R<r> := PolynomialRing(RationalField());
> f1 := r^2 + 5*r + 6; f2 := r^2 + r - 6;
> print XGCD(f1, f2);
r + 3
1/4
-1/4
> print 1/4*f1 - 1/4*f2;
r + 3
```

Table 21.3. Content and Primitive Functions

MAGMA	Meaning
Content(f)	Greatest common divisor of the coefficients of f
PrimitivePart(f)	f divided by its content
Contpp(f)	Returns the content of f together with the primitive part of f

Table 21.4. Functions on polynomials over the integers

MAGMA	Meaning
AbsoluteValue(f), **Abs**(f)	f if leading coefficient of f is positive, else $-f$
Sign(f)	Sign of the leading coefficient of f
SumNorm(f)	Sum of the coefficients of f
MaxNorm(f)	Maximum of the absolute values of the coefficients of f

Table 21.3 (p. 368) lists some functions that apply only to polynomial rings which have a GCD algorithm. The functions compute contents and primtive parts of polynomials.

Table 21.4 (p. 368) lists some other functions that apply only to polynomial rings over the integers. For instance:

```
> print Coefficients(k);
[ 1, 6, 3 ]
> print MaxNorm(k), SumNorm(k);
```

```
6 10
```

21.6 Factorization and Root-Finding

21.6.1 Factorization

The function **Factorization**(f) factorizes the univariate polynomial f into powers of irreducible factors. For instance,

```
> P<x> := PolynomialRing(IntegerRing());
> f := (2*x + 1)^30 - 1;
> print Factorization(f);
[
    <2, 2>,
    <x, 1>,
    <x + 1, 1>,
    <4*x^2 + 2*x + 1, 1>,
    <4*x^2 + 6*x + 3, 1>,
    <16*x^4 + 24*x^3 + 16*x^2 + 4*x + 1, 1>,
    <16*x^4 + 40*x^3 + 40*x^2 + 20*x + 5, 1>,
    <256*x^8 + 896*x^7 + 1344*x^6 + 1152*x^5 + 624*x^4 +
        224*x^3 + 56*x^2 + 8*x + 1, 1>,
    <256*x^8 + 1152*x^7 + 2240*x^6 + 2432*x^5 + 1584*x^4 +
        608*x^3 + 120*x^2 + 8*x + 1, 1>
]
```

This function is available when the coefficient ring is $\mathbb{Z}$, $\mathbb{Q}$, a finite field, a residue ring modulo p, an algebraic number field $\mathbb{Q}(\alpha)$, a p-adic ring or field, or a rational function field over any of the above rings except for the p-adic rings and fields.

The first return value of **Factorization** is a sequence of 2-tuples whose first component is an irreducible factor and whose second component is the multiplicity of that factor. This is the factorization sequence of the canonical associate f' of f. For a polynomial f with coefficient ring a field, f' is the monic polynomial associated with f while for a polynomial f with coefficient ring $\mathbb{Z}$, f' is the associate of f with positive leading coefficient. Like the **Factorization** function for integers, the **Factorization** function for polynomials has a second return value, the unit u of the coefficient ring such that $f = uf'$. For example, the following statement factorizes $-5x^{11} + 25x^{10} + x^8 - 5x^7 - 7x^6 + 31x^5 + 20x^4$ over the integers:

```
> P<x> := PolynomialAlgebra(IntegerRing());
```

```
> a, b := Factorization(-5*x^11 + 25*x^10 + x^8
>   - 5*x^7 - 7*x^6 + 31*x^5 + 20*x^4);
> print a, b;
[
    <x - 5, 1>,
    <x, 4>,
    <5*x^6 - x^3 + 7*x + 4, 1>
]
-1
```

Therefore the factorization of the polynomial is $-(x-5)x^4(5x^6-x^3+7x+4)$.

Polynomials over finite fields may also be factorized:

```
> GF503polys<r> := PolynomialAlgebra(GF(503));
> a, b := Factorization(3*r^200+3);
> print a, b;
[
    <r^2 + 102*r + 502, 1>,
    <r^2 + 184*r + 502, 1>,
    <r^2 + 319*r + 502, 1>,
    <r^2 + 401*r + 502, 1>,
    <r^4 + 68*r^3 + 158*r^2 + 387*r + 1, 1>,
    <r^4 + 116*r^3 + 158*r^2 + 435*r + 1, 1>,
    <r^4 + 172*r^3 + 347*r^2 + 274*r + 1, 1>,
    <r^4 + 229*r^3 + 347*r^2 + 331*r + 1, 1>,
    <r^4 + 274*r^3 + 347*r^2 + 172*r + 1, 1>,
    <r^4 + 331*r^3 + 347*r^2 + 229*r + 1, 1>,
    <r^4 + 387*r^3 + 158*r^2 + 68*r + 1, 1>,
    <r^4 + 435*r^3 + 158*r^2 + 116*r + 1, 1>,
    <r^20 + 68*r^15 + 158*r^10 + 387*r^5 + 1, 1>,
    <r^20 + 116*r^15 + 158*r^10 + 435*r^5 + 1, 1>,
    <r^20 + 172*r^15 + 347*r^10 + 274*r^5 + 1, 1>,
    <r^20 + 229*r^15 + 347*r^10 + 331*r^5 + 1, 1>,
    <r^20 + 274*r^15 + 347*r^10 + 172*r^5 + 1, 1>,
    <r^20 + 331*r^15 + 347*r^10 + 229*r^5 + 1, 1>,
    <r^20 + 387*r^15 + 158*r^10 + 68*r^5 + 1, 1>,
    <r^20 + 435*r^15 + 158*r^10 + 116*r^5 + 1, 1>
]
3
```

The **Factorization** function for univariate polynomials has one parameter, **Al**, which allows the user to specify the desired factorization algorithm. This is relevant for univariate polynomials over a finite field $\mathrm{GF}(q)$. The parameter has 3 possible values: `"BerlekampSmall"`, `"BerlekampLarge"`, and

"CantorZassenhaus", giving respectively the small-prime and large-prime Berlekamp methods, and the Cantor-Zassenhaus method. The default value of **Al** is "BerlekampSmall" if $q < 25$, or "BerlekampLarge" otherwise. For example, *GF503polys* above is over a large-prime finite field, so the default algorithm is "BerlekampLarge":

```
> time a := Factorization(r^250-1: Al := "BerlekampLarge");
Time: 9.630
> time a := Factorization(r^250-1); // same as above
Time: 8.170
> time a := Factorization(r^250-1: Al := "BerlekampSmall");
Time: 38.243
> time a := Factorization(r^250-1: Al:="CantorZassenhaus");
Time: 360.736
```

Note that there a wide difference in execution times between the algorithms. The choice of algorithm depends on the field and the properties of the polynomial.

The function **IsIrreducible**(f) returns **true** if the polynomial f is irreducible. For instance, the following line checks that every polynomial p in the factorization sequence a above is irreducible. Since the polynomials are the first components of the tuples t in the sequence, they are accessible as $t[1]$ for each t:

```
> print forall{p:t in a | IsIrreducible(p) where p is t[1]};
true
```

Another factorization function is **SquareFreeFactorization**(f). It returns the squarefree factorization of the associate f' of f as described above. The return value is a factorization sequence of (not necessarily irreducible) factors with their multiplicities. None of these factors will contain the square of any non-constant polynomial.

A further function, **DistinctDegreeFactorization**(f), takes a squarefree univariate polynomial f over a finite field as its input. It returns a sequence of 2–tuples, each consisting of a degree d, together with the product of the degree–d irreducible factors of the associate f'. If the parameter Degree is given a positive integer value L, then only products of factors up to degree L are returned.

The following example takes a monic polynomial over GF(101),d calculates its squarefree factorization, and then finds the distinct degree factorization of its first squarefree factor:

```
> R101<z> := PolynomialRing(GF(101));
```

```
> pol := (z+96)^6 * (z^7 + 9) * (z^8 + 4*z^3 + z);
> sqff := SquareFreeFactorization(pol); print sqff;
[
    <z^15 + 4*z^10 + 10*z^8 + 36*z^3 + 9*z, 1>,
    <z + 96, 6>
]
> print DistinctDegreeFactorization(sqff[1, 1]);
[
    <1, z^4 + 3*z^3 + 59*z^2 + 39*z>,
    <2, z^2 + 34*z + 94>,
    <3, z^3 + 33*z^2 + 39*z + 36>,
    <6, z^6 + 31*z^5 + 52*z^4 + 97*z^3 + 78*z^2 + 95*z + 16>
]
```

The final output means that the first squarefree factor has factors of degree 1 whose product is $z^4 + 3z^3 + 59z^2 + 39z$, factors of degree 2 whose product is $z^2 + 34z + 94$ (i.e., only one factor), and so on.

Finally, there are some functions associated with the factorization of polynomials over a domain, such as the integers. For univariate polynomials, they are **Discriminant**(f) and **Resultant**(f, g).

For example:

```
> P<x> := PolynomialRing(IntegerRing());
> print Resultant(x^10 + 1, x ^5 - 2);
3125
```

21.6.2 Hensel Lifting

One of the algorithms used in univariate polynomial factorization is Hensel lifting. The function **HenselLift**(f, s, R) performs this. Its arguments are a univariate integer polynomial f, a factorization of f modulo some prime p, given as a sequence s of polynomials over $\mathbb{Z}/p\mathbb{Z}$, and a univariate polynomial ring R over $\mathbb{Z}/p^k\mathbb{Z}$ for some k. The function returns the Hensel lifting into R, that is, a sequence of polynomials over $\mathbb{Z}/p^k\mathbb{Z}$ which is the factorization of f modulo p^k. For example, consider the problem of lifting a modulo–5 factorization of $x^5 + 12x^4 - 22x^3 - 163x^2 + 309x - 119$ into a modulo–5^4 factorization:

```
> P<x> := PolynomialRing(IntegerRing());
> P5<y> := PolynomialRing(ResidueClassRing(5));
> P625<z> := PolynomialRing(ResidueClassRing(625));
> f := P![-119, 309, -163, -22, 12, 1]; print f;
x^5 + 12*x^4 - 22*x^3 - 163*x^2 + 309*x - 119
```

Firstly, a factorization s of f modulo 5 is required. It may be found by applying **Factorization** to the coercion of f into $P5$:

```
> fact5 := Factorization(P5!f); print fact5;
[
    <y + 3, 1>,
    <y^2 + 2*y + 3, 1>,
    <y^2 + 2*y + 4, 1>
]
> s := [t[1]: t in fact5]; print s;
[
    y + 3,
    y^2 + 2*y + 3,
    y^2 + 2*y + 4
]
> print &*s eq P5!f;
true
```

Now the Hensel lifting can take place:

```
> h625 := HenselLift(f, s, P625); print h625;
[
    z + 133,
    z^2 + 12*z + 618,
    z^2 + 492*z + 174
]
> print &*h625 eq P625!f;
true
```

The object of Hensel lifting is generally to find an integer polynomial factorization. If $\frac{p^k}{2} \geq 2^n \sqrt{\sum_{i=0}^{n} f_i^2}$, where n is the degree of f, then the lifted sequence should correspond fairly directly to an integer factorization of f. This is the case for the lifted sequence in the current example:

```
> print 625/2 ge
>    (2^n * Sqrt(&+[Coefficient(f, i): i in [0..n]])
>    where n is Degree(f));
true
```

There is a slight complication, however, because when MAGMA coerces elements of residue rings modulo b into the integers, it always returns non-negative integers. Here what is wanted is the symmetric form, the representation ranging from $-\lfloor \frac{b}{2} \rfloor$ to $\lfloor \frac{b}{2} \rfloor$. The function defined below will do the task. Given an integer polynomial f, it returns the corresponding integer polynomial with the coefficients in 'symmetric form' modulo b,

that is, congruent modulo b to the original coefficients and in the range $\{-(b \text{ div } 2) \text{ .. } +(b \text{ div } 2)\}$:

```
> SymmetricForm := func< f, b |
>     Parent(f) ! [ cmod gt b div 2 select cmod-b else cmod
>       where cmod is c mod b : c in Coefficients(f) ] >;

> ipoly := [SymmetricForm(P!p, 625): p in h625];
> print ipoly;
[
    x + 133,
    x^2 + 12*x - 7,
    x^2 - 133*x + 174
]
```

Now each factor of f is either a polynomial in *ipoly*, or a product of some of these polynomials (with *SymmetricForm* applied to the result):

```
> print [j: j in [1..3] | IsZero(f mod ipoly[j])];
[ 2 ]
> // so ipoly[2] is a factor
> pr := SymmetricForm(ipoly[1] * ipoly[3], 625);
> print f mod pr;
0
> print f eq ipoly[2] * pr;
true
> print ipoly[2], pr;
x^2 + 12*x - 7
x^3 - 15*x + 17
```

Hence the factorization of f over the integers is $(x^2+12x-7)(x^3-15x+17)$.

21.6.3 Roots of Univariate Polynomials

Let f be a univariate polynomial over the ring R, where R may be $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, GF(q) or $\mathbb{Z}/p\mathbb{Z}$. A *root* of f is an object r such that when r is substituted for the indeterminate of f, the result is zero; unless otherwise stated, it is understood that r is an element of R. The function **Roots**(f) returns all the roots of f in R, in a format similar to a factorization sequence, that is, as a sequence of tuples $< r_i, m_i >$ where r_i is a root and m_i is its multiplicity. (There is a slight exception: if f is over the free real field, then the return value may include complex roots.) For example, let f be the polynomial shown below. Its roots in $\mathbb{Q}$ are 3 (with multiplicity 2) and $-\frac{7}{2}$ (with multiplicity 1):

```
> PQ<x> := PolynomialRing(RationalField());
> f := (x - 3)^2*(x^2 + 9)*(x + 7/2);
> print Roots(f);
[ <3, 2>, <-7/2, 1> ]
```

It is also possible to find the roots of f lying in another ring S, if the coefficients of f can be coerced automatically into S (e.g., if R is a subring of S), by means of the function **Roots**(f, S). Continuing the example:

```
> print Roots(f, ComplexField());
[ <-3.49999999999999999999999999999999999917, 1>,
< 0.E-32 - 3.00000000000000000000000000000000000000*i, 1>,
< 0.E-32 + 3.00000000000000000000000000000000000000*i, 1>,
<3.00000000000000000000000000000000000000, 2> ]
```

The output above indicates that the roots of f lying in the complex field are -3.5(multiplicity 1), $-3i$ (multiplicity 1), $3i$ (multiplicity 1), and 3 (multiplicity 2). As another example, let PK be the ring of univariate polynomials in y over $K = \mathrm{GF}(7)$, and compare the roots of $h = y^9 - 1$ over K with those over the splitting field of h:

```
> PK<y> := PolynomialRing(GF(7));
> h := y^9 - 1;
> print Roots(h);
[ <4, 1>, <2, 1>, <1, 1> ]
> spl<w> := SplittingField(h);
> print Roots(h, spl);
[ <w^190, 1>, <4, 1>, <w^266, 1>, <w^304, 1>,
<1, 1>, <w^38, 1>, <w^76, 1>, <2, 1>, <w^152, 1> ]
```

If the coefficient ring of the polynomial f is the free real/complex field, it is recommended that the coefficients of f be given as exact values (integers and rationals) if possible. This will minimize the computational problems associated with floating point numbers. For example, it will increase the likelihood that repeated roots are identified as such by the algorithm:

```
> C<i> := ComplexField();
> PC<z> := PolynomialRing(C);
> print Roots((z - 6/5)^4);
[ <1.19999999999999999999999999999999999964, 4> ]
> print Roots((z - 1.2)^4);
[ <1.20000009287311645800501932758701650792l4, 1>,
<1.19999999999999702652624469204344419090l6 +
0.00000009287311348453096781806583537l8*i, 1>,
<1.19999999999999702652624469204344419090l6 -
```

```
0.00000009287311348453096781806583537 8*i, 1>,
<1.19999990712688948894249128826551059274 45, 1> ]
```

Furthermore, if f is over the free real/complex field, then the user may specify the algorithm to be used, by means of the parameter **Al**. The default value of **Al** is the string `"Schonhage"`, which gives Xavier Gourdon's implementation of Schönhage's algorithm. Other possibilities for **Al** are `"Laguerre"`, `"NewtonRaphson"`, and `"Combination"` (a combination of the Laguerre and Newton-Raphson algorithms. The function **Roots** has a second parameter, **Digits**, which is used to specify the desired accuracy d for the calculation; the default value of d is the current precision of the free real field. When Schönhage's algorithm is used, the results will be correct to within an absolute error of 10^{-d}; for the other algorithms, an attempt will be made to find the roots to an accuracy of d significant figures, but this is not guaranteed.

If the coefficients of f are non-exact elements of the free real/complex field, then it may be preferable to use an alternative function, **RootsNonExact**(f), which is specially designed for this situation. It has a parameter **Digits** (default value being the current precision of the free real field) whose value d is used to specify the accuracy required. **RootsNonExact** returns two sequences $[v_1, \ldots, v_n]$ and $e_1, \ldots, e_n]$, each containing elements of the free complex field. (Note that the multiplicities of the roots are not provided.) The v_i are approximations to the roots of f, such that $|f - a(z - v_1) \ldots (z - v_n)| < 10^{-d}|f|$, where a is the leading coefficient of f. The e_i provide error bounds, having the property that if $\hat{f}$ is any polynomial such that $|f - \hat{f}| < 10^{-d}|f|$, then one can write $\hat{f} = a(z - u_1) \ldots (z - u_n)$ with $|v_i - u_i| < e_i$. In some cases, the second sequence will not be returned, because d is too small (for the given polynomial) for such error bounds to be derived. Increasing d will decrease the errors on the 'roots', but for meaningful answers any non-exact coefficients of f must be given with at least d decimal places. For example:

```
> rts, errs := RootsNonExact((z - 6/5)^4);
> print rts;
[ 1.19999999999999999999999999984,
1.19999999999999999999999999984,
1.20000000000000000000000000000000000023 -
2.028356533497 E-19*i,
1.20000000000000000000000000000000000023 +
2.028356533497 E-19*i ]
> print errs;
[ 0.0000005722045898, 0.0000005722045898,
0.0000005722045898, 0.0000005722045898 ]
```

22. Multivariate Polynomial Rings

Multivariate polynomial rings and their ideals deal with the subject of Commutative Algebra which has a much larger theory than that associated with univariate polynomial rings and their ideals. Consequently, the two types of polynomial rings are implemented with different categories and although they share many simple operations, the major algorithms are quite distinct. See Chapter 21 for more discussion of the differences between univariate and multivariate polynomial rings.

Multivariate polynomials themselves are quite different from univariate polynomials in that they are linearly ordered sums of coefficient-monomial pairs. Furthermore, the orderings on the monomials within the polynomials play an extremely significant role. Also central to computation with multivariate polynomial rings is the concept of the *Gröbner basis*, which is a canonical generating set of an ideal. The *Buchberger Algorithm* allows the computation of the Gröbner basis of any ideal.

The category of multivariate polynomial rings is **RngMPol**.

22.1 Polynomial Rings and Monomial Orders

Let P be the multivariate polynomial ring $R[x_1, \ldots, x_n]$ of rank n over the coefficient ring R. A *monomial* (or *power product*) of P is a product of powers of the variables of P; that is, an expression of the form $x_1^{e_1} \cdots x_n^{e_n}$ with $e_i \geq 0$ for $1 \leq i \leq n$. A *term* of P is a product cm where c is a coefficient from R and m is a monomial of P. Let M be the set of all monomials of P. A *monomial ordering* on M is a total order $<$ on M such that $1 \leq s$ for all $s \in M$, $s \leq t$ implies $su \leq tu$ for all $s, t, u \in M$, and M is a well-ordering (every non-empty subset of M possesses a minimal element with respect to $<$). Within MAGMA, each multivariate polynomial ring P has a fixed monomial order $<$ associated with it.

The main function to create a multivariate polynomial ring in MAGMA is **PolynomialRing**(R, n, S) which takes the coefficient ring R, the rank n, and

the string S specifying the monomial order $<$. These arguments are optionally followed by extra arguments associated with the particular monomial order. Two of the most important monomial orders are the *lexicographical* order and the *graded-reverse-lexicographical* order.

The lexicographical order $<$ (abbreviated `lex`) is defined for monomials s and t as follows: $s < t$ iff there exists $1 \leq i \leq n$ such that the first $i-1$ exponents of s and t are equal but the i-th exponent of s is less than the i-th exponent of t. The order is called 'lexicographical' since it orders the monomials as if they were words in a dictionary. The i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$ so the first variable is the greatest variable. This order is specified by the string `"lex"` as the third argument to the **PolynomialRing** function.

The graded-reverse-lexicographical order $<$ (abbreviated `grevlex`) is defined for monomials s and t as follows: $s < t$ iff the total degree of s is less than the total degree of t or the total degree of s is equal to the total degree of t and $s > t$ with respect to the lexicographical order applied to the exponents of s and t in reverse order. The order is called 'graded reverse lexicographical' since it first grades the monomials by total degree, and then decides ties by the negation of the lexicographical order applied to the variables in reverse order. Again, the i-th variable is greater than the $(i+1)$-th variable for $1 \leq i < n$, so the first variable is the greatest variable. This order is specified by the string `"grevlex"` as the third argument to the **PolynomialRing** function.

Other monomial orders available in MAGMA are the following: graded lexicographical (`glex`), elimination (`elim`), inverse block (`invblock`), univariate (`univ`) and weight (`weight`). See the *Handbook* for details.

A polynomial f of P consists of a linearly ordered sum (internally, a linked list) of terms (i.e., coefficient-monomial pairs) such that the monomials of f are distinct and are sorted with respect to the specific monomial order with the greatest monomial is first.

The following example constructs the polynomial ring of rank 3 over the integer ring in two different ways, so as to illustrate the differences between the `lex` and `grevlex` order. Note that generator assignment may be used to gives names to the indeterminates:

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3, "lex");
> print x + y + z;
x + y + z
> print x + y^20 + z^10*x^9;
x^9*z^10 + x + y^20
> print (x+y+z)^4;
x^4 + 4*x^3*y + 4*x^3*z + 6*x^2*y^2 + 12*x^2*y*z + 6*x^2*z^2 +
```

```
    4*x*y^3 + 12*x*y^2*z + 12*x*y*z^2 + 4*x*z^3 + y^4 +
    4*y^3*z + 6*y^2*z^2 + 4*y*z^3 + z^4
> P<x, y, z> := PolynomialRing(IntegerRing(), 3, "grevlex");
> print x + y + z;
x + y + z
> print x + y^20 + z^10*x^9;
y^20 + x^9*z^10 + x
> print (x+y+z)^4;
x^4 + 4*x^3*y + 6*x^2*y^2 + 4*x*y^3 + y^4 + 4*x^3*z +
    12*x^2*y*z + 12*x*y^2*z + 4*y^3*z + 6*x^2*z^2 +
    12*x*y*z^2 + 6*y^2*z^2 + 4*x*z^3 + 4*y*z^3 + z^4
```

Since the lexicographical order is so important and is usually the order desired, it is the one used if the order argument is omitted. Thus the statements

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3, "lex");
```

and

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3)
```

have exactly the same effect.

Note that a multivariate polynomial ring of rank 1 over the coefficient ring R is *not* the same as the univariate polynomial ring over R. The operations applicable to the former are those discussed in this chapter while those applicable to the latter are discussed in Chapter 21. Usually, if one wishes to compute with univariate polynomials then the univariate polynomial ring should be used (particularly since there are many functions applicable to univariate polynomials which would be useful), and the multivariate polynomial ring of rank 1 should only be used when it occurs, say, as the base case of a recursive algorithm which computes with multivariate polynomial rings.

22.2 Polynomial Creation and Access

The only way to create a multivariate polynomial in MAGMA is to use an expression in the indeterminates, or to coerce an element of the coefficient ring into the polynomial ring to create a constant polynomial.

For example, the following lines create the element $c = 4s^3 + 20s^2t + 19st^4 + 8t + 7$ in the polynomial ring $P = \mathbb{Q}[s, t]$:

```
> P<s, t> := PolynomialRing(RationalField(), 2);
```

```
> c := 20*s^2*t + 4*s^3 - 12*s*t^4 + 8*t - 24;
> print c;
4*s^3 + 20*s^2*t + 19*s*t^4 + 8*t + 7
```

MAGMA provides a large number of functions for accessing multivariate polynomials. These functions are quite different from those applicable to univariate polynomials because the multivariate polynomials have quite a different structure as linearly ordered sums.

Since a multivariate polynomial f is just a linearly ordered sum of coefficient-monomial pairs, f can be viewed in an *absolute* manner as two parallel lists: the list of *base* coefficients (from the coefficient ring) and the list of monomials. Note that in this viewpoint, the sequence of base coefficients on its own is useless since it only corresponds to the sequence of monomials. Table 22.1 lists the functions for accessing polynomials in the absolute manner.

Table 22.1. Absolute access functions for multivariate polynomials

MAGMA	Meaning
Coefficients(f)	Coefficients of f as a sequence
LeadingCoefficient(f)	Leading coefficient of f
TrailingCoefficient(f)	Trailing coefficient of f
MonomialCoefficient(f, m)	Coefficient of monomial m in f
Monomials(f)	Monomials of f as a sequence
LeadingMonomial(f)	Leading monomial of f
Terms(f)	Terms of f as a sequence
LeadingTerm(f)	Leading term of f
TrailingTerm(f)	Trailing term of f
TotalDegree(f)	Total degree of f (maximum of total degrees of monomials of f)
LeadingTotalDegree(f)	Total degree of leading monomial of f

For a fixed variable v, a multivariate polynomial f may be viewed *recursively* as a univariate polynomial with respect to variable v. That is, f may be considered as a linearly ordered sum whose terms are of the form $c_i v^i$ where c_i is the coefficient of the power v^i, which is still in general a polynomial (not containing v though, of course). MAGMA provides functions to access f considered in this way. For each function the variable v can be specified in two ways: as the integer $i \in \{1 \dots n\}$ (where n is the rank of the ring) which is the variable number which corresponds to v, or as the variable v itself (lying in the polynomial ring). Table 22.2 lists the functions for accessing polynomials

in the recursive manner. The functions are only listed in the form in which the variable number i is given; for each function there is also a version which takes the actual variable v instead of the number i.

Table 22.2. Recursive access functions for multivariate polynomials

MAGMA	Meaning
Coefficients(f, i)	(Recursive) coefficients of f w.r.t. variable number i as a sequence
Coefficient(f, i, k)	k-th (recursive) coefficient of f w.r.t. variable number i as a polynomial
LeadingCoefficient(f, i)	(Recursive) leading coefficient of f w.r.t. variable number i
TrailingCoefficient(f, i)	(Recursive) trailing coefficient of f w.r.t. variable number i
Terms(f, i)	(Recursive) terms of f w.r.t. variable number i as a sequence
Term(f, i, k)	(Recursive) k-th term of f w.r.t. variable number i
LeadingTerm(f, i, k)	(Recursive) leading term of f w.r.t. variable number i
TrailingTerm(f, i, k)	(Recursive) leading term of f w.r.t. variable number i
Degree(f, i)	(Recursive) degree of f w.r.t. variable number i

The following example demonstrates the differences between the absolute and recursive access functions for a polynomial ring with `lex` monomial ordering:

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> f := 2*x^3*y*z + 4*y*z^3 + 8*z + 3;
> print f;
2*x^3*y*z + 4*y*z^3 + 8*z + 3
> print Coefficients(f);
[ 2, 4, 8, 3 ]
> print Monomials(f);
[
    x^3*y*z,
    y*z^3,
    z,
    1
]
> print Terms(f);
[
```

```
    2*x^3*y*z,
    4*y*z^3,
    8*z,
    3
]
> print TotalDegree(f);
5
> print LeadingMonomial(f);
x^3*y*z
> print Degree(f, x);
3
> print Coefficients(f, x);
[
    4*y*z^3 + 8*z + 3,
    0,
    0,
    2*y*z
]
> print Coefficient(f, x, 0);
4*y*z^3 + 8*z + 3
```

22.3 Factorization, Resultants and Derivatives

MAGMA incorporates powerful algorithms for factoring multivariate polynomials over various coefficient rings. Currently, the available coefficient rings are: the integer ring $\mathbb{Z}$; the rational field $\mathbb{Q}$; finite fields; cyclotomic, quadratic, and general number fields; and rational function fields or polynomial rings over any of the above.

The factorization algorithms for multivariate polynomials are generally much harder than the corresponding ones for univariate polynomials. Indeed, the easiest coefficient rings for univariate polynomials are the finite fields, while for multivariate polynomials the finite fields become the most difficult coefficient rings!

The function **Factorization**(f) factorizes the polynomial f into powers of irreducible polynomials. The factorization is actually computed for the canonical associate f' of f and the unit u such that $f = uf'$ is also returned. For a polynomial f with coefficient ring a field, f' is the monic polynomial associated with f while for a polynomial f with coefficient ring $\mathbb{Z}$, f' is the associate of f with positive leading coefficient. For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
```

```
> f := 5 * (x^2 + y + z)*(x + y^2 + z)*(x + y + z^2)^2;
> print f;
5*x^5 + 5*x^4*y^2 + 10*x^4*y + 10*x^4*z^2 + 5*x^4*z +
    10*x^3*y^3 + 10*x^3*y^2*z^2 + 5*x^3*y^2 + 10*x^3*y*z^2 +
    10*x^3*y*z + 5*x^3*y + 5*x^3*z^4 + 10*x^3*z^3 + 5*x^3*z
    + 5*x^2*y^4 + 10*x^2*y^3*z^2 + 5*x^2*y^3 + 5*x^2*y^2*z^4
    + 10*x^2*y^2*z + 10*x^2*y^2 + 10*x^2*y*z^3 +
    10*x^2*y*z^2 + 15*x^2*y*z + 5*x^2*z^5 + 10*x^2*z^3 +
    5*x^2*z^2 + 10*x*y^4 + 10*x*y^3*z^2 + 10*x*y^3*z +
    5*x*y^3 + 10*x*y^2*z^3 + 10*x*y^2*z^2 + 15*x*y^2*z +
    5*x*y*z^4 + 20*x*y*z^3 + 10*x*y*z^2 + 5*x*z^5 + 10*x*z^4
    + 5*y^5 + 10*y^4*z^2 + 5*y^4*z + 5*y^3*z^4 + 10*y^3*z^3
    + 5*y^3*z + 5*y^2*z^5 + 10*y^2*z^3 + 5*y^2*z^2 + 5*y*z^5
    + 10*y*z^4 + 5*z^6
> F, u := Factorization(f);
> print F, u;
[
    <x + y + z^2, 2>,
    <x + y^2 + z, 1>,
    <x^2 + y + z, 1>
]
5
```

The function **IsIrreducible**(f) returns **true** if and only if the polynomial f is irreducible. Another factorization function is **SquareFreeFactorization**(f). It returns the square-free factorization (with appropriate unit) of the canonical associate f' of f which is a factorization sequence of square-free (but not necessarily irreducible) factors with their multiplicities. None of the factors will contain the square of any non-constant polynomial. For example:

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> f := (x + y + z)^2 * (x^2 - y^2)^3;
> print f;
x^8 + 2*x^7*y + 2*x^7*z - 2*x^6*y^2 + 2*x^6*y*z + x^6*z^2 -
    6*x^5*y^3 - 6*x^5*y^2*z - 6*x^4*y^3*z - 3*x^4*y^2*z^2 +
    6*x^3*y^5 + 6*x^3*y^4*z + 2*x^2*y^6 + 6*x^2*y^5*z +
    3*x^2*y^4*z^2 - 2*x*y^7 - 2*x*y^6*z - y^8 - 2*y^7*z -
    y^6*z^2
> print SquareFreeFactorization(f);
[
    <x + y + z, 2>,
    <x^2 - y^2, 3>
]
> print Factorization(f);
[
```

```
    <x - y, 3>,
    <x + y, 3>,
    <x + y + z, 2>
]
```

The function **Resultant**(f, g, i) returns the resultant of polynomials f and g with respect to the i-th variable while the function **Resultant**(f, g, v) returns the resultant of polynomials f and g with respect to variable v. Similarly, the functions **Discriminant**(f, i) and **Discriminant**(f, v) compute the discriminant of f with respect to the appropriate variable. For example, the following statements compute the resultant of two bivariate polynomials with respect to one variable. Note that the resultant can also be computed in this case by an elimination ideal computation (see Section 22.8 below).

```
> P<x,y> := PolynomialRing(RationalField(), 2);
> f := x^5 + y^2 + 1;
> g := y^3 + x + 3;
> print Resultant(f, g, y);
x^15 + 3*x^10 + 3*x^5 + x^2 + 6*x + 10
> print I := ideal<P | f, g>;
> I;
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Basis:
[
    x^5 + y^2 + 1,
    x + y^3 + 3
]
> print EliminationIdeal(I, {x});
Ideal of Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: x, y
Basis:
[
    x^15 + 3*x^10 + 3*x^5 + x^2 + 6*x + 10
]
```

The functions **Derivative**(f, i) and **Derivative**(f, v) compute the formal derivative of f with respect to the appropriate variable. Similarly, the functions **Integral**(f, i) and **Integral**(f, v) compute the formal integral of f with respect to the appropriate variable.

22.4 Multivariate Function Fields

A *multivariate rational function field* F of rank n over the ring R is the field of fractions of a multivariate polynomial ring P of rank n over the ring R. That is, it consists of fractions whose numerator and denominator are polynomials over R with n indeterminates. The polynomial ring P is called the *ring of integers* of F. There are two ways to create the field F. Given P, **FieldOfFractions**(P) returns F. Alternatively, **FunctionField**(R, n) returns F, using the lexicographical monomial order for P. The function **IntegerRing**(F) returns P.

As in the univariate case, if $t = \frac{n}{d}$ is an element of a multivariate function field, where a and b are expressed in lowest terms, then **Numerator**(t) and **Denominator**(t) return n and d as elements of the corresponding ring of integers (i.e., the polynomial ring). These functions behave similarly for elements of the rational field.

It is often useful to give the same indeterminate names to the polynomial ring and then to the function field. For example:

```
> P6<a,b,c,d,e,f> := PolynomialRing(RationalField(), 6);
> F6<a,b,c,d,e,f> := FieldOfFractions(P6);
```

The effect of this is subtle. After the two generator assignments, the printnames for both magmas are a, b, c, d, e, f, so that the natural identification can be made in the output. However, the second and final time that the identifiers a, b, c, d, e, f are assign ed, they are given the indeterminates of *F6* as their values, so that in subsequent expressions they will refer to these elements. For instance:

```
> t := 14*a + c;
> print t;
14*a + c
> print Parent(t);
Rational function field of rank 6 over Rational Field
Variables: a, b, c, d, e, f
> num := Numerator(t); print num;
14*a + c
> print Parent(num);
Polynomial ring of rank 6 over Rational Field
Lexicographical Order
Variables: a, b, c, d, e, f
```

Rational functions in several indeterminates may be used for performing algebraic manipulations. For instance, one way of solving the system of

simultaneous equations

$$\begin{array}{rcrcl} ax & + & by & = & e \\ cx & + & dy & = & f \end{array}$$

which can be written in matrix form as

$$\begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} a & c \\ b & d \end{pmatrix} = \begin{pmatrix} e & f \end{pmatrix}$$

is to calculate the inverse of the coefficient matrix and multiply the vector of constants by this inverse. This can always be done using the function field since the determinant $ad - bc$ is a non-zero element of the field, so the inverse exists. If in a particular assignment of the variables a, b, c, d, e, f this determinant remains non-zero, then the solution derived from the function field remains valid. In MAGMA, all of these operations can be done in terms of the function field *F6* defined above:

```
> coeffmat := MatrixRing(F6, 2)![a, c, b, d];
> print coeffmat;
[a c]
[b d]
> rhs := VectorSpace(F6, 2)![e, f];
> print rhs;
(e f)
> print rhs * coeffmat^-1;
((-b*f + d*e)/(a*d - b*c)  (a*f - c*e)/(a*d - b*c))
```

The output equals the vector $\begin{pmatrix} x & y \end{pmatrix}$. Therefore the solution is

$$x = \frac{-bf + de}{ad - bc}, \quad y = \frac{af - ce}{ad - bc},$$

assuming the determinant $ad - bc$ is non-zero. (If the determinant were zero, the solution could be obtained easily by another method.) This example can of course be generalized to much larger examples, and also to non-linear examples, as shown in the following sections.

22.5 Ideals

In this section, assume that P is a polynomial ring in n variables over the *field* K.

For an ideal I of the polynomial ring P, a *basis* B of I is an (ideal) generating set of I, i.e., a subset of I such that any element of I can be expressed as a finite sum of products of the form fg where $f \in P$ and $g \in B$. The Hilbert Basis Theorem states that every ideal I possesses a finite basis. Within MAGMA, a basis of an ideal I is in fact an ordered sequence of elements of I, possibly with repetitions. Thus a basis of an ideal is different from, say, a vector space basis since it may contain repetitions and zero elements.

The most simple but important problem associated with an ideal is the membership test: given a polynomial $f \in P$ and an ideal I of P, is $f \in I$? To answer this question, a multivariate *division algorithm* will be developed.

Suppose B is a basis of an ideal I of P and $f \in P$. If f is non-zero, f is said to be *top-reducible* with respect to B if there exists a $g \in B$ such that the leading monomial (greatest monomial with respect to the monomial ordering) of g divides the leading monomial of f. If such a case holds, then f can be reduced by g by subtracting the appropriate multiple of g from f which cancels the leading term of f. It is easy to show from the properties of the monomial order that the leading monomial of the new polynomial f_1 (if non-zero) is strictly less than that of f. If the new polynomial f_1 is non-zero yet also top-reducible with respect to B, the same procedure can be applied to it to yield f_2, and so on until an f_k is reached which is either the zero polynomial or is non-zero but is not top-reducible with respect to B. The process does terminate because a monomial order is a well-ordering. The polynomial f_k is called a *normal form* of f with respect to the basis B. (If f is already zero, then f is a (in fact, the only) normal form of f.)

If a normal form of a polynomial $f \in P$ with respect to the basis B is zero, then it is easy to see that $f \in I$. However, the converse is in general false; i.e., it is possible that a polynomial f is in the ideal I yet a normal form of f with respect to B is non-zero. As an example, let P be the polynomial ring $\mathbb{Q}[x, y, z]$ with lexicographical order and $x > y > z$, let B be the basis $\{x + y, x + z\}$, and let I be the ideal generated by B. Then the polynomial $f = y - z$ is obviously in I yet a normal form of it with respect to B is still f since the leading monomials x and x respectively of the polynomials of B do not divide the leading monomial y of f.

The notion of Gröbner basis was introduced to remedy this problem: a basis B of a polynomial ideal I of P is called a *Gröbner basis* if, for all $f \in P$, $f \in I$ if and only if every normal form of f with respect to B is zero. The *Buchberger Algorithm* takes an arbitrary basis B of an ideal I and constructs a Gröbner basis for I. This algorithm is of central importance to all computations with ideals of multivariate polynomial rings. A Gröbner basis for an ideal with respect to a fixed monomial order can also be put into a unique form, just like the unique reduced-row echelon form of a matrix. (The basis is *minimalized*, *reduced* and *sorted*—see [CLO92], Chapter 2, §7

for details.) This is always done in MAGMA so that every ideal I possesses a unique Gröbner basis since it has a fixed monomial order.

Table 22.3. Ideal creation and basis functions

MAGMA	Meaning
ideal< P | L **>**	Ideal of P generated by polynomials in L
ideal(Q)	Ideal with user basis Q
Basis(I)	Current basis of I
BasisElement(I, i)	i-th element of the basis of I
Coordinates(I, f)	Coordinates of $f \in I$ with respect to the basis of I
Groebner(I)	Procedure explicitly forcing a Gröbner basis to be constructed for I
GroebnerBasis(I)	(Unique) Gröbner basis of ideal I
GroebnerBasis(S)	(Unique) Gröbner basis of the ideal generated by set or sequence S

MAGMA incorporates an efficient implementation of the Buchberger Algorithm which is automatically invoked when necessary. The usual way to create an ideal is by the **ideal** constructor which is given the polynomial ring P (or an over-ideal) on the left side, and a list on the right side which describes a basis for an ideal. MAGMA remembers the original basis as constructed as the current basis for the ideal until it needs a Gröbner basis: at that point it changes the current basis to the unique Gröbner basis of that ideal and discards the original basis. To force MAGMA to explicitly compute the Gröbner basis of an ideal I, the procedure **Groebner**(I) may be invoked. Note that it is never actually necessary to do this for any ideal computations – MAGMA will automatically compute the Gröbner basis if it needs to do so.

For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x + y, x + z>;
> print I;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Basis:
[
    x + y,
    x + z
]
> print (y-z) in I;
true
```

```
> print I; // NB: Groebner basis of I is now known
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Groebner basis:
[
    x + z,
    y - z
]
```

Table 22.3 lists the functions for creating ideals and accessing their bases. Note that it is possible by the **Ideal**(Q) function to create an ideal with a basis such that the basis is remembered by MAGMA so that polynomials of the ideal can be written in terms of this basis by the **Coordinates**(I, f) function—see the *Handbook* for details.

Verbose output for the Buchberger Algorithm (invoked, for example, directly by the **Groebner** procedure) can be obtained by the procedure call **SetVerbose**(`"Groebner"`, 1).

MAGMA incorporates an implementation of the exciting new *Gröbner Walk* algorithm for changing the Gröbner basis of an ideal with respect to one monomial order to the Gröbner basis of the ideal with respect to another monomial order (see [CKM96]). This gives a very powerful method in MAGMA for computing the Gröbner basis of the fixed monomial order of an ideal: first the Gröbner basis of the ideal is computed with respect to an easy order (usually the `grevlex` order) by the plain Buchberger Algorithm, and then the Gröbner Walk algorithm is invoked to convert this Gröbner basis to the Gröbner basis with respect to the desired final order. Generally this method is much faster than by using the plain Buchberger Algorithm directly with the final desired order. The method is automatically employed within MAGMA whenever a Gröbner basis is needed. The **ChangeOrder** function allows one to manually apply the Gröbner Walk algorithm as desired—see the *Handbook* for details.

Occasionally the above method using the Gröbner Walk can be slower than the method of using the plain Buchberger Algorithm directly with the final desired order. To circumvent this problem, the parameter `Walk` for the **Groebner** function can be used to control the method. Thus if I is an ideal, then the procedure call **Groebner**(`I : Walk := false`) will construct the Gröbner basis for the ideal I without using the Gröbner Walk algorithm and will just use the plain Buchberger Algorithm directly with the final desired order.

Verbose output for the Gröbner Walk algorithm can be obtained by the procedure call **SetVerbose**(`"GroebnerWalk"`, 1). The verbose output indi-

cates each of the steps followed by the algorithm. The verbose output for the Buchberger algorithm is automatically turned off within the Gröbner Walk algorithm since it is called frequently with relatively trivial input. Setting the verbose level to be 2 enables the verbose printing for the Buchberger algorithm as well.

22.6 Ideal Access and Arithmetic

Since ideals of multivariate polynomials are general ideals (and rings) in themselves, many common ideal operations can be performed on them. Table 22.4 lists the basic access and arithmetic functions for multivariate polynomial ideals. The table also lists some operations which can be done on polynomials with respect to an ideal.

An example to demonstrate the ideal arithmetic functions:

```
> P<x, y, z> := PolynomialRing(GF(2), 3);
> I := ideal<P | x^2 + y^2, (y + z)^3 - 1>;
> J := ideal<P | x^2 + y^2, y + z>;
> M := I meet J;
> print M;
Ideal of Polynomial ring of rank 3 over GF(2)
Lexicographical Order
Variables: x, y, z
Basis:
[
    y^4 + y + z^4 + z,
    x^2 + y^2
]
> print M subset I;
true
> print M subset J;
true
> print I * J eq M;
true
> print IsInRadical(x + y, I);
true
> print (x + y) in I;
false
> print (x + y)^2 in I;
true
> print ColonIdeal(J, ideal<P|x+y>);
Ideal of Polynomial ring of rank 3 over GF(2)
```

```
Lexicographical Order
Variables: x, y, z
Basis:
[
    x + y,
    y + z
]
```

Table 22.4. Ideal functions

MAGMA	Meaning
$I + J$	Sum of ideals I and J
$I*J$	Product of ideals I and J
$I\hat{}k$	k-th power of ideal I
ColonIdeal(I, J)	Colon ideal or ideal quotient $I : J$
ColonIdeal(I, f)	Colon ideal or ideal quotient $I : f^\infty$
I **meet** J	Intersection of ideals I and J
&meet S	Intersection of set/sequence S of ideals
Generic(I)	Generic polynomial of which I is an ideal
I **eq** J	True iff ideals I and J are equal
I **subset** J	True iff ideal I is contained in ideal J
IsProper(I)	True iff ideal I is a proper ideal
IsZero(I)	True iff ideal I is the zero ideal
f **in** I	True iff polynomial f is in ideal I
IsInRadical(f, I)	True iff polynomial f is in the radical of ideal I
NormalForm(f, I)	Normal form of polynomial f with respect to ideal I
SPolynomial(f, g)	S-polynomial of polynomials f and g

22.7 Quotient Rings

For an ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, it is possible to form the *quotient ring* $Q = P/I$. This can be done either by the / function or by the **quo** constructor. The new indeterminates of the quotient ring can be named by the angle bracket notation as usual.

For example, the following construction of a quotient ring Q:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> Q<a, b, c> := quo<P | x^3 + y + z, x + y^3 + z,
>                       x + y + z^3>;
```

is completely equivalent to the construction:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x^3 + y + z, x + y^3 + z, x + y + z^3>;
> Q<a, b, c> := P / I;
```

The quotient ring Q consists of the residues (normal forms) of the elements of P with respect to the ideal I. Obviously Q has the structure of a vector space over the field K, just like P has.

An extremely important case arises when the dimension of Q as a vector space over K is finite. In such a case, MAGMA provides the function **VectorSpace**(Q) to create a vector space V together with an isomorphism f from Q onto V. This is useful for investigating further structural properties of Q. Table 22.5 lists the functions for computing with quotient rings.

Table 22.5. Quotient ring functions

MAGMA	Meaning
CoefficientRing(Q)	Coefficient ring of quotient ring Q
Rank(Q)	Rank of quotient ring Q
$Q.i$	i-th indeterminate of quotient ring Q
Dimension(Q)	Dimension of quotient ring as vector space over K
VectorSpace(Q)	Vector space isomorphic to Q together with isomorphism
MatrixAlgebra(Q)	Matrix algebra isomorphic to Q together with isomorphism

Continuing the above example:

```
> print Dimension(Q);
27
> V, f := VectorSpace(Q);
> // Print vector space basis of Q:
> print [V.i@@f: i in [1..27]];
[ 1, c, c^2, c^3, c^4, c^5, c^6, c^7, c^8, c^9, c^10, c^11,
c^12, c^13, c^14, c^15, c^16, b, b*c, b*c^2, b*c^3, b*c^4,
    b*c^5, b*c^6, b*c^7, b*c^8, b^2 ]
```

As well as the functions applicable for standard ring elements, the functions **RepresentationMatrix**(f), **MinimalPolynomial**(f), **IsUnit**(f), and **IsNilpotent**(f) may also be applied to elements of finite-dimensional quotient rings—see the *Handbook* for details.

For example, suppose the user wishes to find the minimal polynomial of $\theta = \sqrt{2}+\sqrt[3]{5}$ over $\mathbb{Q}$. To do this, it suffices to calculate the minimal polynomial of (the residue class of) $x+y$ over $\mathbb{Q}$ in the quotient ring $Q[x,y]/(x^2-2, y^3-5)$:

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> Q<a, b> := quo<P | x^2 - 2, y^3 - 5>;
> UP<t> := PolynomialRing(RationalField());
> print MinimalPolynomial(a + b);
t^6 - 6*t^4 - 10*t^3 + 12*t^2 - 60*t + 17
```

A further example to demonstrate the basic arithmetic in quotient rings when the quotient ring does not have finite dimension:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x^3 + y + z, x + y^3 + z>;
> Groebner(I);
> print I;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Groebner basis:
[
    x + y^3 + z,
    y^9 + 3*y^6*z + 3*y^3*z^2 - y + z^3 - z
]
> Q<a, b, c> := P / I;
> print a;
-b^3 - c
> print b^8;
b^8
> print b^9;
-3*b^6*c - 3*b^3*c^2 + b - c^3 + c
> print c;
c
> print c^100;
c^100
```

22.8 Varieties and Elimination

For an ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, the *variety* of I is defined to be the subset V of K^n consisting of all $(a_1, \ldots, a_n) \in K^n$ such that $f(a_1, \ldots, a_n) = 0$ for all $f \in I$. Thus V is the set of solutions of the simultaneous equations implied by the polynomials of I. It can be easily

shown that V is equal to the set of solutions of the simultaneous equations implied by the polynomials of any finite basis of I. In particular, a Gröbner basis of I may be used.

For example, consider the system of equations Runge-Kutta 2 from the paper [BGK86]. The coefficient field K is the rational function field $\mathbb{Q}(c2, c3)$, and the polynomial ring $K[c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43]$ has 11 variables with the lexicographical ordering on monomials. After creating the ideal I generated by these polynomials, a Gröbner basis of I contains a linear polynomial for each variable so there is exactly one solution to the system given by the polynomials in the Gröbner basis (i.e., the corresponding variety has size 1).

```
> K<c2,c3> := FunctionField(IntegerRing(), 2);
> P<c4,b4,b3,b2,b1,a21,a31,a32,a41,a42,a43> :=
>     PolynomialRing(K, 11);
> I := ideal<P |
>    b1 + b2 + b3 + b4 - 1,
>    b2*c2 + b3*c3 + b4*c4 - 1/2,
>    b2*c2^2 + b3*c3^2 + b4*c4^2 - 1/3,
>    b3*a32*c2 + b4*a42*c2 + b4*a43*c3 - 1/6,
>    b2*c2^3 + b3*c3^3 + b4*c4^3 - 1/4,
>    b3*c3*a32*c2 + b4*c4*a42*c2 + b4*c4*a43*c3 - 1/8,
>    b3*a32*c2^2 + b4*a42*c2^2 + b4*a43*c3^2 - 1/12,
>    b4*a43*a32*c2 - 1/24,
>    c2 - a21,
>    c3 - a31 - a32,
>    c4 - a41 - a42 - a43>;
> Groebner(I);
> print I;
Ideal of Polynomial ring of rank 11 over
    Rational function field of rank 2 over Integer Ring
    Variables: c2, c3
Lexicographical Order
Variables: c4, b4, b3, b2, b1, a21, a31, a32, a41, a42, a43
Dimension 0
Groebner basis:
[
    c4 - 1,
    b4 + (-6*c2*c3 + 4*c2 + 4*c3 - 3)/(12*c2*c3 - 12*c2 -
        12*c3 + 12),
    b3 + (2*c2 - 1)/(12*c2*c3^2 - 12*c2*c3 - 12*c3^3 +
        12*c3^2),
    b2 + (-2*c3 + 1)/(12*c2^3 - 12*c2^2*c3 - 12*c2^2 +
        12*c2*c3),
```

```
    b1 + (-6*c2*c3 + 2*c2 + 2*c3 - 1)/(12*c2*c3),
    a21 - c2,
    a31 + (-4*c2^2*c3 + 3*c2*c3 - c3^2)/(4*c2^2 - 2*c2),
    a32 + (-c2*c3 + c3^2)/(4*c2^2 - 2*c2),
    a41 + (-12*c2^2*c3^2 + 12*c2^2*c3 - 4*c2^2 + 12*c2*c3^2
        - 15*c2*c3 + 6*c2 - 4*c3^2 + 5*c3 - 2)/(12*c2^2*c3^2
        - 8*c2^2*c3 - 8*c2*c3^2 + 6*c2*c3),
    a42 + (-c2^2 + 4*c2*c3^2 - 5*c2*c3 + 3*c2 - 4*c3^2 +
        5*c3 - 2)/(12*c2^3*c3 - 8*c2^3 - 12*c2^2*c3^2 +
        6*c2^2 + 8*c2*c3^2 - 6*c2*c3),
    a43 + (-2*c2^2*c3 + 2*c2^2 + 3*c2*c3 - 3*c2 - c3 +
        1)/(6*c2^2*c3^2 - 4*c2^2*c3 - 6*c2*c3^3 + 3*c2*c3 +
        4*c3^3 - 3*c3^2)
]
```

This example works well because of a phenomenon now discussed further. Again given an ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, for an integer k with $0 \leq k < n$, the k-th elimination ideal I_k of I is defined to be $I \cap K[x_{k+1}, \ldots, x_n]$. Thus I_k consists of all polynomials of I which have the first k variables eliminated. The amazing property of Gröbner bases which allow us to compute the elimination ideals is the following: if B is a Gröbner basis of I with respect to the lexicographical order with $x_1 > \ldots > x_n$, then the set $B \cap K[x_{k+1}, \ldots, x_n]$ is a Gröbner basis of the k-th elimination ideal I_k. Thus the Gröbner basis 'triangularizes' the associated system just like the reduced-row echelon form of a matrix 'triangularizes' a linear system.

The function **EliminationIdeal**(I, k) returns the k-th elimination ideal of I. For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x^2 + y + z - 1, x + y^2 + z - 1,
>     x + y + z^2 - 1>;
> print EliminationIdeal(I, 1);
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Basis:
[
    2*y*z^2 + z^4 - z^2,
    y^2 - y - z^2 + z
]
> print (y^2 - y - z^2 + z) in I;
true
```

The function **EliminationIdeal**(I, S), where S is a set of variables (or set of variable numbers) from the polynomial ring P, returns the elimination ideal $I \cap K[S]$. This allows one to eliminate all variables except for the desired ones easily.

An ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, is called *zero-dimensional* if the quotient ring $Q = P/I$ has finite dimension as a vector space over K. This definition is fundamental and arises in many contexts. The function **IsZeroDimensional**(I) returns whether the ideal I is zero-dimensional. An alternative characterization is that an ideal I is zero-dimensional if and only if its variety over the algebraic closure of the coefficient field is finite. Yet another characterization of zero-dimensionality will be given below in the section on dimension of ideals.

If an ideal I is zero-dimensional then the *univariate elimination ideal* $I \cap K[x_i]$ for each $i = 1, \ldots n$ will be non-trivial. There is a unique monic generator of each ideal. Since it is of frequent importance to know these polynomials, MAGMA provides the function **UnivariateEliminationIdealGenerator**(I, i) to calculate the unique monic generator of the i-th univariate elimination ideal and the function **UnivariateEliminationIdealGenerators**(I) to calculate a sequence containing all n of these polynomials. MAGMA does all of this efficiently since it only needs to calculate one Gröbner basis of the ideal with respect to an easy monomial order and it then applies the Gröbner Walk to change this Gröbner basis to the Gröbner basis with respect to an elimination order which yields the desired polynomial. Note that this method is generally much faster than the common method of using linear algebra presented, for example, in [BeW93].For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x^3 + y + z - 1, x + y^2 + z - 1,
>                x + y + z^4 - 1>;
> print I;
Ideal of Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Basis:
[
    x^3 + y + z - 1,
    x + y^2 + z - 1,
    x + y + z^4 - 1
]
> print IsZeroDimensional(I);
true
> print UnivariateEliminationIdealGenerators(I);
[
    x^23 - 8*x^20 + 4*x^18 + 24*x^17 - 24*x^15 - 32*x^14 +
```

```
        6*x^13 + 50*x^12 + 7*x^11 - 24*x^10 - 16*x^9 + 2*x^8
        + 12*x^7 + 6*x^6 - 9*x^5 - x^3 + 2*x^2,
    y^22 + y^21 - 11*y^20 - 11*y^19 + 55*y^18 + 51*y^17 -
        165*y^16 - 126*y^15 + 330*y^14 + 162*y^13 - 434*y^12
        - 96*y^11 + 342*y^10 + 66*y^9 - 180*y^8 - 111*y^7 +
        112*y^6 + 68*y^5 - 23*y^4 - 44*y^3 - y^2 + 14*y,
    z^22 + z^21 + z^20 + z^19 - 5*z^18 - 5*z^17 - 5*z^16 -
        2*z^15 + 10*z^14 + 10*z^13 + 10*z^12 - 4*z^11 -
        5*z^10 - 5*z^9 - 2*z^8 + z^7 - z^3 - z^2
]
```

Because of the triangularization property mentioned above, it is also easy to compute the (finite) variety of an ideal I when the ideal I is zero-dimensional by root-finding and back-substitution. The function **Variety**(I) returns the variety of ideal I as a (sorted) sequence of vectors from the vector space K^n, while the function **VarietySequence**(I) returns the variety of I as a sequence of sequences over K (sometimes more useful because of clearer printing).

If the variety of an ideal is computed over an extension field of the original field of the ideal, then there may be more solutions than those over the original field. The function **Variety**(I, L) returns the variety of ideal I over the extension field L as a (sorted) sequence of vectors from the vector space L^n, while the function **VarietySequence**(I, L) returns the variety of I over the extension field L as a sequence of sequences over K.

For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x^2 + y + z - 1, x + y^2 + z - 1,
>     x + y + z^2 - 1>;
> print VarietySequence(I);
[
    [ 1, 0, 0 ]
    [ 0, 1, 0 ],
    [ 0, 0, 1 ],
]
> L<w> := QuadraticField(2);
> print VarietySequence(I, L);
[
    [ -1 - w, -1 - w, -1 - w ],
    [ -1 + w, -1 + w, -1 + w ],
    [ 1, 0, 0 ]
    [ 0, 1, 0 ],
    [ 0, 0, 1 ],
]
```

```
> VarietySizeOverAlgebraicClosure(I);
5
```

The function **VarietySizeOverAlgebraicClosure**(I) yields the size of the variety of I over the algebraic closure of its coefficient field. Thus in the case where the coefficient ring is the quadratic field $\mathbb{Q}(\sqrt{2})$, the full set of solutions of the associated system over $\mathbb{C}$ has necessarily been computed.

22.9 Radical and Decomposition of Ideals

For an ideal I of the polynomial ring $P = K[x_1, \ldots, x_n]$, the *radical* of I is defined to be the set of all polynomials $f \in P$ such that $f^k \in I$ for some $k \geq 1$. The radical R is also an ideal of P. One of the most important facts about the radical R is that it is the largest ideal of P containing I which has the same variety as I. The function **Radical**(I) computes the radical of the ideal I.

For example, after constructing an ideal I which is shown to be not zero-dimensional (so the variety of I over an algebraic closure is not finite), computing the radical of I often helps one understand the variety of I since the radical has a simpler structure:

```
> P<x, y, z, t> := PolynomialRing(RationalField(), 4);
> I := ideal<P |
> x + y + z + t,
> x*y + y*z + z*t + t*x,
> x*y*z + y*z*t + z*t*x + t*x*y,
> x*y*z*t - 1>;
> Groebner(I);
> print IsZeroDimensional(I);
false
> print I;
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: x, y, z, t
Groebner basis:
[
    x + y + z + t,
    y^2 + 2*y*t + t^2,
    y*z - y*t + z^2*t^4 + z*t - 2*t^2,
    y*t^4 - y + t^5 - t,
    z^3*t^2 + z^2*t^3 - z - t,
    z^2*t^6 - z^2*t^2 - t^4 + 1
```

```
]
> print Radical(I);
Ideal of Polynomial ring of rank 4 over Rational Field
Lexicographical Order
Variables: x, y, z, t
Radical
Basis:
[
    z^2*t^2 - 1,
    x + z,
    y + t
]
```

Verbose output for the radical algorithm can be obtained by the procedure call **SetVerbose**("Radical", 1). Maximal verbosity can be obtained with a level of 2.

A natural question one asks about an algebraic structure is of course whether it can be decomposed into simpler structures of the same kind. In the ideal case, we wish to decompose a given ideal into the intersection of simpler ideals if possible.

The simple ideals we are interested in are *prime* and *primary* ideals. An ideal I is called *prime* if whenever $fg \in I$ for polynomials f and g, either $f \in I$ or $g \in I$. An ideal I is called *primary* if whenever $fg \in I$ for polynomials f and g, either $f \in I$ or $g^k \in I$ for some $k \geq 1$.

First we consider radical ideals (ideals whose radicals are themselves). Any radical ideal can be decomposed uniquely (up to ordering) as a minimal intersection of prime ideals. The function **RadicalDecomposition**(I), when given an ideal I, computes its radical R, and then returns the the unique minimal (sorted) sequence of prime ideals whose intersection is R. The prime decomposition of the radical corresponds to the decomposition of the variety of I into irreducibles: each variety of a prime component is called irreducible, and the union of these varieties gives the full variety of I.

For example, applying **RadicalDecomposition**(I) to the above ideal I shows that the radical of I decomposes into the intersection of two prime ideals. Thus the variety of the original ideal I can be seen to be the union of two varieties which can be more easily understood geometrically because of their simple structure.

```
> print RadicalDecomposition(I);
[
    Ideal of Polynomial ring of rank 4 over Rational Field
    Lexicographical Order
    Variables: x, y, z, t
```

```
    Dimension 1, Radical, Prime
    Groebner basis:
    [
        x + z,
        y + t,
        z*t + 1
    ],
    Ideal of Polynomial ring of rank 4 over Rational Field
    Lexicographical Order
    Variables: x, y, z, t
    Dimension 1, Radical, Prime
    Groebner basis:
    [
        x + z,
        y + t,
        z*t - 1
    ]
]
```

A general ideal I does *not* necessarily decompose into the intersection of prime ideals. That is why the notion of a primary ideal is necessary. It turns out that an ideal I can be decomposed into an intersection of primary ideals. For each primary ideal J of the decomposition, its radical is a prime ideal and is called the *associated prime* of J. The function **PrimaryDecomposition**(I) returns a primary decomposition of the given ideal, together with the associated primes.

Thus, continuing the above example, a primary decomposition of I has size 8. Notice that the sequence of associated primes (P) contains ideals which contain other ideals of the associated primes. This phenomenon does not occur for radical ideals so it shows the more subtle complications that can arise with non-radical ideals.

```
> Q, P := PrimaryDecomposition(I);
> print #Q, #P;
8 8
> print [Basis(J): J in Q];
[
    [ x + z, y + t, z*t + 1 ],
    [ x + z, y + t, z*t - 1 ],
    [ x + y + 2, y^2 + 2*y + 1, z - 1, t - 1 ],
    [ x + y - 2, y^2 - 2*y + 1, z + 1, t + 1 ],
    [ x + 1, y + z - 2, z^2 - 2*z + 1, t + 1 ],
    [ x - 1, y + z + 2, z^2 + 2*z + 1, t - 1 ],
    [ x + y + 2*t, y^2 + 2*y*t - 1, z - t, t^2 + 1 ],
```

```
    [ x - t, y + z + 2*t, z^2 + 2*z*t - 1, t^2 + 1 ]
]
> print [Basis(J): J in P];
[
    [ x + z, y + t, z*t + 1 ],
    [ x + z, y + t, z*t - 1 ],
    [ x + 1, y + 1, z - 1, t - 1 ],
    [ x - 1, y - 1, z + 1, t + 1 ],
    [ x + 1, y - 1, z - 1, t + 1 ],
    [ x - 1, y + 1, z + 1, t - 1 ],
    [ x + t, y + t, z - t, t^2 + 1 ],
    [ x - t, y + t, z + t, t^2 + 1 ]
]
```

Table 22.6. Radical and decomposition functions

MAGMA	Meaning
Radical(I)	Radical of ideal I
PrimaryDecomposition(I)	Primary decomposition of ideal I together with associated primes
RadicalDecomposition(I)	Prime decomposition of the radical of I
IsRadical(I)	True iff ideal I is radical (equals its radical)
IsPrimary(I)	True iff ideal I is primary
IsPrime(I)	True iff ideal I is prime
IsMaximal(I)	True iff ideal I is maximal

Table 22.6 lists the functions for computing with radicals and primary decompositions. For more information on the relevant theory, see [CLO92], chapter 4, §7, and [BeW93], chapter 8.

Verbose output for the decomposition algorithms can be obtained by the procedure call **SetVerbose**("Decomposition", 1). Maximal verbosity can be obtained with a level of 2.

22.10 Graded Polynomial Rings and Hilbert Series

It is possible within MAGMA to assign weights to the variables of a multivariate polynomial ring. This means that monomials of the ring then have a *weighted degree* with respect to the weights of the variables. Such a multivariate polynomial ring is called *graded* or *weighted*. A polynomial of the

ring whose monomials all have the same weighted degree is called *homogeneous*. The polynomial ring can be decomposed as the direct sum of graded homogeneous components.

Suppose a polynomial ring P has n variables $x_1, \ldots, x_n$ and the weights for the variables are $d_1, \ldots, d_n$ respectively. Then for a monomial $m = x_1^{e_1} \cdots x_n^{e_n}$ of P (with $e_i \geq 0$ for $1 \leq i \leq n$), the *weighted degree* of m is defined to be $\sum_{i=1}^{n} e_i d_i$.

To create a polynomial ring $P = R[x_1, \ldots, x_n]$ with weights $d_1, \ldots, d_n$ corresponding respectively to the variables, one simply uses the function **PolynomialRing**(R, Q), where Q is the sequence $[d_1, \ldots, d_n]$ of weights corresponding to the variables. The rank n of the polynomial ring is determined by the length of the sequence Q.

For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), [1, 2, 4]);
> print P;
Graded Polynomial ring of rank 3 over Rational Field
Lexicographical Order
Variables: x, y, z
Variable weights: 1 2 4
```

A polynomial ring constructed in the usual way without given weights is assumed to have weight 1 for each variable.

The function **WeightedDegree**(f) returns the weighted degree of the polynomial f (the maximum of the weighted degrees of the monomials of f), while the function **LeadingWeightedDegree**(f) returns the weighted degree of the leading monomial of f.

Continuing the above example:

```
> P<x, y, z> := PolynomialRing(RationalField(), [1, 2, 4]);
> print WeightedDegree(x);
1
> print WeightedDegree(y);
2
> print WeightedDegree(z);
4
> print WeightedDegree(x^2*y*z^3);
16
> print TotalDegree(x^2*y*z^3);
6
```

The function **IsHomogeneous**(f) returns whether f is homogeneous with respect to the weights on the variables (i.e., whether the weighted degrees of the monomials of f are all equal). Similarly, the function **IsHomogeneous**(I) returns whether the ideal I is homogeneous with respect to the weights on the variables of the polynomial ring of which it is an ideal—this is equivalent to whether I possesses a basis consisting of homogeneous polynomials alone.

Continuing the above example:

```
> P<x, y, z> := PolynomialRing(RationalField(), [1, 2, 4]);
> print IsHomogeneous(x);
true
> print IsHomogeneous(x + y);
false
> print IsHomogeneous(x^2 + y);
true
> I := ideal<P | y^2 + 2*x^2*y, (x^4 + z)^2, x + y, x^2 + x>;
> print IsHomogeneous(I);
true
```

For a *homogeneous* ideal I of the graded polynomial ring $P = K[x_1, \ldots, x_n]$, the quotient ring P/I is a *graded* vector space in the following way: P/I is the direct sum of the vector spaces V_d for $d = 0, 1, \ldots$ where V_d is the K-vector space consisting of all homogeneous polynomials in P/I (i.e., reduced residues of polynomials of P with respect to I) of weighted degree d. The *Hilbert Series* of the graded vector space P/I is the generating function

$$H_{P/I}(t) = \sum_{d=0}^{\infty} \dim(V_d) t^d.$$

The Hilbert series can be written as a rational function in the indeterminate t.

The function **HilbertSeries**(I) returns the Hilbert Series $H_{P/I}$ of P/I, where I is a homogeneous ideal of the (possibly graded) polynomial ring P, as an element of the univariate function field over the ring of integers. For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), [1, 2, 4]);
> I := ideal<P | x, y^2, z^3>;
> H<t> := HilbertSeries(I);
> print H;
-1/(t^2 - 1)
```

If the weights on the variables of P are all 1, then there exists the Hilbert polynomial $F_{P/I}(d)$ corresponding to the Hilbert series $H_{P/I}(t)$ which is a

univariate polynomial in $\mathbb{Q}[d]$ such that $F_{P/I}(i)$ is equal to the coefficient of t^i in the Hilbert series for all $i \geq k$ for some fixed k.

The function **HilbertPolynomial**(I) returns the Hilbert polynomial $F_{P/I}$ corresponding to $H_{P/I}$, where I is a homogeneous ideal of the (possibly graded) polynomial ring P, as an element of the univariate polynomial ring over the rational field. The function also returns the *index of regularity* k which is the minimal integer $k \geq 0$ such that $F_{P/I}(i)$ equals the coefficient of t^i in the Hilbert series for all $i \geq k$. For example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x^3 + x^2*y + x^2*z, x*y^2 + y^3 + y^2*z>;
> H<t> := HilbertSeries(I);
> print H;
(-t^4 - t^3 + t^2 + t + 1)/(t^2 - 2*t + 1)
> F<d>, k := HilbertPolynomial(I);
> print F;
d + 5
> print k;
3
> // Check that evaluations of F for d >= 3 match coefficients of H:
> SR<u> := PowerSeriesRing(IntegerRing());
> print SR ! H;
1 + 3*u + 6*u^2 + 8*u^3 + 9*u^4 + 10*u^5 + 11*u^6 + 12*u^7 +
    13*u^8 + 14*u^9 + 15*u^10 + 16*u^11 + 17*u^12 + 18*u^13
    + 19*u^14 + 20*u^15 + 21*u^16 + 22*u^17 + 23*u^18 +
    24*u^19 + O(u^20)
> print Evaluate(F, 3);
8
> print Evaluate(F, 4);
9
```

22.11 Dimension of Ideals

Let I be an ideal of the polynomial ring $P = K[x_1, \ldots, x_n]$. Let X be the set $\{x_1, \ldots, x_n\}$ of variables of P. A subset U of X is called *independent modulo* I if $I \cap K[U] = \emptyset$. A subset U of X is called *maximally independent modulo* I if U is independent modulo I, and no proper superset of U is independent modulo I. The *dimension* of I is defined to be the maximum of the cardinalities of all the independent sets modulo I. Note that dimension 0 according to this definition does correspond to the definition of a zero-dimensional ideal given above.

The MAGMA function **Dimension**(I) computes the dimension d of the ideal I. The function also returns a (sorted) sequence U of integers of length d such that the variables of P corresponding to the integers of U constitute a maximally independent set modulo I.

For example, the following ideal has dimension 1 obviously, since the variable z is 'free' so a maximally independent set for the ideal is $\{z\}$.

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x, y^2>;
> print Dimension(I);
1 [ 3 ]
```

23. Power Series and Laurent Series

This chapter explains the operations on power series rings and Laurent series rings in MAGMA. These structures share many operations with polynomial rings $R[x]$, and that there are some similarities with the real and complex fields with regard to the control of precision.

23.1 Power Series and Laurent Series

Power series rings $R[[x]]$ and Laurent series rings $R((x))$ over a commutative ring R contain polynomial-like elements which are, loosely speaking, polynomials with an infinite number of terms. More strictly, a *power series* is an element of the form $a_v x^v + a_{v+1}x^{v+1} + a_{v+2}x^{v+2} + \cdots$, where v is a non-negative integer and $a_v \neq 0$ (unless the series is zero). Here the a_i are elements of R, and x is the indeterminate. The other concept, *Laurent series*, is slightly more general: a Laurent series is permitted to have a finite number of negative powers of x as well, so v may be a negative integer. One way of looking at a Laurent series is to consider it as a power series multiplied by x^k, for some $k \in \mathbb{Z}$.

Since computer memory is finite, power series and Laurent series in MAGMA with an infinite number of non-zero terms must be stored in a truncated form. Thus the infinite series above is stored as $a_v x^v + a_{v+1}x^{v+1} + a_{v+2}x^{v+2} + \cdots + a_{p-1}x^{p-1} + O(x^p)$, for some integer p. The notation $O(x^p)$, known colloquially as 'big-oh' notation, denotes unknown terms in which the exponent of x is at least p.

Table 23.1 (p. 408) lists some functions on power series and infinite series. Given the series $f = a_v x^v + a_{v+1}x^{v+1} + a_{v+2}x^{v+2} + \cdots + a_{p-1}x^{p-1} + O(x^p)$, the number v is called the *valuation* of f, and is the least power of x occurring with a non-zero coefficient in f. As noted above, v may be any integer in a Laurent series, but must be non-negative in a power series. The number p is called the *absolute precision*, and $p - v$ is called the *relative precision*. The relationship between these two kinds of precision is that the relative precision of f equals the absolute precision of $\frac{f}{x^v}$. The *degree* is the highest-

Table 23.1. Precision and coefficients of series

MAGMA	Meaning
Valuation(f)	Given series f with indeterminate x, return least power v of x having a nonzero coefficent in f
AbsolutePrecision(f)	Absolute precision p of a series f concluding $O(p)$
RelativePrecision(f)	$p - v$ if f non-exact, else -1
Degree(f)	Exponent d of the last-known non-zero coefficient of f
Coefficients(f)	Sequence $[a_v, \ldots, a_d]$ of all coefficients of f from the valuation coefficient to the degree coefficient
Coefficient(f, i)	Coefficient a_i of x^i in f
Truncate(f)	Exact power series obtained by truncating f after the $a_d x^d$ term
CoefficientRing(R)	Coefficient ring R of series ring $R[[x]]$ or $R((x))$

known power of x with a non-zero coefficient in f, that is, the largest integer d in the range $v \leq d \leq (p-1)$ such that $a_d \neq 0$.

The function **Coefficients**(f) returns the sequence $[a_v, \ldots, a_d]$. There is an important difference between this function as it applies to a power series or Laurent series compared with the way it applies to a polynomial f. If f is a polynomial with indeterminate x, the coefficient of x^i is always the $(i+1)^{\text{th}}$ entry in the sequence returned by **Coefficients**. However, if f is a series, the first entry of **Coefficients**(f) is the coefficient of x^v, where v is the valuation. By contrast, **Coefficient**(f, i) returns the coefficient of x^i, for both polynomials and series.

If a series has only a finite number of non-zero terms, it is called *exact*, and is written like a polynomial. The function **Truncate**(f) changes a finite-precision series into an exact series by removing the big-oh term and returning the series $a_v x^v + a_{v+1} x^{v+1} + \cdots + a_d x^d$.

The exact zero element, and approximate zero elements of the form $O(x^p)$, behave exceptionally for some of the functions in the table.

23.2 Constructing Series Rings and Their Elements

The functions for constructing a power series ring or Laurent series ring over a commutative ring R are **PowerSeriesRing** and **LaurentSeriesRing**. Since these rings are also R-algebras, the functions have the alternative names **PowerSeriesAlgebra** and **LaurentSeriesAlgebra**. There are several ways to use these functions, depending on how the user wishes MAGMA to handle the precision of the elements of the series ring. The essential distinction is

between *free* rings, in which each element has its own precision, and *fixed* or *limited* precision rings, in which every element has a precision less than or equal to some fixed number.

23.2.1 Free Precision Series Rings

The method for creating a free precision power series ring is to give the function **PowerSeriesRing** a single argument, the coefficient ring R. For example:

```
> Q := RationalField();
> P<x> := PowerSeriesRing(Q);
> print P;
Power Series Algebra in x over Rational Field
```

Exact elements of P may be created like polynomials:

```
> p1 := 6*x^5 + 3/5*x^17 - 4*x^144; print p1;
6*x^5 + 3/5*x^17 - 4*x^144
> print Valuation(p1), Degree(p1);
5 144
```

To create an element with a finite absolute precision p, type $\mathbf{O}(x\hat{}p)$ as one of the terms:

```
> p2 := 5*x^18 - 7*x^22 + 10*x^23 + O(x^47);
> print p2;
5*x^18 - 7*x^22 + 10*x^23 + O(x^47)
> print Valuation(p2), Degree(p2);
18 23
> print AbsolutePrecision(p2), RelativePrecision(p2);
47 29
> print Truncate(p2);
5*x^18 - 7*x^22 + 10*x^23
```

The sequence of coefficients of a series begins at the valuation coefficient and ends at the degree coefficient. For instance:

```
> print Coefficients(p2);
[ 5, 0, 0, 0, -7, 10 ]
```

This kind of coefficient sequence can also be used in the opposite direction, to create a series by means of the **elt** constructor:

elt< $P \mid v, Q, r$ **>**

It creates the element with valuation v, coefficient sequence Q and relative precision r in the power series ring or Laurent series ring P. The value of r must be positive or -1. If v is omitted, then the valuation is set to be zero, and if r is omitted, then the relative precision is set to be the length of Q. For instance, the following element is equal to *p2*:

```
> print elt< P | 18, [ 5, 0, 0, 0, -7, 10 ], 29 >;
5*x^18 - 7*x^22 + 10*x^23 + O(x^47)
```

To create an exact series, type -1 for r:

```
> print elt< P | [4..40 by 3], -1 >;
4 + 7*x + 10*x^2 + 13*x^3 + 16*x^4 + 19*x^5 + 22*x^6
    + 25*x^7 + 28*x^8 + 31*x^9 + 34*x^10 + 37*x^11 + 40*x^12
```

A final way to create elements is to coerce an element of the coefficient ring, or a sequence of such elements, into the series ring P. The coercion of a single element gives an exact result. However, the coercion of a sequence Q gives the same result as **elt<** $P \mid Q$ **>**, since it is assumed that the valuation is zero and that the final entry of Q is the last-known term. For instance:

```
> print P![4, 34, 59, 0, 2, 0];
4 + 34*x + 59*x^2 + 2*x^4 + O(x^6)
```

Elements of Laurent series rings may be created according to the same methods as for power series rings, except that negative powers of the indeterminate are allowed:

```
> L<y> := LaurentSeriesRing(Q);
> print 5*y^-2 + 16*y^5 + O(y^7);
5*y^-2 + 16*y^5 + O(y^7)
> print elt< L | -19, [4,24,0,3,5], 21 >;
4*y^-19 + 24*y^-18 + 3*y^-16 + 5*y^-15 + O(y^2)
> print elt< L | -90, [7,3,4], 75 >;
7*y^-90 + 3*y^-89 + 4*y^-88 + O(y^-15)
```

In some circumstances, a non-exact power series or Laurent series is calculated without a particular relative precision being requested. For instance, the inverse of a series generally has an infinite number of non-zero terms:

```
> p3 := (2 + p1) ^ -1;
> print p3;
1/2 - 3/2*x^5 + 9/2*x^10 - 27/2*x^15 - 3/20*x^17 + O(x^20)
```

It is impossible to return an infinite series, so MAGMA must know what precision to use. This *default relative precision* is stored in the `"Precision"` attribute of the ring. To find the current value of the precision attribute for the free series ring P, use the function **HasAttribute**(P, `"Precision"`), which returns two values. The second return value is the current value of the precision attribute. (The first return value is always **true** in this case, because the precision attribute is always defined for any free series ring that has been created.) For instance:

```
> print HasAttribute(P, "Precision");
true 20
```

Therefore the attribute's value is 20. This number 20 has been chosen by the designers of MAGMA as the default value of the precision attribute for free series rings.

There are two ways to make the precision attribute of a free series ring different from 20. It may either be changed once the ring has been created, or it may be given a non-default value at the same time as the ring is created. The procedure **AssertAttribute**(P, `"Precision"`, p): may be used to change the precision attribute of an existing ring P to p:

```
> AssertAttribute(P, "Precision", 28);
> print (2 + p1) ^ -1;
1/2 - 3/2*x^5 + 9/2*x^10 - 27/2*x^15 - 3/20*x^17 + 81/2*x^20
    + 9/10*x^22 - 243/2*x^25 - 81/20*x^27 + O(x^28)
> print p3;
1/2 - 3/2*x^5 + 9/2*x^10 - 27/2*x^15 - 3/20*x^17 + O(x^20)
```

Notice that the change in the attribute only affects newly-created elements, not pre-existing ones such as *p3*. On the other hand, to create a free series ring with a precision attribute other than 20, assign a value to the parameter **Precision** when calling **PowerSeriesRing**(R) or **LaurentSeriesRing**(R):

```
> Pb<xb> := PowerSeriesRing(Q: Precision := 8);
> Lb<yb> := LaurentSeriesRing(Q: Precision := 8);
```

When the ring has been created in this way, the default relative precision will be 8:

```
> print xb^3 / (1 - xb);
xb^3 + xb^4 + xb^5 + xb^6 + xb^7 + xb^8 + xb^9
    + xb^10 + O(xb^11)
> print yb^3 / (1 - yb);
yb^3 + yb^4 + yb^5 + yb^6 + yb^7 + yb^8 + yb^9
    + yb^10 + O(yb^11)
```

23.2.2 Fixed Absolute Precision Power Series Rings

A different approach to the precision issue for power series rings should be used in cases where all the elements of the ring are to have the same precision, with no exceptions. The function for creating a *fixed absolute precision power series ring*, is **PowerSeriesRing**(R, p):

```
> P30<s> := PowerSeriesRing(Q, 30);
> print P30;
Power Series Algebra of precision 30 in s
over Rational Field
> print Precision(P30);
30
```

Note that the function **Precision** returns p. This integer will be the absolute precision of all elements of the power series ring. For example:

```
> print s^5;
s^5 + O(s^30)
> print P30!0;
O(s^30)
```

A power series ring of fixed absolute precision p over R may be considered mathematically as the quotient structure $R[[x]]/x^p$. That is, all elements of $R[[x]]$ with the same coefficients from x^0 to x^{p-1} are considered to form an equivalence class.

Since every element of a fixed precision power series ring has the same absolute precision, and there are no exact elements, the creation of elements is slightly easier. They may be entered like polynomials, as shown above. Alternatively, the notation **elt<** $P \mid v, Q$ **>** will produce the element with valuation v and coefficient sequence Q, or simply $P!Q$ if the valuation is zero. Although MAGMA will accept both the **O** function and the r option in the **elt** constructor, without giving an error message, it will ignore the given values.

23.2.3 Limited Relative Precision Laurent Series Rings

The final kind of series ring is a *limited relative precision Laurent series ring*. It is created by the function **LaurentSeriesRing**(R, p):

```
> L30<t> := LaurentSeriesRing(Q, 30);
> print L30;
Laurent Series Algebra of precision 30 in t
```

```
over Rational Field
> print Precision(L30);
30
```

All elements of this ring will have relative precision less than or equal to p. (This integer is returned by the function **Precision**.) For example:

```
> print t^5;
t^5 + O(t^35)
> print t^5 + O(t^12);
t^5 + O(t^12)
> print L30!0;
0
```

Observe that if an element is typed as a polynomial, then it is given the relative precision p. The **O** function should be used for creating elements with relative precision less than p.

23.3 Operations on Series

Arithmetic on power series or Laurent series is performed with the usual operators, but the user should be aware of how the precision of the result is calculated. In fixed precision power series rings, the absolute precision of every element is the same, so there is no problem. In the other three kinds of series rings, the absolute precision of $f + g$ is the minimum of the absolute precisions of f and g, and the relative precision of $f*g$ is the minimum of the relative precisions of f and g.

The operator / performs division. For example:

```
> print x^5 / (x^2 + x^3);
x^3 - x^4 + x^5 - x^6 + x^7 - x^8 + x^9 - x^10 + x^11
    - x^12 + x^13 - x^14 + x^15 - x^16 + x^17 - x^18
    + x^19 - x^20 + x^21 - x^22 + O(x^23)
```

To evaluate the expression f/g, MAGMA divides f and g by the valuation of g to obtain f'/g', and then returns $f'*g'^{-1}$. Therefore, in order for the division to succeed, the constant term of g' (and hence the 'valuation coefficient' of g) must be a unit. It follows that if the coefficient ring R is not a field, division is not generally available. Of course, if f and g are power series rather than Laurent series, it is also required that the valuation of the result be non-negative.

Exponentiation $f\hat{}n$ is possible for a series f and an integer n, but if f is a power series rather than a Laurent series then the result cannot contain any negative powers of the indeterminate. In practice this means that n must be non-negative or f must have valuation zero. It is also possible to evaluate $f\hat{}g$ for two series f and g, but only if the valuation of f is zero.

The usual Boolean operators and functions, such as **eq**, **ne** and **IsZero**, apply to elements of series rings. These tests are *strict*. Each ring contains only one zero, and this zero has infinite precision, except in a fixed precision power series ring. Two elements are equal only if their difference is the zero of the ring. To perform a more lenient test for equality, the user should subtract the two elements and then test whether the truncation of the difference is zero. For instance:

```
> f1 := x^5 + O(x^10);
> f2 := x^5 + O(x^12);
> print f1 eq f2;
false
> d := f1 - f2; print d;
O(x^10)
> print IsZero(d);
false
> print IsZero(Truncate(d));
true
```

The function **Evaluate**(f, r) provides a way of substituting the ring element r into the places where the indeterminate occurs in the series f. For instance, the following calculations show a (very inefficient) way of obtaining the constant e:

```
> expon := Exp(x); print expon;
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6
    + 1/5040*x^7 + 1/40320*x^8 + 1/362880*x^9 +
    1/3628800*x^10 + 1/39916800*x^11 + 1/479001600*x^12 +
    1/6227020800*x^13 + 1/87178291200*x^14 +
    1/1307674368000*x^15 + 1/20922789888000*x^16 +
    1/355687428096000*x^17 + 1/6402373705728000*x^18 +
    1/121645100408832000*x^19 + O(x^20)
> print Evaluate(expon, 1.0);
2.71828182845904523492875272830
```

Table 23.2 lists several other operations on series.

Table 23.2. Operations on series

MAGMA	Meaning
Derivative(f)	Derivative of series f
Derivative(f, n)	n^{th} derivative of f
Integral(f)	Anti-derivative of f
SquareRoot(f), **Sqrt**(f)	Approximated square root of f, where f has even valuation
Laplace(f)	Laplace transform $\sum_{i\geq 0}(i!a_i)x^i$ of f, where $f = \sum_{i\geq 0} a_i x^i$
Composition(f, g)	$f \circ g = \sum_{i<p} f_i(g^i)$ where $f = \sum_{i<p} f_i x^i$
Reversion(f), **Reverse**(f)	Inverse of f under composition, i.e., series g such that $f \circ g \approx x$
Convolution(f, g)	$f*g = \sum_{i<\min(p,q)} f_i g_i x^i$, where $f = \sum_{i<p} f_i x^i + O(x^p)$ and $g = \sum_{i<q} g_i x^i + O(x^q)$

23.4 Transcendental Functions

Table 23.3. Transcendental functions on series

MAGMA	Meaning
Sin(f)	Sine of series f
Cos(f)	Cosine of series f
Sincos(f)	Sine and cosine of series f
Tan(f)	Tangent of series f
Sinh(f)	Hyperbolic sine of series f
Cosh(f)	Hyperbolic cosine of series f
Tanh(f)	Hyperbolic tangent of series f
Exp(f)	Exponential of series f
Log(f)	Logarithm of series f of valuation zero

Table 23.3 (p. 415) lists the functions that return transcendental functions of a given series f. The coefficient ring of f must be a field. In particular, if f has valuation zero (that is, a non-zero constant term), then the coefficient field must be one of the real or complex fields, so that the constant term of the result can be evaluated. The precision of the result will be approximately equal to the precision of f.

For example:

```
> PQ<x> := PowerSeriesRing(RationalField());
```

```
> print Sincos(x);
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 -
    1/39916800*x^11 + 1/6227020800*x^13 -
    1/1307674368000*x^15 + 1/355687428096000*x^17 -
    1/121645100408832000*x^19 + O(x^21)
1 - 1/2*x^2 + 1/24*x^4 - 1/720*x^6 + 1/40320*x^8 -
    1/3628800*x^10 + 1/479001600*x^12 - 1/87178291200*x^14 +
    1/20922789888000*x^16 - 1/6402373705728000*x^18 +
    O(x^20)
> print Exp(x + O(x^15));
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6
    + 1/5040*x^7 + 1/40320*x^8 + 1/362880*x^9 +
    1/3628800*x^10 + 1/39916800*x^11 + 1/479001600*x^12 +
    1/6227020800*x^13 + 1/87178291200*x^14 + O(x^15)
> P30R<y> := PowerSeriesRing(RealField(), 30);
> print Log(1 + y);
y - 1/2*y^2 + 1/3*y^3 - 1/4*y^4 + 1/5*y^5 - 1/6*y^6 +
    1/7*y^7 - 1/8*y^8 + 1/9*y^9 - 1/10*y^10 + 1/11*y^11 -
    1/12*y^12 + 1/13*y^13 - 1/14*y^14 + 1/15*y^15 -
    1/16*y^16 + 1/17*y^17 - 1/18*y^18 + 1/19*y^19 -
    1/20*y^20 + 1/21*y^21 - 1/22*y^22 + 1/23*y^23 -
    1/24*y^24 + 1/25*y^25 - 1/26*y^26 + 1/27*y^27 -
    1/28*y^28 + 1/29*y^29 + O(y^30)
```

These series may also be used to check identities, although the results may not be exact when floating point arithmetic is used:

```
> print Derivative(Tan(x)) eq Cos(x)^-2;
true
> C<i> := ComplexField();
> PC<z> := PowerSeriesRing(C);
> print Exp(i*z) eq (Cos(z) + i*Sin(z));
true
> R8 := RealField(8);
> PR8<t> := PowerSeriesRing(R8);
> print Sin(Pi(R8)/2 - t) - Cos(t);
-0.00000001 + 3.673205e-6*y - 6.12201e-7*y^3 + 3.061e-8*y^5
    - 7.288006e-10*y^7 + 1.1222e-11*y^9 - 9.22113e-14*y^11 +
    5.8988e-16*y^13 - 2.88949e-18*y^15 + 1.0327e-20*y^17 -
    3.1957e-23*y^19 + O(y^20)
```

23.5 Series as Generating Functions

One of the applications of series is in counting theory. A *generating function* is a series that carries in its coefficients the values of some function τ whose domain is the integers (or just the non-negative integers, for a power series). The n^{th} coefficient is the function value $\tau(n)$.

For example, the sequence of Catalan numbers arises in several combinatorial contexts. The n^{th} Catalan number is the number of balanced arrangements of n left brackets and n right brackets, that is, arrangements such as (()((((()))))(((()))) in which all n pairs of brackets are nested legally. It can be shown that

$$C(x) = \frac{1 - \sqrt{1 - 4x}}{2x}$$

is a closed form generating function for the Catalan numbers. The following lines construct this power series and print its coefficents:

```
> PQ<x> := PowerSeriesRing(RationalField());
> C := (1 - Sqrt(1-4*x)) / (2*x);
> print Coefficients(C);
[ 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786,
208012, 742900, 2674440, 9694845, 35357670, 129644790,
477638700 ]
```

It can also be shown that the n^{th} Catalan number is equal to $\frac{1}{n+1}\binom{2n}{n}$. The following lines show that the results from the generating function and this identity are equal in the case $n = 7$:

```
> print Coefficient(C, 7);
429
> print Binomial(2*7, 7) div (7+1); // division is exact
429
```

The example below is a verification (within the precision) of the identity $C - 1 = xC^2$. It shows that the difference between the two sides of the equation is an approximation to zero:

```
> D := x*C^2 - (C - 1);
> print D;
O(x^19)
> print IsZero(Truncate(D));
true
```

24. Finite Fields

A *finite field* or *Galois field* is a field containing only a finite number of elements q. Such a field exists only if q equals p^n, a positive power of a prime number p and, up to isomorphism, there is exactly one field for each such q. The numbers p and n for a field of size p^n are called the *characteristic* and the *degree*. A finite field of p^n elements contains a subfield of cardinality p^m when m divides n. In particular, any finite field contains a subfield of prime cardinality p; such a field is called a *prime field*

In MAGMA it is well possible to work in a finite field created from the specification of p^n only. The first section of this Chapter provides the details. In many situations, however, it is convenient to have more control over the representation of elements of finite fields and of the relations between finite fields. Later sections of this Chapter contain the more advanced mechanism in MAGMA for dealing with that.

24.1 Finite Fields by Cardinality

24.1.1 Constructing Finite Fields

The simplest way of creating the Galois field with q elements in MAGMA is to use the function **FiniteField** (with which **GaloisField** is synonymous, allowing the abbreviation **GF**). The function may be used as **GF**(q) or as **GF**(p, n), with exactly the same result except that in the second version MAGMA does not have the extra task of factorizing q; this may save some time if p is large.

```
> F32 := GF(2, 5); print F32;
Finite field of size 2^5
> print F32 eq GF(32);
true
```

Given a finite field F with p^n elements, the MAGMA functions **Degree**(F) and **Characteristic**(F) return p and n. The cardinality $q = p^n$ of F is returned by $\#F$.

Once a finite field of cardinality p^n has been created, it is possible to define extensions and subfields by use of **ext** and **sub**, specifying just the desired degree. For extensions any degree is allowed, for subfields the degree must divide n of course.

```
> G := GF(11, 3);
> H := ext< G | 4 >;
> I := sub< H | 6 >;
> print I;
Finite field of size 11^6
```

There exist many more sophisticated ways of creating extensions and subfields, which we will encounter below.

Arithmetic in a prime field of cardinality p is just arithmetic modulo p. It is possible to do such arithmetic without creating the finite field, with integer operations (see ??).

To define elements in larger fields we use some distinguished element, a generator. Any finite field F in MAGMA is built up as the extension of another finite field G (called the *ground field*) by an element called the *generator* of the field. The ground field is either explicit in the construction, for example when we use **ext**, or implicit, such as in the **sub** construction and in the **GF**(p, n) definition. In such implicit cases the ground field is the smallest subfield, i.e., the prime subfield.

Any prime field will be its own ground field, with generator simply 1. There are functions for retrieving the ground field and prime subfield.

```
> G := GF(11, 3);
> H := ext< G | 4 >;
> print PrimeField(G), PrimeField(H);
Finite field of size 11
Finite field of size 11
> print GroundField(G), GroundField(H);
Finite field of size 11
Finite field of size 11^3
```

24.1.2 Creation of Elements

The generator of a finite field F has the property that every other element of the field can be written as a polynomial expression in this generator, with

coefficients from the ground field G. This *polynomial representation* is unique if we allow polynomials of degree less than $n - m$ only, where n and m are the degrees of F and G.

The generator can be extracted from the field using **Generator**(G) or G**.**1. but it is much better to assign it to a variable and at the same time determine the way in which elements of the field will be printed by the familiar angle bracket construction on the left-hand side in the definition of the finite field:

```
> G<g> := GF(11, 3);
> print G.1;
g
> print Random(G);
g^29
> print g^2-g+10;
g^1158
```

As can be seen here, the elements of this finite field are *not* printed as a polynomial of degree less than 3 in the generator g. The reason is that there is another representation of elements that is often convenient. The non-zero elements of a finite field form a finite (multiplicative) group that is cyclic of order $q - 1$. Once a generator v of this group is found, every element of the field except zero can be written uniquely as v^k with $0 \leq k < q - 1$. We will call this the *power presentation*. An element of the field that generates the multiplicative group is called *primitive*.

In principle we then have two representations for non-zero elements in any finite field: the polynomial presentation and the power presentation. For small fields (for q less than around 10^6) both may be used; by default elements are given in the power presentation, but by changing an attribute on the field it is possible to change this. Continuing the previous example:

```
> AssertAttribute(G, "PowerPrinting", false);
> print g^29, g^1158;
g^2 + 9*g + 10
g^2 + 10*g + 10
```

For large fields elements will always be shown in polynomial representation. The reason is that it is not always easy to find a primitive polynomial, and it may be hard to convert between polynomial and power presentations. In general it is not guaranteed that field generators are primitive. The function **IsPrimitive** may be used to check primitivity of an element. Moreover, with **PrimitiveElement**(F) a primitive element can be found.

```
> H<h> := ext< G | 4>;
```

```
> print Random(H);
> print IsPrimitive(h);
true
> print IsPrimitive(h+1);
false
> print FactoredOrder(h+1);
[ <2, 4>, <3, 2>, <5, 1>, <13, 1>, <19, 1>, <37, 1>, <61, 1>, <1117, 1> ]
> print FactoredOrder(h);
[ <2, 4>, <3, 2>, <5, 1>, <7, 1>, <13, 1>, <19, 1>, <37, 1>, <61, 1>, <1117, 1> ]
> print IsPrimitive(Root(h+1, 7));
true
```

In the above example the element h is primitive, but $h + 1$ is not. In fact $h + 1$ is the 7-th power of a primitive element.

24.2 Finding Special Elements

We have seen already how to find random elements (in the example above), primitive elements, and a generator for a field over its ground field. The function **Generator** can also be used to produce a generator over a subfield S other than the prime field; in that case **Generator**(F, S) should be used.

Two more elements of a special kind have their own intrinsic function in MAGMA's language. With **NormalElement** an element that generates a *normal basis* can be found, that is, an element α such that $\alpha, \alpha^p, \alpha^{p^2}, \ldots, \alpha^{p^{n-1}}$ forms a basis for F over its ground field. Again, **NormalElement**(G, S) can be used to find an element generating a normal basis over an intermediate field.

Other ways of obtaining the generator of a finite field are **Generator**(F) and $F.1$.

A *primitive element* of a finite field F is an element a that generates (multiplicatively) the group of the non-zero elements of F. The function **IsPrimitive**(a) tests if $a \in F$ is primitive. For example:

```
> print IsPrimitive(g), IsPrimitive(g^2);
true false
```

Notice that in this case of this field g is not only a generator but also a primitive element; this may be confirmed by calculating the powers of g:

```
> print Set(gf9);
{ 1, g, g^2, g^3, 2, g^5, g^6, g^7, 0 }
```

Although the generator g of *gf9* is a primitive element, *it is not always true that the generator supplied by* MAGMA *for a finite field will be primitive.* However, the generator will definitely be primitive if MAGMA is able to use a Conway polynomial to define the field, because of the properties of these polynomials. Conway polynomials are available for reasonably small fields. (See Section 24.6 for more information.) As for fields whose Conway polynomials are not known, such as the fields in the following example, the primitivity of the generator depends on the field and on the generator that MAGMA chooses:

```
> print IsPrimitive(Generator(GF(2, 128)));
true
> print IsPrimitive(Generator(GF(2, 100)));
false
```

Given a field F, the function **PrimitiveElement**(F) will return a primitive element of F, even if the Conway polynomial of F is not known.

24.3 Extensions of Finite Fields

24.3.1 Extensions by a Given Defining Polynomial

Every Galois field E of size p^n, where $n > 1$, is a simple algebraic extension of some field F of size p^m, where $m < n$ and m divides n. F is called the *ground field* of E, and is isomorphic to a subfield of E. If F is not a prime field, that is, if $m > 1$, then it is in turn an extension of a smaller field with the same characteristic p. This chain will eventually terminate at GF(p), the prime field of E. The integer $d = \frac{m}{n}$ is the *degree of the extension field E relative to the ground field F*, whereas n is the *absolute degree of E*, that is, its degree relative to the prime field.

The field E is created as a simple algebraic extension of degree d of F. In other words, a root of an irreducible polynomial f of degree d with coefficients in F is adjoined to F in order to create E. (For any given finite field and degree, there always exists at least one irreducible polynomial of degree d over the field.) In MAGMA, the **ext** constructor is used to create an extension. One form of this constructor is

ext< F | f **>**

If F is a finite field and f is an irreducible polynomial f of degree d with coefficients in F, this creates an extension of degree d of the ground field F, with f as its *defining polynomial*.

For example, suppose that the user wishes to construct a degree-4 extension of *gf9* (as defined above) using the defining polynomial $f = x^4+g^6x^2+g^5$ over *gf9*:

```
> gf9<g> := GF(9);
> Pgf9<x> := PolynomialRing(gf9);
> f := x^4 + g^6*x^2 + g^5;
```

A check should be made that f is irreducible, and then the extension may be defined:

```
> print IsIrreducible(f);
true
> gf9e4<alpha> := ext< gf9 | f >;
> print gf9e4;
Finite field of size 3^8
```

The new extension, *gf9e4*, has *alpha* as its generator. This generator is a root of f, the defining polynomial, so if it is substituted into f, the result is zero:

```
> print Evaluate(f, alpha);
0
```

In fact, although f has no roots over its own coefficient field, the corresponding polynomial with coefficients in the extension field has $d = 4$ roots, including *alpha*:

```
> print Roots(f);
[]
> print Roots(PolynomialRing(gf9e4)!f);
[ <2*alpha, 1>, <alpha^3 + alpha, 1>, <alpha, 1>,
<2*alpha^3 + 2*alpha, 1> ]
```

The functions **PrimeField**, **GroundField** and **Degree** may be used to gain information about a field. Continuing the example:

```
> print PrimeField(gf9e4);
Finite field of size 3
> print GroundField(gf9e4);
Finite field of size 3^2
> print Degree(gf9e4);
8
> print Degree(gf9e4, gf9);
4
```

Note that **Degree** has two versions: **Degree**(E) is the absolute degree n over the prime field of a field E with p^n elements, and **Degree**(E, F) is the degree $d = \frac{m}{n}$ of E considered as an extension of a field F with p^m elements.

It is possible to avoid explicitly creating the polynomial ring for the defining polynomial by using the constructor **ExtensionField**, a variant of the **ext** constructor. It has the form

ExtensionField< F, indeterminate | f **>**

where the polynomial f is expressed in terms of the indeterminate. For instance, the following statement demonstrates how to create *gf9e4* without first creating *Pgf9* and its indeterminate x:

```
> gf9e4<alpha>:=ExtensionField< gf9, x | x^4+g^6*x^2+g^5 >;
```

The indeterminate (x in this example) does not remain defined as a identifier or printname. It is only used for stating the defining polynomial.

24.3.2 Extensions by a Given Degree

Another way of creating an extension E of a finite field F is to specify the required degree d instead of the defining polynomial in the **ext** constructor:

ext< F | degree d of extension over ground field F **>**

When the **ext** constructor is used in this fashion, MAGMA selects a suitable defining polynomial for the extension. The irreducible polynomial chosen, which is available from the function **DefiningPolynomial**(E), is the same as the polynomial returned by **IrreduciblePolynomial**(F, d). For instance, *gf9e4e3* may be constructed as a degree-3 extension of *gf9e4*, without specifying a defining polynomial:

```
> gf9e4e3<beta> := ext< gf9e4 | 3 >;
> print gf9e4e3;
Finite field of size 3^24
> defgf9e4e3 := DefiningPolynomial(gf9e4e3);
> print defgf9e4e3;
$.1^3 + (g^6*alpha^3 + g^7*alpha^2 + g)*$.1^2
    + (g*alpha^3 + 2*alpha^2 + 2)*$.1
    + g*alpha^3 + g^5*alpha^2 + g^7*alpha + g^7
> print defgf9e4e3 eq IrreduciblePolynomial(gf9e4, 3);
true
```

Notice from the output form of *defgf9e4e3* that there is no printname for the indeterminate in the defining polynomial, so the special symbol **$.1** is employed. If the user wishes to give the indeterminate a printname and assign it to an identifier, there are two methods: either the parent polynomial ring may be defined, together with its indeterminate, or (more simply) angle brackets may be used when assigning *defgf9e4e3*:

```
> defgf9e4e3<y> := DefiningPolynomial(gf9e4e3);
> print defgf9e4e3;
y^3 + (g^6*alpha^3 + g^7*alpha^2 + g)*y^2 +
    (g*alpha^3 + 2*alpha^2 + 2)*y +
    g*alpha^3 + g^5*alpha^2 + g^7*alpha + g^7
```

Both forms of **ext**, as well as **ExtensionField**, optionally return a second value, which is the embedding monomorphism $\phi : F \to E$. This map has the same effect as coercion of an element of F into E using the ! operator.

24.3.3 Splitting Fields

The defining polynomial for an extension formed using the **ext** constructor has to be irreducible. However, it is also possible to form an extension using a non-irreducible polynomial. Given any polynomial p over F, **SplittingField**(p) constructs the smallest-degree extension E of F such that the polynomial p factors completely into linear factors when considered as a polynomial over E.

For instance, suppose p is the polynomial defined below. The coefficient field of p is *gf9*. Over this field, p has two non-linear irreducible factors, but no roots:

```
> p := MinimalPolynomial(alpha^3+alpha, gf9) *
>     MinimalPolynomial(g*beta^2, gf9);
> print Factorization(p);
[
    <x^4 + g^6*x^2 + g^5, 1>,
    <x^12 + g^3*x^11 + g^2*x^10 + g*x^8 + g^6*x^7 + g^3*x^6
        + g^2*x^5 + x^4 + 2*x^3 + g*x^2 + g^6*x + g^2, 1>
]
> print Roots(p);
[]
```

Over the field *gf9e4*, which contains $alpha^3 + alpha$, p has the 4 roots corresponding to the degree-4 irreducible factor shown above:

```
> print Roots(PolynomialRing(gf9e4)!p);
[ <2*alpha, 1>, <alpha^3 + alpha, 1>, <alpha, 1>,
    <2*alpha^3 + 2*alpha, 1> ]
```

However, over the splitting field of p, the degree-16 polynomial p has 16 roots:

```
> Spl := SplittingField(p); print Spl;
Finite field of size 3^24
> print Roots(PolynomialRing(gf9e4e3)!p);
[
    <(g*alpha^3 + g^6*alpha^2 + g^3*alpha + g^5)*beta^2
        + (2*alpha^3 + g^3*alpha^2 + alpha + 1)*beta
        + g*alpha^3 + g^6*alpha + g^5, 1>,
    <2*alpha, 1>,
    <(g*alpha^3 + g^2*alpha^2 + g^7*alpha + g^5)*beta^2
        + (g^3*alpha^3 + g^6*alpha^2 + g^3*alpha + 1)*beta
        + alpha^3 + g^7*alpha^2 + 2*alpha + g^6, 1>,
    <(2*alpha^3 + g*alpha^2 + g^7*alpha)*beta^2
        + (alpha^3 + g^3*alpha + g^3)*beta
        + g^3*alpha^3 + g^7*alpha^2 + g*alpha + g^2, 1>,
    <2*alpha^3 + 2*alpha, 1>,
    <alpha, 1>,
    <(alpha^3 + g^5*alpha^2 + g^3*alpha + g^5)*beta^2
        + (2*alpha^3 + g^7*alpha + g^7)*beta
        + g^3*alpha^3 + g*alpha^2 + g^2*alpha + 2, 1>,
    <(alpha^3 + g*alpha^2 + g^7*alpha + g^2)*beta^2
        + (2*alpha^3 + g^7*alpha^2 + alpha + g^2)*beta
        + g^3*alpha^3 + g^6*alpha^2 + g^6*alpha + 2, 1>,
    <(g^6*alpha^3 + g^3*alpha^2 + 2*alpha + g^7)*beta^2
        + (g^6*alpha^3 + 2*alpha^2 + g)*beta
        + 2*alpha^3 + g^7*alpha^2 + g*alpha + g^3, 1>,
    <(g*alpha^3 + g^7*alpha^2 + g^3*alpha + g)*beta^2
        + (g^7*alpha^3 + alpha^2 + g^5*alpha + g^6)*beta
        + g^3*alpha^2 + alpha + g, 1>,
    <(g^3*alpha^3 + g^2*alpha^2 + g^5*alpha + 1)*beta^2
        + (g^7*alpha^3 + g^7*alpha^2 + 2*alpha + 1)*beta
        + g^2*alpha^3 + g^3*alpha^2 + g*alpha + g^6, 1>,
    <(alpha^3 + g^5*alpha^2 + g*alpha + 1)*beta^2
        + (g^7*alpha^3 + g*alpha^2 + 2*alpha + g^6)*beta
        + g^7*alpha^3 + g*alpha^2 + g*alpha + 2, 1>,
    <(g^6*alpha^3 + g^2*alpha^2 + g^7)*beta^2
        + (g^6*alpha^3 + g^5*alpha^2 + g^7*alpha)*beta
        + g^3*alpha^3 + alpha^2 + g*alpha + 2, 1>,
    <g*beta^2, 1>,
    <(2*alpha^3 + g^2*alpha^2 + alpha + g^7)*beta^2
```

```
        + (2*alpha^2 + g*alpha + 1)*beta
        + g^7*alpha^3 + g^2*alpha^2 + g^3*alpha, 1>,
    <alpha^3 + alpha, 1>
]
```

In this collection of roots, the 4 roots found previously appear once again. As expected from the construction of the polynomial, $alpha^3 + alpha$ and $g \cdot beta^2$ are among the roots.

Given a set T of polynomials over a finite field F, **SplittingField**(T) creates the smallest-degree extension field E such that every polynomial in T factors completely into linear factors over E.

24.3.4 Roots of Unity in Extension Fields

Given an $n \in \mathbb{Z}$ and a finite field K, the function **RootOfUnity**(n, K) returns a primitive n^{th} root of unity in the smallest possible extension field of K. For example:

```
> gf9<g> := GF(9);
> r := RootOfUnity(23, gf9);
> L<e> := Parent(r);
> print L;
Finite field of size 3^22
> print r;
g^3*e^10 + g^5*e^9 + g*e^8 + g^3*e^7 + g^3*e^6 + g^5*e^5 +
    2*e^4 + e^3 + g^2*e^2 + g^2*e + g^3
> print Order(r);
23
```

24.4 Subfields of Finite Fields

The operation approximately opposite to the creation of an extension is the creation of a subfield. The **sub** constructor is used for this purpose, but in the case of finite fields, it has two versions (like the **ext** constructor):

sub< field | a **>**

where a is an element of the field, and

sub< field | m **>**

where m is a positive integer dividing the absolute degree of the original field. The first of these returns the subfield generated by a, and the second returns a subfield of absolute degree m. Both forms of **sub** optionally return the embedding monomorphism from the subfield to the main field as a second value. This map has the same effect as a standard coercion.

For instance, the following lines create two fields of size 3^4, as subfields of *gf9e4e3*:

```
> gf9e4e3s4A<gammaA> := sub< gf9e4e3 | alpha^2 >;
> gf9e4e3s4B<gammaB> := sub< gf9e4e3 | 4 >;
> print gf9e4e3s4A;
Finite field of size 3^4
> print gf9e4e3s4B;
Finite field of size 3^4
```

Although these fields are isomorphic, because they have the same size, they happen to have different defining polynomials, and their generators are not equal. The coercion mapping gives the isomorphism between the fields:

```
> print DefiningPolynomial(gf9e4e3s4A);
$.1^4 + 2*$.1 + 2
> print DefiningPolynomial(gf9e4e3s4B);
$.1^4 + 2*$.1^3 + 2
> print gammaA eq gammaB;
false
> print gf9e4e3s4A ! gammaB;
gammaA^31
> print gf9e4e3s4B ! gammaA;
gammaB^31
```

With respect to the field *gf9e4*, the generator of *gf9e4e3s4A* is $alpha^2$, as would be expected, but the generator of *gf9e4e3s4B* is $g^7\,alpha^2$:

```
> print gf9e4 ! gammaA;
alpha^2
> print gf9e4 ! gammaB;
g^7*alpha^2
```

24.5 Embedding a Finite Field in Another Finite Field

When an extension or subfield is created, MAGMA automatically creates an embedding monomorphism from the smaller field into the bigger field. This

monomorphism permits the user to perform coercions of elements, test relationships between fields, and so on. For instance:

```
> print gf9e4e3s4A subset gf9e4e3;
true
> print alpha in gf9e4e3s4B;
false
```

However, if two fields F and F' are created independently of one another, then MAGMA will not see them as related. Even if they have the same characteristic p, and the degree of F divides the degree of F'. MAGMA will not be aware of the mathematical relationship between them. Error messages will arise if the user attempts any operations involving both F and F'. For example:

```
> gf9e4e11<delta> := ext< gf9e4 | 11 >;
> print gf9e4e11;
Finite field of size 3^88
> print gf9e4e3s4A;
Finite field of size 3^4
> print gf9e4e3s4A subset gf9e4e11;

>> print gf9e4e3s4A subset gf9e4e11;
                     ^
Runtime error in 'subset': Arguments have no covering field
> print gf9e4e11 ! gammaA;

>> print gf9e4e11 ! gammaA;
                  ^
Runtime error in '!': Illegal coerce
```

To solve this problem, F must be embedded in F', using the procedure **Embed**(F, F'). This procedure sets up an isomorphism between F and the subfield of F' with the same cardinality:

```
> Embed(gf9e4e3s4A, gf9e4e11);
```

Now MAGMA is aware that *gf9e4e3s4A* is a subfield of *gf9e4e7*. After an embedding has taken place, computations and tests involving the two fields may be performed:

```
> print gf9e4e3s4A subset gf9e4e11;
true
> print gf9e4e11 ! gammaA;
g^3*alpha^2 + 1
> print Degree(gf9e4e11), Degree(gf9e4e11, gf9e4e3s4A);
```

```
88 22
```

The embedding isomorphism is always chosen so that diagrams 'commute', that is, so that all the homomorphisms involving the fields are compatible.

In MAGMA, the **ext** constructor generates a different field each time it is called, even if the arguments given to the constructor are exactly the same. This facility allows the user to create two fields which are isomorphic but notionally distinct. If a relationship between the two fields is to be established the **Embed** procedure must be invoked:

```
> F1 := ext< gf9 | 5 >;
> F2 := ext< gf9 | 5 >;
> print F1 eq F2;

>> print F1 eq F2;
             ^
Runtime error in 'eq': Arguments have no covering field
> Embed(F1, F2);
> print F1 eq F2;
true
> print DefiningPolynomial(F1) eq DefiningPolynomial(F2);
true
> print F1.1 eq F2.1;
false
> print F2 ! (F1.1);
F2.1^81
> print F1 ! (F2.1);
F1.1^729
```

Although these two fields were created with the same expression, and their defining polynomials are the same, their generators do not correspond in the isomorphism. For each field, the algorithm involved has randomly selected one of the five roots of the defining polynomial to be the generator of the field.

The **GF** function is an exception. Every call to **GF** creates the default field of the given size, so all fields created as $\mathbf{GF}(p, n)$ or $\mathbf{GF}(q)$, for the same p, n pair or the same $q = p^n$, are identical:

```
> F3 := GF(7, 3);
> F4 := GF(7, 3);
> print F3 eq F4;
true
> print F3.1 eq F4.1;
true
```

There is a variant of the **Embed** function in which the user can specify the image of the generator. **Embed**(F, F', x) embeds F in F' in such a way that the embedding isomorphism maps the generator of F to the element x of F'.

24.6 Associated Polynomials

Table 24.1. Polynomials associated with finite fields

MAGMA	Meaning
DefiningPolynomial(F)	Polynomial over the ground field of F, defining F as an extension of the ground field
DefiningPolynomial(F, S)	Polynomial over subfield S of F, defining F as an extension of S
IrreduciblePolynomial(F, m)	Irreducible polynomial of degree m over F
AllIrreduciblePolynomials(F, m)	Set of the monic irreducible polynomials of degree m over F
IsIrreducible(f)	**true** if univariate polynomial f over F is irreducible
PrimitivePolynomial(F, m)	A primitive polynomial of degree m over F (i.e., irreducible, and having a primitive root of the degree-m extension field of F as a root)
IsPrimitive(f)	**true** if univariate polynomial f over F is primitive
MinimalPolynomial(a)	Minimal polynomial of $a \in F$ over ground field of F
MinimalPolynomial(a, S)	Minimal polynomial of $a \in F$ over subfield S of F
CharacteristicPolynomial(a)	Characteristic polynomial of $a \in F$ over ground field of F
CharacteristicPolynomial(a, S)	Characteristic polynomial of $a \in F$ over subfield S of F

Several polynomials are associated with finite fields, including defining polynomials, irreducible polynomials and Conway polynomials as explained above. Table 24.1 (p. 432) gives a list of the polynomial functions for finite fields.

A few of these functions resemble the **Degree** function in that they can take one argument or two arguments. If only one argument is given, the

operation is performed with respect to the ground field, whereas if two arguments are given, the operation is performed with respect to the specified subfield. (However, if no subfield is stated in the **Degree** function, it is understood to be the prime field, not the ground field.) For instance, the function **MinimalPolynomial** returns the minimal polynomial of a given element a of the finite field F. If a subfield S of F is given as the second argument, the value returned is the minimal polynomial over the coefficient field S. Otherwise, it is the minimal polynomial over the ground field of F:

```
> e := beta^3 + 2*g*beta + alpha;
> print MinimalPolynomial(e); // i.e., over gf9e4
y^3 + (2*alpha^3 + alpha^2 + g^2*alpha + g^2)*y^2 +
    (alpha^3 + g^6*alpha^2 + alpha)*y + g^6*alpha^3 + alpha
> print MinimalPolynomial(e, gf9);
x^12 + 2*x^10 + g^7*x^8 + g^6*x^7 + g^6*x^6 + 2*x^5 + x^4 +
    g^7*x^3 + 2*x^2 + g^5*x + g^3
> print MinimalPolynomial(e, GF(3));
$.1^24 + $.1^22 + $.1^18 + $.1^17 + $.1^16 + 2*$.1^15 +
    $.1^14 + 2*$.1^11 + 2*$.1^10 + 2*$.1^9 + 2*$.1^7 +
    2*$.1^5 + 2*$.1^4 + $.1^2 + 2
> print MinimalPolynomial(e, gf9e4e3);
$.1 + (g^6*alpha^3 + g^7*alpha^2 + g)*beta^2 +
    (g*alpha^3 + 2*alpha^2 + g^7)*beta + g*alpha^3 +
    g^5*alpha^2 + g^2*alpha + g^7
```

Note that the degree of this polynomial indicates the degree of the corresponding extension. In the last example, the given element e is in the given subfield, so the polynomial is linear.

The minimal polynomial of the generator of a field F is the defining polynomial of F. For example:

```
> print MinimalPolynomial(beta) eq defgf9e4e3;
true
```

Some remarks should now be made on Conway polynomials. As was shown above, any irreducible polynomial of degree d over a finite field F may be used in MAGMA to create an extension of F of degree d. Since this freedom can be problematic, there has been defined, for each moderately small prime p and exponent n, a special irreducible polynomial of degree n with coefficients in GF(p). It is known as the *Conway polynomial* $C_{p,n}$. (The details of the definition are given in the *Handbook*.) MAGMA contains a list of the known Conway polynomials. In order to avoid the ambiguities that might arise when embeddings take place, MAGMA uses a Conway polynomial (if it is known) whenever a field is being defined as an extension of a prime field. For example:

Table 24.2. Conway polynomial functions

MAGMA	Meaning
ConwayPolynomial(p, n)	Degree-n Conway polynomial over GF(p)
ExistsConwayPolynomial(p, n)	If a Conway polynomial of degree n over GF(p) is known, returns **true** and the polynomial, else returns **false**
IsConway(F)	**true** if defining polynomial of F relative to its prime field is a Conway polynomial

```
> print ConwayPolynomial(5, 6) eq DefiningPolynomial(gf5e6);
true
```

Table 24.2 contains the Conway polynomial functions.

24.7 Operations on Finite Field Elements

MAGMA has functions for several standard operations on elements of finite fields. Some of them are listed in Table 24.3. Notice that for those functions that take an optional subfield as a second argument, whenever the subfield is omitted the default subfield is the prime field, not the ground field as in the polynomial functions explained above.

One of the most important functions for finite field elements is **Log**(a). It uses the primitive element $w \in F$ that MAGMA returns from the function **PrimitiveElement**(F). By definition of w, every non-zero $a \in F$ may be expressed as w^n for a unique $n \in \{0, \ldots, \#F - 2\}$. **Log**$(a)$ returns this value n. For instance:

```
> pr := PrimitiveElement(gf9e4);
> print pr;
alpha^3 + g^5*alpha^2 + g^2*alpha + g^7
> print Log(alpha);
4879
> print pr^4879;
alpha
> print Log(gf9e4!g);
2460
> print pr^2460;
g
```

The **Log** function only applies to moderately small fields, for which MAGMA can calculate the complete table of logarithms. These logarithms are called

Table 24.3. Functions on elements of finite fields

MAGMA	Meaning
Norm(a)	Absolute norm of a (norm relative to prime field)
Norm(a, S)	Norm of a relative to subfield S
Trace(a)	Absolute trace of a (trace relative to prime field)
Trace(a, S)	Trace of a relative to subfield S
Log(a)	Discrete logarithm of a ($\neq 0$), with 'base' the defining primitive element w, i.e., n such that $w^n = a$
Order(a)	Multiplicative order of a ($\neq 0$)
FactoredOrder(a)	Multiplicative order of a ($\neq 0$) in factored form (as a factorization sequence)
Sqrt(a)	A square root of a, if an answer exists
IsSquare(a)	If a is square, returns **true** and a square root, else returns **false**
Root(a, n)	An n^{th} root of a, if an answer exists
Eltseq(a)	Length-d sequence containing the coefficients of $a \in F$ as a polynomial over S in the generator of F over S, where S is the ground field of F and d is the degree of F over S
Eltseq(a, S)	Length-d sequence containing the coefficients of $a \in F$ as a polynomial over S in the generator of F over S, where S is a subfield of F and d is the degree of F over S
Eltseq(a)	Sequence as for **Eltseq**(a, S), where S is the ground field of F
Seqelt(Q, F)	Given a sequence Q as described for **Eltseq**, returns the corresponding element of F

Zech logarithms, and they allow arithmetic in the field to be done very efficiently by MAGMA.

ElementToSequence(a, S) and **SequenceToElement**(Q, S), which are commonly abbreviated to **Eltseq**(a, S) and **Seqelt**(Q, F), convert between an element a of a finite field F and its representation as a polynomial in the generator of F over a subfield S, with coefficients in S. **Eltseq** returns the coefficients of the polynomial in a length-d sequence Q, where d is the degree of F over S, and **Seqelt** returns the field element a. If **Eltseq** is used in the form **Eltseq**(a), then the subfield S is understood to be the ground field of F. For instance:

```
> print Eltseq(pr);
[ g^7, g^2, g^5, 1 ]
> print Eltseq(pr, GF(3));
[ 0, 0, 2, 0, 0, 2, 2, 1 ]
> print Seqelt([ 2*alpha, 1, g*alpha], gf9e4e3);
```

```
g*alpha*beta^2 + beta + 2*alpha
```

24.8 Printing Finite Field Elements

There are two ways of printing an element of a finite field in MAGMA. It can either be printed as a power of the generating element, or else as a polynomial in the generator over the ground field, having degree less than the relative degree of the field. For instance, in *gf9*, $2g + 2$ and g^6 are different ways of expressing the same element:

```
> print (2*g + 2) eq g^6;
true
```

Note that the issue does not arise for prime fields, because their elements are printable as 'integers'.

MAGMA is always able to express elements in the polynomial form. (This form resembles the vector representation explained in the next section and the **Eltseq** representation discussed above.) However, it can only express them as powers of the generator if the field is small enough for the Zech logarithms to be stored. To specify the default style of printing preferred by the user for all subsequently-created (small) finite fields, the procedure **AssertAttribute**(**FldFin**, `"PowerPrinting"`, b) should be used. The argument b takes a Boolean value: it should be **true** if elements are to be printed as powers of the generator, and **false** if they are to be printed as polynomials. However, this procedure does not change the printing style for any fields that have been already created. To change the printing style for a particular small and non-prime field F, the procedure **AssertAttribute**(F, `"PowerPrinting"`, b) may be used, with b a Boolean as before.

To find the current value of the attribute `"PowerPrinting"` for a field F or for the **FldFin** category, the user should call the **HasAttribute** function, as demonstrated below:

```
> fld := GF(81);
> print HasAttribute(fld, "PowerPrinting");
true true

> print HasAttribute(FldFin, "PowerPrinting");
true true
```

This function, like the procedure **AssertAttribute**, is used for several attributes throughout the MAGMA system. Its first return value is a Boolean

indicating whether the attribute is defined for the object in question; this value will be **true** except for prime fields and fields too large for the Zech algorithms to be stored. If the attribute is defined, then the function also returns the value of the attribute.

The following example illustrates the use of the "PowerPrinting" attribute:

```
> print HasAttribute(FldFin, "PowerPrinting");
true true
> gf9<g> := GF(9);
> print Set(gf9);
{ 1, g, g^2, g^3, 2, g^5, g^6, g^7, 0 }
> AssertAttribute(FldFin, "PowerPrinting", false);
> gf8<h> := GF(8);
> print HasAttribute(gf8, "PowerPrinting");
true false
> print HasAttribute(gf9, "PowerPrinting");
true true
> print Set(gf8), Set(gf9);
{ 1, h, h^2, h + 1, h^2 + h, h^2 + h + 1, h^2 + 1, 0 }
{ 1, g, g^2, g^3, 2, g^5, g^6, g^7, 0 }
> gf4<e> := GF(4);
> print Set(gf4);
{ 1, e, e + 1, 0 }
> AssertAttribute(gf4, "PowerPrinting", true);
> print Set(gf4);
{ 1, e, e^2, 0 }
```

24.9 Finite Fields as Vector Spaces

If E is a degree-d extension of F, then there is a natural isomorphism between E and the d-dimensional vector space $E^{(d)}$. Every element of E may be written uniquely as a linear combination of the basis elements $1, \omega, \omega^2, \ldots, \omega^{d-1}$, where ω is the generator of E over F and the scalars are elements of F. The function **VectorSpace**(E, F) returns this vector space, and the corresponding isomorphism from E to the vector space. For instance:

```
> V, phi := VectorSpace(gf9e4, gf9);
> print V, phi;
Full Vector space of degree 4 over GF(3^2)
Mapping from: FldFin: gf9e4 to ModTupFld: V
```

The mapping *phi* converts a given element of *gf9e4* to a vector of V:

```
> print (g^5*alpha^3 + g^5*alpha^2 + g^3*alpha + 2 ) @ phi;
(  2 g^3 g^5 g^5)
```

The correspondence between the element and its vector space representation is obvious.

Instead of using the powers of the generator as a basis, the user may choose a basis for the vector space. If B is a suitable sequence of elements of F, then **VectorSpace**(E, F, B) returns the vector space and isomorphism for that basis.

For example, a *normal basis* for E over F is a basis of the form $a, a^q, \ldots, a^{q^{d-1}}$. Such a basis has the property that exponentiation of an element by a power of 2 can be achieved by rotating the vector representing that element. If $a \in E$ forms a normal basis, it is called a normal element.

The next example shows how to construct a normal basis for a very large field, the field with 2^{1000} elements. Firstly, the field must be constructed:

```
> K<w> := FiniteField(2, 1000);
```

The function **IsNormal**(a) tests whether a forms a normal basis over the prime field, and **NormalElement**(E) returns a normal element for the field E considered over its prime field. (For calculations relative to an arbitrary subfield S, the functions **IsNormal**(a, S) and **NormalElement**(E, S) should be used instead.) In this example, it will be convenient if the generator of K is normal, so this should be tested:

```
> print IsNormal(w);
false
```

Unfortunately, w is not normal. Therefore it is necessary to find an element e which is normal, and then form the basis B:

```
> e := NormalElement(K);
> B := [IsZero(i) select e else Self(i)^2 : i in [0..999]];
```

Now that B is known, the vector space may be constructed:

```
> VS, iso := VectorSpace(K, GF(2), B); print VS, iso;
Full Vector space of degree 1000 over GF(2)
Mapping from: FldFin: K to ModTupFld: VS
```

Since VS is a vector space with a normal basis, there should be a strong relationship between the vectors corresponding to r and $r^{(2^t)}$, for any field element r and integer t. In fact, the first vector rotated t places will equal the second vector, as the following output verifies:

```
> r := Random(K); t := Random(1, 10);
> r2t := r^(2^t);
> rVS := r @ iso;
> r2tVS := r2t @ iso;
> print r2tVS eq VS!Rotate(Eltseq(rVS), t);
true
```

24.10 Finite Fields as Matrix Algebras

Suppose E is a degree-d extension of a finite field F. (F does not have to be the ground field of E; it may be isomorphic to any subfield of E.) As was shown above, E is isomorphic to the d-dimensional vector space $E^{(d)}$. However, a stronger mathematical statement can be made. E is also isomorphic to a matrix algebra of degree d and dimension d over F. The advantage of this representation is that field elements as matrice can be multiplied and inverses can be found, whereas field elements as vectors can only be added and made negative. Of course, elements of E which are in the ground field F correspond to multiples of the identity matrix, so they still have a scalar action. The function **MatrixAlgebra**(E, F) returns this matrix algebra A, and the corresponding isomorphism from E to A, such that the generator of E maps to the generator of A. For instance:

```
> M<G>, isoM := MatrixAlgebra(gf9e4, gf9);
> print M, isoM;
Matrix Algebra of degree 4 and dimension 4 with 1 generator
    over GF(3^2)
Mapping from: FldFin: gf9e4 to AlgMat: M
> print G eq isoM(alpha);
true
> print isoM(alpha^2+1) eq G^2+1;
true
```

Observe that G, the generator of M, is isomorphic to *alpha*, the generator of *gf9e4*. The isomorphism is *isoM*.

The generator of the matrix algebra is constructed as the companion matrix of the field's defining polynomial. It has a line of 1s above the main diagonal, and 0s elsewhere, except for the last row.

```
> print G;
[  0   1   0   0]
[  0   0   1   0]
[  0   0   0   1]
```

```
[  g   0 g^2   0]
> defgf9e4 := DefiningPolynomial(gf9e4); print defgf9e4;
x^4 + g^6*x^2 + g^5
> print G eq CompanionMatrix(defgf9e4);
true
```

The last row of the generating matrix contains the coefficients of the negative of the defining polynomial modulo x^d. In the present example, the last row represents the polynomial $g + 0x + g^2x^2 + 0x^3$:

```
> print -defgf9e4 mod x^4;
g^2*x^2 + g
> print Coefficients(-defgf9e4 mod x^4);
[ g, 0, g^2 ]
```

In general, the top row of the matrix indicates the vectorial representation of the element as a sum of powers of the generator up to $d-1$. For instance:

```
> m := 2 + G^2 + G^3; print m;
[  2   0   1   1]
[  g   2 g^2   1]
[  g   g   g g^2]
[g^3   g g^7   g]
```

Functions such as **MinimalPolynomial**, **CharacteristicPolynomial** and **Order** can be applied to elements of the matrix algebra just as for field elements:

```
> print Order(m);
656
> print Order(2 + alpha^2 + alpha^3);
656
```

There is a further way of constructing a matrix algebra that involves finite fields. If A is a matrix algebra A over a finite field E, and S is a subfield of E, then there exists another matrix algebra N over S which is isomorphic to A. N is constructed by expanding each entry of a matrix of A into the block matrix associated with it.d The function **MatrixAlgebra**(A, S) returns this matrix algebra, and the corresponding isomorphism from E to the matrix algebra, such that the generator of E maps to the generator of the matrix algebra. For instance:

```
> gf3 := GF(3);
> N<GN>, isoN := MatrixAlgebra(M, gf3);
> print N;
```

```
Matrix Algebra of degree 8 with 1 generator over GF(3)
> print GN;
[0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
[0 1 0 0 1 1 0 0]
[1 1 0 0 1 2 0 0]
> print GN eq isoN(G);
true
```

Each 2×2 block in GN corresponds to an entry of G. Most obviously, a 2×2 zero matrix corresponds to a 0 and a 2×2 identity matrix corresponds to a 1. In fact, the correspondence comes from yet another matrix algebra, the one relating *gf9* to *gf3*:

```
> M9rel3, isoM9rel3 := MatrixAlgebra(gf9, gf3);
> print isoM9rel3(g); // block corresponding to g
[0 1]
[1 1]
> print isoM9rel3(g^2); // block corresponding to g^2
[1 1]
[1 2]
```

25. Number Fields

MAGMA contains very sophisticated machinery for computing in algebraic number fields. To the casual user, the specialist features may not be of immediate interest. Therefore the earlier sections of this chapter deal mainly with the easier issues of basic constructions and arithmetic, whereas subsequent sections treat more complicated *structural* computations relating to number fields and their orders (maximal order, ideal class group, unit group, and Galois group).

A *number field* is by definition a finite algebraic extension of the rational field $\mathbb{Q}$. This means that it is a field containing the rational numbers, in which every element satisfies an equation of the form $p(x) = 0$, where p is a polynomial with rational coefficients. In fact, any number field N consists of all elements of the form $a_0 + a_1\alpha + \cdots + a_{n-1}\alpha^{n-1}$, for rational numbers a_i, some non-negative integer n, and a fixed α (called a *primitive element* for N). The element α can always be chosen such that $f(\alpha) = 0$ for some monic irreducible polynomial f with *integer* coefficients; in that case n above will equal the degree of f and is called the *degree* of the number field. The field N is then usually denoted $\mathbb{Q}(\alpha)$, and is entirely determined by the defining polynomial f. As a ring, N is isomorphic to $\mathbb{Q}[x]/f(x)$, the quotient of the polynomial ring over $\mathbb{Q}$ and the ideal generated by f.

For example, if f equals $x^3 - 2$, an irreducible monic polynomial of degree 3, a cubic number field may be obtained by *adjoining* a root α of f to $\mathbb{Q}$. Such a root is usually referred to as $\sqrt[3]{2}$, and the resulting field $\mathbb{Q}(\sqrt[3]{2})$ consists of all rational combinations $a_0 + a_1\sqrt[3]{2} + a_2\sqrt[3]{2}^2$. Higher powers of $\sqrt[3]{2}$ can simply be reduced, using the fact that the third power equals 2.

For the sake of efficiency, number fields in MAGMA are implemented in four different categories: the field of rational numbers; quadratic number fields; cyclotomic number fields; and 'general' number fields. Since rational numbers and ordinary integers have been discussed in detail in Chapter 20, they will only be mentioned in this chapter occasionally. Most functions for number fields apply to the rational field as well, with trivial results.

The simplest number field is $\mathbb{Q}$ itself; the MAGMA category for it is **FldRat**. Next in terms of complexity are quadratic fields, defined by an

irreducible polynomial f of degree 2. These are the familiar fields obtained by adjoining the square root of an integer to the rational field, such as $\mathbb{Q}(\sqrt{5})$. Quadratic fields have their own category, **FldQuad**, to take advantage of the existence of faster algorithms than those available for general number fields, which form the category **FldNum**. There is one more category of special number fields, namely that of cyclotomic fields, **FldCyc**. These are obtained by adjoining a root of unity to $\mathbb{Q}$, that is, an element ζ satisfying $\zeta^n = 1$ for some $n > 2$. Cyclotomic fields can have arbitrary degree, but because their elements are combinations of roots of unity they arise frequently, for example in representation theory. Quadratic fields, cyclotomic fields and general number fields will be described in detail below.

Besides these fields, this chapter also discusses *orders*, which arise as subrings of the ring of integers of a field. They have their own MAGMA categories, **RngQuad** and **RngOrd**, corresponding to the quadratic and the general number fields, respectively.

25.1 Elementary Constructions of Number Fields

This section explains how to create number fields and their elements in the various categories. The constructions are kept at their simplest level; more sophisticated constructions are discussed later in the chapter.

25.1.1 Number Fields

Table 25.1. Number field functions

MAGMA	Meaning
NumberField(f)	Number field $\mathbb{Q}(\alpha)$, where α is a root of the monic irreducible integer polynomial f
Degree(K)	Absolute degree $m = [K : Q]$ of number field K
DefiningPolynomial(K)	Monic irreducible degree-m polynomial defining K
Discriminant(K)	Discrimant of K
ReducedDiscriminant(K)	Reduced discrimant of K
Signature(K)	Signature r, s of K
PrimitiveElement(K)	A primitive element for K over ground field
Eltseq(a)	Sequence $[a_0, a_1 \ldots, a_{m-1}]$, where $a = a_0 + a_1\alpha + \cdots + a_{m-1}\alpha_m^{m-1}$

The number field $K = \mathbb{Q}(\alpha)$ is defined to be the field obtained by adjoining to the rational field $\mathbb{Q}$ a root α of the monic irreducible integer polynomial f. K may be created in MAGMA using the function **NumberField**(f), where f is a monic polynomial over $\mathbb{Z}$ or $\mathbb{Q}$ which is irreducible over $\mathbb{Q}$. If the coefficients of f are all integral, a root of f is adjoined to $\mathbb{Q}$ to create the extension K; otherwise, if denominators greater than 1 occur among the coefficients of f, an equivalent integral polynomial is used. For example, the following lines create the field $K = \mathbb{Q}(\sqrt{2} + \sqrt{5})$, by adjoining to $\mathbb{Q}$ a root *alpha* of the polynomial $f = x^4 - 14x^2 + 9$ over $\mathbb{Q}$:

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^4 - 14*x^2 + 9;
> K<alpha> := NumberField(f);
> print K;
NumberField( alpha^4 - 14*alpha^2 + 9 )
> print Degree(K);
4
```

Here *alpha* is an identifier whose value is the root that has been adjoined, and also the printname for this root.

The easiest way to construct elements of K is in terms of *alpha*. Arithmetic on these elements involves the usual operations.

```
> t := alpha^2 + 5;
> print 5*alpha*t^3 - 2*t + 4*alpha;
2360*alpha^3 - 2*alpha^2 - 676*alpha - 10
```

Another way to create elements of K is to coerce into K a sequence of integers or rationals whose length is the degree of K:

```
> print K![5, 0, 1, 0];
alpha^2 + 5
```

When elements are created in this way, MAGMA interprets the contents of the sequence as coefficients of a polynomial in α, starting with the constant term. If the sequence is $[a_0, a_1 \ldots, a_{m-1}]$ then the corresponding element is $a_0 + a_1\alpha + \cdots + a_{m-1}\alpha^{m-1}$. The function **Eltseq**(a) reverses this process:

```
> print Eltseq(2*t + 3);
[ 13, 0, 2, 0 ]
```

Table 25.1 lists some access functions for a number field K. The meaning of all of these should be clear, except perhaps in the case of **Signature**. The signature of K consists of two natural numbers, usually called r and s,

being the number of real roots and of pairs of complex roots of the defining polynomial of the field. Real roots correspond to ways of embedding the field in the real numbers, and complex non-real roots to (pairs of) embeddings in the complex numbers. Since the total number of roots equals the degree m of the polynomial (and the field), it follows that $m = r + 2 \cdot s$.

25.1.2 Quadratic Fields

Table 25.2. Quadratic field functions

MAGMA	Meaning
QuadraticField(m)	Quadratic field $\mathbb{Q}(\sqrt{d})$, where d is the square-free part of the non-square integer m
Degree(K)	Degree of quadratic field K over $\mathbb{Q}$ (always equals 2)
DefiningPolynomial(K)	Monic irreducible quadratic polynomial defining K
Discriminant(K)	Discrimant of $K = Q(\sqrt{d})$ (equals d if $d \equiv 0$ or $1 \pmod 4$, else $4d$
Signature(K)	Signature r, s of K ($2, 0$ if $d > 0$, else $0, 1$)
Eltseq(a)	Sequence $[a_0, a_1]$, where $a = a_0 + a_1\sqrt{d}$

The quadratic field $\mathbb{Q}(\sqrt{d})$, where d is a (positive or negative) square-free integer, may be created in MAGMA with the function **QuadraticField**. This is an alternative and preferable way of constructing the number field obtained by adjoining a root of $x^2 - d$ to $\mathbb{Q}$. For example:

```
> QF7<rt7> := QuadraticField(7);
> print QF7;
Quadratic Field Q(rt7)
> print rt7^2;
7
```

If the argument m of **QuadraticField**(m) contains any square factors, MAGMA will divide out the largest possible square from m to obtain the square-free integer d, and then create $\mathbb{Q}(\sqrt{d})$. For instance, if $m = 20$, then the generator of the resulting quadratic field will be equal to $\sqrt{5}$:

```
> Q<w> := QuadraticField(20);
> print Q, w^2, (1+w)^2;
Quadratic Field Q(w)
5  6 + 2*w
```

It is usual to create quadratic fields (and other number fields) with a generator assignment statement. If no generator name is given, then the generator is printed using `sqrt-`d:

```
> Gaussians := QuadraticField(-1);
> print Gaussians.1;
sqrt-1
```

The elements of a quadratic field are of the form $a_0 + a_1\sqrt{m}$, where a_0 and a_1 are rational numbers. These elements may be created in MAGMA either as expressions in $\sqrt{d}$ (as above), or by coercing into the field a sequence of rationals $[a_0, a_1]$. Conversely, the (rational) coefficients $[a_0, a_1]$ of an element a may be extracted with the function **Eltseq**(a), which is short for **ElementToSequence**(a). Elements are printed in the form $(1/m) \cdot (u + v\sqrt{d})$, where u, v, m are integers, the denominator m is positive and $\gcd(u, v, m) = 1$. For example, the following lines construct and print the elements $f = \frac{1}{3} - \frac{5}{6}\sqrt{7}$ and $g = \frac{5}{8} + 6\sqrt{7}$, their sum, and the sequence representation of their product:

```
> f := QF7![1/3, -5/6]; print f;
1/6*(2 - 5*rt7)
> g := QF7![5/8, 6]; print g;
1/8*(5 + 48*rt7)
> print f + g;
1/24*(23 + 124*rt7)
> print Eltseq(f * g);
[ -835/24, 71/48 ]
```

The main access functions for quadratic fields are listed in Table 25.2.

25.1.3 Cyclotomic Fields

The field $K = \mathbb{Q}(\zeta_m)$, where m is an integer greater than 2, is called the cyclotomic number field of cyclotomic order m. It is the field created by adjoining to $\mathbb{Q}$ a primitive m^{th} root of unity, and it is constructed with the function **CyclotomicField**(m). The degree d of K (and of its defining polynomial) is equal to $\phi(m)$; it may be obtained as **Degree**(K) or **EulerPhi**(m). The primitive root of K may be considered as the complex number $e^{\frac{2\pi \mathrm{i}}{m}}$. For example:

```
> C8<z> := CyclotomicField(8);
> print C8;
Cyclotomic Field of order 8 and degree 4
```

Table 25.3. Cyclotomic field functions

MAGMA	Meaning
CyclotomicField(m)	Cyclotomic field $\mathbb{Q}(\zeta_m)$
CyclotomicOrder(K)	m for $K = Q(\zeta_m)$
Degree(K)	Degree $d = \phi(m)$ of $K = Q(\zeta_m)$
DefiningPolynomial(K)	Monic irreducible degree-d polynomial defining K
Discriminant(K)	Discrimant of K
Signature(K)	Signature 0, $\frac{d}{2}$ of K
Conductor(K)	Smallest n such that $K \subset Q(\zeta_n)$, i.e., $\frac{m}{2}$ if $m \equiv 2 \bmod 4$, else m
Eltseq(a)	Sequence $[a_0, a_1 \ldots, a_{d-1}]$, where $a = a_0 + a_1\zeta_m + \cdots + a_{d-1}\zeta_m^{d-1}$
RootOfUnity(n, K)	n^{th} root of unity, as an element of K (if it exists)
VectorSpace(K, E)	Vector space $V = E^{(n)}$ and mapping from K onto V, where E is a subfield of cyclotomic field K such that K has degree n over E (uses powers of generator of K as basis)
VectorSpace(K, E, Q)	As above, using sequence Q of field elements as a basis

```
> print z^8;
1
> print DefiningPolynomial(C8);
$.1^4 + 1
```

Elements of $K = \mathbb{Q}(\zeta_m)$ can be created as the values of expressions involving the primitive element and rational numbers. Alternatively, the user can coerce into K a sequence $[a_0, a_1 \ldots, a_{t-1}]$ of rational numbers, where t is any non-negative integer, to represent the element $a_0 + a_1\zeta_m + \cdots + a_{t-1}\zeta_m^{t-1}$. In either case, the elements will be stored and printed in the form $a_0 + a_1\zeta_m + \cdots + a_{d-1}\zeta_m^{d-1}$, where d is the degree of K and the a_i are rational; this representation is unique, since $\{\zeta_m^0, \ldots, \zeta_m^{d-1}\}$ forms a $\mathbb{Q}$-basis for K. Given $a \in K$, the function **Eltseq**(a) returns the sequence $[a_0, a_1 \ldots, a_{d-1}]$ representing a in this way. For example:

```
> print 3*z^10 - 5 + 2/5*z^3;
2/5*z^3 + 3*z^2 - 5
> print C8 ! [2/7, 3/10];
3/10*z + 2/7
> ee := C8![9, 56, 0, 1, 7/10, 0, 2/3, 0]; print ee;
z^3 - 2/3*z^2 + 56*z + 83/10
> print Eltseq(ee);
[ 83/10, 56, -2/3, 1 ]
```

A **VectorSpace** function is provided for viewing a cyclotomic field as a vector space over some subfield. If K is a cyclotomic field with generator w and E is a subfield of K such that K is of degree n over E, then **VectorSpace**(K, E) returns the vector space $V = E^{(n)}$ together with the isomorphism from K onto V such that w^i is mapped to the $(i+1)^{\text{th}}$ unit vector of V. Similarly, if Q is a length-n sequence of elements of K describing a basis of K over E, then **VectorSpace**(K, E, Q) returns $V = E^{(n)}$ together with the isomorphism from K onto V such that $Q[i]$ is mapped to the $(i+1)^{\text{th}}$ unit vector of V.

Table 25.3 shows the simple functions for cyclotomic fields.

25.2 Equation Orders and Maximal Orders

Certain subrings of number fields are of great importance in number theory, namely *orders*. They form the analogue to the ring $\mathbb{Z}$ with respect to the field $\mathbb{Q}$. The most important orders in a number field $N = \mathbb{Q}(\alpha)$ are the *equation order* E_N and the *maximal order* M_N. The equation order consists of all combinations of powers of α with *integer* coefficients: $E_N = \mathbb{Z}[\alpha]$. The maximal order M_N is the subring of N consisting of all elements β of N that satisfy a polynomial equation $g(\beta) = 0$ for some polynomial g with integer coefficients. Since α itself satisfies such a polynomial equation (with $g = f$, the defining polynomial of N), it is clear that $E_N \subseteq M_N$. In a sense the ring M_N consists of all 'integers' in N (satisfying an integer relation), and so it is often called the *ring of integers* of N. All orders of N are subrings of M_N.

In the simplest number field, $\mathbb{Q}$, both the equation order and the maximal order equal the ring of ordinary integers, $\mathbb{Z}$; this is the only order in $\mathbb{Q}$. In cyclotomic fields, it is also the case that the equation order and the maximal order are equal. Therefore, if K is a cyclotomic or rational field and $a \in K$, then a is in the maximal order of K if and only if its coefficients are integral. MAGMA does not offer any functions for orders of $\mathbb{Q}$ or cyclotomic fields.

However, if K is a quadratic field or a general number field then the two orders may be different. Consequently, MAGMA offers the functions **EquationOrder**(K) and **MaximalOrder**(K) in these categories. (The number field can be retrieved from the equation order with the function **FieldOfFractions**.) Orders in quadratic fields and in general number fields have their own categories, **RngQuad** and **RngOrd**. Table 25.4 lists several functions relating to orders of number fields; some of these functions are only provided for certain categories.

For example, consider the field $\mathbb{Q}(\sqrt{5})$. The equation order is $\mathbb{Z}[\sqrt{5}]$, which consists of all elements $a + b\sqrt{5}$, where a and b are integers. Now, the element

Table 25.4. Orders of number fields

MAGMA	Meaning
EquationOrder(K)	Equation order of number field K
EquationOrder(f)	Equation order defined by monic irreducible integer polynomial f
MaximalOrder(K)	Maximal order (ring of integers) of K
MaximalOrder(f)	Equation order defined by monic irreducible integer polynomial f
IntegerRing(K)	Maximal order (ring of integers) of K
FieldOfFractions(O)	Field of fractions of order O
IsEquationOrder(O)	**true** if O is an equation order (i.e., its basis is a power basis)
IsMaximal(O)	**true** if O equals maximal order over its field of fractions
IntegralBasis(K)	A basis for K consisting of integral elements
IsIntegral(a)	**true** if element a is in the maximal order of K
Conductor(K)	Order of smallest cyclotomic field containing K
DefiningPolynomial(O)	Defining polynomial for order O over its ground ring
Degree(O)	Degree of order O
Discriminant(O)	Discriminant of order O
ReducedDiscriminant(O)	Reduced discriminant of order O
Signature(O)	Signature r, s of order O

$(1 + \sqrt{5})/2$ is not in the equation order, but since it satisfies $x^2 - x - 1 = 0$, it is contained in the maximal order. Therefore the maximal order is strictly greater than the equation order for this field. The following lines show how to define these orders, firstly in a quadratic field, and secondly in a general number field:

```
> // quadratic field
> Q5<rt5> := QuadraticField(5);
> EO := EquationOrder(Q5);
> print EO;
Order of conductor 2 in Quadratic Field Q(sqrt5)
> MO := MaximalOrder(Q5);
> print MO;
Order of conductor 1 in Quadratic Field Q(sqrt5)
> a := (1 + rt5) / 2;
> print a in EO;
false
> print a in MO;
true
```

```
> print IsIntegral(a);
true

> // number field
> P<x> := PolynomialRing(RationalField());
> Q5<rt5> := NumberField(x^2 - 5);
> EO := EquationOrder(Q5);
> print EO;
Equation Order of Q5
> MO := MaximalOrder(Q5);
> print MO;
Maximal Order of Q5
> a := (1 + rt5) / 2;
> print a notin EO and a in MO;
true
```

The facilities for orders in the different categories will now be discussed in more detail.

25.2.1 Orders in Quadratic Fields

For a quadratic field $K = \mathbb{Q}(\sqrt{d})$, the computation of the maximal order M_K is easy. It depends solely on the residue class of d modulo 4, irrespective of whether d is positive or negative. If $d \equiv 2, 3 \pmod 4$ then M_K equals the equation order E_K; for $d \equiv 1 \pmod 4$, $M_K = \mathbb{Z} + \mathbb{Z} \cdot (1 + \sqrt{d})/2$, so it is larger than E_K. Let $\epsilon_d = \sqrt{d}$ for $d \equiv 2, 3 \pmod 4$ and $\epsilon_d = (1 + \sqrt{d})/2$ for $d \equiv 1 \pmod 4$; then the maximal order will always consist of integer linear combinations of 1 and ϵ_d. That is, $[1, \epsilon_d]$ forms an integral basis for the field, returned by the function **IntegralBasis**(K) (as a sequence of field elements) or **Basis**(M_K) (as a sequence of elements of M_K). This ϵ_d will be the 'generator' for the maximal order. It may be given an identifier and printname by means of generator assignment; otherwise, it will be printed as `epsilon`d. For instance:

```
> Q<w> := QuadraticField(-3);
> M<z> := MaximalOrder(Q);
> print IntegralBasis(Q);
[ 1, 1/2*(1 + w) ]
> print z, Q ! z;
z 1/2*(1 + w)
> print M ! w;
-1 + 2*z
```

Unlike in the case of general number fields, it is easy to describe all orders of a quadratic field K. For every positive integer f there exists a suborder of index f in the maximal order, consisting of all elements of the form $a + bf\epsilon_d$. The index f is called the *conductor* of the order, and is returned by the function **Conductor**(K). This order can be created with the **sub**-constructor, and the resulting 'generator' will be equal to $f\epsilon_d$. For instance:

```
> S<s> := sub< M | 7 >;
> print S;
Order of conductor 7 in Quadratic Field Q(sqrt3)
> print M ! s;
7*z
> print Q ! s;
1/2*(7 + 7*w)
```

Every order O of a quadratic field K has a basis $[1, f\epsilon_d]$, where f is the conductor of O and ϵ_d is as defined above for the maximal basis. This basis is returned by the function **Basis**(O), as a sequence of elements of O. It follows that each $a \in O$ may be expressed uniquely as $a = a_0 + a_1 f\epsilon_d$, where $a_0, a_1 \in \mathbb{Z}$, and may be created in MAGMA in this way. The element a may also be created by coercing the integer-sequence $[a_0, a_1]$ into O; conversely, **Eltseq**(a) returns this sequence.

```
> el_S := S ! [7, 15]; print el_S;
7 + 15*s
> print Eltseq(el_S);
[ 7, 15 ]
> el_M := M ! el_S; print el_M;
7 + 105*z
> print el_M eq el_S;
true
> print Eltseq(el_M);
[ 7, 105 ]

> el_Q := Q ! el_M; print el_Q;
7 + 105*w
> print el_Q eq el_S;
true
```

As the examples above demonstrate, elements of orders of quadratic fields are printed as expressions of the form $a_0 + a_1 f\epsilon_d$. Any order element may be coerced into the field from which it derives, and any field element may be coerced into an order if it is an element of that order. Equality tests of elements of orders and/or fields are performed by lifting them into a common overstructure.

25.2.2 Orders in Number Fields

Just as for quadratic fields, elements of orders in general number fields are represented as linear combinations of basis elements, where the coefficients are integers. If O is an order of a number field K, and $a \in O$, then **Basis**(O) returns the basis used for O, as a sequence of elements of K. (Alternatively, **BasisMatrix**(O) returns this basis in matrix form.) If $a \in O$, then **Eltseq**(a) returns the coefficients for a in terms of this basis, as a sequence of integers, and such a sequence may be coerced into O in order to create a. Unlike the case for quadratic fields, elements of orders in general number fields are printed as a list of coefficients.

In the following example, a suitable element of the number field N generated by $x^2 - 5$ is coerced into the equation order E of N. Observe that the basis of E consists of the powers of a root of the defining polynomial. This is always the case for an equation order.

```
> P<x> := PolynomialRing(RationalField());
> N<z> := NumberField(x^4 - 5);
> E := EquationOrder(N);
> print Basis(E);
[
    1,
    z,
    z^2,
    z^3
]
> r_N := 3*z^2 - 5*z + 2; print r_N;
3*z^2 - 5*z + 2
> r_E := E ! r_N; print r_E;
[2, -5, 3, 0]
> print Eltseq(r_E);
[ 2, -5, 3, 0 ]
```

Given orders O and P of a degree-n field, **TransformationMatrix**(O, P) returns an $n \times n$ integral matrix and a common integer denominator. This matrix expresses the n basis elements of O as linear combinations of the n basis elements of P. Continuing the example:

```
> O := MaximalOrder(N);
> print Index(O, E);
4
> print BasisMatrix(O);
[   1    0    0    0]
[   0    1    0    0]
```

```
[-1/2    0  1/2    0]
[-1/2 -1/2 -1/2  1/2]
> r_O := O ! r_E; print r_O;
[5, -5, 6, 0]

> TM, d := TransformationMatrix(E, O); print TM, d;
[1 0 0 0]
[0 1 0 0]
[1 0 2 0]
[2 1 2 2]
1
> V := RSpace(IntegerRing(), 4);
> vec_r_E := V ! Eltseq(r_E);
> print vec_r_E * TM;
( 5 -5  6  0)
> print Eltseq($1) eq Eltseq(r_O);
true
```

Table 25.5. Basis and matrix functions for orders of number fields

MAGMA	Meaning
Basis(O)	Basis of field elements for the order O
BasisMatrix(O)	Rational matrix representing a basis for O
TraceMatrix(O)	Matrix consisting of traces of products of basis elements
RepresentationMatrix(a)	Representation matrix for order element or field element a
MultiplicationTable(O)	Length-n sequence of representation matrices for basis elements of O
TransformationMatrix(O, P)	Transformation matrix from order O to order P
IsRelative(O)	**true** if O is given by a transformation matrix (i.e., it is not an equation order)

If a is a field element or order element, **RepresentationMatrix**(a) returns the square matrix representing the linear transformation induced by multiplying by a. For field elements the representation matrix is with respect to the (possibly relative) power basis for the field, whereas for order elements the (absolute) basis for the order is used. For example:

```
> print RepresentationMatrix(z); // field element
[0 0 0 5]
[1 0 0 0]
```

```
[0 1 0 0]
[0 0 1 0]

> print r_O * O!z;
[1, 5, -4, 6]
> rm_z_O := RepresentationMatrix(O!z); // order element
> print rm_z_O;
[ 0  1  1  1]
[ 1  0  0 -1]
[ 0  2  1 -2]
[ 0  0  1 -1]
> vec_r_O := V ! Eltseq(r_O);
> print vec_r_O * Transpose(rm_z_O);
( 1  5 -4  6)
```

Table 25.5 lists the basis and matrix functions for orders of number fields.

25.3 Elements of Number Fields and Their Orders

Table 25.6. Functions on elements of number fields or orders

MAGMA	Meaning
Norm(a)	Product of all the conjugates of a (over ground field)
Trace(a)	Sum of all the conjugates of a (over ground field)
MinimalPolynomial(a)	Minimal polynomial of a (over ground field)
NormEquation(O, m)	Seeks elements of order O with norm m up to a unit; returns Boolean stating whether any solutions exist, and (if **true**) a sequence of the solutions
NormEquation(O, m)	As above, where O is the equation order of field K (category **FldQuad** or **FldNum**)
IsIntegral(a)	**true** if element a is integral
IsUnit(a)	**true** if a is a unit

Most functions and arithmetic operators for elements of a number field K are also available for elements of orders of K. Table 25.6 lists the main functions for field elements and order elements in the various categories.

As will be explained later, in the **FldNum** category it is possible to construct *relative extensions*, that is, extensions of a field N other than $\mathbb{Q}$. In this case, the return values of **Norm**, **Trace** and **MinimalPolynomial** are given

in terms of the ground field N. For computations in terms of $\mathbb{Q}$, the functions **AbsoluteNorm**, **AbsoluteTrace** and **AbsoluteMinimalPolynomial** are provided.

The function **NormEquation**(O, m) or **NormEquation**(K, m) makes it possible to search for order elements with a specified norm m, where m is a positive integer. If the first argument is a quadratic field or general number field K, then the function searches for elements of the equation order of K; otherwise, it searches for elements in the given order O. By default, this function look for solutions whose norm equals m up to a unit, but if the parameter **Exact** is set to **true**, then the norm must equal m exactly. (See the *Handbook* for the other parameters.) The function returns a Boolean stating whether a solution exists; and if the first return value is **true** then it also returns a sequence of solutions. For instance:

```
> P<x> := PolynomialRing(RationalField());
> E := EquationOrder(x^4 - 5);
> ok, n := NormEquation(E, 4 : Exact := true);
> print ok;
false
> ok, n := NormEquation(E, 4);
> print ok;
true
> print #n;
1
> soln := n[1]; print soln;
[-1, -1, 0, 0]
> print Norm(soln);
-4
```

From the output, there is no element in $\mathbb{Z}[\sqrt[4]{5}]$ of norm 4, but there is an element *soln* of norm -4 (which equals 4, up to a unit). Since *soln* is an element of the order E, it is printed in terms of the basis of the order E. The basis of an equation order always consists of the powers of a root of the polynomial, so the element is equal to $-1 - \sqrt[4]{5}$:

```
> K<w> := FieldOfFractions(E);
> print K ! soln;
-w - 1
```

Table 25.7 lists additional functions available only in the categories **FldNumElt** and **RngOrdElt** (general number fields and their orders).

Complex conjugates are available in quadratic fields and orders, and in cyclotomic fields, by means of the function **ComplexConjugate**(a). For instance:

Table 25.7. Element functions for general number fields only

MAGMA	Meaning
Sqrt(a)	A square root of a, if it exists
Root(a)	An n^{th} root of a, if it exists
IsPrimitive(a)	**true** if a is primitive
IsPower(a, n)	**true** if a in order O equals b^n for some $b \in O$; if **true**, also returns such a b
IsPowerTimesUnit(a, n)	As above, but the equality is up to units

```
> R<r> := QuadraticField(3);
> I<rr> := QuadraticField(-3);
> print ComplexConjugate(r), ComplexConjugate(rr);
r -rr

> C<z> := CyclotomicField(12);
> print ComplexConjugate(z);
-z^3 + z
> print z^11 eq ComplexConjugate(z);
true
```

As for conjugates, each $a \in \mathbb{Q}(\sqrt{d})$ has only a single conjugate, given by **Conjugate**(a). However, each $a \in \mathbb{Q}(\zeta_m)$ has a conjugate given by **Conjugate**(a, k) for every k coprime to m, since the map sending ζ_m to ζ_m^k and leaving $\mathbb{Q}$ invariant defines an automorphism. For example:

```
> print Conjugate(r), Conjugate(rr);
-r -rr

> print [ Conjugate( z + z^-1, k) : k in [1, 5, 7, 11] ];
[ -z^3 + 2*z, z^3 - 2*z, z^3 - 2*z, -z^3 + 2*z ]
> print Seqset($1);
{ z^3 - 2*z, -z^3 + 2*z }
```

Note that the cyclotomic example above demonstrates an explicit way to act with the *Galois group* of a cyclotomic field $\mathbb{Q}(\zeta_m)$ over $\mathbb{Q}$; this group is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$. In this case there are only 2 different conjugates, although there are 4 automorphisms.

The functions for complex conjugates and conjugates for elements of quadratic fields and orders and of cyclotomic fields are given in Table 25.8. The conjugate functions for elements of general number fields and their orders are not listed here but in Table 25.15 (p. 481); they return approximations in the complex field rather than exact values.

Table 25.8. Conjugate functions in quadratic and cyclotomic fields

MAGMA	Meaning
Conjugate(a)	Given $a = a_0 + a_1\sqrt{d}$ in $\mathbb{Q}(\sqrt{d})$ or an order, return $a_0 - a_1\sqrt{d}$
ComplexConjugate(a)	Given $a = a_0 + a_1\sqrt{d}$ in $\mathbb{Q}(\sqrt{d})$ or an order, return a if $d > 0$, else $a_0 - a_1\sqrt{d}$
Conjugate(a, k)	k^{th} conjugate of $a \in \mathbb{Q}(\zeta_m)$, where $\gcd(k, m) = 1$, obtained by applying field automorphism $\zeta_m \mapsto \zeta_m^k$
ComplexConjugate(a)	Complex conjugate of $a \in \mathbb{Q}(\zeta_m)$

Quadratic and cyclotomic field elements can be automatically coerced into a complex field (or $\mathbb{R}$, $\mathbb{Q}$, or $\mathbb{Z}$ if they lie in that subfield or subring). MAGMA's designers have made a choice of embedding: ζ_n is sent to $e^{2\pi i/n}$, and $\sqrt{d}$ for $d < 0$ is sent to $\sqrt{-d}i$. These embeddings are customary and convenient (but not necessary). For instance:

```
> print ComplexField() ! rr;
1.732050807568877293527446341495*i
> print IntegerRing() ! ( rr * ComplexConjugate(rr));
3
> print ComplexField() ! z;
0.866025403784438646763723170759426544989 +
 0.499999999999999999999999999999987*i
```

25.4 Advanced Constructions of Fields and Elements

This section discusses advanced methods of creating number fields: the use of roots of unity in cyclotomic fields; the construction of subfields; and the construction of relative and multiple extensions.

25.4.1 Cyclotomic Fields and Roots of Unity

In situations where (only) roots of unity are involved in computations, the function **RootOfUnity** is of assistance in constructing such elements in the appropriate cyclotomic field. One version of this function, **RootOfUnity**(n) takes a positive integer n and returns a primitive n^{th} root of unity ζ_n in $\mathbb{Q}(\zeta_n)$. For example:

```
> r := RootOfUnity(5); print r;
```

```
zeta_5
> s := RootOfUnity(7); print s;
zeta_7
> print Parent(r), Parent(s);
Cyclotomic Field of order 5 and degree 4
Cyclotomic Field of order 7 and degree 6
```

Automatic coercion enables arithmetic with various roots of unity, mixed with rationals:

```
> print (r + s - 2/3)^2;
zeta_35^14 + 2*zeta_35^12 + zeta_35^10
   - 4/3*zeta_35^7 - 4/3*zeta_35^5 + 4/9
```

There is a second version of this function, **RootOfUnity**(n, K), which creates the desired element ζ_n inside a given cyclotomic field $K = \mathbb{Q}(\zeta_m)$. Of course, this is only possible if such an element exists in K, so n must divide m (or $2m$ if m is odd).

Suppose that the user wishes to work with matrices whose entries are of the form $\cos(2\pi\mathrm{i}/r)$ and $\sin(2\pi\mathrm{i}/r)$ for certain integers r. It is helpful to begin by defining short functions that create these sines and cosines as algebraic numbers. MAGMA's designers have made a choice of embedding ζ_n in the complex numbers, by sending ζ_n to $e^{2\pi\mathrm{i}/n}$, which is customary and convenient. This makes it possible to express the sines and cosines in terms of roots of unity:

```
> cos := func< r | (x + x^-1)/2 where x is RootOfUnity(r) >;
> sin := func< r | -RootOfUnity(4) * (x - x^-1)/2
      where x is RootOfUnity(r) >;
> print sin(6);
-1/2*zeta_12^3 + zeta_12
> print sin(6)^2 + cos(6)^2;
1
```

The following lines use the functions *cos* and *sin* to build 3×3 matrices whose entries are sines and cosines of $2\pi\mathrm{i}/3$ and $2\pi\mathrm{i}/5$. To create the appropriate matrix ring, it is necessary to find a single cyclotomic field containing all these algebraic elements. Fortunately it is easy to compute such a field, since r^{th} and s^{th} roots of unity will be contained in the field of L^{th} roots of unity, where $L = \mathrm{lcm}(r, s)$. In the present example, third roots and fifth roots of unity are required, as well as fourth roots for the *sin* function. Therefore the field $\mathbb{Q}(\zeta_{60})$ will suffice, 60 being the least common multiple of 3, 5, and 4:

```
> MR := MatrixRing(CyclotomicField(60), 3);
```

```
> m1 := MR ! [ cos(3),sin(3),0,sin(3),-cos(3),0,0,0,-1 ];
> m2 := MR ! [cos(5),sin(5),0,-sin(5),cos(5),0,0,0,1 ];
> print m1 * m2 * m1 eq m2^-1;
true
```

It should be remarked that MAGMA's automatic coercion mechanism is also capable of finding the smallest cyclotomic field containing various roots of unity. For example, if the user constructs a sequence containing $\cos(2\pi/5)$ and $\sin(2\pi/3)$, MAGMA will calculate its universe as $\mathbb{Q}(\zeta_{60})$, and an element a extracted from the sequence will have this field as its parent. Furthermore, given any cyclotomic field element a, it is possible to compute the element which equals a but is a member of the smallest possible cyclotomic field: the function **Minimize**(a) returns this element; and the procedure **Minimize**($\tilde{}a$) mutates a into this element. (**Minimize** may also be applied to a set of cyclotomic field elements.) For instance:

```
> s := [ cos(5), sin(3) ];
> print Universe(s);
Cyclotomic Field of order 60 and degree 16
> a := s[1];
> print a;
-1/2*zeta_60^14 + 1/2*zeta_60^6 + 1/2*zeta_60^4 - 1/2
> print Parent(a);
Cyclotomic Field of order 60 and degree 16
> Minimize(~a);
> print a;
-1/2*zeta_5^3 - 1/2*zeta_5^2 - 1/2
> print Parent(a);
Cyclotomic Field of order 5 and degree 4
```

The function **MinimalField**, which may be applied to a cyclotomic field element or a set of such elements, performs a similar task to **Minimize**. It returns the smallest cyclotomic field containing the element(s):

```
> print MinimalField(s[2]);
Cyclotomic Field of order 12 and degree 4
```

25.4.2 Subfields of Cyclotomic Fields

Sometimes the need arises to mix roots of unity and other algebraic numbers. Sums of roots of unity may define elements in *subfields* of cyclotomic fields, and it may be convenient to recognize such a situation (e.g., when the subfield is a quadratic field).

In fact, MAGMA handles automatic coercion from quadratic fields to cyclotomic fields: if a quadratic field element is contained in a cyclotomic field, it can be coerced into it. This provides a way to write quadratic field elements as sums of roots of unity. For instance:

```
> Q<w> := QuadraticField(5);
> C<z> := CyclotomicField(5);
> print C ! w;
-2*z^3 - 2*z^2 - 1
```

It is not hard to determine the smallest cyclotomic field containing a quadratic field, but the intrinsic function **Conductor**(K) is also provided for this:

```
> Q<w> := QuadraticField(-13);
> C<z> := CyclotomicField(Conductor(Q));
> print C;
Cyclotomic Field of order 52 and degree 24
> print C ! w;
-2*z^21 + 2*z^19 + 2*z^15 - z^13 + 2*z^11 + 2*z^7 - 2*z^5
```

It can be more difficult to go the other way, and recognize the subfield in which a combination of roots of unity lies. The following examples are intended to suggest helpful techniques.

To find out if a given cyclotomic field element a is contained in some subfield, the function **MinimalPolynomial**(a) can be used. The polynomial resulting from this function will define a number field containing the element. For example, the following lines show that $\zeta_7 + \zeta_7^6$ is contained in the cubic field defined by adjoining a root of $x^3 + x^2 - x - 1$, and that $\zeta_7 + \zeta_7^2 + \zeta_7^6$ is contained in a quadratic field which is isomorphic to $\mathbb{Q}(\sqrt{-7})$:

```
> R<x> := PolynomialRing(RationalField());
> C<z> := CyclotomicField(7);
> print MinimalPolynomial(z + z^6);
x^3 + x^2 - x - 1
> mp := MinimalPolynomial(z + z^2 + z^4); print mp;
x^2 + x + 2
> print Discriminant(mp);
-7
```

The next example is more elaborate. By using a homomorphism from the cyclotomic field to a general number field, it is possible to coerce elements into subfields, add elements from various subfields, and so on. The starting point will be the character table for the dihedral group of order 48; what

matters in the present context is that the entries are sums of 24^{th} roots of unity.

```
> T := CharacterTable(DihedralGroup(24));
> print T;

Character Table
---------------

------------------------------------------------------------------------
Class|  1  2  3  4  5  6  7   8   9  10  11    12    13    14    15
Size |  1  1 12 12  2  2  2   2   2   2   2     2     2     2     2
Order|  1  2  2  2  3  4  6   8   8  12  12    24    24    24    24
------------------------------------------------------------------------
p  = 2  1  1  1  1  5  2  5   6   6   7   7    10    11    11    10
p  = 3  1  2  3  4  1  6  2   9   8   6   6     8     9     8     9
------------------------------------------------------------------------
X.1  +  1  1  1  1  1  1  1   1   1   1   1     1     1     1     1
X.2  +  1  1 -1  1  1  1  1  -1  -1   1   1    -1    -1    -1    -1
X.3  +  1  1 -1 -1  1  1  1   1   1   1   1     1     1     1     1
X.4  +  1  1  1 -1  1  1  1  -1  -1   1   1    -1    -1    -1    -1
X.5  +  2  2  0  0 -1  2 -1   2   2  -1  -1    -1    -1    -1    -1
X.6  +  2  2  0  0 -1  2 -1  -2  -2  -1  -1     1     1     1     1
X.7  +  2  2  0  0 -1 -2 -1   0   0   1   1    Z2   -Z2   -Z2    Z2
X.8  +  2  2  0  0 -1 -2 -1   0   0   1   1   -Z2    Z2    Z2   -Z2
X.9  +  2  2  0  0  2 -2  2   0   0  -2  -2     0     0     0     0
X.10 +  2 -2  0  0  2  0 -2  Z1 -Z1   0   0   -Z1    Z1   -Z1    Z1
X.11 +  2 -2  0  0  2  0 -2 -Z1  Z1   0   0    Z1   -Z1    Z1   -Z1
X.12 +  2 -2  0  0 -1  0  1 -Z1  Z1  Z2 -Z2    Z3  Z3#5 -Z3#5   -Z3
X.13 +  2 -2  0  0 -1  0  1 -Z1  Z1 -Z2  Z2 -Z3#5   -Z3    Z3  Z3#5
X.14 +  2 -2  0  0 -1  0  1  Z1 -Z1  Z2 -Z2   -Z3 -Z3#5  Z3#5    Z3
X.15 +  2 -2  0  0 -1  0  1  Z1 -Z1 -Z2  Z2  Z3#5    Z3   -Z3 -Z3#5

Explanation of Symbols:
-----------------------

# denotes algebraic conjugation, that is,
#k indicates replacing the root of unity w by w^k

Z1     = -zeta_8^3 + zeta_8 where zeta_8 is RootOfUnity(8)
Z2     = -zeta_12^3 + 2*zeta_12 where zeta_12 is RootOfUnity(12)
Z3     = zeta_24^7 - zeta_24^3 - zeta_24
                               where zeta_24 is RootOfUnity(24)


> C := CyclotomicField(24);
> z1 := C ! T[10][8];       // or C ! T[10, 8]
> z2 := C ! T[7][12];
> z3 := C ! T[12][12];
> R<x> := PolynomialRing(RationalField());
> N<n> := NumberField(DefiningPolynomial(C));
> print N;
```

```
Number Field with defining polynomial x^8 - x^4 + 1
over the Rational Field
> h := hom< C -> N | n >;
```

The homomorphism between the cyclotomic field and its general counterpart N has now been established. Now the three elements coming from the character table will be investigated, by defining subfields of N generated by them. From their minimal polynomials, *z1* corresponds to $\sqrt{2}$ and *z2* to $\sqrt{3}$.

```
> print [ MinimalPolynomial(z) : z in [z1, z2, z3] ];
[
    x^2 - 2,
    x^2 - 3,
    x^4 - 4*x^2 + 1
]
> Q1<rt2> := sub< N | h(z1) >;
> Q2<rt3> := sub< N | h(z2) >;
> K<k> := sub< N | h(z3) >;
> print h(z1), h(z1) eq rt2;
-n^5 + n^3 + n
true
> rt2_p_rt3 := rt2 + rt3; print rt2_p_rt3;
-n^6 - n^5 + n^3 + 2*n^2 + n
> print rt2_p_rt3 in K;
true
> print K ! rt2_p_rt3;
-k^3 + k^2 + 3*k - 2
> N23 := sub< N | rt2_p_rt3 >; print N23;
Number Field with defining polynomial x^4 - 10*x^2 + 1
over the Rational Field
> print k in N23;
true
```

It can be concluded that the element $z3$ lies in the degree-4 field generated by $\sqrt{2} + \sqrt{3}$.

Knowing that both $\sqrt{2}$ and $\sqrt{3}$ are contained in N (or, for that matter, K) the user might want to know what $\sqrt{6}$ looks like in N. Of course, in this example it suffices to multiply out $\sqrt{2} \cdot \sqrt{3}$, but the general method to find such an algebraic element in a given number field is to factor its defining polynomial over the number field and look for a linear factor:

```
> A<a> := PolynomialRing(N);
> F := Factorization(a^2 - 6);
> print F;
```

```
[
    <a - 2*n^7 + n^5 + n^3 + n, 1>,
    <a + 2*n^7 - n^5 - n^3 - n, 1>
]
> rt6 := Coefficient(F[1][1], 0);
> print rt6^2;
6
> print rt6 eq rt2*rt3;
true
```

Factorization of polynomials over number fields may be very time consuming, and is only feasible for low-degree polynomials over small fields.

25.4.3 Relative Extensions and Subfields

The number fields seen up to this point have been *absolute extensions*, that is, simple extensions of the rational field $\mathbb{Q}$. In the category of general number fields, it is also possible to construct *relative extensions*, by extending number fields other than $\mathbb{Q}$. Finite algebraic extensions of number fields produce number fields again, and it is often beneficial to create large degree number fields by successive relative extension rather than by a single absolute extension. The *ground field* of an extension K, returned by the function **GroundField**(K), is the field that was extended to produce K; thus a relative extension is one whose ground field is larger than $\mathbb{Q}$.

Relative extensions may be created in two ways. The first possibility is to create a multiple extension in one step. If Q is a sequence of polynomials $[f_1, f_2, \ldots, f_m]$ with rational or integer coefficients, then **NumberField**(Q) creates a tower of field extensions by successively adjoining roots of f_m, f_m, ..., f_1. It returns the relative extension $K_m = K_{m-1}(\alpha_1)$, where $K_{m-1} = K_{m-2}(\alpha_2)$, and so forth, up to $K_1 = K_0(\alpha_m)$, and $K_0 = \mathbb{Q}$, where α_i is a root of f_i. The roots are adjoined in this order, so that the i^{th} generator of the resulting field corresponds to α_i. It is necessary for each f_i to be irreducible over K_{i-1}. For example:

```
> P<x> := PolynomialRing(RationalField());
> M<a1, a2, a3> := NumberField([ x^2-2, x^3-3, x^5-5 ]);
> print M;
Number Field with defining polynomial x^2 - 2
over its ground field
> print GroundField(M);
Number Field with defining polynomial x^2 - 2
over its ground field
> print GroundField(GroundField(M));
```

```
Number Field with defining polynomial x^5 - 5
over the Rational Field
> print a1^2, a2^3, a3^5;
2
3
5
> print (a1 + a2 - a3)^2;
(2*a2 - 2*a3)*a1 + a2^2 - 2*a3*a2 + a3^2 + 2
```

Note that the intermediate fields do not have 'names' assigned to them. Also note that in this construction all polynomials need to have rational (or integer) coefficients.

As another example of multiple extensions, the problem of constructing a field containing both $\sqrt{2}$ and $\sqrt{3}$ was discussed in the previous section. It may be achieved more directly as follows:

```
> N<n, m> := NumberField([x^2-2, x^2-3]);
> print n^2 eq 2 and m^2 eq 3;
true
```

The second way to create a relative extension is to apply the **ext**-constructor, where the left side is an existing number field K (which may be $\mathbb{Q}$) and the right side is a monic polynomial g which is irreducible over K. It returns the field created by adjoining a root of g to K. For instance, the field M above could also be created as follows:

```
> S<a2, a3> := NumberField([ x^3-3, x^5-5 ]);
> Y<y> := PolynomialRing(S);
> M1 := ext< S | y^2 - (a3^5 - a2^3) >;
> print M1;
Number Field with defining polynomial x^2 - 2
over S
```

For the sake of efficiency, MAGMA *assumes that whenever a relative extension is defined (using* **ext** *or the sequence version of* **NumberField***) the polynomial is irreducible over the ground field.* This is because it is in general an expensive operation (similar to factorization) to test the irreducibility of a polynomial over a number field. If this is not the case, a magma may be created that is not a number field. For example, MAGMA will not object if the user tries to extend $F = \mathbb{Q}(\sqrt{3})$ by $x^2 - 3$, that is, by $\sqrt{3}$ again, but the resulting strange object will not be a field!

```
> F<rt3> := NumberField(x^2 - 3);
> H<h> := ext< F | x^2 - 3 >;
```

```
> print (rt3 * h) ^ 2;
9
> print (rt3 * h) in { H | 3, -3 };
false
```

If L is defined as an extension of K and K is an absolute extension, then **AbsoluteField**(L) returns L as an absolute extension. For multiple extensions, the construction has to be repeated. In the case of the fields M and S defined above, S is an extension of a relative extension, but M is not:

```
> A<a> := AbsoluteField(S);
> print A;
Number Field with defining polynomial x^15 - 15*x^12
 - 15*x^10 + 90*x^9 - 1350*x^7 - 270*x^6 + 75*x^5
 - 6075*x^4 + 405*x^3 - 2250*x^2 - 2025*x - 368 over
 the Rational Field
> B<b> := AbsoluteField(M);

>> B<b> := AbsoluteField(M);
                        ^
Runtime error in 'AbsoluteField': This routine only works
for simple relative extensions
> M2<m2> := ext< A | y^2 - 2 >;
> B<b> := AbsoluteField(M2);
> print Degree(B);
30
```

Extensions of orders may be calculated in an entirely analogous manner, using the **ext**-constructor. The left side of this constructor should be an order O, and the right side should be a polynomial that is irreducible over the field of fractions of O. The result will be a relative order. **AbsoluteOrder** exists as an analogue to **AbsoluteField**.

If K is a relative number field, with ground field G say, the subfield of K generated by a single element $a \in K$ may be created using the **sub**-constructor. The result will be the relative field $H = G(a)$ over G, together with an embedding of H into K. For instance:

```
> G := NumberField(x^5 - 8);
> K<k> := ext< G | x^4 + 4 >;
> H, H2K := sub< K | k^2 >;
> print H, H2K;
Number Field with defining polynomial x^2 + 4 over G
Mapping from: FldNum: H to FldNum: K
```

The entries in the lattice of subfields of an absolute number field K can be obtained through the use of **Subfields**(K). It returns a sequence of pairs, consisting of a subfield together with a homomorphism from it to the field K. The sequence may contain entries with isomorphic subfields. It is possible to obtain subfields of a specific degree d too, by means of the function **Subfields**(K, d).

Some of the number field constructions produce fields with 'bad' defining polynomials. Given an absolute number field K with defining polynomial f, the **BetterPolynomial**(K) function attempts to find a monic irreducible integer polynomial g defining a field L isomorphic to K, such that the discriminant of g is smaller (in absolute value) than that of f. Another version of this function, **BetterPolynomial**(K, d), allows the user to specify that a given index divisor d should be avoided; here, the additional restriction on g is that d must not divide the index $[O_L : E_L]$ of the equation order E_L of L in the maximal order O_L. The principal return value of this function is a Boolean indicating whether such a polynomial can be found. If this is **true**, the function also returns the number field L defined by g.

Table 25.9 lists the functions for relative extensions, subfields, and embeddings of number fields. As an example of several of these functions, consider the fields K and L with defining polynomials $x^6 - 2$ and $x^4 - 8$ respectively. The following lines find another field E containing subfields isomorphic to K and L, and compute the embedding maps of K and L into E. The construction begins with an invocation of **MergeFields**(K, L), which provides an alternative method for creating multiple absolute extensions. This function takes a pair of (absolute) number fields K and L and returns a sequence of number fields, each of which contains a subfield isomorphic to K and a subfield isomorphic to L. In general there are several possibilities for this, hence the sequence, since different choices for the roots of polynomials may exist.

```
> P<x> := PolynomialRing(RationalField());
> K<a> := NumberField(x^6 - 2);
> L<b> := NumberField(x^4 - 8);
> mf := MergeFields(K, L); print mf;
[
    Number Field with defining polynomial
        x^12 - 24*x^8 - 96*x^7 - 4*x^6 + 192*x^4 - 1280*x^3
        + 672*x^2 - 96*x - 508 over the Rational Field,
    Number Field with defining polynomial
        x^12 - 24*x^8 + 96*x^7 - 4*x^6 + 192*x^4 + 1280*x^3
        + 672*x^2 + 96*x - 508 over the Rational Field
]
```

Table 25.9. Relative extensions and subfields

MAGMA	Meaning
AbsoluteField(K)	Absolute number field isomorphic to simple relative extension K
AbsoluteOrder(O)	Absolute equation order isomorphic to simple relative order O
GroundField(K)	Field over which K was defined as an extension
CoefficientRing(O), **BaseRing**(O)	Order over which order O was defined by single equation (i.e., last in a possible tower of orders); equals $\mathbb{Z}$ for an absolute order
Simplify(O)	Given order O obtained by chain of transformations from an equation order E, return isomorphic order obtained by a single transformation over E
MergeFields(K, L)	Sequence of number fields, each of which contains K and L as subfields
IsIsomorphic(K, L)	**true** if absolute number fields K and L are isomorphic; if **true**, also returns the isomorphism from K to L
BetterPolynomial(K)	**true** if there is a field L isomorphic to K with a 'better' polynomial; if **true**, also returns L
BetterPolynomial(K, d)	As above, with extra requirement that d cannot be an index divisor
Subfields(K)	Sequence of tuples $< K_i, m_i >$ giving all subfields K_i of K (except $\mathbb{Q}$) and the embedding homomorphisms m_i into K; some of the subfields may be isomorphic
Subfields(K, m)	As above, but only the subfields of K of degree m

The two resulting fields both have degree 12. (In fact, they are isomorphic.) Arbitrarily, the first field will be chosen as a starting point. Rather than taking it directly as E, an isomorphic field with a better polynomial will be constructed:

```
> isbp, E := BetterPolynomial(mf[1]);
> print isbp;
true
> print E;
Number Field with defining polynomial x^12 - 2
over the Rational Field
> E<e> := E; // name the primitive element
```

The next stage is to find the subfields of E. Among them should be fields isomorphic to K and L:

```
> sf := Subfields(E); print sf;
```

```
[
    <Number Field with defining polynomial x^12 - 2
    over the Rational Field,
    Mapping from: FldNum: E to FldNum: E>,
    <Number Field with defining polynomial x^2 - 2
    over the Rational Field,
    Mapping from:
    Number Field with defining polynomial x^2 - 2
    over the Rational Field to FldNum: E>,
    <Number Field with defining polynomial x^3 + 2
    over the Rational Field,
    Mapping from:
    Number Field with defining polynomial x^3 + 2
    over the Rational Field to FldNum: E>,
    <Number Field with defining polynomial x^4 - 2
    over the Rational Field,
    Mapping from:
    Number Field with defining polynomial x^4 - 2
    over the Rational Field to FldNum: E>,
    <Number Field with defining polynomial x^6 - 2
    over the Rational Field,
    Mapping from: FldNum: K to FldNum: E>
]
```

From the degrees of these fields, the fourth subfield is isomorphic to L and the fifth subfield is isomorphic (indeed, equal) to K. The next lines confirm this, and construct the embeddings:

```
> print K eq sf[5, 1];
true
> K2E := sf[5, 2]; print K2E;
Mapping from: FldNum: K to FldNum: E
> print K2E(a);
-e^2
> print IsZero(Evaluate(DefiningPolynomial(K), -e^2));
true

> isiso, L2temp := IsIsomorphic(L, sf[4, 1]);
> print isiso;
true
> print L2temp;
Mapping from: FldNum: L
to Number Field with defining polynomial x^4 - 2
over the Rational Field
> L2E := L2temp * sf[4, 2];
```

```
> print L2E;
Mapping from: FldNum: L to FldNum: E
> print L2E(b);
-e^9
> print IsZero(Evaluate(DefiningPolynomial(L), -e^9));
true
```

The embeddings of K and L in E are the mappings *K2E* and *L2E*. In particular, the primitive elements of K and L map to $-e^2$ and $-e^9$, respectively.

25.5 Ideals of General Number Fields

The **ideal**-constructor allows the construction of ideals in orders of general number fields. For instance:

```
> R<x> := PolynomialRing(RationalField());
> N<n> := NumberField(x^3 - 2);
> O := MaximalOrder(N);
> I := ideal< O | n^2 - 5, 6*n^2 + 3*n - 7 >;
> print I, Norm(I);
Ideal of O
Basis:
[  1   0  24]
[  0   1  58]
[  0   0 121]
121
```

Table 25.10. Some ideal invariants

MAGMA	Meaning
Index(O, I)	Index of the ideal in the order O
Denominator(I)	Denominator of a fractional ideal
Norm(I)	Norm of a (fractional) ideal
Trace(I)	Trace of a (fractional) ideal
RamificationIndex(p, P)	Valuation of pO at prime ideal P
Degree(P)	Residue class degree of prime ideal P
MinimalInteger(I)	The least integer contained in an ideal

Table 25.10 lists the main functions for obtaining invariants of ideals in MAGMA.

Table 25.11. Functions relating to the representation of ideals

MAGMA	Meaning
Basis(I)	$\mathbb{Z}$-basis for ideal I as field elements
BasisMatrix(I)	Rational matrix representing a basis for I
TransformationMatrix(I)	Transformation matrix for I, with denominator
Generators(I)	Ideal generators as order elements
TwoElement(I)	Ideal generators as field elements
TwoElementNormal(I)	Two 'special' generators for I

Ideals in orders have two important representations, as listed in Table 25.11. The first, as shown in the example above, is by a $\mathbb{Z}$-basis of I. This is returned by the function **Basis**(I):

```
> print Basis(I);
[
    [121, 0, 0],
    [48, 1, 0],
    [116, 0, 1]
]
```

The other representation, which may be more expensive to compute initially but is often cheaper to use, is that of a two-element representation. (Any ideal in an order can be generated by at most two elements.) The functions **Generators**(I) and **TwoElement**(I) return two elements in the order and in the field, respectively, that generate the ideal I. For instance:

```
> print Generators(I);
[
    [121, 0, 0],
    [-169, -1, 0]
]
> print TwoElement(I);
121
-n - 169
```

The even more sophisticated **TwoElementNormal** presentation is also available.

Ideals may be multiplied, raised to a power, or multiplied by an order element (which is equivalent to multiplication by the principal ideal generated by the element). The prime factor decomposition of an ideal is returned by **Factorization**(I). In the example below, I is factorized into J^2:

```
> F := Factorization(I);
> print F;
[
    <Ideal of O
    Two element generators:
        [11, 0, 0]
        [4, 1, 0], 2>
]
> J := F[1][1];
> print J;
Ideal of O
Two element generators:
    [11, 0, 0]
    [4, 1, 0]
> print J^2 eq I;
true
```

Ideals can also be divided; the result, in general, is a *fractional ideal*, which has the property that for some integer d (the *denominator*) $d \cdot J$ is an ordinary, integral ideal. Fractional ideals are represented like integral ideals, but with an integral denominator. The factorization of the ideal generated by an ordinary prime number is given by **Decomposition**(O, p):

```
> print I/J eq J;
true
> print J^-1;
Fractional Ideal of O
Two element generators:
    [1, 0, 0]
    [16, -4, 1] / 11
> print Decomposition(O, 11);
[
    <Ideal of O
    Two element generators:
        [11, 0, 0]
        [4, 1, 0], 1>,
    <Ideal of O
    Two element generators:
        [11, 0, 0]
        [5, 7, 1], 1>
]
```

The function **IsPrincipal** returns **true** if a given fractional ideal can be generated by a single element (in which case such a generator is returned as

a second value), and **false** otherwise. (This test may consume some time in large orders.) For instance:

```
> print IsPrincipal(J);
true -n^2 - n + 1
```

Table 25.12. Other operations involving ideals

MAGMA	Meaning
Factorization(I)	Factorization of integral ideal I
Decomposition(O, p)	Factorization of ideal pO for prime p
Valuation(I, J)	Valuation of ideal I at prime ideal J
DegreeOnePrimeIdeals(O, b)	All degree-1 prime ideals in O up to norm b
IsZero(I)	**true** if I is zero ideal
IsIntegral(I)	**true** if ideal I is integral
IsPrincipal(I)	**true** (and generator) if I is principal
IsPrime(I)	**true** if I is prime
MultiplicatorRing(I)	Subring of elements multiplying I into itself
pRadical(O, p)	Ideal of elements with some power in pO

The decomposition functions, predicates, and some other functions are summarized in Table 25.12. The multiplicator ring and p-radical play a role in the algorithm for determining the maximal order.

25.6 Unit Group and Class Group

Two of the main tasks in the analysis of a number field are the determination of the class group and the unit group. It is customary to abuse terminology by speaking of the 'unit/class group of a number field'. Of course, every non-zero element of a field is a unit, and fields do not have non-trivial ideals. What is meant is the unit/class group for the ring of integers (maximal order) of the field. The class group of a number field gives information on the 'ideal structure' of the maximal order, and knowing the units of the maximal order is important for many applications.

MAGMA provides class group and unit group functions for both general number fields and quadratic fields, as listed in Table 25.13. *All these functions may also be applied to orders, except that* **ClassGroup** *may only be applied to an order of a general number field if it is known to be maximal.* The algorithms in the quadratic case are more efficient than the general ones.

Table 25.13. Class group and unit group

MAGMA	Meaning
ClassNumber(K)	The class number of the ring of integers of K
ClassGroup(K)	The class group of the ring of integers of K (as an abelian group) and a mapping
ClassGroupStructure(K)	Invariants of the (abelian) class group of K
UnitGroup(K)	The unit group of the ring of integers of K (as an abelian group) and a mapping
TorsionSubgroup(K)	Finite part of the unit group of K
UnitRank(K)	The free rank of the unit group of K
Regulator(K)	The regulator of the ring of integers of K

No special class group or unit group algorithms have been implemented for cyclotomic fields; if these structures are required, it is necessary to compute them via the general number field corresponding to the given cyclotomic field.

The *(ideal) class group* of an order in a number field is a finite abelian group of ideal classes equivalent under multiplication by principal ideals (ideals generated by a single element). The class number is the order (i.e., cardinality) of the class group. In a sense it measures how far the order is from being a principal ideal domain (a domain in which every ideal is principal). The methods used to determine class groups in MAGMA are variants of the relation method. A generating set of ideals is taken, relations between these generators are produced (usually by factoring the ideals into prime ideals), and finally it is established that no more relations exist (which usually involves unit computations and principal ideal testing). In general it is hard to determine the class group of (an order in) a number field with certainty. However, if the user does not insist on certainty, it is possible to take short-cuts (taking an initial small set of ideals that will most likely generate the class group, omitting expensive checks at the end, and so on) and find answers that are likely to be correct.

According to basic algebraic number theory, the units in an order form a finitely-generated abelian group, known as the *unit group*. Such a group will consist of a finite part (the *torsion subgroup*) and an infinite part, isomorphic to a finite number of copies (the *unit rank*) of the infinite cyclic group. Moreover, this unit rank is completely determined by the signature of the field: it equals $r + s - 1$. The work in determining the unit group then consists of two parts: finding the order of the torsion part and generators for it (these must be roots of unity, and this part is relatively easy); and finding generators for the free infinite part.

A few remarks on general features are required here. The computation of groups associated with number fields incorporates two aspects. In the first

place there is the structure of the group as an *abstract group*. In the second place there is the actual *representation* of that abstract group, in terms of the field (or a subring, or its ideals). In MAGMA these two aspects are separated: functions such as **ClassGroup** and **UnitGroup** return two values accordingly. The first value is the 'abstract group', consisting of a group A in MAGMA's abelian group category **GrpAb**. The second is a homomorphism, mapping A to its 'realization' (ideals, elements) inside an appropriate ring. For example, (the maximal order of) the quadratic field $\mathbb{Q}(\mathrm{i})$, where $\mathrm{i} = \sqrt{-1}$, contains 4 units, namely the powers of i. The function **UnitGroup** applied to this field will therefore return a cyclic group A of order 4, as well as a homomorphism sending a generator of A to either i or $-\mathrm{i}$:

```
> Q<i> := QuadraticField(-1);
> U, m := UnitGroup(Q);
> print U;
Abelian Group isomorphic to Z/4
Defined on 1 generator
Relations:
    4*U.1 = 0
> print m;
Mapping from: GrpAb: U to FldQuad: Q
> print m(U.1);
i
```

25.6.1 Units in Quadratic Orders

Quadratic fields $\mathbb{Q}(\sqrt{d})$ come in two kinds: real quadratic (with $d > 0$) and imaginary quadratic (with $d < 0$). In the real case there are two real roots (signature $r = 2, s = 0$), and in the imaginary case there is one pair of complex roots ($r = 0, s = 1$). Since it is also well known that the torsion consists of ± 1 (with two exceptions in the imaginary case, as given below), the sole remaining problem is that of finding a *fundamental unit*, that is, a generator for the infinite cyclic part of the unit group in the real case. One of the two exceptional torsion cases was seen above, namely $\mathbb{Q}(\mathrm{i})$, whose ring of integers $\mathbb{Z}[\mathrm{i}]$ contains the 4 roots of unity $1, \mathrm{i}, -1, -\mathrm{i}$. The other case is $\mathbb{Q}(\sqrt{-3})$:

```
> Q<w> := QuadraticField(-3);
> U, m := UnitGroup(Q);
> print U;
Abelian Group isomorphic to Z/6
Defined on 1 generator
Relations:
    6*U.1 = 0
```

```
> print m(U.1);
1/2*(1 + w)
```

Thus $\mathbb{Z}[\sqrt{-3}]$ contains the sixth roots of unity, the powers of $(1+\sqrt{-3})/2$.

Given a real quadratic field K, the function **FundamentalUnit**(K) returns a fundamental unit. Finding such an element in the real case is a difficult problem, since its coefficients may be very large. For instance:

```
> Q<z> := QuadraticField(100001);
> print FundamentalUnit(Q);
319588146987925012951271253501434553702636460448000624 66\
135444719474287192833499264395114934391894033480593751 +
101062140458013340064241220962154915379791795353179619 53\
309543166366332375187628985809498561038088435549950 0*z
```

The fundamental unit must have norm ± 1, that is, it provides a solution to an equation of the form $x^2 + dy^2 = \pm 4$ in integers (the 4 arises because ϵ_d may have denominator 2). The following statements print the values of $D < 50$ for which the norm is -1:

```
> D := { SquareFree(d) : d in [2..100] | not IsSquare(d) };
> print { d : d in D |
>   IsMinusOne(Norm(FundamentalUnit(QuadraticField(d)))) };
{ 2, 5, 10, 13, 17, 26, 29, 37, 41, 53, 58, 61, 65, 73,
74, 82, 85, 89, 97 }
```

In imaginary quadratic fields it is also possible to search for solutions to other equations of the form $x^2 + dy^2 = m$, using the function **NormEquation**:

```
> print NormEquation(QuadraticField(-3), 7);
true 1/2*(1 + 3*sqrt-3)
```

25.6.2 Class Groups of Quadratic Fields

An important aspect of the difference between quadratic fields and general number fields lies in the area of class groups. For quadratic fields, binary quadratic forms offer an attractive alternative to ideals for computation in the class group. It suffices to mention here that there exists a one-to-one correspondence between ideal classes and reduced binary quadratic forms of given discriminant in the imaginary quadratic case, and a one-to-one correspondence between ideal classes and 'cycles' of reduced forms in the real case.

Consequently, the **ClassGroup** function for the quadratic field $K = \mathbb{Q}(\sqrt{d})$ returns not only the abstract group A isomorphic to the class group, but also a mapping from A to the magma of binary quadratic forms of the same discriminant as K. For example:

```
> Q<w> := QuadraticField(-65);
> C, h := ClassGroup(Q);
Abelian Group isomorphic to Z/2 + Z/4
Defined on 2 generators
Relations:
    4*C.1 = 0
    2*C.2 = 0
> print h;
Mapping from: GrpAb: C
to The magma of binary quadratic forms
with discriminant D = -260
> print [ h(c) : c in C ];
[ <1,0,65>, <2,2,33>, <6,2,11>, <3,2,22>, <9,8,9>, <5,0,13>,
<6,-2,11>, <3,-2,22> ]
> F1 := h(C.1); print F1;
<6,2,11>
> print F1^3 eq -F1;
true
```

As the above output shows, binary quadratic forms $aX^2 + bXY + cY^2$ are printed in MAGMA as < a, b, c >. Such forms of fixed *discriminant* $D = b^2 - 4ac$ constitute a magma under an operation called *composition*. The magma of forms of a given discriminant D can be created with the function **QuadraticForms**(D). Its category is **MagForm**, and individual forms are in a category **MagFormElt**. See Table 25.14 for the operations on binary quadratic forms. Note that the **Reduction**(f) operation produces a form that is $\mathrm{SL}(2, \mathbb{Z})$-equivalent to f, and either unique (in the imaginary case) or in a cycle of reduced forms (in the real case).

The example below demonstrates how elements in real quadratic class groups are cycles of reduced forms, in the case of $\mathbb{Q}(\sqrt{145})$. The four cycles together contain all reduced forms of discriminant 145. These cycles are obtained by successive repetitions of **ReductionStep** until the cycle comes full circle:

```
> Q<w> := QuadraticField(145);
> C, h := ClassGroup(Q);
> print C;
Abelian Group isomorphic to Z/4
Defined on 1 generator
```

Table 25.14. Operations on binary quadratic forms

MAGMA	Meaning
Composition(f, g), f*g	Composition of forms f and g
Power(f, n), f^n	f*$\cdots$*f (n times)
$-f$, f^-1	Inverse < $a, -b, c$ >, where $f =$ < a, b, c >
Reduction(f)	A reduced form $\mathrm{SL}(2, \mathbb{Z})$-equivalent to f
ReductionStep(f)	A single step in the reduction of f
Discriminant(f)	Discriminant of f
PrimeForm(M, p)	Form < p, b, c > in M (if it exists)
Distance(f, g)	Distance between real quadratic forms f and g
IsReduced(f)	**true** if f is reduced
IsOne(f)	**true** if f is the principal form
IsMinusOne(f)	**true** if f is inverse of the principal form

```
Relations:
    4*C.1 = 0
> M := Codomain(h); print M;
The magma of binary quadratic forms
with discriminant D = 145
> print ClassNumber(Q);
4
> for c in C do
>     F := h(c); G := F; S := [ ];
>     repeat
>         Append(~S, G);
>         G := ReductionStep(G);
>     until G eq F;
>     printf "Cycle of reduced forms:\n%o\n", S;
>     printf "Size of cycle: %o\n", #S;
> end for;
Cycle of reduced forms:
[ <1,11,-6>, <-6,1,6>, <6,11,-1>, <-1,11,6>, <6,1,-6>,
<-6,11,1> ]
Size of cycle: 6
Cycle of reduced forms:
[ <-8,9,2>, <2,11,-3>, <-3,7,8>, <8,9,-2>, <-2,11,3>,
<3,7,-8> ]
Size of cycle: 6
Cycle of reduced forms:
[ <6,7,-4>, <-4,9,4>, <4,7,-6>, <-6,5,5>, <5,5,-6>,
<-6,7,4>, <4,9,-4>, <-4,7,6>, <6,5,-5>, <-5,5,6> ]
Size of cycle: 10
```

```
Cycle of reduced forms:
[ <3,11,-2>, <-2,9,8>, <8,7,-3>, <-3,11,2>, <2,9,-8>,
<-8,7,3> ]
Size of cycle: 6

> H := M ! < 6, -11, -1>;
> print IsReduced(H), IsReduced(-H);
false true
> print Reduction(H);
<-1,11,6>
```

From the output, there are 4 cycles of reduced forms, whence the class number is 4. Each cycle contains a (variable) number of equivalent, reduced classes.

25.6.3 Units of General Number Fields

Recall that the unit group of an order consists of a finite part (the torsion group) and a finitely-generated free infinite part. Given a general number field or an order of such a field, the function **TorsionSubgroup** usually returns the finite part very quickly. For the free part, the function **UnitGroup**,, **pFundamentalUnits** and **IndependentUnits** are provided. Each of these functions returns two values: an abelian group A, together with a homomorphism from A to the order.

The function **UnitGroup** determines the torsion and free parts of the unit group, thus providing all information about the unit group at once. However, the computations necessary may be very lengthy, and it is therefore possible to obtain partial information about the units in several ways. Firstly, **IndependentUnits** determines a subgroup of finite index in the full unit group. Hence it provides a maximal set of independent elements in the unit group, but the user may have to extract certain roots in order to obtain true generators. The function **pFundamentalUnits** returns a subgroup of the unit group that is p-fundamental at some prime p, i.e., its index is coprime to p (so no p^{th} roots need be taken to obtain the full unit group). It may be much faster to determine such subgroups.

Another useful feature enables the user to supply information about units, through the function **MergeUnits**. The effect is to incorporate any unit the user provides into the stored information about the units. The function returns **true** if the rank of the subgroup of units increases, else **false**.

For example, consider the field $\mathbb{Q}(\sqrt[4]{-3})$:

```
> R<x> := PolynomialRing(RationalField());
> M<m> := NumberField(x^4+3);
```

```
> T, t := TorsionSubgroup(M);
> print T;
Abelian Group isomorphic to Z/6
Defined on 1 generator
Relations:
    6*T.1 = 0
> print t(T.1);
[1, 0, 1, 0]
> u0 := M ! t(T.1);
> print u0;
1/2*m^2 + 1/2
> print UnitRank(M);
1
```

From the output, the ring of integers of M (which has not been created explicitly!) contains only the (expected) sixth roots of unity, namely the powers of $(\sqrt{-3}+1)/2$. It is also now known that one free generator is required. In the next lines, a unit is found, and used to extend the known subgroup:

```
> print { <a, b> : a, b in [-5..5] | Norm(m^2+a*m+b) eq 1 };
{ <-2, 2>, <2, 2> }
> print MergeUnits(M, m^2-2*m+2);
true
> I, i := IndependentUnits(M);
> u1 := M ! i(I.2);
> print u1;
m^2 - 2*m + 2
```

This furnishes independent units. Now the 2-fundamental part of the unit group and the full unit group are determined:

```
> V, v := pFundamentalUnits(M, 2);
> u2 := M ! v(V.2);
> print u2;
-1/2*m^3 + 1/2*m - 1
> U, u := UnitGroup(M);
> u3 := M ! u(U.2);
> print u3;
1/2*m^3 - 1/2*m^2 + 1/2*m + 1/2
```

It emerges that the group V was the full unit group already, since $u2$ and $u3$ differ by a root of unity, and that the subgroup I was of index 2:

```
> tU := { u0^i : i in [1..6] }; // roots of unity in M
> print u3*u2^-1 in tU;
```

```
true
> print u3*u1^-1 in tU;
false
> print Logs(u1), Logs(u3);
[ -1.6628858910586214, 1.6628858910586211 ]
[ -0.83144294552931051793, 0.83144294552931051793 ]
> print u3^2*u1^-1 in tU;
true;
```

If only the regulator of an order is required, **Regulator** can be invoked. This function will trigger the computation of units (and of the maximal order) if necessary.

The table lists the real-valued functions we mentioned, and some others.

Table 25.15. Some real-valued functions

MAGMA	Meaning
Regulator(O)	Regulator of order O
Conjugates(a)	Sequence of conjugates of a as complex numbers
Conjugate(a, k)	k^{th} term of above sequence
AbsoluteValues(a)	Sequence of absolute values of conjugates of a, as real numbers
Logs(a)	Sequence of logarithms of absolute values of conjugates of a, as real numbers
Length(a)	T_2 norm of a
Zeros(O, n)	Zeros of defining polynomial of O up to precision at least n

There is another way the user may supply information to MAGMA in order to avoid expensive computation. The functions **SetOrderMaximal**, **SetOrderUnitsAreFundamental**, and **SetOrderTorsionUnits** may be used to set flags indicating that an order is maximal, or that the units found so far are fundamental, or to supply all torsion units.

MAGMA also provides a function to calculate exceptional units and to determine S-units and S-class groups. See the *Handbook*.

25.6.4 Class Groups of General Number Fields

Fractional ideals, under equivalence up to multiplication by principal ideals, form a group known as the *ideal class group* of the order. In MAGMA, class group computations for general number fields are restricted to the maximal

order of the field. The main method employed to determine the ideal class group of the maximal order is the relation method. It can be briefly described as follows.

In the first step a list of prime ideals is generated, all having norm below some given bound. This is the *factor basis*. In the second step a search is conducted to find a few elements in each of these prime ideals such that the principal ideal generated factors completely over the factor basis. These provide relations between the ideals in the factor basis, and from them, a basis for the class group is determined. Finally, it is checked that the correct orders for the basis elements are found. Here knowledge about units in the order is required.

Table 25.16. Class group functions for general number fields

MAGMA	Meaning
MinkowskiBound(K)	Minkowski bound for K
BachBound(K)	Bach bound for K
FactorBasis(K, b)	Factor basis of primes of norm below b in maximal order of K
RelationMatrix(K, b, n)	Matrix representing n relations for each ideal in factor basis with bound b over maximal order of K

To obtain results that are guaranteed to be correct, the factor basis needs to contain all prime ideals below the Minkowski bound; assuming certain Riemann hypotheses the (usually much lower) Bach bound may be used. For both values a function is provided, as listed in Table 25.16. The functions **FactorBasis** and **RelationMatrix** allow the user to generate the necessary relation matrix, after which matrix functions can be used to complete the relation algorithm. (All these functions can take the maximal order as an alternative to the field.)

The functions **ClassNumber**, **ClassGroupStructure** and **ClassGroup** provide a complete implementation of the method. They differ in the form of the output, yielding the mere cardinality of the group, the structure of the abelian group, and the group itself, respectively. (As its second return value, **ClassGroup** provides a mapping from the abelian group to the set of ideals.) In the default version all three take just a general number field as their argument (the maximal order is allowed as substitute for the field). The default version uses the Minkowski bound and executes the complete relation method. In practice, for even moderately sized fields, this takes a very long time to finish. Moreover, with a much smaller bound correct results will usually be obtained, in a reasonable time. By means of parameters, the user

can instruct MAGMA to employ such a modified method. The most important parameters are **Bound** (for an alternative to the Minkowski bound) and **Check**, which, when set to **false**, instructs MAGMA not to perform potentially expensive checks at the end of the algorithm. It needs to be stressed that the use of these parameters may invalidate the results: if the bound is too small, it may be that only a subset of the generators is used (and hence the result will be too small), whereas omitting the check at the end may mean that not all relations are found and that the result is too large.

A typical example is given below:

```
> N<n> := NumberField(x^8 - 15);
> C, h := ClassGroup(N : Bound := 1000, Check := false);
> print C;
Abelian Group isomorphic to Z/4 + Z/4
Defined on 2 generators
Relations:
    4*C.1 = 0
    4*C.2 = 0
```

26. The Real and Complex Fields

Section 3.5 provided a brief introduction to real and complex numbers in MAGMA. This chapter explains how to calculate in a real field or complex field. By the nature of $\mathbb{R}$ and $\mathbb{C}$, the real and complex fields constructed by MAGMA are different from most other MAGMA fields: computation in them is not exact, only approximate. Therefore these 'fields' are not true fields; in fact, they are not even commutative rings. However, MAGMA's algorithms for real and complex arithmetic have been designed to produce results that are accurate within whatever precision is chosen by the user.

26.1 The Two Kinds of Real and Complex Fields

There are two kinds of real fields in MAGMA: *free fields* and *fields of fixed precision*. In the free real field, each element has its own relative precision, which may be as high as the user specifies, or else infinite (for integers and rationals considered as exact reals). MAGMA's implementation of these fields is based on the PARI package. On the other hand, in a fixed precision real field every element of the field has the same precision, which once again may be arbitrarily high. MAGMA's implementation of fixed precision fields is based on R. P. Brent's MP-package.

There are also two kinds of complex fields. The elements of the free complex field have real and imaginary parts which are each members of a free real field. The real and imaginary parts may have different relative precisions, and the overall relative precision of the element is the smaller of the two. By contrast, the elements of a fixed precision complex field of precision p have real and imaginary parts which are each members of a fixed precision real field of the same precision p.

The functions for creating these fields are **RealField** and **ComplexField**. For the free real and complex fields, the function takes no arguments, but for a fixed precision real or complex field, it takes a precision p as its argument:

```
> R := RealField();
```

```
> print R, Category(R);
Real Field
FldPr
> R32 := RealField(32);
> print R32, Category(R32);
Real Field of precision 32
FldRe
> C<i> := ComplexField();
> print C, Category(C);
Complex Field
FldPr
> C32<j> := ComplexField(32);
> print C32, Category(C32);
Complex Field of precision 32
FldCom
```

When a complex field is created, the angle bracket notation (generator assignment) is the best way of giving the element $\sqrt{-1}$ the special name i (or j, or any other chosen name).

Given a fixed precision field, the function **Precision** returns its precision. For instance:

```
> print Precision(R32), Precision(C32);
32 32
```

26.2 Working in the Default Real Field

For many purposes, it is sufficient to know how to work within the *default real field*, that is, the real field which MAGMA uses unless there is an instruction to the contrary. The notation $i.j$, where i and j are each a succession of decimal digits, always creates the corresponding real number as an element of the default real field. For example:

```
> a := -18.394;
> print Parent(a);
Real Field
```

This assigns the real number -18.394 to the identifier a. The parent of a will be the default real field, which in this case is the free field.

Scientific notation should be used in order to express very large or very small real numbers. Immediately after the decimal number $i.j$ the user may

put `e` followed by an integer s. MAGMA evaluates this as the real number which is $i.j \times 10^s$. For instance, the following line assigns to b the real number 901.2×10^{-14}:

```
> b := 901.2e-14;
```

Again, the parent of this element will be the default real field.

It is sometimes useful to go in the other direction, that is, to express a real number x as $m \cdot 10^e$, where m is a real number and e is an integer. The function **MantissaExponent**(x) returns both these values, but since there are an infinite number of correct responses, it returns m and e such that $1 \leq m < 10$:

```
> mant, exp := MantissaExponent(b);
> print mant, exp;
9.011999999999999999999999998481 -12
```

The output indicates that b equals 9.012×10^{-12}. (The trailing 9's in the output should not be a matter for concern. They often arise as the consequence of converting from the internal base to which free reals are stored.)

The concept of *relative precision* is based on the concept of mantissa and exponent. The relative precision of a real number x, returned by **Precision**(x), is the number of significant decimal digits in the mantissa. For instance, b and *mant* must have the same relative precision:

```
> print Precision(b);
28
> print Precision(mant);
28
```

The next section explains why the relative precision of these numbers is 28.

When MAGMA starts up, the default real field is the free field. The following example demonstrates how to obtain the current default, and how to reset it:

```
> print GetDefaultRealField();
Real Field
> SetDefaultRealField(RealField(48));
> print GetDefaultRealField();
Real Field of precision 48
> print Parent(a);
Real Field
> c := 1.4;
```

```
> print Parent(c);
Real Field of precision 48
```

Notice that a change in the default real field does not change the parent of elements that already exist.

This chapter contains many functions that take a real number as an argument and return a real number. Whether the parent of the return value is a free field or fixed precision field depends on the parent of the input. For instance:

```
> print Parent(Sin(a));
Real Field
> print Parent(Sin(c));
Real Field of precision 48
```

However, these functions often accept other categories of input (such as integers, rationals, quadratic field elements and cyclotomic field elements) if the input can be coerced into a real number. If this is possible, the input is coerced into the default real field, so the output will also belong to that field. For instance, the current default real field in the example is the real field of fixed precision 48. When a real function is applied to a rational number, the rational is coerced automatically into that field, and the return value will belong to it too:

```
> print Parent(Sin(3/4));
Real Field of precision 48
```

The rules for free fields and fixed precision fields are explained in the following sections.

26.3 The Free Real Field

In the examples of this section it will be assumed that the default real field has been set to the free real field (as is the case at the beginning of a MAGMA session). The following command ensures that this is true:

```
> SetDefaultRealField(RealField());
```

In the free field, a real number may be created with a relative precision of arbitrary size (within reason). If the real number has a precision greater than 28 (or the current default precision), it may be typed directly, as explained above. MAGMA will automatically provide sufficient precision to store every digit of the number. For example:

```
> d := 1.23456789012345678901234567890123456789012345678e22;
> print d;
12345678901234567890123.45678901234567890123456793305
> print Precision(d);
48
```

If the mantissa of the number ends with many trailing but significant zeros, it can be inconvenient to type them all. However, omitting the zeros gives a different result:

```
> f1 := 5.55500000000000000000000000000000000000e6;
> print Precision(f1);
38
> f2 := 5.555e6;
> print Precision(f2);
28
```

The solution to this problem is to follow the real number immediately by `p` and the desired precision:

```
> f3 := 5.555e6p38;
> print Precision(f3);
38
```

Not every integer may be used internally as a precision, because of the way that MAGMA stores these numbers. Therefore MAGMA will sometimes increase the requested precision slightly. This should not cause any major difficulties. For example:

```
> print Precision(7.9p50), Precision(7.9p80);
57 86
```

If no precision is given and the number of digits in the mantissa is smaller than 28 (or the current default precision), then MAGMA will choose 28 as the precision rather than the number of mantissa digits. This is what happened in the calculation of *f2*, above. In order to override this, the desired precision must be specified using `p`:

```
> f4 := 5.555e6p9;
> print Precision(f4);
9
```

The free real field can also contain exact elements. They are created when an integer or rational is coerced into the field. The relative precision of an exact real is infinity, but MAGMA represents it as -1. For instance:

```
> f5 := R!(22/7); print f5;
22/7
> print Parent(f5);
Real Field
> print Precision(f5);
-1
```

The discussion above included several instances where the free field makes use of the number 28. This number is the default value of the *default precision attribute for the free field*. If the user finds that 28 is an unsuitable default value for this attribute, then the value should be changed. It is returned and altered by the function **HasAttribute** and the procedure **AssertAttribute**. These intrinsics are used in several contexts within MAGMA. When they are used with reference to the default precision attribute of the free field, the first argument must be the category of the free field, **FldPr**, and the second argument must be the string `"Precision"`. **AssertAttribute** takes a third argument, the new default precision. The function **HasAttribute** returns two values: a Boolean specifying whether the attribute is defined (this will always be **true** in this case, so it can be ignored); and the current value of the attribute. For example:

```
> print HasAttribute(FldPr, "Precision");
true 28
> AssertAttribute(FldPr, "Precision", 57);
> print HasAttribute(FldPr, "Precision");
true 57
> print Precision(f2);
28
> print Precision(7.9);
57
```

Note carefully that the change to the attribute does not affect the precision of previously existing elements of the free real field. For elements created after the change in attribute, the new attribute value (57 in this example) will behave in exactly the same way as 28 did before.

One of the applications of the default precision attribute is to state the precision when an object that is not an element of the free real field R is *coerced* into R. This can happen as an explicit coercion:

```
> Qu3<rt3> := QuadraticField(3);
> print R!rt3;
1.732050807568877293527446341505872366942805253810380628055
> print Precision(R!rt3);
57
```

or when a real function accepts several categories of input (e.g., reals, integers, rationals) but coerces the input into the free real field before performing the operation:

```
> print Exp(30);
10686474581524.462146990468650741401650024495005473054990\
24923
> print Precision(Exp(30));
57
```

(The above example has been calculated using the free real field as the default field.)

26.4 The Fixed Precision Real Fields

In a fixed precision real field, every real number in the field has the same precision. The function **RealField**(p) returns the real field of fixed precision p. More precisely, although any positive integer may be given as the precision, MAGMA always rounds the input up to the nearest multiple of 4 to find the precision, because it stores elements of fixed precision real fields internally to a base of 10^4. For instance, a request for a precision of 29 or 30 or 31 will result in a precision of 32, just as if the user had requested that precision directly.

Before working within a real field R of fixed precision p, it is advisable to make that field into the default real field, with the function **SetDefaultRealField**(R). (The resulting value may be checked with the function **GetDefaultRealField**().) For instance, the instruction used to specify a real field of precision 16 is:

```
> SetDefaultRealField(RealField(16));
> print GetDefaultRealField();
Real Field of precision 16
```

After the default field has been set in this way, every input of the form $i.j$ or $i.j$*es* will produce a member of this field, with that precision:

```
> g1 := 3.893276e-217;
> print g1;
3.893276e-217
> print Parent(g1);
Real Field of precision 16
> g2 := 1234.567812345678123456781234567812354678;
```

```
> // NB: some digits are lost
> print g2;
1234.567812345678
> print Precision(g2);
16
```

It is important to realize that the interpretation of the `p` notation depends on the current category of default real field. If the default field is the free field, then `p` means to create a free field element with an unusual precision; this was demonstrated in the previous section. If the default field is one of the fixed precision fields, then `p` indicates a request to create an element in a field of given fixed precision, rather than in the default field. For example, if the default field is the field of precision 16, then the `p` notation may be used to create a real number of some other precision, as an element of another fixed precision field:

```
> g3 := 1234.56781234567812345678123456781235467 8p60;
> print g3;
1234.567812345678123456781234567812354678
> print Precision(g3);
60
> print Parent(g3);
Real Field of precision 60
```

It is only approximately true that the precision of fixed precision reals is the same as the number of significant figures in decimal form. The truth is a little more complicated. Since the digits on the left and right sides of the decimal point are stored separately, every block or part-block of 4 digits on each side requires 4 digits of precision. The following example demonstrates this:

```
> print 61532.61437p8;
61533
> print 61532.61437p12;
61532.6144
> print 61532.61437p16;
61532.61437
```

In this example, the part on the left of the decimal point in 61532.61437 consists of 5 digits, so 8 precision digits are required to store it. In the first statement, since only 8 digits are available, MAGMA evaluates only this left part, rounded up because the part on the right of the decimal point is not less than a half. In the second example, the 12 precision digits are used to represent the left part and the first four digits of the right part, rounded once again. In the last example, there are sufficient digits for the complete

number. It follows that to store a fixed precision real number successfully, the precision must be set at least 6 digits higher than the required number of significant figures.

Many kinds of ring elements may be coerced into a fixed precision real field. For example:

```
> Qu5<rt5> := QuadraticField(5);
> print RealField(52)!(2 + 3*rt5);
8.708203932499369089227521006193828706321855078836
> print 2 + 3*Sqrt(5p52); // comparison
8.708203932499369089227521006193828706321855078836
```

Coercion may also be used to convert a real number into the equivalent number with a different precision, by coercing it into the appropriate fixed precision real field. If the new precision is less than the old precision, then rounding will occur if necessary, and if the new precision is greater than the old precision, the number is padded with trailing zeros until the required precision is reached. (Consequently, once the precision of a number is reduced, the 'lost' digits can never be regained by coercing into a higher precision.) For example:

```
> rt3 := Sqrt(3.0p12); print rt3;
1.73205081
> rt3prec8 := RealField(8)!rt3; print rt3prec8;
1.7321
> rt3prec48 := RealField(48)!rt3; print rt3prec48;
1.73205081
> print RealField(100)!rt3prec8;
1.7321
```

The `e` and `p` notations are limited to contexts where the exponent and precision are known as literals. In situations where they are only known as expressions, such as in the course of a function, the **elt** constructor should be used instead:

elt< RealField(p) | t, s >

This constructor creates the real number $t \cdot 10^s$ of precision p, where t is an expression coercible into a real, and p and s are integer-valued expressions. (The user should be careful about the precision of t, if it is given as a real literal in the default real field, to ensure that the result is sufficiently precise.) The following example shows how to express the number $\frac{42}{9} \times 10^7$ as an element of the real field of fixed precision given by the identifier *prec*:

```
> prec := 52;
```

```
> print elt< RealField(prec) | 42/9, 7 >;
46666666.666666666666666666666666666666666666666667
```

The coercion expression $R!t$, where R is a fixed precision real field and t is coercible into R, is equivalent to **elt<** $R \mid t, 0$ **>**.

26.5 The Complex Fields

To create a complex number it is usually necessary to call the function **ComplexField**() or **ComplexField**(p) so as to define the parent field of the number:

```
> C<i> := ComplexField();
> print C;
Complex Field
> C32<j> := ComplexField(32);
> print C32;
Complex Field of precision 32
```

MAGMA considers an element z of a complex field to have two parts, the real part $x = \mathbf{Re}(z)$ and the imaginary part $y = \mathbf{Im}(z)$, such that $z = x + iy$. If the parent of z is a fixed precision complex field, then x and y both belong to the corresponding real field with the same fixed precision. If the parent of z is the free complex field, then x and y both belong to the free real field, and they may have different precisions (including -1).

The easiest way to construct the complex number $x+iy$ is to type it as $x+i*y$, where i is the identifier that has been assigned $\sqrt{-1}$ in the complex field F. Another method is to use the expression $F![x, y]$. Both of these methods coerce x and y into the free or fixed real field corresponding to F. For instance:

```
> print 2 + 3*j;
2 + 3*j
> print C32 ! [2, 3];
2 + 3*j
> print 5/7 + 2.4*i;
5/7 + 2.399999999999999999999999999969*i
> print C ! [5/7, 2.4];
5/7 + 2.399999999999999999999999999969*i
```

The last example has been calculated when the default real field is the free field, so $\frac{5}{7}$ has remained an exact value.

Another way of creating a complex number is to give its polar coordinates. The function **PolarToComplex**(m, a) returns the complex number $me^{ai} = m(\cos a + i \sin a)$. To make use of this function, the function **Pi**(R) is helpful; it returns π to the same precision as the real field R. Therefore a number such as $8(\cos(\frac{\pi}{6}) + i \sin(\frac{\pi}{6}))$ may be created in the following way:

```
> pi := Pi(R32); print pi;
3.14159265358979323846264338 33
> print PolarToComplex(8, pi/6);
6.92820323027550917410978536 6 + 4*j
```

Within the precision, the result equals $4\sqrt{3} + 4j$.

Coercion into a complex field works in the same way as coercion into a real field. Note that all elements of quadratic and cyclotomic fields are coercible into a complex field, whereas only some of them (those with zero imaginary parts) are coercible into a real field:

```
> Q7<sev> := CyclotomicField(7);
> print C ! sev;
0.623489801858733530525004884 00
+ 0.781831482468029808708444526665775860168*i
```

26.6 Elementary Real and Complex Operations

Table 26.1. Elementary real-number functions

MAGMA	Meaning
Abs(r)	Absolute value $\|r\|$ of r
Norm(r)	$\|r\|$ if r is a fixed precision real, else $\|r\|^2$
Sign(r)	Returns $-1, 0$, or 1 depending on whether r is negative, zero, or positive respectively
Sqrt(r)	Square root of non-negative real r
Root(r, n)	n^{th} root of r, if a real root exists
Floor(r)	The greatest integer $\lfloor r \rfloor$ less than or equal to r
Ceiling(r)	The smallest integer $\lceil r \rceil$ greater than or equal to r
Round(r)	The integer nearest r ($n+\frac{1}{2}$ is rounded to $n+1$, for $n \in \mathbb{Z}$)
Truncate(r)	The integer part of r, or r rounded towards zero, i.e., **Sign**(r)*$\lfloor \|r\| \rfloor$

Table 26.2. Elementary complex-number functions

MAGMA	Meaning
Re(c), **Real**(c)	Real part of c
Im(c), **Imaginary**(c)	Imaginary part of c
IsReal(c)	**true** if **Im**(c) is zero
IsIntegral(c)	**true** if **Im**(c) is zero and **Re**(c) is an integer
PolarToComplex(m, a)	Complex number $c = me^{ia}$, with modulus m and argument a
ComplexToPolar(c)	Returns modulus m and principal argument a of c such that $c = me^{ia}$
Modulus(c)	$\lvert c\rvert$
Arg(c), **Argument**(c)	Principal argument of c, in radians
Norm(c)	$\lvert c\rvert^2$
ComplexConjugate(c)	Complex conjugate $x - yi$ of $c = x + yi$
Sqrt(c)	Square root of c

The operations of elementary arithmetic on real and complex numbers can be performed using the usual MAGMA operators. There are also the functions **Zero**(F) and **One**(F), which return the 0 and 1 of a real or complex field F. If a binary operation is performed on a combination of a real/complex number and an object coercible into a real/complex field, then MAGMA will automatically coerce the second object into a suitable field, with appropriate precision.

The customary Boolean functions for field elements apply to the real and complex fields: **IsZero**(a), **IsOne**(a), and **IsMinusOne**(a). The operators **eq** and **ne** and all the membership operators can also be used in both the real fields and the complex fields. The ordering operators such as **lt** are applicable to the real fields but not the complex fields, since ordering is not a property of the complex field. All of these Boolean tests examine the number as it is, with its given precision; that is, they require the test to hold within the precision of the argument(s).

The functions **PolarToComplex**(m, a) and **ComplexToPolar**(c) are inverses, in a sense: the first takes the modulus and argument of a complex number and returns the number, and the second takes the number and returns the modulus and argument. These values may also be obtained separately, from the functions **Modulus**(c) and **Arg**(c) (or **Argument**(c)). Since the identity $me^{ia} = me^{i(a+2\pi k)}$ for all $k \in \mathbb{Z}$ poses an ambiguity for **Arg**(c), the value a that is returned is the principal argument, i.e., the value satisfying $-\pi < a \leq \pi$.

Table 26.1 and Table 26.2 (p. 496) list the elementary functions on real and complex numbers.

26.7 Standard Real Constants

Three functions return standard real constants, to the precision of a given real or complex field F. If F is complex then the return value is a complex number whose imaginary part is zero and whose real part approximates the constant. The most important of these functions is **Pi**(F):

```
> print Pi(RealField(28));
3.1415926535897932384626433
```

The other two functions are **EulerGamma**(F), which returns Euler's Γ constant, and **Catalan**(F), which returns Catalan's constant, the sum given by $\sum_{k=0}^{\infty}(-1)^k(2k+1)^{-2}$:

```
> print EulerGamma(R);
0.57721566490153286060651209008
> print Catalan(RealField(52));
0.91596559417721901505460351493238411077414937428167 21
```

26.8 Advanced Real and Complex Functions

The functions in this section return results with the same precision as the precision of the arguments. Most of them accept both real and complex input, from either kind of field, but in some cases only elements of the free fields are accepted. If the function is given integer or rational arguments, then the return value will belong to the default real field (or its complex version).

These functions are designed to be fast, numerically stable, and accurate to the very last digit.

26.8.1 Transcendental Functions

Given a real or complex number c, **Exp**(c) returns e^c, the exponential function of c. The inverse function, **Log**(c), returns $\log_e c$. Moreover, **Log**(b, c) calculates $\log_b c$, the logarithm of c to the base b. For instance:

```
> print Exp(1p48);
2.7182818284590452353602874713526624977572470936977
> print Log(10);
2.3025850929940456840179914546 58
> print Log(10, 1000);
3.0000000000000000000000000000
```

Table 26.3. Exponential and logarithmic functions

MAGMA	Meaning
Exp(c)	Exponential e^c of c
Log(c)	Principal value of logarithm of c to base e
Log(b, c)	Logarithm of c to real base b
Dilog(c)	Principal value of dilogarithm $\mathrm{Li}_2(c)$
Polylog(m, c)	Principal value of polylogarithm $\mathrm{Li}_m(c)$
PolylogD(m, c)	Principal value of modified version $\tilde{D}_m$ of $\mathrm{Li}_m(c)$
PolylogDold(m, c)	Principal value of modified version D_m of $\mathrm{Li}_m(c)$
PolylogP(m, c)	Principal value of modified version P_m of $\mathrm{Li}_m(c)$

Table 26.3 lists these and other logarithmic functions.

26.8.2 Trigonometric and Hyperbolic Functions

Table 26.4. Trigonometric and hyperbolic functions

Sin(c)	**Arcsin**(c)	**Sinh**(c)	**Argsinh**(c)
Cos(c)	**Arccos**(c)	**Cosh**(c)	**Argcosh**(c)
Tan(c)	**Arctan**(c)	**Tanh**(c)	**Argtanh**(c)
Cosec(c)	**Arccosec**(c)	**Cosech**(c)	**Argcosech**(c)
Sec(c)	**Arcsec**(c)	**Sech**(c)	**Argsech**(c)
Cot(c)	**Arccot**(c)	**Coth**(c)	**Argcoth**(c)
Sincos(c)	**Arctan2**(b, a)		

As shown in Table 26.4, MAGMA provides the full range of trigonometric and hyperbolic functions such as **Sin** and **Sinh**, together with their inverses such as **Arcsin** and **Argsinh**. The only two functions whose meanings are not immediately apparent are **Sincos** and **Arctan2**. **Sincos**(c) returns two values, the sine and the cosine of c. **Arctan2**(b, a) returns a number whose tangent is $\frac{b}{a}$ and which is in the quadrant indicated by the signs of b and a.

The angles are measured in radians, so to calculate in degrees it is necessary to multiply by the conversion factor $\frac{\pi}{180}$:

```
> DegToRad := Pi(RealField(20)) / 180; print DegToRad;
0.01745329251994329556
> print Sin(30 * DegToRad);
0.49999999999999999456
> print Cos(45 * DegToRad);
```

```
0.70710678118654753 10
```

The examples indicate that $\sin 30^\circ = \frac{1}{2}$ and $\cos 45^\circ = \frac{1}{\sqrt{2}}$. The answers are not exact, since *DegToRad* does not equal $\frac{\pi}{180}$ to infinite precision.

The inverse trigonometric functions return values in the customary ranges, that is, 0 to π for **Arccos** and **Arcsec**, and $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ for the other four.

26.8.3 Summation of Infinite Series

Magma includes functions for evaluating infinite sums: **InfiniteSum**(m, i), **PositiveSum**(m, i), and **AlternatingSum**(m, i). Here m is a map from the integers to the free real field, such that $m(n)$ is the n^{th} term of the sum. The summation starts at term i, where i is a non-negative integer.

Each of the three functions returns essentially the same result. The first of them, **InfiniteSum**, simply evaluates successive terms of the series and adds them together, until the result is within the default precision of the free real field. The other two functions employ special techniques to do this more effectively: **PositiveSum** is designed for series in which every term is positive, and **AlternatingSum** is designed for series in which the terms alternate in sign.

For example, consider the series

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

and

$$e = 1 + \frac{1}{2!} + \frac{1}{3!} + \cdots$$

They may be verified as follows:

```
> Z := IntegerRing();
> R := RealField();
> AssertAttribute(FldPr, "Precision", 57);
> m := map< Z -> R | n :-> (-1)^n / (2*n+1) >;
> print Pi(R)/4 - AlternatingSum(m, 0);
1.5930919107 E-58

> f := map< Z -> R | n :-> 1/Factorial(n) >;
> print Exp(1) - InfiniteSum(f, 0);
-4.3769182618 E-58
```

26.8.4 Continued Fractions

MAGMA provides several functions for continued fractions. Given a free real number r, **ContinuedFraction**(r) returns the partial quotients in the continued fraction expansion of r, as a sequence of integers. As many partial quotients will be given as the precision of r permits. For example:

```
> pi := Pi(RealField());
> cf := ContinuedFraction(pi);
> print cf;
[ 3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2,
1, 1, 2, 2, 2, 2, 1, 84, 2, 1, 1, 15, 3 ]
```

The function **BestApproximation**(r, n) returns the best rational approximation to r with denominator not exceeding n. This calculation is made using continued fraction techniques. Continuing the example:

```
> print BestApproximation(pi, 10);
22/7
> print BestApproximation(pi, 130);
355/113
> print 355/113 - pi;
0.00000026676418906242231236899895735 6
```

Finally, **Convergents**(Q) returns the convergents matrix of the integer sequence Q, where Q is interpreted as a sequence of partial quotients. This matrix is a 2×2 integer matrix m such that m_{12}/m_{22} and $m_{11}/m21$ form the last two convergents for r as provided by Q. For instance:

```
> cv := Convergents(cf);
> print cv;
[428224593349304 139755218526789]
[136308121570117  44485467702853]
> print cv[1,2]/cv[2,2] - pi;
1.9965585740 E-28
> print cv[1,1]/cv[2,1] - pi;
5.21276888810 E-29
```

26.8.5 Special Functions

Table 26.5 (p. 501) shows MAGMA's intrinsics for some other standard real functions. Many of them are available for elements of the free field only.

Table 26.5. Special functions

MAGMA	Meaning
JacobiTheta(q, z)	First Jacobi θ function $\theta(q, z)$
JacobiThetaNull(q, k)	k^{th} derivative of $\theta(q, z)$ at $z = 0$
DedekindEta(c)	Dedekind's function $\eta(c)$
jInvariant(c)	Elliptical j-invariant $j(c)$
WeberF(c)	Weber's function $f(c)$
WeberF2(c)	Weber's function $f_2(c)$
Gamma(c)	$\Gamma(c)$
Gamma(x, y)	Incomplete γ function $\gamma(x, y)$
GammaD(c)	$\Gamma(c + \frac{1}{2})$
LogGamma(c)	Principal value of $\log \Gamma(c)$
LogDerivative(c), **Psi**(c)	Principal value of $\Psi(c) = \frac{d \log \Gamma(c)}{dc}$
BesselFunction(n, r)	Bessel function of the 1st kind $J_n(r)$, where r is in a fixed precision real field
JBessel(n, r)	Bessel function of the 1st kind of $\frac{1}{2}$-integral index $J_{n+\frac{1}{2}}(r)$, where r is in a free real field
KBessel(n, r)	Modified Bessel function of the 2nd kind $K_n(r)$
KBessel2(n, r)	Another implementation of **KBessel**(n, r)
HypergeometricU(a, b, r)	Confluent hypergeometric fn $U(a, b, r)$
AGM(c, d)	Arithmetic-geometric mean of c and d
BernoulliNumber(n)	n^{th} Bernoulli number B_n, as a rational
BernoulliApproximation(n)	Approximation to B_n, as a finite-precision free real
DawsonIntegral(r)	Dawson's integral at r
ErrorFunction(r), **Erf**(r)	Error function erf(r)
Erfc(r)	Complementary error function $1 - \text{erf}(r)$
ExponentialIntegral(r)	Exponential integral ei(r)
LogIntegral(r)	Logarithmic integral li(r)
ZetaFunction(c)	Riemann ζ function $\zeta(c)$, for c in free field
ZetaFunction(R, n)	Riemann ζ function $\zeta(n)$ for integer $n \neq 1$, evaluated in fixed precision real field R

As an example, consider the beta function $B(r, s)$. MAGMA does not have a function for $B(r, s)$, but it is related to the gamma function by

$$B(r, s) = \frac{\Gamma(r)\Gamma(s)}{\Gamma(r + s)}$$

so a user-defined function *BetaFunction*(r, s) may be defined as follows:

```
> BetaFunction := func< r, s |
>     Exp(LG(r) + LG(s) - LG(r+s)) where LG is LogGamma>;
```

The function could be defined using **Gamma** rather than **LogGamma**, but **LogGamma** is preferable since **Gamma** is a rapidly increasing function.

26.8.6 Example: The Prime Number Theorem

The prime number theorem states that $\frac{\pi(n)}{\mathrm{li}(n)} \to 1$ as $n \to \infty$, where $\pi(n)$ is the number of primes less than or equal to n and $\mathrm{li}(n)$ is the logarithmic integral of n. Riemann extended the prime number theorem by defining a function $R(n) = \sum_{k=1}^{\infty} \frac{\mu(k)}{k} \mathrm{li}(\sqrt[k]{n})$, where $\mu(n)$ is the Möbius μ function, which also tends asymptotically to $\pi(n)$ but at a faster rate. (The first term of the sum is the simple approximation $\mathrm{li}(n)$.)

The following procedure prints statistics of the prime number theorem up to a given limit. For n =10, 100, 1000, ... such that $n \leq$ *limit*, it prints

$$n, \pi(n), \mathrm{li}(n), \frac{\pi(n)}{\mathrm{li}(n)}, R(n), \frac{\pi(n)}{R(n)}.$$

In their printed version only, $\mathrm{li}(n)$ and $R(n)$ are rounded to the nearest integer. The identifiers for this procedure are as follows: `final_prime` stores the greatest prime $\leq n$; `true_c` stores the number of primes $\leq n$; `li_c` stores the $\mathrm{li}(n)$ approximation to `true_c`; and `rie_c` stores the Riemann approximation $R(n)$ (summing the first 15 terms only) to `true_c`.

To perform this example online, type `load "I96c26e1";`

```
PrimeNumberTheorem := procedure(limit)
  n := 10;
  final_prime := 7;
  true_c := 4;
  // fields of lower precision than the default,
  // used for convenience of printing
  R8 := RealField(8);
  R4 := RealField(4);

  // the heading for the output
  print "n\tpi(n)\tli(n)\tpi(n)/li(n)\tR(n)\tpi(n)/R(n)";
  print "-"^58;

  while n le limit do
    p := NextPrime(final_prime);
    while p le n do
      true_c +:= 1;
      final_prime := p;
      p := NextPrime(final_prime);
    end while;
    li_c := LogIntegral(R8!n);
```

```
    true_li := true_c / li_c;
    rie_c :=
      &+[LogIntegral(Root(R8!n, k)) * MoebiusMu(k)/k :
        k in [1..15]];
    true_rie := true_c / rie_c;
    printf "%o\t%o\t%o\t%o\t\t%o\t%o\n",
      n, true_c,
      Round(li_c), (true_li lt 1 select R4 else R8)!true_li,
      Round(rie_c), (true_rie lt 1 select R4 else R8)!true_rie;
    n *:= 10;
  end while;
end procedure;
```

A suitable call to this procedure would be:

```
> PrimeNumberTheorem(1000000);
n       pi(n)   li(n)   pi(n)/li(n)     R(n)    pi(n)/R(n)
-----------------------------------------------------------
10      4       6       0.6488          4       0.8911
100     25      30      0.8299          26      0.9752
1000    168     178     0.9459          168     0.9978
10000   1229    1246    0.9863          1227    1.0017
100000  9592    9630    0.9961          9587    1.0005
1000000 78498   78626   0.9984          78525   0.9997
```

Part VI

Modules and Algebras

27. Vector Spaces and Matrix Spaces

The most general kind of module which can be constructed in MAGMA is a module of dimension n over a ring R. MAGMA offers two ways of viewing such structures: as *(R-)tuple spaces* with an embedded basis, or as *(R-)modules with a reduced basis*. This chapter discusses the former view only.

If the ring over which a tuple space is defined is a field K, then the structure which emerges is a K-space, more commonly known as a *(finitely-generated) vector space*. Its elements are *row vectors*. The category of such structures in MAGMA is **ModTupFld**. (The field K must be defined as a member of a **Fld** category, so that MAGMA knows it is a field.) For the sake of clarity and familiarity, this chapter concentrates on vector spaces, although most of the operations are also available for R-spaces, that is, tuple spaces over a general ring R (category **ModTupRng**).

A further topic of this chapter is matrix spaces. Again, the emphasis will be on K-matrix spaces (**ModMatFld**), but the operations for R-matrix spaces in general (**ModMatRng**) closely resemble them. The elements of matrix spaces are represented as rectangular matrices, and may be considered either as vector-like objects printed as matrices, or as homomorphisms from one vector space to another.

27.1 Constructing the Full Vector Space

Every vector space of finite dimension n over a field K is isomorphic to the vector space $K^{(n)}$, that is, the space of n–dimensional coordinates whose entries are elements of K. This space is known as the *full vector space*. It may be created in MAGMA using the function **VectorSpace**(K, n), or **KSpace**(K, n). For example, the full 5–dimensional vector space over the rational field Q may be constructed as follows:

```
> Q := RationalField();
> V5 := VectorSpace(Q, 5);
```

It is not necessary to assign the field to an identifier before creating the vector space. Thus the following line will create the 3–dimensional vector space *V3F11* over the finite field with eleven elements:

```
> V3F11 := VectorSpace(GF(11), 3);
```

The function **Basis**(V) returns the basis of a vector space V. If V is a full vector space, then it returns the *standard basis*, in which the i^{th} basis vector is the vector that has the identity of the field K as its i^{th} entry, and the zero of K in all other coordinate positions. For example:

```
> print V3F11;
Full Vector space of degree 3 over GF(11)
> print Basis(V3F11);
[
    ( 1  0  0),
    ( 0  1  0),
    ( 0  0  1)
]
```

For full vector spaces created using **VectorSpace**(K), the generators are the same as the basis vectors, and the i^{th} generator, $V.i$, is the i^{th} basis vector:

```
> print Generators(V3F11);
{
    ( 0  1  0),
    ( 1  0  0),
    ( 0  0  1)
}
> print V3F11.2;
( 0  1  0)
```

Later in this chapter are examples of vector spaces where the generators are not the basis vectors or where the basis is not the standard basis.

27.2 Constructing the Full Matrix Space

Computations with $m \times n$ matrices in MAGMA may occur in several ways. Firstly, such matrices are elements of a matrix space, most obviously the *full matrix space* containing all $m \times n$ matrices over the field K. The function **KMatrixSpace**(K, m, n) returns this space. For instance, the following line assigns the matrix space of 2×3 matrices over the rationals to the identifier *M23*:

```
> M23 := KMatrixSpace(Q, 2, 3);
```

The standard basis of a full matrix space consists of all the 2×3 matrices with a single 1 entry and zeros elsewhere. Over the basis sequence, this 1 moves along each position within each row, starting with all of the first row, followed by the second row, and so on. This order is called *row-major order*. For example:

```
> print Basis(M23);
[
    [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]
]
```

Matrix spaces containing $m \times n$ matrices can also be regarded by MAGMA as vector spaces $K^{(m \times n)}$. Therefore all the vector operations can be used for matrices too. These operations will be explained in this chapter.

Thirdly, MAGMA can view $m \times n$ matrices as *linear transformations* or *homomorphisms* between two vector spaces $K^{(m)}$ and $K^{(n)}$. Therefore the matrix space $K^{(m \times n)}$ can also be created by means of the homomorphism module function $\mathbf{Hom}(M, N)$. Continuing our example, *M23* may be compared with the homomorphism module consisting of the homomorphisms from the vector space $Q^{(2)}$ to the vector space $Q^{(3)}$:

```
> V2 := VectorSpace(Q, 2);
> V3 := VectorSpace(Q, 3);
> H23 := Hom(V2, V3);
> print H23 eq M23;
true
```

27.3 Tuple Spaces and Matrix Spaces over a Ring

If R is a ring and n is a positive integer, then the function **RSpace**(R, n) creates the full tuple space $R^{(n)}$ of dimension n over R. This space is like a vector space, except that the coefficients do not come from a field. Similarly, the function **RMatrixSpace**(R, m, n) creates the full matrix space containing all $m \times n$ matrices over R. The categories of these magmas are **ModTupRng** and **ModMatRng** respectively. For example:

```
> Rsp := RSpace(IntegerRing(), 4);
> print Rsp;
Full RSpace of degree 4 over Integer Ring

> Msp := RMatrixSpace(ResidueClassRing(6), 2, 3);
> print Msp;
Full RMatrixSpace of 2 by 3 matrices
over Residue class ring of integers modulo 6
```

If one of these functions is called on a ring R which is in a **Fld** *category, then* MAGMA *will automatically create the resulting object as a vector space or K-matrix space.*

Tuple spaces and matrix spaces over a general ring R will not be discussed further in this chapter. However, many of the operations available for them are the same as those for spaces over a field K. One key difference is that intrinsics involving the word 'Field' for the K case use the word 'Ring' instead for the R case. An example is **CoefficientRing** instead of **CoefficientField**:

```
> print CoefficientRing(Rsp);
Integer Ring
```

27.4 Vectors and Matrices

27.4.1 Creating Vectors and Matrices

A vector may be created in MAGMA by coercing the sequence of its components into its parent vector space, using a `!` symbol. For instance, the vector $(-2\ \frac{3}{4}\ 12\ 0\ -\frac{8}{5})$ in *V5* may be assigned to the identifier v as follows:

```
> v := V5![-2, 3/4, 12, 0, -8/5];
```

Vectors whose components follow some mathematical pattern can be expressed easily by exploiting the sequence constructors:

```
> w := V5![3/4*(-1)^i: i in [1..5]];
> print w;
(-3/4  3/4 -3/4  3/4 -3/4)
```

The zero vector of a vector space V is $V!0$. The **IsZero**(v) function returns **true** when its argument is the zero vector.

Matrices are constructed in the same way, by regarding the matrix space as a vector space of dimension mn. The entries must be listed in row-major order, the same order as for the standard basis. (This is the same method for matrix creation as is used in matrix rings and matrix groups.) For instance, the following lines construct two 3×7 matrices with real coefficients:

```
> R20 := RealField(20);
> M37 := KMatrixSpace(R20, 3, 7);
> m1:=M37![0,0,0,4.5,1,1,-6.7,0,0,0,0,0,0,0,1,1,0,0,0,1,0];
> print m1;
[   0    0    0  4.5    1    1 -6.7]
[   0    0    0    0    0    0    0]
[   1    1    0    0    0    1    0]
> m2 := M37![ (21 - t) / 5 : t in [0..20] ];
> print m2;
[4.2   4 3.8 3.6 3.4 3.2   3]
[2.8 2.6 2.4 2.2   2 1.8 1.6]
[1.4 1.2   1 0.8 0.6 0.4 0.2]
```

Given a vector or matrix u, the function **Eltseq**(u) returns the sequence of its components:

```
> print Eltseq(w);
[ -3/4, 3/4, -3/4, 3/4, -3/4 ]
> print Eltseq(m2);
[ 4.2, 4, 3.8, 3.6, 3.4, 3.2, 3, 2.8, 2.6, 2.4, 2.2, 2, 1.8,
1.6, 1.4, 1.2, 1, 0.8, 0.6, 0.4, 0.2 ]
```

If U is a vector space or matrix space defined over a finite field, then **Random**(U) returns a random element of U. (U has to be finite because otherwise MAGMA has no way of constructing random elements.) For example:

```
> for n in [1..4] do
for> print Random(V3F11);
for> end for;
( 9 10  2)
( 5  4  2)
```

```
( 8  0  6)
( 6  8  6)
```

27.4.2 Arithmetic and Functions

The operators and functions in this subsection apply to both matrices and vectors. If the operations are performed on matrices, MAGMA considers the matrices to be vectors, by treating them in row-major order, but the output for matrices remains in rectangular form.

Arithmetic on MAGMA vectors and matrices is performed with the usual operators. Addition and subtraction are performed component-by-component, so the vectors or matrices in the operation must be of the same size. Scalar multiplication is also available. For instance, the vector $y = v - \frac{2}{5}w$ can be assigned as follows:

```
> y := v - 2/5*w; print y;
(-17/10   9/20 123/10  -3/10 -13/10)
```

Normalization is a particular type of automatic scalar multiplication. Given a vector or matrix u, the function **Normalize**(u) returns u divided by its first non-zero component, so that the first non-zero entry of the result is the unit of $K5$. For instance:

```
> print Normalize(y);
(      1   -9/34 -123/17    3/17   13/17)
```

The normalized form of a zero vector is the same as the vector itself.

Given u_1 and u_2 in the same space, **InnerProduct**(u, v) returns their inner product according to whatever inner product is defined on the space (usually the Euclidean norm). For example, the following line calculates the inner product of vectors v and w in $V5$ with respect to the Euclidean norm, so the output is the sum of the products of corresponding components of v and w:

```
> print InnerProduct(v, w);
-459/80
```

27.4.3 Indexing Vectors and Matrices

The usual $v[i]$ notation may be used to inspect or modify the i^{th} component of a vector v. For instance:

```
> print y[2];
9/20
> y[2] := 1000;
> print y;
(-17/10   1000 123/10  -3/10 -13/10)
```

The indexing of matrices is slightly different. To obtain an individual entry of a matrix, two coordinates are required. Given a matrix m, the entry in the i^{th} row and j^{th} column is $m[i, j]$:

```
> print m2[2, 6];
1.8
```

By contrast, $m[i]$ means the vector which is the i^{th} row of m:

```
> print m2[3];
[1.4 1.2   1 0.8 0.6 0.4 0.2]
> print IsZero(m1[2]);
true
```

There is no way of obtaining a column of m directly, since MAGMA's vectors are row vectors. The best method is to transpose m, using the function **Transpose**(m), and then extract a row from the transposed form. The next example shows how to obtain the fourth column of m as a row vector:

```
> m2tr := Transpose(m2);
> print m2tr[4];
(3.6 2.2 0.8)
```

The function **Support**(u) returns the set containing the indexes of all the non-zero entries of a vector or matrix u. If u is a matrix, then the indexes are given as tuple pairs `< `i, j` >`:

```
> print Support(v);
{ 1, 2, 3, 5 }
> print Support(m1);
{ <1, 6>, <3, 1>, <1, 5>, <1, 4>, <3, 6>, <1, 7>, <3, 2> }
```

27.4.4 Blocks within Matrices

Given an element A of a matrix space over K, any $p \times q$ block within it may be considered as a matrix (in particular, an element of the space of $p \times q$ matrices over K). Table 27.1 lists the operations connected with blocks within matrices, and some examples are given below:

Table 27.1. Blocks within matrices

MAGMA	Meaning
Submatrix(A, i, j, p, q)	$p \times q$ submatrix of A, starting at row i and column j
InsertBlock$(\tilde{}A, R, i, j)$	Given $p \times q$ matrix R, changes matrix A by replacing the $p \times q$ submatrix starting at row i and column j with R
HorizontalJoin(A, B)	Matrix formed by joining matrices A and B horizontally, where A and B have the same number of rows
HorizontalJoin(Q)	Matrix formed by joining sequence Q of matrices horizontally, where the matrices all have the same number of rows
VerticalJoin(A, B)	Matrix formed by joining matrices A and B vertically,
where A and B have the same number of columns	
VerticalJoin(Q)	Matrix formed by joining sequence Q of matrices vertically,
where the matrices all have the same number of columns	
DiagonalJoin(A, B)	Matrix formed by joining matrices A and B along main diagonal, with zeros elsewhere
DiagonalJoin(Q)	Matrix formed by joining sequence Q of matrices along main diagonal, with zeros elsewhere

```
> ms := Submatrix(m2, 2, 3, 2, 4); print ms;
[2.4 2.2   2 1.8]
[  1 0.8 0.6 0.4]
> InsertBlock(~m1, ms, 2, 1);
> print m1;
[   0    0    0  4.5    1    1 -6.7]
[ 2.4  2.2    2  1.8    0    0    0]
[   1  0.8  0.6  0.4    0    1    0]
> print VerticalJoin(m1, m2);
[   0    0    0  4.5    1    1 -6.7]
[ 2.4  2.2    2  1.8    0    0    0]
[   1  0.8  0.6  0.4    0    1    0]
[ 4.2    4  3.8  3.6  3.4  3.2    3]
[ 2.8  2.6  2.4  2.2    2  1.8  1.6]
[ 1.4  1.2    1  0.8  0.6  0.4  0.2]
> print DiagonalJoin(ms, m2);
[2.4 2.2   2 1.8   0   0   0   0   0   0   0]
[  1 0.8 0.6 0.4   0   0   0   0   0   0   0]
[  0   0   0   0 4.2   4 3.8 3.6 3.4 3.2   3]
[  0   0   0   0 2.8 2.6 2.4 2.2   2 1.8 1.6]
```

```
[  0   0   0   0 1.4 1.2   1 0.8 0.6 0.4 0.2]
```

27.5 Subspaces and Quotient Spaces

Vector subspaces and quotient spaces may be formed in MAGMA using the customary **sub** and **quo** constructors. For instance, the following statement assigns to *V5s* the subspace of *V5* generated by w, the third generator of *V5* and the vector $(-5\ 5\ 7\ 5\ -5)$:

```
> V5s := sub< V5 | w, V5.3, [-5, 5, 7, 5, -5] >;
```

The information listed to the right of the | symbol specifies the generators for the subspace. In this example they are given explicitly as vectors. Other possible objects on the right of the | symbol are subspaces of the main space, and sets or sequences containing these; in this case, the vectors used to generate the subspace will be the generating elements of these subspaces.

Not all vector spaces are represented internally using the generators provided by the user. For the sake of efficiency, MAGMA employs an echelonized basis for this purpose, and the vectors of this basis may differ from those vectors given by the user. However, the generators are all stored (even if they are not linearly independent), and are available in the user-specified order under the names *V5s*.1, *V5s*.2 and *V5s*.3. For example:

```
> print V5s;
Vector space of degree 5, dimension 2 over Rational Field
Generators:
(-3/4  3/4 -3/4  3/4 -3/4)
(   0    0    1    0    0)
(  -5    5    7    5   -5)
Echelonized basis:
( 1 -1  0 -1  1)
( 0  0  1  0  0)
> print 10*V5s.2 + V5s.3;
(-5  5 17  5 -5)
```

The quotient constructor **quo** creates the quotient of the main space by the subspace described by whatever is on the right of the | symbol. For instance, the following lines demonstrate two ways to create the quotient *V5q* of *V5* by *V5s*:

```
> V5q := quo< V5 | V5s >;
> // or
```

```
> V5q := quo< V5 | w, V5.3, [-5, 5, 7, 5, -5] >;
> print V5q;
Full Vector space of degree 3 over Rational Field
```

The **sub** and **quo** constructors can return mappings as their second return value. These mappings are the inclusion homomorphism and the natural homomorphism, respectively. For instance, we can construct the natural homomorphism *phi* which maps *V5* to the quotient space *V5q*. In accordance with vector space theory, the complement C of *V5s* is isomorphic to *V5q*, since C and *V5q* have the same dimension:

```
> V5q, phi := quo< V5 | w, V5.3, [-5, 5, 7, 5, -5] >;
> print phi;
Mapping from: Full Vector space of degree 5 over Rational
Field to Full Vector space of degree 3 over Rational Field
> C := Complement(V5, V5s); print C;
Vector space of degree 5, dimension 3 over Rational Field
Echelonized basis:
(0 1 0 0 0)
(0 0 0 1 0)
(0 0 0 0 1)
> print C@phi;
Full Vector space of degree 3 over Rational Field
> print C@phi eq V5q;
true
```

The following line verifies that the kernel of *phi* is *V5s*:

```
> print Kernel(phi) eq V5s;
true
```

If the user has constructed a subspace or quotient space U of a vector space V, without obtaining the corresponding mapping, the mapping can be obtained at some later stage from the function **Morphism**(U, V). It returns the homomorphism as a matrix.

27.6 Operations on Vector Spaces

Table 27.2 lists several of the Magma operations on vector spaces. Note carefully the difference between generators and basis vectors.

Table 27.2. Vector space operations

MAGMA	Meaning
CoefficientField(V), **BaseField**(V)	Field K of K-vector space V
Dimension(V)	Dimension of V
Generic(V)	Full vector space $K^{(n)}$ of which V is a subspace
OverDimension(V), **Degree**(V)	Given subspace V of $K^{(n)}$, returns n
OverDimension(v), **Degree**(V)	Given element v of subspace of $K^{(n)}$, returns n
Generators(V)	Set of the generators of V, as given to MAGMA (e.g., in **sub**)
Ngens(V)	Number of generators of V
$V.i$	i^{th} generator of V
Basis(V)	Sequence of basis vectors of V, as calculated by MAGMA
BasisMatrix(V)	Matrix whose rows are the basis vectors of V
BasisElement(V, i)	i^{th} basis vector of V
$U_1 + U_2$	Sum of subspaces U_1 and U_2
U_1 **meet** U_2	Intersection of subspaces U_1 and U_2
Complement(V, U)	Complement for subspace U of V, as a subspace of V
Transversal(V, U)	Transversal in V for subspace U of V, as a set of vectors; V must be over a finite field

27.7 Operations with Linear Transformations

Given any two K-vector spaces V and W of over-dimension (degree) m and n, the set of linear transformations from V to W is the homomorphism module **Hom**(V, W). Every linear transformation $V \to W$ can be written as a matrix a of size $m \times n$ with entries in K. The effect of a on V may be investigated using mapping operations.

For example, let the field K be the quadratic field $Q(\sqrt{7})$, and let V and W be certain subspaces of the vector spaces $K^{(4)}$ and $K^{(6)}$:

```
> K<rt7> := QuadraticField(7); print K;
Quadratic Field Q(rt7)
> V := sub< VectorSpace(K, 4) |
>   [rt7, 1, 0, 0], [0, 3 - 2*rt7, 0, 1] >;
> print V;
Vector space of degree 4, dimension 2
```

```
over Quadratic Field Q(rt7)
Generators:
(      rt7         1         0         0)
(        0 3 - 2*rt7         0         1)
Echelonized basis:
(1 0 0 1/133*(14 + 3*rt7))
(0 1 0 1/19*(-3 - 2*rt7))
> W := sub< VectorSpace(K, 6) |
>   [2,0,0,1,0,0], [0,0,0,0,-1,1], [0,1,0,0,0,0] >;
> print W;
Vector space of degree 6, dimension 3
over Quadratic Field Q(rt7)
Generators:
( 2  0  0  1  0  0)
( 0  0  0  0 -1  1)
( 0  1  0  0  0  0)
Echelonized basis:
(  1   0   0 1/2   0   0)
(  0   1   0   0   0   0)
(  0   0   0   0   1  -1)
>
> H := Hom(V, W);
> print H;
KMatrixSpace of 4 by 6 GHom matrices and dimension 18 over
Quadratic Field Q(rt7)
```

Let a equal the sum of the second generator of H and twice the thirteenth generator of H. It has domain V and codomain W: be the us construct the following matrix a:

```
> a := H.2 + 2*H.13;
> print a;
[0 1 0 0 0 0]
[0 0 0 0 0 0]
[0 0 0 2 0 0]
[0 0 0 0 0 0]
> print Domain(a) eq V and Codomain(a) eq W;
true
```

The product of an element of V and a is an element of W. For instance:

```
> v := V![ -8*rt7, -2 - 4*rt7, 0, 2];
> w := v * a; print w;
(     0 -8*rt7      0      0      0      0)
> print Coordinates(W, w);
```

```
[ 0, -8*rt7, 0 ]
```

Observe that w is $-8\sqrt{7}$ times the second basis vector of W.

It is also possible to multiply a subspace of V by a, so as to obtain a subspace of W.

The image of a is returned by the function **Image**(a). It is equal to the product of the whole domain by a:

```
> I := Image(a); print I;
Vector space of degree 6, dimension 1
over Quadratic Field Q(rt7)
Echelonized basis:
(0 1 0 0 0 0)
> print I eq V*a;
true
```

The image of a should be distinguished from the row space of a. The function **RowSpace**(a) returns the space generated by the rows of a, disregarding the domain of a. Therefore the row space may be of higher dimension than the image, as it is in the following example:

```
> print RowSpace(a);
Vector space of degree 6, dimension 2
over Quadratic Field Q(rt7)
Echelonized basis:
(0 1 0 0 0 0)
(0 0 0 1 0 0)
```

The dimension of the row space indicates that the rank of a is 2, but this may be verified using the function **Rank**:

```
> print Rank(a);
2
```

It is possible to find the echelon form of a using **EchelonForm**(a). This function returns two values: the echelon form e and a matrix b (a product of elementary matrices) such that $ba = e$:

```
> e, b := EchelonForm(a); print e, b;
[0 1 0 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
```

```
[  1   0   0   0]
[  0   0 1/2   0]
[  0   0   0   1]
[  0   1   0   0]
> print b*a eq e;
true
```

Also available are **Kernel**(a) (or **NullSpace**(a)) and **Cokernel**(a). The kernel is the subspace of the domain which is mapped to zero, and the cokernel is the subspace of the codomain which is not in the image. For instance:

```
> Ker := Kernel(a); print Ker;
Vector space of degree 4, dimension 1
over Quadratic Field Q(rt7)
Echelonized basis:
(0 1 0 1/19*(-3 - 2*rt7))
> print Ker*a;
Vector space of degree 6, dimension 0
over Quadratic Field Q(rt7)
> print Cokernel(a);
Vector space of degree 6, dimension 2
over Quadratic Field Q(rt7)
Echelonized basis:
(  1   0   0 1/2   0   0)
(  0   0   0   0   1  -1)
```

Thus the kernel is generated by the second basis element of V, the one which maps to zero. The cokernel is generated by the first and third basis elements of W, whereas the second basis element of W generates the image of a. Notice that the dimensions of the spaces sum correctly:

```
> Dim := Dimension;
> print Dim(Ker)+Dim(Image(a)) eq Dim(Domain(a));
true
> print Dim(Cokernel(a))+Dim(Image(a)) eq Dim(Codomain(a));
true
```

The function **Transpose**(a) returns the matrix whose columns are the rows of a.

```
> aTr := Transpose(a); print aTr;
[0 0 0 0]
[1 0 0 0]
[0 0 0 0]
[0 0 2 0]
```

```
[0 0 0 0]
[0 0 0 0]
```

Multiplying matrices from different homomorphism modules is another way of creating elements of new homomorphism modules. The process corresponds to the composition of mappings. If U is a subspace of some other K-vector space and b is an element of $\mathbf{Hom}(W, U)$, then the matrix product $a*b$ represents the mapping in $\mathbf{Hom}(V, U)$ which is the composition of a and b.

27.8 Row and Column Operations

Table 27.3. Row and column operations

MAGMA	Meaning
AddRow($\tilde{}a, u, i, j$)	Adds u times row i to row j in matrix a
AddColumn($\tilde{}a, u, i, j$)	Adds u times column i to row j in matrix a
MultiplyRow($\tilde{}a, u, i$)	Multiplies row i of matrix a by u
MultiplyColumn($\tilde{}a, u, i$)	Multiplies column i of matrix a by u
SwapRows($\tilde{}a, i, j$)	Interchanges rows i and j of matrix a
SwapColumns($\tilde{}a, i, j$)	Interchanges columns i and j of matrix a

The three elementary row and column operations on matrices are available as procedures in MAGMA. Table 27.3 shows them. Each procedure destructively modifies the given matrix, so a ~ must precede its identifier name in the procedure call. In the following example, a copy *a1* of a is made, and then 17 times row 1 of *a1* is added to row 4, and columns 2 and 5 are swapped:

```
> a1 := a; print a1;
[0 1 0 0 0 0]
[0 0 0 0 0 0]
[0 0 0 2 0 0]
[0 0 0 0 0 0]
> AddRow(~a1, 17, 1, 4); print a1;
[ 0  1  0  0  0  0]
[ 0  0  0  0  0  0]
[ 0  0  0  2  0  0]
[ 0 17  0  0  0  0]
> SwapColumns(~a1, 2, 5); print a1;
[ 0  0  0  0  1  0]
```

```
[ 0  0  0  0  0  0]
[ 0  0  0  2  0  0]
[ 0  0  0  0 17  0]
```

27.9 Simultaneous Systems of Linear Equations

MAGMA provides an easy way of solving a simultaneous system of linear equations that is expressed as a matrix equation $VX = W$, where X is a matrix, W is a vector with the same number of entries as X has columns, and V is an unknown vector with the same number of entries as X has rows. The function **Solution**(X, W) returns a particular solution v, and also returns the kernel K of X, regarding X as a homomorphism from the vector space of V to the vector space of W. The kernel is the space of vectors which when multiplied by X equal the zero vector, so the general solution to $VX = W$ is $V = v + k$ where k is any vector in K. For example, suppose the user wants to solve the system

$$\begin{array}{rcrcrcrcl} a & + & 2b & + & 2c & - & 3d & = & 5 \\ 2a & - & b & + & c & + & 2d & = & 1 \\ 3a & + & b & + & 3c & - & d & = & 6 \end{array}$$

which is equivalent, once the vectors are written as row vectors, to

$$\begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 2 & -1 & 1 \\ 2 & 1 & 3 \\ -3 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 5 & 1 & 6 \end{pmatrix}$$

In MAGMA this system may be solved as follows:

To perform this example online, type `load "I96c27e1";`

```
> Q := RationalField();
> V4 := VectorSpace(Q, 4);
> V3 := VectorSpace(Q, 3);
> H43 := Hom(V4, V3);
> X := H43![1, 2, 3, 2, -1, 1, 2, 1, 3, -3, 2, -1];
> print X;
[ 1  2  3]
[ 2 -1  1]
[ 2  1  3]
```

```
[-3  2 -1]
> W := V3![5, 1, 6];
> v, K := Solution(X, W);
> print v;
(7/5 9/5   0   0)
> print K;
Vector space of degree 4, dimension 2 over Rational Field
Echelonized basis:
(   1    0 -8/7 -3/7)
(   0    1 -1/7  4/7)
```

The identifier v now contains a particular solution to the equation, and K contains the kernel of X:

```
> print v*X eq W and Kernel(X) eq K;
true
```

Therefore the general solution is $V = (\frac{7}{5}\ \frac{9}{5}\ 0\ 0) + t_1(1\ 0\ \frac{-8}{7}\ \frac{-3}{7}) + t_2(0\ 1\ \frac{-1}{7}\ \frac{4}{7})$.

If the user is not sure whether there is a solution to the equation $VX = W$, then **IsConsistent**(X, W) should be used instead. It returns **true** if a solution exists, and if so, then it also returns a particular solution and the kernel, just as **Solution** does:

```
> print IsConsistent(X, V3![5, 1, 7]);
false
> print IsConsistent(X, W);
true (7/5 9/5   0   0)

Vector space of degree 4, dimension 2 over Rational Field
Echelonized basis:
(   1    0 -8/7 -3/7)
(   0    1 -1/7  4/7)
```

27.10 Changing the Basis

27.10.1 Defining the Chosen Basis

The function **VectorSpaceWithBasis**(B), where B is a sequence of linearly independent vectors, allows the user to define a chosen basis B for a vector space. (An alternative version of this function takes as its argument not a sequence but a matrix whose rows are the basis vectors.) For example, the following lines show how to build a vector space Sw of degree 3 over the

rationals which has the basis $[[1,0,1],[2,1,1],[1,1,1]]$, as compared to the same vector space S with the standard basis:

To perform this example online, type `load "I96c27e2";`

```
> S := VectorSpace(RationalField(), 3);
> print Basis(S);
[
    (1 0 0),
    (0 1 0),
    (0 0 1)
]
> Sw:=VectorSpaceWithBasis([S | [1,0,1], [2,1,1], [1,1,1]]);
> print Basis(Sw);
[
    (1 0 1)
    (2 1 1)
    (1 1 1)
]
```

The vector spaces S and Sw are considered by MAGMA to be the same in most respects. Note especially that when a vector is being created it makes no difference whether the sequence of its entries is coerced into S or Sw:

```
> print S eq Sw;
true
> print S![2, 5, 72] eq Sw![2, 5, 72];
true
```

The only time that the basis of Sw is significant, apart from when using the functions that access the basis and generators, is for the function **Coordinates**(V, v). Given a vector v in a vector space V, it returns the coordinates of v relative to the current basis of V. For example:

```
> v := Sw![-2, -1, 2];
> print v;
(-2 -1  2)
> print Coordinates(Sw, v);
[ 3, -4, 3 ]
```

Observe that $v = 3 \cdot (1\ 0\ 1) - 4 \cdot (2\ 1\ 1) + 3 \cdot (1\ 1\ 1)$.

To simulate a vector space whose entries are radically connected with a non-standard basis, the user should create a matrix which maps the vectors of the standard basis to those of the required basis:

```
> M := EndomorphismAlgebra(S); print M;
Full Matrix Algebra of degree 3 over Rational Field
> m := M![1,0,1, 2,1,1, 1,1,1];
> /* the rows are the non-standard basis vectors */
> mInv := m^-1; print mInv;
[ 0  1 -1]
[-1  0  1]
[ 1 -1  1]
```

Now, given a vector, it may be multiplied on the right by *mInv* to make it into as a vector expressed relative to the other basis:

```
> print S![-2, -1, 2]*mInv;
( 3 -4  3)
```

The output is a vector of the simulated vector space.

27.10.2 Constructing a Basis Gradually

MAGMA provides some functions to help the user construct linearly independent vectors. The simplest of these is **IsIndependent**(S), which returns **true** if the elements of the set or sequence S are linearly independent. For example:

```
> vecseq := [V5 | [4/3, 8, -2, 0, 1/2], [5, 7, 10, -2, 0]];
> print IsIndependent(vecseq);
true
```

The function **ExtendBasis**(Q, V) constructs a basis for a vector space V such that the linearly independent sequence of vectors Q is the first part of the resulting basis sequence:

```
> print ExtendBasis(vecseq, V5);
[
    (4/3   8  -2   0 1/2),
    ( 5  7 10 -2  0),
    (0 0 1 0 0),
    (0 0 0 1 0),
    (0 0 0 0 1)
]
```

The space V need not be a full vector space.

Another form of this function is **ExtendBasis**(U, V), which takes a subspace U of V as its first argument. The basis sequence that it constructs begins with the basis of U.

27.11 Changing the Coefficient Field

The functions **ExtendField**(V, L) and **RestrictField**(V, L) change the field K of a vector space V to the field L, where K and L are both finite fields. Both functions return two values, namely the L-vector space U and a homomorphism $V \to U$.

For **ExtendField**(V, L), L must be an extension of K. The vector space returned is $V \otimes_K L$ and the mapping returned is the inclusion homomorphism.

For **RestrictField**(V, L), L must be a subfield of K, where the generating vectors for K lie over L. U is obtained from V by restricting the scalars to L, and the mapping returned is the restriction homomorphism.

A more general function for this task is **ChangeRing**(V, R). It returns the vector space V' over R obtained by coercing the components of elements of V into V', together with the homomorphism from V to V'. If the user wishes to use a special homomorphism f from the old coefficient ring to R, then use the function should be used in the form **ChangeRing**(V, R, f).

28. Matrix Algebras

This chapter discusses matrix algebras (matrix rings) in MAGMA. In general, it does not list the operations that also apply to other categories of rings. See Chapter 19 for these.

28.1 The Generic Matrix Algebra and Its Elements

28.1.1 Constructing the Generic Matrix Algebra

The set of all $n \times n$ matrices with entries in a ring R forms an R-algebra $M_n(R)$ which can be constructed in MAGMA as **MatrixAlgebra**(R, n). Since $M_n(R)$ may also be regarded as a (non-commutative) ring, another name for this function is **MatrixRing**(R, n). The category of M is **AlgMat**.

For example, the algebra M below is the *generic* or complete matrix algebra of degree 3 over F, the finite field of five elements:

```
> F := FiniteField(5);
> M := MatrixAlgebra(F, 3);
> print M;
Full Matrix Algebra of degree 3 over GF(5)
> print #M;
1953125
```

This algebra has $5^{3^2} = 1953125$ elements.

28.1.2 Constructing Matrices

The method for defining an element of a matrix algebra is to create a sequence of its entries, listed in row-major order, and coerce it into the algebra. (This is the same method for matrix creation as is used in matrix spaces and matrix groups.) Continuing the example:

```
> m1 := M![2, 3, 0, 1, 1, 0, 0, 3, 4]; print m1;
[2 3 0]
[1 1 0]
[0 3 4]
```

Notice that for this coefficient ring, MAGMA is able to coerce the integer elements of the sequence into elements of F, so that the universe of the given sequence need not be F.

Matrices which have the same entry t down the main diagonal and zeros everywhere else are called *scalar* matrices, because they behave very much like the element t in the coefficient ring R. For this reason, such matrices may be created by coercing t into the matrix algebra:

```
> m2 := M!4; print m2;
[4 0 0]
[0 4 0]
[0 0 4]
```

An alternative for $M!t$ is **ScalarMatrix**(M, t).

Being a scalar matrix, the zero matrix may be constructed as $M!0$. **Zero**(M) also returns this matrix.

Another special kind of matrix is a diagonal matrix. Its entries are all zeros except for those down the main diagonal (from top left to bottom right). The function for creating a diagonal matrix is **DiagonalMatrix**(M, Q). Given a matrix algebra M over R of degree n, and a sequence Q of n elements of R, it returns the matrix in M with elements of Q down the main diagonal and zeros elsewhere. For instance:

```
> print DiagonalMatrix(M, [4, 2, 3]);
[4 0 0]
[0 2 0]
[0 0 3]
```

MAGMA also has a function **MatrixUnit**(M, i, j) for constructing a matrix unit, that is, a matrix in M which has the one of the coefficient ring in row i, column j and zeros everywhere else:

```
> print MatrixUnit(M, 3, 2);
[0 0 0]
[0 0 0]
[0 1 0]
```

Scalar matrices, diagonal matrices and matrix units may all be constructed by coercing a sequence of length n^2 into the matrix algebra, but the functions above make the task easier in these frequently-occurring cases.

The function **ElementToSequence**(m) or **Eltseq**(m) converts a matrix m into the sequence of its entries, in row-major order:

```
> print Eltseq(m1);
[ 2, 3, 0, 1, 1, 0, 0, 3, 4 ]
```

28.1.3 Arithmetic on Matrices

Arithmetic on elements of matrix algebras is performed with the usual ring operators for addition and multiplication. (Remember, however, that matrix multiplication is not commutative in general.) For example:

```
> print m1 * m2;
[3 2 0]
[4 4 0]
[0 2 1]
> print m1^-1;
[4 3 0]
[1 3 0]
[3 4 4]
```

Observe that the inverse of an invertible matrix is calculated by raising it to the power of -1. The function **IsUnit**(m) is provided for testing whether a matrix m has an inverse.

It was shown above that scalar matrices may be constructed by coercing the scalar into the algebra. This is an instance of forced coercion. However, automatic coercion of scalars into matrix algebras may also occur. When a scalar matrix is involved in an arithmetic operation with another matrix, the scalar matrix may be referred to by means of the scalar alone, without explicitly coercing the scalar into the matrix algebra:

```
> print m1 - 2;
[0 3 0]
[1 4 0]
[0 3 2]
```

It is possible to access and modify specific entries or rows of a matrix. The entry in row i, column j of matrix m is $m[i, j]$, and the i^{th} row is $m[i]$. For instance:

```
> r := m1[3]; print r;
(0 3 4)
> m2[3,2] := 2; print m2;
[4 0 0]
[0 4 0]
[0 2 4]
> m2[1] := r; print m2;
[0 3 4]
[0 4 0]
[0 2 4]
```

Multiplication involving matrices is possible not only when the two matrices are elements of (or coercible into) the same matrix algebra, but also whenever the coefficient rings are compatible and the number of columns of the first matrix equals the number of rows of the second matrix. For examples of this, using rectangular matrices belonging to matrix spaces, see Chapter 27. The operations on blocks within matrices listed in Table 27.1 are also available.

28.2 Constructing Other Matrix Algebras

28.2.1 Subalgebras

The matrix algebra discussed to this point has been the generic or full matrix algebra $M_n(R)$, which is constructed using the function **MatrixAlgebra**(R, n). There are two main ways to create subalgebras of this. One method is to build $M_n(R)$ first, then use a **sub** constructor. The other method is to use the **MatrixAlgebra** or **MatrixAlgebra** constructor

MatrixAlgebra< R, n | generator specification >

which builds the subalgebra of the generic algebra in one step. In both versions, the generators of the algebra must be given on the right side of the | symbol in the constructor. The result is the same in either case:

```
> Ms := MatrixAlgebra< F, 3 |
>   [2,3,0, 1,1,0, 0,3,4] >;
> print Ms;
Matrix Algebra of degree 3 with 1 generator over GF(5)
> print #Ms;
125
> print Ms eq sub< M | m1 >;
true
```

Now *Ms* is a matrix algebra with one generator, *m1*.

Unlike M, the subalgebra *Ms* does not contain all the 3×3 matrices over F, so it is not a generic algebra. The function **Generic**(S) returns the generic algebra corresponding to a matrix algebra S:

```
> print Generic(Ms) eq M;
true
```

Matrix algebras do not have to contain the multiplicative identity of their generic algebra. The generators are exactly those given by the user; the one of the generic algebra is not inserted automatically among the generators. For example:

```
> print M!1 in
>   sub< M | [0,0,0,0,2,0,0,2,0], [0,0,0,0,0,2,0,0,0] >;
false
```

28.2.2 Ideals

Since matrix algebras are generally non-commutative, there are three kinds of ideals: left ideal, right ideal and two-sided ideal. The constructors for these are respectively **lideal**, **rideal** and **ideal**. See Section **??**. A left ideal of a matrix algebra M is a subalgebra I of M with the property that for all $i \in I$ and $m \in M$, the product mi is in I. A right ideal satisfies the property $im \in I$, and a two-sided ideal (or simply 'ideal') satisfies both these properties. For example:

```
> LI := lideal< M | M.1 >;
> print LI;
Matrix Algebra [lideal of M] of degree 3 and dimension 3
over GF(5)
> m := Random(M);
> print forall{ i: i in LI | m*i in LI };
true
> print forall{ i: i in LI | i*m in LI };
false
```

The generic matrix algebra over a field has only two ideals, itself and the trivial algebra, so examples of two-sided ideals must come from non-generic algebras. Consider the set T consisting of all the 5×5 matrices with zeros below the diagonal over the finite field with 7 elements. This set, which forms a ring, is conveniently generated by all the matrix units in which the one is above or the diagonal. The subset B of T consisting of the matrices that also

have zeros on the diagonal may be shown to be an ideal of T. B is generated by those matrix units which are above the diagonal. In MAGMA, T and B may be constructed as follows:

```
> T := MatrixAlgebra< GF(7), 5 |
>     { MatrixUnit($, i, j) : i in [1..j], j in [1..5]} >;
> print T;
Matrix Algebra of degree 5 with 15 generators over GF(7)
> B := ideal< T |
>     { MatrixUnit($, i, j) : i in [1..j-1], j in [1..5]} >;
> print B;
Matrix Algebra [ideal of T] of degree 5 and dimension 10
over GF(7)
```

Table 28.1. Operations involving ideals

MAGMA	Meaning
JacobsonRadical(M)	Intersection of all maximal ideals of algebra M
MaximalIdeals(M)	Maximal (two-sided) ideals of M
MaximalLeftIdeals(M)	Maximal left ideals of M
MaximalRightIdeals(M)	Maximal right ideals of M
MinimalIdeals(M)	Minimal (two-sided) ideals of M
MinimalLeftIdeals(M)	Minimal left ideals of R
MinimalRightIdeals(M)	Minimal right ideals of R
IsSimple(M)	**true** if M has no non-trivial ideals
IsSemisimple(M)	**true** if M is a direct sum of simple ideals (i.e., has trivial Jacobson radical)

Table 28.1 (p. 532) lists functions relating to ideals of matrix algebras. The functions returning the maximal and minimal ideals (two-sided, left, and right) return them as a sequence of ideals, in non-decreasing size. Each of these functions has a parameter **Limit** (default ∞): if the user assigns n to **Limit**, then at most n ideals will be calculated. The second return value of the function will be **true** if all the ideals have been returned, else **false**. For instance:

```
> print MinimalIdeals(Ms);
[
    Matrix Algebra [ideal of Ms] of degree 3
and dimension 1 over GF(5),
    Matrix Algebra [ideal of Ms] of degree 3
and dimension 2 over GF(5)
```

```
]
true

> T := MatrixAlgebra< GF(7), 5 |
>     { MatrixUnit($, i, j) : i in [1..j], j in [1..5]} >;
> print MaximalIdeals(T);
[
    Matrix Algebra [ideal of T] of degree 5
and dimension 14 over GF(7),
    Matrix Algebra [ideal of T] of degree 5
and dimension 14 over GF(7),
    Matrix Algebra [ideal of T] of degree 5
and dimension 14 over GF(7),
    Matrix Algebra [ideal of T] of degree 5
and dimension 14 over GF(7),
    Matrix Algebra [ideal of T] of degree 5
and dimension 14 over GF(7)
]
true
> print MaximalIdeals(T : Limit := 2);
[
    Matrix Algebra [ideal of T] of degree 5
and dimension 14 over GF(7),
    Matrix Algebra [ideal of T] of degree 5
and dimension 14 over GF(7)
]
false
```

28.2.3 Quotients

The quotient of a matrix algebra by an ideal is performed in MAGMA using the **quo** constructor. It returns an algebra, and a homomorphism from the original algebra to the quotient. Continuing the example above:

```
> Q, f := quo< T | B >;
> print Q;
Algebra of dimension 5 with base ring GF(7)
> print f;
Mapping from: AlgMat: T to AlgCon: Q
```

The kernel of this homomorphism f is the ideal B used to construct the quotient:

```
> print Kernel(f) eq B;
```

```
true
```

Q is isomorphic to the image $f(D)$ of the subalgebra D of T consisting of all the diagonal matrices. The reason is that T may be seen as the disjoint union of the cosets $B + d$, where $d \in D$. This may be seen by calculating the matrix that gives the correspondence between them, using the function **Morphism**. The matrix returned by **Morphism** is square and has full rank, indicating an isomorphism:

```
> D := sub< T | { MatrixUnit($, i, i) : i in [1..5]} >;
> print D;
Matrix Algebra of degree 5 with 5 generators over GF(7)
> print f(D);
Algebra of dimension 5 with base ring GF(7)
> print Morphism(Q, f(D));
[1 0 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]
```

28.2.4 Other Matrix Algebra Constructions

Table 28.2. Other matrix algebra constructions

MAGMA	Meaning
DirectSum(M, N)	Direct sum of algebras M and N over same base ring, with action given by direct sum of their actions
TensorProduct(M, N)	Tensor product of M and N over same base ring
DirectSum(a, b)	Direct sum of matrices a and b, as an element of the direct sum of their parents
TensorProduct(a, b)	Tensor product of a (n_1 rows) and b (n_2 rows) as an element of $M_{n_1 n_2}(R)$
ExteriorSquare(a)	Given $a \in M_n(R)$, return exterior square of a as an element of $M_{n(n+1)}(R)$
SymmetricSquare(a)	Given $a \in M_n(R)$, return symmetric square of a as an element of $M_{n(n-1)}(R)$

Table 28.2 lists some constructions that are special to matrix algebras. For example:

```
> matrand := Random(Ms); print DirectSum(matrand, matrand);
```

```
[2 1 0 0 0 0]
[2 0 0 0 0 0]
[1 3 2 0 0 0]
[0 0 0 2 1 0]
[0 0 0 2 0 0]
[0 0 0 1 3 2]
```

28.3 General Facts About a Matrix Algebra

Table 28.3. General facts about a matrix algebra

MAGMA	Meaning
Degree(M)	Degree n of M, an algebra of $n \times n$ matrices
Dimension(M)	Dimension of M (base ring must be a field)
CoefficientRing(M), **BaseRing**(M)	Coefficient ring or base ring R of M
Generic(M)	Given M of degree n and coefficient ring R, return full matrix algebra $M_n(R)$ in which M is naturally embedded

Table 28.3 summarizes the functions giving facts about a matrix algebra, and Table 28.4 gives the functions for individual matrices.

For example, if A is a square matrix over $\mathbb{C}$, then A is *Hermitian* if it equals the sum of its complex conjugate and its transpose, and it is *anti-Hermitian* if the negative of A equals this sum. It can be shown that every square matrix over $\mathbb{C}$ may be written as the sum of a Hermitian matrix and an anti-Hermitian matrix. The following functions (ignoring issues of precision) may be used to represent a matrix in this way:

```
> HermitianPart := function(A)
>     H := Transpose(A);
>     n := Nrows(A);
>     for i in [1..n] do
>         for j in [1..n] do
>             H[i,j] := ComplexConjugate(H[i,j]);
>         end for;
>     end for;
>     return H;
> end function;
> // OR
```

Table 28.4. General facts about elements of matrix algebras

MAGMA	Meaning
NumberOfRows(a), **Nrows**(a)	Number of rows of a
NumberOfColumns(a), **Ncolumns**(a)	Number of columns of a
Rank(a)	Rank of a
Determinant(a)	Determinant of a
Trace(a)	Sum of diagonal entries of a
Transpose(a)	Transpose of a
Order(a)	Given invertible matrix a over a finite field, return the order of a
FactoredOrder(a)	Given invertible matrix a over a finite field, return the order of a, in a factorization sequence
IsDiagonal(a)	**true** if the only non-zero entries of a are on the diagonal
IsScalar(a)	**true** if a is a diagonal matrix, and all the entries on the diagonal are the same
IsSymmetric(a)	**true** if a equals its transpose
IsSimilar(a, b)	**true** if a and b are similar; if **true**, also returns matrix t such that $tat^{-1} = b$

```
> HermitianPart := func< A |
>   Parent(A) ! [ ComplexConjugate(A[j,i]) :
>                   i, j in [1..Nrows(A)] ] >;

> HermAnti := func< A | B, A - B
>   where B is 1/2*(A + HermitianPart(A)) >;

> C<i> := ComplexField();
> M4C := MatrixAlgebra(C, 4);
> s1 := [ Random(20): n in [1..16] ];
> s2 := [ Random(20): n in [1..16] ];
> A := M4C!s1 + i*M4C!s2;
> print A;
[18 + 11*i 16 + 12*i        20  11 + 4*i]
[      9*i 11 + 16*i  2 + 20*i 18 + 19*i]
[       18  20 + 3*i 20 + 10*i  6 + 13*i]
[  1 + 7*i  5 + 15*i  19 + 1*i   7 + 5*i]
> B, C := HermAnti(A);
> print A eq B+C;
true
> print B eq HermitianPart(B);
```

```
true
> print -C eq HermitianPart(C);
true
```

28.4 Linear Algebra Concepts

It has already been seen that it is possible to multiply a square matrix by a scalar, or to multiply two square or rectangular matrices of suitable sizes. Another kind of multiplication for matrix algebras is the product $v*m$ of a (row) vector v with a matrix m. Here v and m must have compatible coefficient rings, and v must have the same number of columns as m has rows. For instance, the vector r constructed on p. 529 is suitable for multiplication with a matrix of M:

```
> print r * m1;
(3 0 1)
```

The value returned belongs to the generic vector space V over the coefficient ring of M and with the same degree as M:

```
> V := Parent(r); print V;
Full Vector space of degree 3 over GF(5)
> print V eq VectorSpace(F, 3);
true
```

V is the natural structure on which M acts, and may also be obtained using the function **BaseModule**(M):

```
> print V eq BaseModule(M);
true
```

For any matrix algebra M of degree n over a ring R, **BaseModule**(M) returns the generic R-module of rank n, consisting of n-tuples over R. When R is a field, the R-module will be a vector space over R.

Given any m in a matrix algebra, the function given by $v \mapsto vm$ for $v \in V$ is an endomorphism of V, where V is the base module of the parent of m. The functions **Kernel**(m) and **Image**(m) return the standard submodules of V relating to this endomorphism. For example:

```
> print Kernel(m2);
Vector space of degree 3, dimension 1 over GF(5)
Echelonized basis:
```

```
(1 1 4)
> print Image(m2);
Vector space of degree 3, dimension 2 over GF(5)
Echelonized basis:
(0 1 0)
(0 0 1)
```

The kernel contains the elements of V which map to zero, and the image contains all the elements to which some element of V is mapped. Alternative names for these functions are **NullSpace**(m) and **RowSpace**(m) respectively.

It is also possible to multiply a submodule and a matrix. For instance, the following line checks that the kernel as a whole maps onto the trivial submodule:

```
> print Kernel(m2) * m2;
Vector space of degree 3, dimension 0 over GF(5)
```

For further examples, see Section 27.7.

Some of these linear algebra concepts do not depend on the matrices in question being square. When this is so, the matrix algebra M is being considered as a matrix module (or matrix space, if the coefficient ring is a field) by ignoring the ability to multiply two matrices in M, as if the matrices were not square. The operations on matrix modules are explained in Chapter 27, and most of them are available for matrix algebras too. They include functions associated with linear algebra, such as row, column and submatrix operations, and the solution of linear equations. Note especially **EchelonForm**(a), which returns two values: the (row) echelon form of a and a matrix b such that $ba = 1$. For instance:

```
> a, b := EchelonForm(m1);
> print a, b;
[1 0 0]
[0 1 0]
[0 0 1]

[4 3 0]
[1 3 0]
[3 4 4]
> print IsOne(b*m1);
true
```

Table 28.5. Eigenspace functions and canonical forms

MAGMA	Meaning
CharacteristicPolynomial(a)	Characteristic polynomial of a
MinimalPolynomial(a)	Minimal polynomial of a
Eigenvalues(a)	Eigenvalues of a, as a set of tuples each giving an eigenvalue and its multiplicity
Eigenspace(a, e)	Eigenspace of a for eigenvalue e
JordanForm(a)	Returns (i) (generalized) Jordan canonical form j of a (ii) matrix b such that $bab^{-1} = j$ (iii) sequence corresponding to blocks of j, each entry being a tuple containing the irreducible polynomial and multiplicity
RationalForm(a)	Returns (i) rational canonical form f of a (ii) matrix b such that $b * a * b^{-1} = f$ (iii) sequence containing the polynomials corresponding to each block (each one dividing the next)
InvariantFactors(a)	Third return value of **RationalForm**(a)
PrimaryRationalForm(a)	Returns (i) primary rational canonical form p of a (ii) matrix b such that $b * a * b^{-1} = p$ (iii) sequence of 2-tuples corresponding to the blocks of p where each pair consists of the irreducible polynomial and its multiplicity
PrimaryInvariantFactors(a)	Third return value of above function

Finally, Table 28.5 (p. 539) lists some functions connected with eigenspace theory and canonical forms. These functions do require the matrix to be square.

Part VII

Groups

29. Overview of Groups

There are five major group categories installed in MAGMA: finitely presented groups (**GrpFP**), abelian groups (**GrpAb**), soluble groups defined by a polycyclic presentation (**GrpPC**), permutation groups (**GrpPerm**) and matrix groups (**GrpMat**). A number of minor categories including blackbox groups and Coxeter groups are under construction; the reader should consult the *Handbook* for details of these.

It is important to note that the types of information that it is possible to compute for a group G depends upon the category to which G belongs. In the case where G is an fp-group, it is not possible, in general, to even determine whether or not two elements are equal. Since the techniques available for attempting to construct a normal form of the elements of an fp-group almost never works when applied to an arbitrary group, MAGMA does not automatically try to construct such a normal form. Instead, if the user believes that G belongs to a class of fp-groups for which such techniques are likely to succeed, then he/she must invoke them explicitly.

It is useful to introduce a number of terms to distinguish computationally significant properties of groups.

A category equipped with a normal form for the elements of its magmas will be referred to as an *ENF-category*. Thus, of the group categories, all but (**GrpFP**) are NF-categories.

The group operation is denoted by `*` in all the group categories except for the category of abelian groups. It will sometimes be convenient to refer to the group categories having `*` as their group operation as *multiplicative group categories*.

The group categories may be further classified into those defined by presentations (fp-groups, pc-groups and abelian groups) and those given in some concrete representation (permutation groups and matrix groups). For brevity, the two classes will be termed *abstract groups* and *concrete groups*, respectively. An abstract group is defined in terms of generators and relations, while a concrete group is taken to be the group generated by a subset of elements of $\mathrm{Sym}(X)$ (permutation group) or $\mathrm{GL}(n, R)$ (matrix group).

Most structural computation requires the category to possess an element normal form algorithm and, in addition, a *structure normal form*. At the most elementary level the availability of such a normal form for a group G implies that, membership in, and equality of, arbitrary subgroups of G can be determined. A category possessing a structure normal form will be referred to as a *SNF-category*. The categories of polycyclic groups, abelian groups and permutation groups are all SNF-categories. Recalling that matrix groups are not a single category, but rather a family of categories indexed by the coefficient ring, some matrix group categories are SNF and some are not. Currrently, the following matrix categories are SNF:

– Matrix groups over finite fields and finite euclidean rings;

– Finite matrix groups over fields and euclidean rings.

Finally, fp-groups are neither ENF nor SNF.

In this chapter, the operations that are common to most group categories will be presented. However, the semantics of some operators will differ slightly between categories and the reader should consult the relevant *Handbook* chapter where relevant.

29.1 Construction of Groups

29.1.1 Abstract Groups

There are five simple ways to construct the various kinds of abstract group. Firstly, the function **FreeGroup**(n) creates a free group of rank n, in the multiplicative group category **GrpFP**. The function **FreeAbelianGroup**(n) creates a free abelian group of rank n, in the additive group category **GrpAb**. The other three methods are constructors which may be used for arbitrary groups in the categories **GrpFP**, **GrpAb** and **GrpPC**. In these constructors, temporary names for the generators $x_1, \ldots, x_n$ are given on the left side, and a comma-separated list of relations in terms of the x_i is given on the right side:

Group< $x_1, \ldots, x_i$ | relations **>**
AbelianGroup< $x_1, \ldots, x_i$ | relations **>**
PCGroup< $x_1, \ldots, x_i$ | relations **>**

Each relation may be any of the following, where $w, w_1, \ldots, w_k$ denote words in the x_i: $w_1{=}w_2$; w (meaning that w equals the identity); or $w_1{=}w_2{=}\cdots{=}w_k$. Of course, if a group is being created using the **PCGroup** constructor, then the relations must define a polycyclic group.

For example, the free abelian group F of rank 5 may be defined by the statement:

```
> F := FreeAbelianGroup(5);
> print F;
Abelian Group isomorphic to Z (5 copies)
Defined on 5 generators (free)
```

The 2-dimensional space group *p6* with presentation

$$\langle s, t \mid s^3 = t^2 = (st)^6 = 1 \rangle$$

may be defined by the statement

```
> p6<s, t> := Group<s, t | s^3 = t^2 =(s*t)^6 = 1 >;
> print p6;
Finitely presented group p6 on 2 generators
Relations
    s^3 = Id(p6)
    t^2 = Id(p6)
    (s * t)^6 = Id(p6)
```

The infinite abelian group G with generators a and b satisfying the single relation $a + 2b = 0$ may be defined by the statement

```
> G<a, b> := AbelianGroup<a, b | a+2*b = 0 >;
> print G;
Abelian Group isomorphic to Z
Defined on 2 generators
Relations:
    a + 2*b = 0
```

An element of an abstract group G is normally given as a word in the generators of G. The following assignments define w as the conjugate of s by t in *p6*, and define u as the sum (i.e., product) of the generators a and b:

```
> w := t * s * t;
> u := a + b;
```

29.1.2 Concrete Groups

The symmetric group of degree n may be created in the **GrpPerm** category using the function **Sym**(n), if the G-set of the group is $\{1, \ldots, n\}$, or **Sym**(T),

where the G-set of the group corresponds to the enumerated or indexed set T with cardinality n.

The general linear group of $n \times n$ matrices over the ring R may be created using the function **GeneralLinearGroup**(n, R), which has the abbreviation **GL**(n, R). If the ring is a finite field GF(q), then the function may also be called as **GL**(n, q) or **GL**(V), where V is a vector space with n coordinates over GF(q).

Every arbitrary concrete group G definable in MAGMA is either a subset of a symmetric group $L = \mathrm{Sym}(n)$ or $L = \mathrm{Sym}(T)$ (in the permutation group case) or a subset of a general linear group $L = \mathrm{GL}(n, R)$ (in the matrix group case). The standard method for defining an arbitrary concrete group G is by means of a constructor, in which L is specified on the left side, and a subset S of L that generates G is given on the right side. The syntax for these constructors is:

PermutationGroup< n | generators **>**
PermutationGroup< T | generators **>**
MatrixGroup< n, R | generators **>**

The generators must be given in a comma-separated list. For permutation groups, generating elements may be represented as sequences or as products of cycles (in terms of the G-set); for matrix groups, they may be represented as length-n^2 sequences of elements of R. The list of generators can also contain suitable elements of related structures, sets/sequences of elements, or subgroups (implying the generators of the subgroups).

For example, the affine group $\mathrm{AGL}_1(7)$, which is generated by the permutations $(1, 2, 3, 4, 5, 6, 7)$ and $(2, 4, 3, 7, 5, 6)$, may be created in MAGMA by the statement:

```
> AGL := PermutationGroup< 7 | (1, 2, 3, 4, 5, 6, 7),
>                              (2, 4, 3, 7, 5, 6) >;
> print AGL;
Permutation group AGL acting on a set of cardinality 7
    (1, 2, 3, 4, 5, 6, 7)
    (2, 4, 3, 7, 5, 6)
```

The element $(1, 5)(2, 4)(6, 7)$ of this group may be defined as follows:

```
> c := AGL!(1, 5)(2, 4)(6, 7);
```

As a matrix group example, the integer matrices

$$x = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad y = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

generate the group of isometries that fix the square with vertices $(\pm 1, 0)$ and $(0, \pm 1)$. This group may be constructed as follows:

```
> Sq := MatrixGroup< 2, IntegerRing()  |
>         [-1, 0, 0, 1], [0, 1, -1, 0] >;
> print Sq;
MatrixGroup(2, Integer Ring)
Generators:
    [-1  0]
    [ 0  1]

    [ 0  1]
    [-1  0]
```

The matrix $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ belonging to Sq may be created by the statement:

```
> d := Sq![0, 1, 1, 0];
> print d;
[0 1]
[1 0]
```

29.1.3 Some Standard Groups

Some groups are so frequently needed that their definitions are built into MAGMA. The functions for these standard groups are summarized in Table 29.1. Observe that in addition to taking one or two arguments to describe the size of the group, these functions take as their first argument a group category, such as **GrpPerm** for permutation groups or **GrpFP** for finitely-presented groups. The specification of the category indicates the desired representation for the group.

For example, the following three assignments all create the same group, namely, the alternating group on nine elements generated by $(3, 4, 5, 6, 7, 8, 9)$ and $(1, 2, 3)$. However, it is best to use the third version (the function **AlternatingGroup**), because MAGMA will be able to construct the group quickly and will know additional information about it such as its order:

```
> A9 := sub< Sym(9) | (3,4,5,6,7,8,9), (1,2,3) >;
> A9 := PermutationGroup< 9 | (3,4,5,6,7,8,9), (1,2,3) >;
> A9 := AlternatingGroup(GrpPerm, 9);
> print A9;
Permutation group A9 acting on a set of cardinality 9
Order = 181440 = 2^6 * 3^4 * 5 * 7
```

Table 29.1. Some standard groups

MAGMA	Meaning
CyclicGroup(C, n)	Cyclic group of order n in category C
DihedralGroup(C, n)	Dihedral group of degree n and order $2n$ in category C
AbelianGroup(C, Q)	Abelian group in category C which is the direct product of cyclic groups $C_{n_1} \times C_{n_2} \times \cdots \times C_{n_r}$, given sequence $Q = [n_1, \ldots, n_r]$ of positive integers
AlternatingGroup(C, n), **Alt**(C, n)	Alternating group of degree n in category C
SymmetricGroup(C, n), **Sym**(C, n)	Symmetric group of degree n in category C

```
(3, 4, 5, 6, 7, 8, 9)
(1, 2, 3)
```

Similarly, the following two assignments create the same finitely-presented group, namely, the dihedral group of order 10:

```
> D5 := Group<x, y | x^5, y^2, (x*y)^2 >;
> D5 := DihedralGroup(GrpFP, 5);
> print D5;
Finitely presented group D5 on 2 generators
Relations
    D5.1^5 = Id(D5)
    D5.2^2 = Id(D5)
    (D5.1 * D5.2)^2 = Id(D5)
```

Generators from these standard groups may be given special names by means of a generator assignment statement. These names will become identifiers whose values are the generators, and in the categories **GrpAb**, **GrpFP** and **GrpPC** they will also become printnames for these generators. For example, the group **DihedralGroup**(**GrpFP**, 5) corresponds to the set of all symmetries of a regular pentagon. As shown above, this finitely-presented group has two generators. The first generator has order 5 and corresponds geometrically to a rotation; the second has order 2 and corresponds to a reflection. Therefore it may assist the user's geometric intuition to give the generators names such as *rot* and *refl*:

To perform this example online, type `load "I96c29e1";`

```
> D5<rot, refl> := DihedralGroup(GrpFP, 5);
```

```
> print D5;
GrpFP: D5 on 2 generators
Relations
    rot^5 = Id(D5)
    refl^2 = Id(D5)
    (rot * refl)^2 = Id(D5)
```

29.2 Operations on Group Elements

29.2.1 Arithmetic

Table 29.2. Arithmetic in multiplicative groups

MAGMA	Meaning
x*y	Product of x and y
x^−1	Inverse of x
x^n	n^{th} power of x, for $n \in \mathbb{Z}$
x^y	Conjugate of x by y, i.e., $y^{-1}xy$
x/y	Quotient of x by y, i.e., xy^{-1}
(x, y)	Commutator of x and y, i.e., $x^{-1}y^{-1}xy$
$(x_1, \ldots, x_n)$	Left-normed commutator of $x_1, \ldots, x_n$
Id(G),G!1	Identity of group G

Table 29.3. Arithmetic in additive groups

MAGMA	Meaning
x+y	Sum of x and y
-x	Inverse (negative) of x
n*x	x added to itself n times, for $n \geq 0$; else $-(\lvert n \rvert * x)$
x-y	Sum of x and $-y$
Id(G),G!0	Identity of abelian group G

Table 29.2 and Table 29.3 display the arithmetic operations on the elements of multiplicative groups and additive groups (i.e., abelian groups), respectively. In these tables, $x, y, x_1, \ldots, x_n$ denote elements of a commmon group. It should be noted that for a category without an ENF (namely, the category of fp-groups), the elements returned will not be canonical.

For instance, the following lines form the product of the inverse of $c = (2,4)(5,6)$ with $(3,4,5)$:

```
> S6 := Sym(6);
> c := S6!(2,4)(5,6);
> print c^-1 * S6!(3,4,5);
(2, 5, 6, 3, 4)
```

The ^ operator has an additional use in all the multiplicative group categories. The *conjugate* of x by y, namely $y^{-1}xy$, is customarily written by mathematicians as x^y. MAGMA's notation $x\hat{}y$ is based on this notation. For instance, the next line uses a set constructor to form the conjugacy class containing the element e^3 of *S6*, that is, the set of all elements of *S6* which are conjugate to e^3:

```
> e := S6!(1, 2, 4, 3, 5, 6);
> ConjClass := { (e^3)^x : x in S6 };
> print ConjClass;
{
    (1, 3)(2, 5)(4, 6),
    (1, 5)(2, 3)(4, 6),
    (1, 2)(3, 5)(4, 6),
    (1, 2)(3, 6)(4, 5),
    (1, 3)(2, 6)(4, 5),
    (1, 6)(2, 3)(4, 5),
    (1, 2)(3, 4)(5, 6),
    (1, 3)(2, 4)(5, 6),
    (1, 5)(2, 4)(3, 6),
    (1, 5)(2, 6)(3, 4),
    (1, 6)(2, 5)(3, 4),
    (1, 4)(2, 3)(5, 6),
    (1, 6)(2, 4)(3, 5),
    (1, 4)(2, 5)(3, 6),
    (1, 4)(2, 6)(3, 5)
}
```

However, it is more efficient in practice to use the conjugacy class function **Class**(H, x) for this purpose:

```
> print Class(S6, e^3) eq ConjClass;
true
```

Another product that frequently arises in group theory is the *commutator* $x^{-1}y^{-1}xy$ of group elements x and y. The MAGMA syntax for the commutator is (x, y). Note that the MAGMA notation differs from the traditional square

bracket notation, $[x, y]$, since MAGMA uses square brackets for sequences. For instance, the following statement computes the commutator of c and e in two different ways:

```
> print (c, e), c^-1 * e^-1 * c * e;
(1, 6, 5)(2, 3, 4)
(1, 6, 5)(2, 3, 4)
```

The identity element for any group G is **Identity**(G) or **Id**(G).

29.2.2 Boolean Operations

Table 29.4. Boolean operators for group elements

MAGMA	Meaning
x **eq** y	**true** if x and y are equal
x **ne** y	**true** if x and y are not equal
x **cmpeq** y	**true** if x and y are compatible and equal
IsIdentity(x), **IsId**(x)	**true** if $x \in G$ is the identity of G

The Boolean operators for elements are presented in Table 29.4. While they apply to every category, it should be noted that in the category of fp-groups, elements x and y are equal if and only if they reduce to the same word under *free reduction*.

The use of equality will be illustrated in the following code, which looks for a cube root of $(1,3)(2,5)(4,6)$ in *S6*.

```
> S6 := Sym(6);
> print exists(x){x: x in S6 | x^3 eq S6!(1,3)(2,5)(4,6)};
true
> print x;
(1, 2, 4, 3, 5, 6)
```

29.2.3 Order of an Element

If x is an element of finite order in an ENF-group G, the function **Order**(x) will return that order.

29.3 Generating Random Elements

Various group theoretic algorithms require a means of generating pseudo-random elements of a group. There are two separate approaches to this problem depending upon whether or not a SNF is available for the group. A SNF allows elements of group G to be sampled with equal probability of being chosen. Assuming that a SNF has been constructed for G, the function **Random**(G) will return such an element. Note that if the user has not created a SNF for G, then this function will not do so: instead it simply evaluates a short random word in the generators of G. In this case the element returned by **Random**(G) is usually far from being random.

In the case where a SNF is not known, an alternative procedure should be followed. Firstly, the user must initialize a process that will be used to produce random elements. The process creation function, **RandomProcess**(G), builds an expanded generating set X consisting of randomized elements of G and returns a process$P$. If the function **Random**(P) is now involved, a new random element E is produced at the cost of one multiplication. At the same time X is further randomized by replacing one of its existing elements by e. This random element generator will not, in general, sample the elements of G with equal probability. The user can control 'randomness' of the elements produced to a degree by setting values for the parameters **Slots** (the size of X) and **Scramble** (the number of multiplications used to construct X).

The random process will be illustrated by using it to examine the order of 5 random elements in the matrix group $\Omega^+(20, 4)$:

```
> G := OmegaPlus(20, 4);
> RP := RandomProcess(G);
> print [ FactoredOrder(Random(RP)) : i in [1..5] ];
[
    [ <3, 1>, <5, 2>, <11, 1>, <31, 1>, <41, 1> ],
    [ <2, 2>, <3, 1>, <5, 1>, <17, 1>, <241, 1> ],
    [ <3, 1>, <11, 1>, <31, 1> ],
    [ <3, 2>, <7, 1>, <19, 1>, <73, 1> ],
    [ <3, 2>, <5, 1>, <7, 1>, <17, 1> ]
]
```

29.4 Constructing a Subgroup

29.4.1 Defining a General Subgroup

Let G be a group belonging to any category and let $S = h_1, \ldots, h_r$ be a set of elements of G. The subgroup H generated by S may be defined using the **sub**-constructor:

sub< G | generators **>**

The subgroup generators are given in the form of a list where the items of the list may be any of:

– An element of G;

– A sequence defining an element of G;

– A set or sequence of either of the above;

– A subgroup of G (contributing its generators to the definition of H);

– A set or sequence of subgroups of G.

This constructor returns two values: the group H, as a group belonging to the same category as G; and a monomorphism $\phi : H \to G$ giving the embedding of H in G.

For instance, the group *U2* of all 2×2 integer matrices having determinant ±1 has the presentation

$$\langle r_1, r_2, r_3, z \mid r_1^2, r_2^2, r_3^3, (r_1r_2)^3 = (r_1r_3)^2 = z, z^2 = 1\rangle.$$

The words r_1r_2 and r_1r_3 generate the 2-dimensional modular group *M2* as a subgroup of *U2* ([CoM72] 7.2). These two groups may be defined as follows:

```
> U2<r1, r2, r3, z> := Group<r1, r2, r3, z |
>     r1^2, r2^2, r3^2, (r1*r2)^3 = (r1*r3)^2 = z, z^2 >;
> M2<s, t>, f := sub<U2 | r1*r2, r1*r3 >;
> print M2;
Finitely presented group M2 on 2 generators
Generators as words in group U2
    s = r1 * r2
    t = r1 * r3
```

The function *GLSyl* below constructs the Sylow p-subgroup for the general linear group $\mathrm{GL}(n, k)$, where K is a finite field of characteristic p:

```
GLSyl := function(G)
   K := CoefficientRing(G);
   n := Degree(G);
   R := MatrixRing(K, n);
   e := func< i, j | MatrixUnit(R, i, j) >;
   return sub< G |  { R!1 + a*e(i,j) : a in K, j in [i+1],
         i in [1 .. n - 1] | a ne 0 } >;
end function;
```

29.4.2 Constructing a Normal Closure

The **sub**-constructor creates the smallest subgroup containing the designated subset S of elements of G. By contrast, MAGMA also offers a constructor that treats the elements of S as *normal* generators is provided: the **ncl**-constructor creates the smallest normal subgroup of G containing S. It has the same arguments as **sub**.

29.4.3 Constructing a Standard Subgroup

Table 29.5. Standard subgroups

MAGMA	Meaning
$H\hat{}g$	Conjugate $g^{-1}Hg$ of subgroup H with respect to element g
H **meet** K	Intersection $H \cap K$ of H and K
Centralizer(G, x)	$\{g \in G : x^g = x\}$
Centralizer(G, H)	$\{g \in G : h^g = h, \forall h \in H\}$
Core(G, H)	Maximal normal subgroup of G contained in the subgroup H
Normalizer(G, H)	$\{g \in G : H^g = H\}$
$H\hat{}G$	Normal closure of the subgroup H in the group G
pCore(G, P)	Maximal normal p-subgroup of G (p-core of G) for prime P dividing $\|G\|$
SylowSubgroup(G, p)	Sylow p-subgroup of G, p a prime

MAGMA has a very large number of functions and operators for computing various standard subgroups. Generally, very subtle and efficient algorithms are employed for finding these subgroups. Table 29.5 gives details of some subgroup constructions. To remember the order of the arguments, notice that the group in which the subgroup is to be calculated is typically the

first argument. Not all of these functions are available in every category. For example, functions **Centralizer**, **pCore** and **SylowSubgroup** are not applicable to fp-groups.

For example, the following lines demonstrate how to create the normalizer of the Sylow 2-subgroup of the permutation group of degree 9 generated by (1,2,3,4)(5,6,7,8), (1,8)(3,6)(5,7), (1,4,7)(2,3,5)(6,9,8):

```
> gp9 := PermutationGroup< 9 | (1,2,3,4)(5,6,7,8),
              (1,8)(3,6)(5,7), (1,4,7)(2,3,5)(6,9,8) >;
> syl2 := SylowSubgroup(gp9, 2);
> print syl2;
Permutation group syl2 of degree 9
Order = 16 = 2^4
    (3, 7)(4, 9)(5, 6)
    (2, 5)(4, 7)(6, 8)
    (2, 4, 8, 7)(3, 6, 9, 5)
> norm := Normalizer(gp9, syl2);
> print norm eq syl2;
true
```

Therefore the Sylow 2-subgroup is self-normalizing.

29.5 Constructing a Quotient Group

29.5.1 General Quotients

Let G be a group belonging to any category, let $S = \{h_1, \ldots, h_r\}$ be a subset of G, and let N be the normal closure of S. The quotient group G/N may be created by using the **quo**-constructor:

quo< G | generators **>**

The generators of N are given as a list whose items satisfy the same rules as the **sub**-constructor. The **quo**-constructor also returns two values: the quotient group $Q = G/N$; and the natural surjection $\tau : G \to Q$.

In the current version of MAGMA, the quotient Q is returned as a group in the same category as G, *except* when G is a matrix group, in which case Q is returned as a permutation group. In the near future, quotients of both permutation and matrix groups will be returned as members of a new category.

For instance, the following statement constructs the quotient of Sym(4) by the Klein 4-group:

```
> S4 := Sym(4);
> Q, f := quo< S4 | (1,2)(3,4), (1,3)(2,4) >;
> print Q;
Permutation group Q acting on a set of cardinality 6
Order = 6 = 2 * 3
    (1, 2)(3, 5)(4, 6)
    (1, 3)(2, 4)(5, 6)
```

29.5.2 Special Quotients

Table 29.6. Special quotient functions

MAGMA	Meaning
AbelianQuotient(G)	Quotient of G by its derived subgroup (quotient lies in category **GrpAb**)
pQuotient(G, p, c)	p-quotient of G calculated to lower exponent-p central class c (quotient lies in category **GrpPC**)
SolubleQuotient(G)	Quotient of G by the stationary term of its derived series (the soluble residual) (quotient lies in **GrpPC**)
RadicalQuotient(G)	Quotient of permutation group G by its soluble radical (the largest normal soluble subgroup) (quotient lies in **GrpPerm**)

In the case of certain characteristic subgroups, special algorithms are available for constructing the corresponding quotients. These functions are summarized in Table 29.6. Not all of these functions are available for every category – the user should check availability with the *Handbook* or online help system.

In addition to returning the quotient group Q, each of these functions also returns the natural surjection $\tau : G \to Q$.

In the example below, several of the quotient functions are applied to $\mathrm{Sym}(5) \wr \mathrm{Sym}(3)$. Note that since the group has no proper soluble subgroup, the radical quotient is the whole group.

```
> G := WreathProduct(Sym(5), Sym(3));
> A := AbelianQuotient(G);
> print A;
```

```
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*A.1 = 0
    2*A.2 = 0
> B := SolubleQuotient(G);
GrpPC : B of order 48 = 2^4 * 3
PC-Relations:
    B.1^2 = Id(B),
    B.2^2 = Id(B),
    B.3^3 = Id(B),
    B.4^2 = Id(B),
    B.5^2 = Id(B),
    B.2^B.1 = B.2 * B.5,
    B.3^B.1 = B.3^2,
    B.3^B.2 = B.3 * B.4,
    B.4^B.1 = B.4 * B.5,
    B.4^B.3 = B.4 * B.5,
    B.5^B.3 = B.4
> C := RadicalQuotient(G);
> print Order(C) eq Order(G);
true
```

As another example, the **pQuotient** function applied to the group

$$\langle a, b \mid a^{(b,a)} = a^5, b^{(a,b)} = b^5 \rangle$$

constructs a 2-quotient order 2048. The progress of the calculation may be displayed by setting the parameter **Print** to 1:

```
> G := Group< a, b | a^(b,a) = a^5, b^(a,b) = b^5>;
> Q := pQuotient(G, 2, 10 : Print := 1);
Lower exponent-2 central series for G
Group: G to lower exponent-2 central class 1 has order 2^2
Group: G to lower exponent-2 central class 2 has order 2^5
Group: G to lower exponent-2 central class 3 has order 2^8
Group: G to lower exponent-2 central class 4 has order 2^10
Group: G to lower exponent-2 central class 5 has order 2^11
Completed. Lower exponent-2 central class = 5, Order = 2^11
```

29.6 Order and Index Functions

For a finite group G, one of the most useful invariants is its order. The order is returned as an integer by the function **Order**(G), or as a factorization

sequence by the function **FactoredOrder**(G). If either of these functions is applied to an infinite matrix group, it will fail.

Care should be exercised when applying the function to an fp-group G, since the order is determined by using the Todd-Coxeter procedure to enumerate the cosets of the identity subgroup. If G is infinite or has large order, or if the enumeration is difficult, the function may exhaust memory without returning an answer. For this reason, when **Order** calls the Todd-Coxeter procedure, it does so with a limit of a million cosets. The user may override this limit by calling the Todd-Coxeter function directly.

Similarly, if H is a subgroup of G having finite index, the index $[G : H]$ is returned as an integer by the function **Index**(G, H) or as a factorization sequence by the function **FactoredIndex**(G, H).

The index functions may only be applied to matrix groups for which a BSGS can be computed (see Section **??**). In practice, this requires that the matrix group be finite and defined over either a field or a euclidean ring.

The example below uses the *GLSyl* function defined above to construct the Sylow 2-subgroup of $\mathrm{GL}(3, 8)$, and then applies order and index functions:

```
> G := GeneralLinearGroup(3, GF(8));
> print Order(G);
115379712
> print FactoredOrder(G);
[ <2, 9>, <3, 2>, <7, 3>, <73, 1> ]
> T := GLSyl(G);
> print FactoredOrder(T);
[ <2, 9> ]
> print Index(G, T);
225351
> print FactoredIndex(G, T);
[ <3, 2>, <7, 3>, <73, 1> ]
```

Another useful numerical invariant of a group is its exponent, that is, the lowest-common multiple of the orders of its elements. The MAGMA function **Exponent**(G) computes the exponent of a group G for which a SNF is computable. For example, when applied to the group $\mathrm{GL}(3, 8)$ above, it gives the answer 18 396.

29.7 Properties of Groups and Subgroups

The functions listed in Table 29.7 determine abstract properties of a group. Note that, with the exception of the function **IsPerfect**, these functions are **not** applicable to fp-groups.

Table 29.7. Some tests on properties of a group

IsAbelian(G)	**IsElementaryAbelian**(G)
IsNilpotent(G)	**IsPerfect**(G)
IsCyclic(G)	**IsSimple**(G)
IsSpecial(G)	**IsExtraSpecial**(G)
IsSoluble(G), **IsSolvable**(G)	

There are also functions which test properties relating to the way in which a subgroup H or pair of subgroups H and K are embedded in group G. Table 29.8 lists some of them.

Table 29.8. Some tests on properties of a subgroup

IsConjugate(G, H, K)	**IsNormal**(G, H)
IsSubnormal(G, H)	**IsSelfNormalizing**(G, H)
IsCentral(G, H)	**IsMaximal**(G, H)

29.8 Characteristic Subgroups, Series and Normal Structure

29.8.1 Characteristic Subgroups

MAGMA provides functions for determining the standard characteristic subgroups, as listed in Table 29.9. All but one of these functions require a SNF for the group G; the only exception is that **DerivedSubgroup** may be applied to an fp-group. Furthermore, note that the functions **Agemo** and **Omega** are only defined for p-groups, while **FrattiniSubgroup** is only implemented for p-groups.

The example below applies some of these functions to the soluble primitive group *gp9* introducted above:

Table 29.9. Characteristic subgroups

MAGMA	Meaning
Centre(G)	Centre of G
Hypercentre(G)	Stationary term of upper central series of G
DerivedSubgroup(G)	Derived subgroup of G
FittingSubgroup(G)	Maximal normal nilpotent subgroup of G
FrattiniSubgroup(G)	Intersection of maximal subgroups of p-group G
Radical(G)	Maximal normal soluble subgroup of G
SolubleResidual(G)	Stationary term of derived series
Agemo(G, i)	Subgroup of p-group G generated by the p^i-th powers
Omega(G, i)	Subgroup of p-group G generated by the elements of order dividing p^i

```
> gp9 := PermutationGroup< 9 | (1,2,3,4)(5,6,7,8),
             (1,8)(3,6)(5,7), (1,4,7)(2,3,5)(6,9,8) >;
> print Centre(gp9);
Permutation group acting on a set of cardinality 9
Order = 1
    Id($)
> print Hypercentre(gp9);
Permutation group acting on a set of cardinality 9
Order = 1
    Id($)
> print DerivedSubgroup(gp9);
Permutation group acting on a set of cardinality 9
Order = 36 = 2^2 * 3^2
    (1, 8, 3, 6)(2, 7, 4, 5)
    (1, 3, 9)(2, 6, 7)(4, 5, 8)
> print FittingSubgroup(gp9);
Permutation group acting on a set of cardinality 9
Order = 9 = 3^2
    (1, 8, 2)(3, 4, 6)(5, 7, 9)
    (1, 7, 4)(2, 5, 3)(6, 8, 9)
> print Radical(gp9);
Permutation group acting on a set of cardinality 9
Order = 144 = 2^4 * 3^2
    (2, 5)(4, 7)(6, 8)
    (1, 8, 2)(3, 4, 6)(5, 7, 9)
    (2, 4, 8, 7)(3, 6, 9, 5)
    (1, 7, 4)(2, 5, 3)(6, 8, 9)
    (2, 5, 8, 6)(3, 4, 9, 7)
```

```
    (2, 8)(3, 9)(4, 7)(5, 6)
> SolubleResidual(gp9);
Permutation group acting on a set of cardinality 9
Order = 1
    Id($)
```

29.8.2 Series

Table 29.10 lists the MAGMA functions creating various series of characteristic subgroups (as a sequence of subgroups). They are applicable only to groups for which a SNF can be computed, so none of them are available for fp-groups. Moreover, certain of the series are only defined for p-groups or soluble groups, while not all of the functions are available for all categories of groups possessing SNFs.

Table 29.10. Series

MAGMA	Meaning
ChiefSeries(G)	Chief series for G
CompositionSeries(G)	Composition series for G
ElementaryAbelianSeries(G)	Series $G = E_0 \rhd E_1 \rhd \cdots \rhd E_r$, where E_i/E_{i-1} is elementary abelian (G must be soluble)
DerivedSeries(G)	Series $G = G^{(0)} \rhd G^1 \rhd G^2 \rhd \cdots$, where $G^{(i)} = [G^{(i-1)}, G^{(i-1)}]$
DerivedLength(G)	If G is soluble, returns least integer l such that $G^{(l)}$ is trivial; else returns number of terms up to and including stationary term
JenningsSeries(G)	Series $G = J_1 \rhd J_2 \rhd J_3 \rhd \cdots$, where $J_{i+1} = \langle (J_i, G), J_k^p \rangle$ with $k = \lceil (i+1)/p \rceil$ (G must be a p-group)
LowerCentralSeries(G)	Series $G = G_1 \rhd G_2 \rhd G_3 \rhd \cdots$, where $G_i = [G_{i-1}, G]$
NilpotencyClass	If G is nilpotent, returns least integer c such that term G_c of lower central series is trivial; else returns -1
pCentralSeries(G, p)	Series $G = P_1 \rhd P_2 \rhd P_3 \rhd \cdots$, where $P_{i-1} = [G, P_i]P_i$ (G must be soluble and prime p must divide $\lvert G \rvert$
SubnormalSeries(G, H)	Series $N_1 = H \lhd N_2 \lhd \ldots \lhd N_r = G$, where H is a subgroup of G
UpperCentralSeries(G)	Series $Z_0 = 1, Z_i/Z_{i-1} = Z(G/Z_{i-1})$

For example:

```
> gp9 := PermutationGroup< 9 | (1,2,3,4)(5,6,7,8),
              (1,8)(3,6)(5,7), (1,4,7)(2,3,5)(6,9,8) >;
> print ChiefSeries(gp9);
[
    Permutation group gp9 acting on a set of cardinality 9
    Order = 144 = 2^4 * 3^2
        (1, 2, 3, 4)(5, 6, 7, 8)
        (1, 8)(3, 6)(5, 7)
        (1, 4, 7)(2, 3, 5)(6, 9, 8),
    Permutation group acting on a set of cardinality 9
    Order = 72 = 2^3 * 3^2
        (3, 7)(4, 9)(5, 6)
        (2, 8)(3, 4)(7, 9)
        (2, 6)(3, 9)(5, 8)
        (1, 7, 4)(2, 5, 3)(6, 8, 9),
    Permutation group acting on a set of cardinality 9
    Order = 9 = 3^2
        (1, 8, 2)(3, 4, 6)(5, 7, 9)
        (1, 7, 4)(2, 5, 3)(6, 8, 9),
    Permutation group acting on a set of cardinality 9
    Order = 1
        Id($)
]
> print DerivedSeries(gp9);
print DS;
[
    Permutation group gp9 acting on a set of cardinality 9
    Order = 144 = 2^4 * 3^2
        (1, 2, 3, 4)(5, 6, 7, 8)
        (1, 8)(3, 6)(5, 7)
        (1, 4, 7)(2, 3, 5)(6, 9, 8),
    Permutation group acting on a set of cardinality 9
    Order = 36 = 2^2 * 3^2
        (1, 6, 3, 8)(2, 5, 4, 7)
        (2, 5, 8, 6)(3, 4, 9, 7)
        (1, 7, 2, 3)(4, 6, 5, 9),
    Permutation group acting on a set of cardinality 9
    Order = 9 = 3^2
        (1, 5, 6)(2, 9, 4)(3, 8, 7)
        (1, 7, 4)(2, 5, 3)(6, 8, 9),
    Permutation group acting on a set of cardinality 9
    Order = 1
        Id($)
]
```

```
> print LowerCentralSeries(gp9);
[
    Permutation group gp9 acting on a set of cardinality 9
    Order = 144 = 2^4 * 3^2
        (1, 2, 3, 4)(5, 6, 7, 8)
        (1, 8)(3, 6)(5, 7)
        (1, 4, 7)(2, 3, 5)(6, 9, 8),
    Permutation group acting on a set of cardinality 9
    Order = 36 = 2^2 * 3^2
        (1, 6, 3, 8)(2, 5, 4, 7)
        (2, 5, 8, 6)(3, 4, 9, 7)
        (1, 7, 2, 3)(4, 6, 5, 9),
    Permutation group acting on a set of cardinality 9
    Order = 18 = 2 * 3^2
        (1, 3)(2, 4)(5, 7)(6, 8)
        (1, 2)(3, 7)(4, 5)(6, 9)
        (2, 8)(3, 9)(4, 7)(5, 6),
    Permutation group acting on a set of cardinality 9
    Order = 9 = 3^2
        (1, 5, 6)(2, 9, 4)(3, 8, 7)
        (1, 7, 4)(2, 5, 3)(6, 8, 9)
]
```

From the derived series, the group *gp9* is soluble with soluble length 3 (since this series reaches the trivial subgroup). However, from the lower central series, *gp9* is not nilpotent (since this series fails to reach the trivial subgroup).

29.9 Abstract Structure of a Group

A knowledge of the isomorphism types of the composition factors of a group G gives important information about its abstract structure. The function **CompositionFactors**(G), applicable to permutation and matrix groups, returns a sequence of tuples $F_1, \ldots, F_r$ identifying the composition factors of G. The order of terms in the sequence corresponds to some composition series

$$1 = T_0 \lhd T_1 \lhd \ldots \lhd T_r = G$$

for G. The factor tuple F_i identifies the composition factor $S_i = T_i/T_{i-1}$ for $i = 1, \ldots, r$. In each tuple $F_i = \texttt{<}\ f_i, d_i, q_i\ \texttt{>}$, the integer f_i defines the family of simple groups to which S_i belongs, and the integers d_i and q_i are parameters of the family. The exact interpretation of the tuples is given in Table 29.11, with the classification for the sporadic groups given in Table 29.12.

Table 29.11. Group families for **CompositionFactors**

f	Family name	f	Family name
1	A(d,q)	11	2B$(2,q)$
2	B(d,q)	12	2D(d,q)
3	C(d,q)	13	3D$(4,q)$
4	D(d,q)	14	2G$(2,q)$
5	G$(2,q)$	15	2F$(4,q)$
6	F$(4,q)$	16	2E$(6,q)$
7	E$(6,q)$	17	Alternating(d)
8	E$(7,q)$	18	Sporadic group - see Table 29.12
9	E$(8,q)$	19	Cyclic(q)
10	2A(d,q)		

Table 29.12. Sporadic groups for **CompositionFactors**

d	Group name	d	Group name	d	Group name
1	M_{11}	10	SUZ	19	LY
2	M_{12}	11	J_3	20	RU
3	M_{22}	12	CO_1	21	ON
4	M_{23}	13	CO_2	22	TH
5	M_{24}	14	CO_3	23	HA
6	J_1	15	HE	24	BM
7	HS	16	$M(22)$	25	M
8	J_2	17	$M(23)$	26	J_4
9	MCL	18	$M(24)$		

For example, consider the problem of determining the structure of the group G of 9×9 matrices generated by these matrices over GF(3):

$$\begin{pmatrix} 2&2&0&0&0&0&0&0&0\\ 2&2&1&0&0&0&0&0&0\\ 0&1&1&0&0&0&0&0&0\\ 0&0&0&0&1&2&0&0&0\\ 0&0&0&2&1&0&0&0&0\\ 0&0&0&2&2&1&0&0&0\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1 \end{pmatrix} \qquad \begin{pmatrix} 0&0&0&1&0&0&0&0&0\\ 0&0&0&0&1&2&0&0&0\\ 0&0&0&2&1&1&0&0&0\\ 1&2&0&0&0&0&0&0&0\\ 0&0&2&0&0&0&0&0&0\\ 2&2&2&0&0&0&0&0&0\\ 0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&1 \end{pmatrix}$$

It may be solved in Magma as follows:

To perform this example online, type `load "I96c29e2";`

```
> G := MatrixGroup<9, GF(3) |
>            [ 2, 2, 0, 0, 0, 0, 0, 0, 0,
>              2, 2, 1, 0, 0, 0, 0, 0, 0,
>              0, 1, 1, 0, 0, 0, 0, 0, 0,
>              0, 0, 0, 0, 1, 2, 0, 0, 0,
>              0, 0, 0, 2, 1, 0, 0, 0, 0,
>              0, 0, 0, 2, 2, 1, 0, 0, 0,
>              0, 0, 0, 0, 0, 0, 1, 0, 0,
>              0, 0, 0, 0, 0, 0, 0, 1, 0,
>              0, 0, 0, 0, 0, 0, 0, 0, 1 ],
>            [ 0, 0, 0, 1, 0, 0, 0, 0, 0,
>              0, 0, 0, 0, 1, 2, 0, 0, 0,
>              0, 0, 0, 2, 1, 1, 0, 0, 0,
>              1, 2, 0, 0, 0, 0, 0, 0, 0,
>              0, 0, 2, 0, 0, 0, 0, 0, 0,
>              2, 2, 2, 0, 0, 0, 0, 0, 0,
>              0, 0, 0, 0, 0, 0, 1, 0, 0,
>              0, 0, 0, 0, 0, 0, 0, 1, 0,
>              0, 0, 0, 0, 0, 0, 0, 0, 1 ] >;
> cf := CompositionFactors(G);
> // 2-dimensional layout
> print cf;
    G
    |  Cyclic(2)
    *
    |  A(2, 3)               = L(3, 3)
    *
    |  A(2, 3)               = L(3, 3)
    *
    |  Cyclic(2)
    *
    |  Cyclic(2)
    1
> // show tuples in cf
> for i in [1..#cf] do print cf[i]; end for;
<19, 0, 2>
<1, 2, 3>
<1, 2, 3>
<19, 0, 2>
<19, 0, 2>
```

The items in the middle of the chain indicate the proper subgroups of a composition series for G. The isomorphism type of the corresponding composition

factor is indicated on the right. It can be seen that the group has a soluble quotient isomorphic to the cyclic group of order 2. After this, there are two copies of PSL$(3, 3)$ sitting above an abelian group of order 4.

If G is known to be a simple group, the function **NameSimple**(G) returns a tuple identifying G according to the conventions described above.

29.10 Conjugacy Classes of Elements

The three major issues concerning the conjugacy classes of elements of a group G are:

– to determine whether two elements are conjugate in G;

– to determine the classes of G, in the sense of constructing a representative and basic invariants for each class;

– having determined these classes, to determine quickly the class in which a given element of G falls.

These problems will be considered in turn.

29.10.1 Testing Conjugacy

If x and y are elements of a group G, the function **IsConjugate**(G, x, y) tests whether x and y are conjugate in G. If x and y are conjugate, the function returns as its second value, an element g such that $x^g = y$.

MUST FIND A DIFFERENT EXAMPLE. THIS IS USED EXTENSIVELY IN Iperm.tex

For instance, the group below is is the group of symmetries of a cube:

```
> cube<a, b, c> := PermutationGroup< 8 |
>   (1, 2, 3, 4)(5, 6, 7, 8), (2, 4, 5)(3, 8, 6),
>   (1, 5)(2, 6)(3, 7)(4, 8) >;
> conj, z := IsConjugate(cube, b, b^-1);
> print conj;
true
> print z;
(3, 6)(4, 5)
```

If it is necessary to determine for a large number of elements the conjugacy class to which they belong, it may be more efficient to use the function **ClassMap** (Section 29.10.4).

29.10.2 Constructing One Class

The set of conjugates of x under the action of the subgroup H is given by **Class**(H, x) or **Conjugates**(H, x). If the value of H is set to G, the function will return the entire conjugacy class of x. For example, calculations above showed that b^{-1} lies in the same class as b. The statement below confirms this, by displaying the entire conjugacy class of b (including b^{-1}):

```
> print Class(cube, b);
{
    (1, 6, 8)(2, 7, 4),
    (1, 3, 6)(4, 7, 5),
    (1, 6, 3)(4, 5, 7),
    (2, 4, 5)(3, 8, 6),
    (1, 8, 6)(2, 4, 7),
    (1, 8, 3)(2, 5, 7),
    (2, 5, 4)(3, 6, 8),
    (1, 3, 8)(2, 7, 5)
}
```

Since **Class**(H, x) and Conjugates(H, x) return the class containing the element x as an enumerated set. Consequently, this function should not be used on large classes. If the user merely needs to know the size of the class containing x, it is much more efficient to use the expression **Index**$(G,$ **Centralizer**(G, x).

29.10.3 Constructing All the Classes

Let $K_1, \ldots, K_r$ denote the conjugacy classes of G. For the purposes of this discussion, constructing the classes of G will be taken to mean determining, for each class K_i a representative element x_i and the cardinality of K_i (the *length* of the class).

The function **ConjugacyClasses**(G) or **Classes**(G) constructs all the conjugacy classes in the group G. It does not return the classes as such; it merely returns sufficient information about each class so that the user can easily compute an entire class if it is wanted explicitly. The return value of **Classes**(G) is a sequence of tuples, one tuple for each conjugacy class; the components of each tuple are the order of the elements in that class, the length of the class, and a representative element for the class.

Individual entries of this sequence should be treated by the user as normal tuples. However, when the whole classes sequence is printed, there is a special display format provided for the convenience of the user:

```
> cubecl := Classes(cube);
> cl9info := cubecl[9]; // information on class 9
> print cl9info;
<4, 6, (1, 3, 8, 6)(2, 4, 7, 5)>
> print cl9info[2]; // length of class 9
6
> print cubecl[4, 1]; // order of class 4
2

> // special display format
> print cubecl;
Conjugacy Classes of group cube
--------------------------------
[1]     Order 1       Length 1
        Rep Id(cube)

[2]     Order 2       Length 1
        Rep (1, 7)(2, 8)(3, 5)(4, 6)

[3]     Order 2       Length 3
        Rep (1, 3)(2, 4)(5, 7)(6, 8)

[4]     Order 2       Length 3
        Rep (1, 2)(3, 4)(5, 6)(7, 8)

[5]     Order 2       Length 6
        Rep (1, 7)(2, 3)(4, 6)(5, 8)

[6]     Order 2       Length 6
        Rep (3, 6)(4, 5)

[7]     Order 3       Length 8
        Rep (1, 8, 3)(2, 5, 7)

[8]     Order 4       Length 6
        Rep (1, 2, 3, 4)(5, 6, 7, 8)

[9]     Order 4       Length 6
        Rep (1, 3, 8, 6)(2, 4, 7, 5)

[10]    Order 6       Length 8
        Rep (1, 2, 3, 7, 8, 5)(4, 6)
```

To illustrate how one may loop through the classes sequence, the *class equation*, which states that the order of a group G equals $\sum[G : C_G(x)]$, where x ranges over the class representatives, will be verified for *cube*:

```
> print Order(cube);
48
> print &+[Index(cube, Centralizer(cube, cl[3])) :
>   cl in cubecl];
48
```

The determination of classes is known to be a hard problem. Several algorithms are available in MAGMA. A particular algorithm may be selected by means of an **Al** parameter on the **Classes** function. The value `"Action"` will invoke an algorithm that finds the classes by computing the orbits of the elements of G under conjugation by G. Consequently, it is only useful for small groups but in some cases it will be faster than the alternative methods. The value `"Random"` specifies a random algorithm which attempts to locate an element in each conjugacy class. Finally, the **Al** value `"Extension"` employs an inductive algorithm that computes classes in a sequence of successively larger quotients of G determined by an elementary abelian series for G.

Finally, if the user already knows a set of representatives for the classes, the **AssertAttribute** command may be used to define a value for the `"Classes"` attribute for the group, by giving the classes in the form of a sequence containing exactly one representative from each conjugacy class,

29.10.4 The Class Map and Power Map

For certain applications (such as constructing a class matrix) it is necessary to identify the class to which each member of a large set of elements belongs. Let $R = [x_1, \ldots, x_r]$ be the sequence of class representatives for G constructed by the **Class** function. Let $f : G \to R$ be defined by $f(x) = x_i$ where x belongs to the class having representative x_i. The mapping f is called the *class map* of G and may be created by the function **ClassMap**(G). The function precomputes certain information in addition to the class representatives, which may greatly speed up the identification of the class representative for x. Note that if the classes are not known at the time **ClassMap**(G) is invoked, they will be first computed.

A function related to **ClassMap** is **ClassRepresentative**(G, x). It returns the class representative of x, that is, the representative for the class containing x that appears in the sequence of tuples returned by **Classes**.

For example, the MAGMA code below calculates that b lies in the class numbered seven by the **Class** function, and verifies that the class representative of b is the representative for the seventh class:

```
> cmcube := ClassMap(cube);
> print cmcube;
Mapping from: GrpPerm: cube to { 1 .. 10 }
> print cmcube(b);
7
> print cubecl[7, 3] eq ClassRepresentative(cube, b);
true
```

The mapping f returned by **ClassMap** may be applied not only to individual elements of G but also to sets and sequences of elements of G:

```
> print cmcube([b, c, b*c]);
[ 7, 4, 10 ]
```

Suppose the conjugacy classes of G are denoted $K_1, \ldots, K_r$. If t is a fixed positive integer, and K_i is a class of G, there exists a unique class K_j such that $\{x^t : x \in K_i\} \subseteq K_j$. The behaviour of the classes of G under the taking of powers may be represented by a mapping known as the *powermap* ϕ. The powermap may be defined formally as follows: Let $C = \{1, \ldots, r\}$ denote the set of indices of the classes and suppose the exponent of G is e. The power map ϕ for G is the mapping $\phi : C \times \{1, \ldots, r\} \to C$ where $\phi(i, j)$ is the index of the class containing x_i^j (x_i is a representative for K_i). This map is created by the function **PowerMap**(G). If the class map is not already known, **PowerMap** will cause it to be created.

The use of the power map is illustrated by displaying how the classes of *cube* are mapped under the taking of squares, cubes and fouth powers.

```
> phi := PowerMap(cube);
> print [ phi(i, 2) : i in [1..#cubecl] ];
[ 1, 1, 1, 1, 1, 1, 7, 3, 3, 7 ]
> print [ phi(i, 3) : i in [1..#cubecl] ];
[ 1, 2, 3, 4, 5, 6, 1, 8, 9, 2 ]
> print [ phi(i, 4) : i in [1..#cubecl] ];
[ 1, 1, 1, 1, 1, 1, 7, 1, 1, 7 ]
```

Thus, inspecting the cubes, it can be seen that the cubes of the elements of class 10 fall into class 2.

For each class K_i of G define $\mathcal{K}_i = \sum x$, where x runs through the elements of K_i. It is a standard result from representation theory that the formal sums

$\mathcal{K}_i$, $i = 1, \ldots, r$ form a basis for the centre of the group algebra $\mathbb{C}[G]$. Thus, the product of any two class sums can be written as follows:

$$\mathcal{K}_i \mathcal{K}_j = \sum c_{ijk} \mathcal{K}_k$$

where the sum is over k. The *structure constants* c_{ijk} correspond to the number of solutions (x, y) of the equation $xy = z$, where $z \in \mathcal{K}_k$ is fixed and $x \in \mathcal{K}_i$, $y \in \mathcal{K}_j$. The matrix M_i, whose (j, k)th entry is c_{ijk}, is called the structure constant matrix for class i. It may be computed using the function **ClassMatrix**(G, i) as shown in the following example in which the class matrix for *cube* for the unique conjugacy class of elements of order 3 (class 7) is computed.

```
> print ClassMatrix(cube, 7);
[0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 3 0 0 0]
[0 0 0 0 0 0 0 0 0 3]
[0 0 0 0 4 0 0 4 0 0]
[0 0 0 0 0 4 0 0 4 0]
[8 0 8 0 0 0 4 0 0 0]
[0 0 0 0 4 0 0 4 0 0]
[0 0 0 0 0 4 0 0 4 0]
[0 8 0 8 0 0 0 0 0 4]
```

The matrix may be interpreted as follows: Consider row 6 of the matrix. If the elements of class 7 are multiplied by an element from class 6, then four of the products belong to class 6 and four belong to class 9 as is seen from the calculation below (*phi* is the power map defined above).

```
> x := cubecl[6][3];
> [ phi(x*y) : y in Class(cube, cubecl[7][3]) ];
[ 9, 6, 6, 9, 6, 9, 6, 9 ]
```

29.11 Conjugacy Classes of Subgroups

29.11.1 Determining the Conjugacy Classes

Knowledge of the conjugacy classes of subgroups of a group G provides a detailed picture of the internal structure of G. Because of the huge number of subgroups that a complicated group may possess, it is only practical to find all conjugacy classes of subgroups in groups that either have moderate order

or relatively few classes of subgroups. The algorithm employed by MAGMA for permutation or matrix groups is restricted to groups whose radical quotient (i.e., quotient by the maximal normal soluble subgroup) is less than some bound (20,160 at the time of writing, but to be increased).

The basic function, **SubgroupClasses**(G), determines a representative subgroup for each conjugacy class. The subgroups are returned as a sequence of records where the i^{th} record contains:

1. A representative subgroup H for the i^{th} conjugacy class (field name: `subgroup`);
2. The order of the subgroup (field name: `order`);
3. The number of subgroups in the class (field name: `length`);
4. (Optional, and only relevant for permutation groups) A presentation for H (field name `presentation`).

The function is illustrated by computing the subgroups of Alt(5):

```
> G := Alt(5);
> S := SubgroupClasses(G);
> print S;
Conjugacy classes of subgroups
------------------------------

[ 1]    Order 1            Length 1
        Permutation group acting on a set of cardinality 5
            Id($)
[ 2]    Order 2            Length 15
        Permutation group acting on a set of cardinality 5
            (1, 4)(3, 5)
[ 3]    Order 3            Length 10
        Permutation group acting on a set of cardinality 5
            (1, 3, 2)
[ 4]    Order 5            Length 6
        Permutation group acting on a set of cardinality 5
            (1, 4, 3, 5, 2)
[ 5]    Order 4            Length 5
        Permutation group acting on a set of cardinality 5
            (1, 4)(3, 5)
            (1, 5)(3, 4)
[ 6]    Order 6            Length 10
        Permutation group acting on a set of cardinality 5
            (1, 3, 2)
            (2, 3)(4, 5)
[ 7]    Order 10           Length 6
        Permutation group acting on a set of cardinality 5
            (1, 4, 3, 5, 2)
            (1, 4)(2, 3)
[ 8]    Order 12           Length 5
```

```
        Permutation group acting on a set of cardinality 5
            (1, 4)(3, 5)
            (3, 4, 5)
[ 9]    Order 60           Length 1
        Permutation group acting on a set of cardinality 5
            (1, 4)(3, 5)
            (1, 3, 2)
```

It can be seen that Alt(5) has 9 conjugacy classes of subgroups. Class 6 consists of a class containing 10 subgroups of order 6. The chosen representative for this class is the subgroup generated by the permutations $(1, 3, 2)$ and $(2, 3)(4, 5)$.

Since the function returns a sequence, it is an easy matter to iterate over the subgroup classes. This is illustrated by determining the orders of the non-abelian simple subgroups of the Mathieu group M_{12}:

```
> m12 := PermutationGroup< 12 |
>          (1, 11, 5, 2, 8, 3, 9, 7, 4, 10, 12),
>          (1, 8, 12, 9, 6)(2, 4, 10, 11, 3) >;
> print Order(12);
95040
> S := SubgroupClasses(G);
> print [ <i, Order(sub)> : i in [1..#S] |
>              not IsAbelian(sub) and IsSimple(sub)
>              where sub := S[i]`subgroup ];
[ <90, 60>, <91, 60>, <92, 60>, <93, 60>, <119, 660>,
  <120, 660>, <131, 360>, <132, 360>, <145, 7920>,
  <146, 7920>, <147, 95040> ]
```

Table 29.13. Restriction parameters for subgroup functions

Parameter	Meaning
OrderEqual	If assigned integer n, only construct subgroups of order n
OrderDividing	If assigned integer n, only construct subgroups having order dividing n
IsNormal	If assigned **true**, only construct normal subgroups
IsRegular	If assigned **true**, only construct regular subgroups (for permutation groups only)

Additional restrictions on the subgroups constructed may be imposed by means of the parameters shown in Table 29.13. These parameters apply to all of the subgroup functions listed above, including **SubgroupClasses**.

The central algorithm for computing the subgroup classes allows various restrictions to be imposed on the kind of subgroups that are constructed. Different versions of the subgroup function allow the user to construct specific subgroup varieties of subgroups. These varieties are displayed in Table 29.14.

Table 29.14. Varieties of subgroups

ElementaryAbelianSubgroups(G)	**AbelianSubgroups**(G)
NilpotentSubgroups(G)	**SolubleSubgroups**(G)
PerfectSubgroups(G)	**SimpleSubgroups**(G)
RegularSubgroups(G)	

These functions will be used to determine the conjugacy classes of elementary abelian subgroups of order 16 in the group $Z/2Z \wr \mathrm{Alt}(5)$:

```
> G := WreathProduct(CyclicGroup(GrpPerm, 2), Alt(5));
> #G;
1920
> S := ElementaryAbelianSubgroups(G: OrderEqual := 16);
> print #S;
7
```

29.11.2 The Poset of Subgroup Classes

In addition to finding representatives for conjugacy classes of subgroups, MAGMA allows the user to create the poset L of subgroup classes. The elements of the poset correspond to the conjugacy classes of subgroups. Two lattice elements a and b are joined by an edge if either some subgroup of the conjugacy class a is a maximal subgroup of some subgroup of conjugacy class b or vice-versa. The elements of L are called *subgroup-poset elements* and are numbered from 1 to n, where n is the cardinality of L. Various functions allow the user to identify maximal subgroups, normalizers, centralizers etc in the lattice. Given $e \in L$, it is possible to create the subgroup H of G corresponding to e, and to create the element of L corresponding to a subgroup of G.

If a large number of poset operations are to be performed, it may be more efficient to construct the complete subgroup class poset first.

The function **SubgroupLattice**(G) determines the conjugacy classes of subgroups and the builds the poset data structure. The construction of the poset can be very time-consuming and should only be used with care. The function has the following Boolean-valued parameters: **Properties** – if **true**,

record the abstract properties of each subgroup as it is stored in the poset; **Centralizers** – if **true**, record the class in which the centralizer of each poset element lies; **Normalizers** – if **true**, record the class in which the normalizer of each poset element lies. The default value of each of these parameters is **false**.

The **SubgroupLattice** function will be applied to the simple group PSL(2, 8) of order 504:

```
> G := PSL(2, 8);
> L := SubgroupLattice(G: Properties := true,
       Centralizers := true, Normalizers := true);
> print L;

Partially ordered set of subgroup classes
-----------------------------------------

[ 1]  Order 1    Length 1              C = [12]  N = [12]
      Maximal Subgroups:
---
[ 2]  Order 2    Length 63  Cyclic     C = [ 9]  N = [ 9]
      Maximal Subgroups: 1
[ 3]  Order 3    Length 28  Cyclic     C = [ 7]  N = [10]
      Maximal Subgroups: 1
[ 4]  Order 7    Length 36  Cyclic     C = [ 4]  N = [ 8]
      Maximal Subgroups: 1
---
[ 5]  Order 4    Length 63  Elem Abel  C = [ 9]  N = [ 9]
      Maximal Subgroups: 2
[ 6]  Order 6    Length 84  Soluble    C = [ 1]  N = [ 6]
      Maximal Subgroups: 2 3
[ 7]  Order 9    Length 28  Cyclic     C = [ 7]  N = [10]
      Maximal Subgroups: 3
[ 8]  Order 14   Length 36  Soluble    C = [ 1]  N = [ 8]
      Maximal Subgroups: 2 4
---
[ 9]  Order 8    Length 9   Elem Abel  C = [ 9]  N = [11]
      Maximal Subgroups: 5
[10]  Order 18   Length 28  Soluble    C = [ 1]  N = [10]
      Maximal Subgroups: 6 7
---
[11]  Order 56   Length 9   Soluble    C = [ 1]  N = [11]
      Maximal Subgroups: 4 9
---
[12]  Order 504  Length 1   Simple     C = [ 1]  N = [12]
```

```
Maximal Subgroups: 8 10 11
```

From the poset L of subgroup classes of $\mathrm{PSL}(2,8)$, it can be seen, for example, that class 7 contains 28 subgroups of order 9. These are cyclic, their normalizers lie in class 11, and their maximal subgroups belong to class 3.

The term *class* will be used in the remainder of this section when referring to poset elements. If G has r conjugacy classes of subgroups, the poset elements (classes) are indexed by the integers 1 to r. Indeed, a poset element is represented by this integer for input and output purposes. So if L is such a poset, the coercion $L\,!\,i$ will denote the ith class of the poset. Operations on subgroup posets are shown in Table 29.15.

Table 29.15. General subgroup poset operations

MAGMA	Meaning
$\#L$	Number of conjugacy classes of subgroups
$L\,!\,i$	i^{th} element (class) of the poset
$Z\,!\,i$	Integer corresponding to poset e, where Z is integer ring
$L\,!\,H$	Element of the poset containing subgroup H
Bottom(L)	Trivial subgroup class
Top(L)	Class corresponding to G

The standard relational operations for poset elements are denoted by **eq**, **ne**, **ge**, **gt**, **le**, and **lt**. For instance, the value of e **ge** f is **true** if the class e contains class f.

Table 29.16. Extracting information from a subgroup poset

MAGMA	Meaning
Group(e)	Representative for class e
Centralizer$e, f)$	Class containing centralizers of f in e
Normalizer(e, f)	Class containing normalizers of f in e
Length(e)	Number of subgroups in class e
Order(e)	Order of subgroups in class e
MaximalSubgroups(e)	Classes containing maximal subgroups of e
MinimalOvergroups(e, f)	Classes containing minimal overgroups of e
NumberOfInclusions(e, f)	Number of subgroups of the class e lying in a fixed representative of the class f

The following code constructs a chain from the bottom to the top of the subgroup classes poset for PSL$(2, 8)$ displayed above and then prints the indices of successive terms of the chain:

```
> H := Bottom(L);
> C := [H];
> while H ne Top(L) do
>     H := Representative(MinimalOvergroups(H));
>     C := Append(C, H);
> end while;
> print C;
[ 1, 2, 5, 9, 11, 12 ]
> print [ Order(C[i+1]) div Order(C[i]) : i in [1..#C-1]];
[ 2, 2, 2, 7, 9 ]
```

The Burnside matrix of a subgroup lattice displays information about the containment relationship between all pairs of conjugacy classes of subgroups. This matrix is of some interest to combinatorial theorists. In what follows it will be assumed that L is a subgroup poset for the group G. Let $S_1, \ldots, S_n$ denote the conjugacy classes of subgroups corresponding to the elements of L, where S_1 is the trivial subgroup. For each class S_i, where $i = 1, \ldots, n$, let s_i be the number of subgroups in S_i and let r_i denote the representative subgroup for the class S_i. For $i, j \in \{1, \ldots, n\}$, define b_{ij} to be the number of subgroups in class S_i which either contain r_j or are contained in r_j. The matrix (b_{ij}) will be termed the *Burnside matrix* of the subgroup poset L and it may be displayed using the function **PrintBurnsideMatrix**(L). The example below displays the Burnside matrix for PSL$(2, 8)$:

```
> PrintBurnsideMatrix(L);

CL ORD  1 2 3 4 5 6 7 8 9 10 11 12

 1   1  1 1 1 1 1 1 1 1 1  1  1  1
 2   2 63 1 . . 3 3 . 7 7  9  7 63
 3   3 28 . 1 . . 1 1 . .  1  . 28
 4   7 36 . . 1 . . . 1 .  .  8 36
 5   4 63 3 . . 1 . . . 7  .  7 63
 6   6 84 4 3 . . 1 . . .  3  . 84
 7   9 28 . 1 . . . 1 . .  1  . 28
 8  14 36 4 . 1 . . . 1 .  .  . 36
 9   8  9 1 . . 1 . . . 1  .  1  9
10  18 28 4 1 . . 1 1 . .  1  . 28
11  56  9 1 . 2 1 . . . 1  .  1  9
12 504  1 1 1 1 1 1 1 1 1  1  1  1
```

29.11.3 Normal Subgroups

Current techniques for finding all normal subgroups are restricted in their application to groups of moderate order. The function **NormalSubgroups**(G) returns the normal subgroups of G as a sequence of records having the same format as the subgroup records returned by **SubgroupClasses**. For instance:

```
> G := WreathProduct(Sym(3), Sym(5));
> print #G;
933120
> N := NormalSubgroups(G);
> print [ r`order : r in N ];
[ 1, 243, 486, 3888, 7776, 233280, 466560, 466560,
  466560, 933120 ]
```

It can be seen that G has 8 proper normal subgroups.

In the case of permutation groups and matrix groups, the function **MinimalNormalSubgroups**(G) finds the minimal normal subgroups of G. When G is a primitive permutation group, this function is quite efficient, but for other groups, a naive algorithm is employed.

29.12 Characters

29.12.1 Class Functions and Characters

Let K be a field and G be a group. A *representation* of G is a homomorphism $\psi : G \to \mathrm{GL}(n, K)$, for some positive integer n. The *character* of G afforded by the representation ψ is the mapping $\chi : G \to K$ defined by $\chi : g \to \mathrm{Tr}\psi(g)$. An *irreducible character* is afforded by an irreducible representation.

In this section, K will be taken to be the complex field $\mathbb{C}$. Over $\mathbb{C}$, a finite group possesses a finite number k of irreducible representations, where k is the number of conjugacy classes of G.

A *class function* on a group G is a mapping $\phi : G \to \mathbb{C}$. The set of class functions on G forms a vector space W over $\mathbb{C}$ having dimension k, where the irreducible characters for G constitute a basis. The standard inner product of characters makes W an inner product space. The set of distinct irreducible characters over $\mathbb{C}$ of G will be written as $\mathrm{Irr}(G)$.

For both theoretical and computational purposes, it is convenient to work in the space W of class functions. The function **ClassFunctionSpace**(G) creates the space of class functions for G as a k-dimensional vector space

over some cyclotomic field (the particular field depends upon G). Note that this function will create the conjugacy classes of G but not the irreducible characters.

The MAGMA character theory machinery provides tools for computing character tables, both automatically and interactively, and facilities for performing a range of standard operations on characters.

29.12.2 Creating Individual Characters

Class functions may be defined using the usual constructors. If $(a_1, \ldots, a_k)$ denotes a class function a for G taking the value a_i on the i^{th} class, it may be created either by the expression

elt< W | $a_1, \ldots, a_k$ **>**

or by the expression

W **!** Q

where $Q = [a_1, \ldots, a_k]$. Unless the irreducible characters of G are known, it is not possible to determine whether a is a character (or generalized character). A Boolean-valued parameter, **Character**, on the **elt**-constructor may be used to flag a as a character:

elt< W | $a_1, \ldots, a_k$ **: Character := true >**

The following code defines a number of characters for the group Alt(5):

```
> G := Alt(5);
> cls := Classes(G);
> R := CharacterRing(G);
> x := R ! [ 6, -2, 0, 1, 1 ];
> y := R ! [ 13, 1, 1, -2, -2 ];
> Q5<w> := CyclotomicField(5);
> z := R ! [ 3, -1, 0, w^3 + w^2 + 1, -w^3 - w^2 ];
> t := R ! [ 4, 0, 1, -1, -1];
```

29.12.3 Operations on Characters

The standard vector operations (addition, subtraction, scalar multiplication) apply to class functions. A componentwise product $\chi\psi$ of class functions χ and ψ is defined. When χ and ψ are characters, $\chi * \psi$ is the character afforded

by the tensor product of the representations affording χ and ψ. Similarly, χ^i denotes the product $\chi * \chi * \ldots * \chi$ (i factors). Other elementary operations are summarized in Table 29.17. In this table, x and y denote class functions.

Table 29.17. Elementary character operations

MAGMA	Meaning
InnerProduct(x, y)	Inner product of class functions x and y
Norm(x)	Inner product of class function x with itself
Degree(x)	Value of class function x on identity of G
Schur(x, k)	Generalized Frobenius-Schur indicator (see *Handbook*)

Continuing with the above example, the arithmetic operations are applied to identify irreducible characters and to determine if a given character occurs as a summand in another character. Recall that a character has norm 1 if and only if it is irreducible.

```
> print Norm(x), Norm(y), Norm(z), Norm(t);
2 5 1 1
```

Therefore z and t are irreducible characters. Next the **InnerProduct** function is used to determine if either of z and t occur in x or y:

```
> print InnerProduct(z,x), InnerProduct(t,x);
1 0
```

So z is a summand of x. It may be removed using the operation of character difference:

```
> u := x - z;
> print u;
( 3, -1, 0, -w^3 - w^2, w^3 + w^2 + 1 )
> print Norm(u);
1
```

The character u is irreducible. Now the process is repeated with the non-irreducible character y:

```
> print InnerProduct(z,y), InnerProduct(t,y);
0 2
// Two copies of t occur in y.
> v := y - 2*t;
> print v;
```

```
( 5, 1, -1, 0, 0 )
> print Norm(v);
1
```

The value of the class function x on class i is obtained by indexing x in the usual way. Since a class function is also a mapping, the expression $x(g)$ (or g`@`x) will give the value of x on element g:

```
> print x(G!(2,3,4));
0
> print x(G!(1,2)(3,4));
-2
> print (G!(1,2,3,4,5)) @ x;
1
```

29.12.4 Properties of Characters

Table 29.18 contains a self-explanatory list of functions that test a class function for various properties. Note that the functions **IsCharacter** and **IsGeneralizedCharacter** require the irreducible characters of G and therefore will cause the character table to be created if it is not already known.

Table 29.18. Properties of characters

MAGMA	Meaning
IsCharacter(x)	**true** if x is a character
IsConjugate(x, y)	**true** if x and y are Galois conjugates
IsGeneralizedCharacter(x)	**true** if character x is generalized
IsFaithful(x)	**true** if x is a character of a faithful representation
IsIrreducible(x)	**true** if character x is irreducible
IsLinear(x)	**true** if character x is linear
IsOne(x)	**true** if x is the principal character
IsReal(x)	**true** if value of x on every class is real

29.12.5 Automatic Determination of Irreducible Characters

The table of characters of G may be constructed automatically using the Dixon-Schneider algorithm which is invoked by **CharacterTable**(G). The characters are returned in the form of an enumerated sequence T which has

attached to it a special printing routine. Thus, whenever the sequence T is printed, the characters are displayed in the traditional manner.

For example:

```
> G := PSL(2, 7);
> X := CharacterTable(G);
> print X;

Character Table of Group g
--------------------------

------------------------------
Class |   1  2  3  4    5    6
Size  |   1 21 56 42   24   24
Order |   1  2  3  4    7    7
------------------------------
p  =  2   1  1  3  2    5    6
p  =  3   1  2  1  4    6    5
p  =  7   1  2  3  4    1    1
------------------------------
X.1   +   1  1  1  1    1    1
X.2   0   3 -1  0  1   Z1 Z1#3
X.3   0   3 -1  0  1 Z1#3   Z1
X.4   +   6  2  0  0   -1   -1
X.5   +   7 -1  1 -1    0    0
X.6   +   8  0 -1  0    1    1


Explanation of Symbols:
-----------------------

# denotes algebraic conjugation, that is,
#k indicates replacing the root of unity w by w^k

Z1     = zeta_7^4 + zeta_7^2 + zeta_7
```

The character table is printed in three sections separated by broken lines. The first section contains a description of the conjugacy class including the number of elements (*size*) and the order of the elements (*order*) in a given class. The second section contains the class maps for all primes dividing the group order. Thus, for example, it can be seen that taking third powers swaps classes 5 and 6. The third section contains the actual characters. The second column of this section contains the Frobenius-Schur indicators. The symbols are interpreted as follows: A + indicates that the character is real; a - indicates

that while the character is real, it is not afforded by a real representation; and, finally, a `0` indicates a character that is not real. Character values that are complex are given as elements of some cyclotomic number field. In the above example, the complex values are written in terms of a primitive 7^{th} root of unity ζ_7. The value of `Z1` is $\zeta_7^4 + \zeta_7^2 + \zeta_7$, while `Z1#3` has the value $\zeta_7^6 + \zeta_7^5 + \zeta_7^3$ (i.e., the conjugate of `Z1` obtained by taking the cube of ζ_7).

As a second example, consider the group of order 27 defined by the presentation $\langle s, t \mid t^3 = 1, t^{-1}st = s^{-2} \rangle$. The group is given initially in the category **GrpFP**:

```
> G<s, t> := Group< s, t | t^3 = 1, s^t = s^-2 >;
> print G;
Finitely presented group G on 2 generators
Relations
    t^3 = Id(G)
    s^t = s^-2
> print Order(G);
27
```

In order to find the character table of G, it is necessary to construct a group P isomorphic to G in a category that provides SNFs. This is done by using the function **pQuotient** to construct an isomorphic group in the category **GrpPC**:

```
> P := pQuotient(G, 3, 3);
27
> print CharacterTable(P);

Character Table of Group P
--------------------------

------------------------------------------------------------------
Class |   1      2      3    4    5    6    7    8    9   10   11
Size  |   1      1      1    3    3    3    3    3    3    3    3
Order |   1      3      3    3    3    9    9    9    9    9    9
------------------------------------------------------------------
p  =  3   1      1      1    1    1    2    2    2    3    3    3
------------------------------------------------------------------
X.1   +   1      1      1    1    1    1    1    1    1    1    1
X.2   0   1      1      1    J -1-J    1    J -1-J    1    J -1-J
X.3   0   1      1      1 -1-J    J    1 -1-J    J    1 -1-J    J
X.4   0   1      1      1    1    1    J    J    J -1-J -1-J -1-J
X.5   0   1      1      1    J -1-J    J -1-J    1 -1-J    1    J
X.6   0   1      1      1 -1-J    J    J    1 -1-J -1-J    J    1
X.7   0   1      1      1    1    1 -1-J -1-J -1-J    J    J    J
X.8   0   1      1      1    J -1-J -1-J    1    J    J -1-J    1
X.9   0   1      1      1 -1-J    J -1-J    J    1    J    1 -1-J
X.10  0   3 -3-3*J    3*J    0    0    0    0    0    0    0    0
```

```
X.11  0   3    3*J -3-3*J    0    0    0    0    0    0    0    0

Explanation of Symbols:
-----------------------

J = RootOfUnity(3)
```

29.12.6 Standard Constructions for Characters

For situations where it is too expensive to compute $\mathrm{Irr}(G)$ automatically, or where only certain characters are of interest, MAGMA provides standard constructions which allow characters to be generated interactively. For example, the character tables of many of the sporadic simple groups have been constructed by writing down some characters, generating new ones using the constructions to be described in this section and then attempting to extract new irreducibles from these characters.

The function **PrincipalCharacter**(G) defines the principal character of G while **LinearCharacters**(G) returns the linear characters of G. If H is a subgroup of G, **PermutationCharacter**(G, H) constructs the character corresponding to the permutation representation afforded by the action of G on the cosets of H. Finally, if G is a permutation group, **PermutationGroup**(G) constructs the character corresponding to the defining permutation representation of G.

Suppose that H is a subgroup of G. If x is a class function on H, **Induction**(x, G) returns the class function on G induced from x. If y is a class function on G, **Restriction**(y, H) returns the class function on H obtained by restricting y to H. Let Q be a quotient of G with natural homomorphism $f : G \to Q$ and suppose x is a class function on Q. Then the function **LiftCharacter**(x, f, G) lifts x to G. Note that irreducible characters of G/N correspond to irreducible characters of G that have N in their kernel.

The following example uses these operations to find the characters of $G = \mathrm{Sym}(4)$. The main operation used is to lift the characters of the quotient of G by the Klein 4-group, thereby yielding three irreducible characters of G.

```
> G := Sym(4);
> print #Classes(G);
5
> x1 := PrincipalCharacter(G);
> print x1;
( 1, 1, 1, 1, 1 )
> Q, f := quo< G | (1,2)(3,4), (1,3)(2,4) >;
```

```
> X := [ Extension(x, f, G) : x in CharacterTable(Q) ];
> print X;
[ ( 1, 1, 1, 1, 1 ), ( 1, 1, -1, 1, -1 ),
  ( 2, 2, 0, -1, 0 ) ]
```

At this point, two characters have still to be found. The permutation character yields one of them:

```
> x4 := PermutationCharacter(G) - x1;
> x4;
( 3, -1, 1, 0, -1 )
> print IsIrreducible(x4);
true
> Append(~X, x4);
> print 24 - &+[ Degree(x)^2 : x in X ];
9
```

So the one remaining character has degree 3. It may therefore be obtained by taking the tensor product of $X[2]$ with $X[4]$:

```
> x5 := X[2]*X[4];
( 3, -1, -1, 0, 1 )
> print IsIrreducible(x5);
true
> Append(~X, x5);
> print X;
[ ( 1, 1, 1, 1, 1 ), ( 1, 1, -1, 1, -1 ),
  ( 2, 2, 0, -1, 0 ), ( 3, -1, 1, 0, -1 ),
  ( 3, -1, -1, 0, 1 ) ]
```

Let $\mathbb{Q}(x)$ denote the cyclotomic field generated by the rational field $\mathbb{Q}$ and the values of the character x. The set of Galois conjugates of x under the action of the Galois group $\mathrm{Gal}(\mathbb{Q}(x)/\mathbb{Q})$ are all characters of G. These conjugates may be constructed using either **GaloisConjugate**(x, j), which computes the conjugate of x under the action of the element of the Galois group determined by the integer j (where j must be coprime to the exponent of G), or **GaloisOrbit**(x), which returns a sequence containing all of the Galois conjugates of x.

29.12.7 Decomposition of Tensor Powers

An important theorem due to Burnside states that if χ is a faithful character for G such that $\chi(g)$ takes precisely t distinct values as g runs through

the elements of G, then every irreducible character of G occurs as a constituent of one of the powers $\chi^0, \chi^1, \ldots, \chi^{t-1}$. The practical application of this result is limited by the difficulty of decomposing χ^l into irreducibles. A partial decomposition of χ^l may be obtained through the technique of *symmetrization*. Briefly, each irreducible character of $\mathrm{Sym}(l)$ determines a character summand of χ^l. For example, since $\mathrm{Sym}(2)$ has two classes and hence two irreducible characters, it is possible to write down characters χ_S and χ_A such that $\chi^2 = \chi_S + \chi_A$. The hope is that, given this initial decomposition of χ^2 into characters having smaller norm, there is a better chance of extracting new irreducibles. If χ is a character afforded by an orthogonal representation, then formulae due to Weyl use characters of the real orthogonal group to obtain a finer decomposition of powers χ^l than that provided by symmetrization. Similar formulae exist if χ is afforded by a symplectic representation.

Table 29.19 summarizes the functions implementing these formulae. In this table, x denotes a character; if x is orthogonal, then its Schur indicator is $+1$, and if x is symplectic, then its Schur indicator is -1. Note that these functions only apply to powers χ^n, $2 \leq n \leq 6$. This is not a real restriction since the technique is almost never useful for larger values of n.

Table 29.19. Tensor components of characters

Magma	Meaning
OrthogonalComponent(x, p)	Given an orthogonal character x and a partition p of n ($2 \leq n \leq 6$), return orthogonal component of x^n corresponding to p (result may be a generalized character)
OrthogonalComponents(x, n)	Given an orthogonal character x and n such that $2 \leq n \leq 6$, return orthogonal components of x^n (result may include generalized characters)
SymmetricComponent(x, p)	Symmetric component of n^{th} power of character x, determined by partition p of n ($2 \leq n \leq 6$)
Symmetrizations(x, n)	Symmetrizations of character of n^{th} power of character x ($2 \leq n \leq 6$)
SymplecticComponent(x, p)	Given symplectic character x and a partition p of n ($2 \leq n \leq 6$), return symplectic component of x^n corresponding to p (result may be a generalized character)
SymplecticComponents(x, n)	Given a symplectic character x and n such that $2 \leq n \leq 6$, return symplectic components of x^n (result may include generalized characters)

To illustrate these operations, consider the unitary group $U(3,3)$. Starting with the character $(6,-2,-3,0,-2,-2,2,1,-1,-1,0,0,1,1)$, its Frobenius indicator is seen to be $+1$, so its tensor powers may be decomposed using **SymplecticComponents**:

```
> G := PSU(3, 3);
> CF := CharacterRing(G);
> x := CF![6,-2,-3, 0, -2,-2, 2, 1, -1, -1, 0, 0, 1, 1];
> Schur(x6, 2);
-1
> S := SymplecticComponents(x, 2);
[ ( 14, -2, 5, -1, 2, 2, 2, 1, 0, 0, 0, 0, -1, -1 ),
  ( 21, 5, 3, 0, 1, 1, 1, -1, 0, 0, -1, -1, 1, 1 ) ]
> Norm(S[1]), Norm(S[2]);
1 1
```

In this example, the function **SymplecticComponents** produces a splitting of x^2 into two irreducible characters.

29.12.8 Interactive Determination of Irreducible Characters

Once the user has produced a character y (using one of the techniques explained above), the next step is to remove all known irreducibles from y. The function **Decomposition**(T, y) takes a sequence T of m characters and some character y and attempts to express y as a linear combination of the elements of T. Usually T is a sequence of irreducible characters. **Decomposition** returns a sequence $Q = [q_1, \ldots, q_m]$ of cyclotomic field elements and a character z such that

$$y = q_1 T[1] + \ldots + q_m T[m] + z$$

If y involves only irreducibles in T (i.e., if T contains all irreducibles) then q gives the coordinates of y with respect to the basis $T[1], \ldots, T[m]$. If y is not expressible in terms of the characters $T[i]$, then z is the residue of y after the characters of T have been removed.

29.13 Transferring Between Group Categories

In certain circumstances, given a group G, it may be possible to construct an isomorphic group belonging to a different group category. In particular, this provides the possibility of transferring G to a category in which computation is much more efficient.

29.13.1 Transferring to the Polycyclic Group Category

Suppose G is an fp-group, p is a prime and c is some positive integer. Then **pQuotient**(G, p, c) returns two values: the class c p-quotient P of G; and the natural homomorphism $\phi : G \to P$. Thus, if G is a finite p-group having class less than c, this function will return a polycyclic group P isomorphic to G. This technique allows G to be investigated using the very powerful tools provided for computing structural information for polycyclic groups.

If G is a finite soluble permutation or matrix group, then **PCGroup**(G) also returns two values: a polycyclic group H isomorphic to G; and an isomorphism $\phi : G \to H$.

29.13.2 Transferring to the Permutation Group Category

Let G belong to any group category and suppose H is a subgroup of G (of finite index). Then the function **CosetAction**(G, H) returns the homomorphism $\phi : G \to T$ defined by the action of G on the space of (right) cosets. In addition, it returns T and, where possible, the kernel of ϕ. If H is a core-free subgroup of G (i.e., H does not contain any proper normal subgroups of G), then the homomorphism ϕ will be faithful. This function may sometimes be used to construct a faithful permutation representation of a finite fp-group G. In general, it will be difficult to locate a non-trivial core-free subgroup H of G unless a good deal is known about G. Taking H to be the trivial subgroup ensures that it is core-free, but the application of **CosetAction** will then only be practical if G has moderate order (100 000 at most).

For instance, consider the group $G55$ defined by the presentation

$$\langle a, b, c \mid a^5, b^5, c^5, (ab)^2, (bc)^2, (ca)^2, (abc)^2 \rangle$$

The following lines define this group in MAGMA, and compute its order:

```
> G55<a, b, c> := Group< a, b, c | a^5, b^5, c^5, (a * b)^2,
>          (b*c)^2, (c*a)^2, (a*b*c)^2 >;
> print Order(G55);
660
```

This computation shows that the group is finite of reasonable order. Hence it is practical to transfer it to the permutation group category and try to identify it. Since the homomorphism is not required, it is sufficient to use the variant **CosetImage**, which returns the image group T only:

```
> T := CosetImage(G55, sub< G55 | >);
> print Order(T);
```

```
660
> print CompositionFactors(T);
    G
    |  A(1, 11)               = L(2, 11)
    1
```

Therefore $G55$ is $\mathrm{PSL}(2, 11)$.

30. Finitely-Presented Groups

A well-known theorem due to Novikov and Boone states that, in general, the word problem for finitely-presented groups (fp-groups) is undecidable ([Boo59], [Rot73], Chap. 12). This, in turn, implies that a normal form for the elements of an fp-group is usually unavailable. Consequently, algorithms for fp-groups are fundamentally different in nature to algorithms for groups given in some *concrete* form (e.g., permutation groups or groups of matrices over finite fields).

Typical of the elementary questions mathematicians wish to answer about an fp-group G are the following:

– Is G finite? Is G trivial?

– If G is finite, what is its order and structure?

– What are the abelian (nilpotent, soluble, perfect) quotients of G?

– Is G abelian (nilpotent, soluble, perfect)?

– Can a permutation or matrix representation for G be constructed?

– Is G isomorphic to another given fp-group?

Answering such questions is much more of an art than a science. Indeed, finding the answers is seldom possible through application of a single 'push-button' function; rather, several of the available tools have to be brought to bear on the problem. It should be stated here that in order to determine detailed structural information about a finite fp-group G, the user must explicitly construct a group isomorphic to G in one of the categories of groups for which structure normal forms are available (see Chapter 29).

The algorithms available to compute with fp-groups roughly fall under three headings: quotient group methods; Todd-Coxeter methods, based on coset enumeration; and Knuth-Bendix or term-rewriting methods. This chapter begins by discussing the construction of free groups and (combinatorial) operations on words. Tools for constructing an fp-group from various starting points are next considered. The following two sections cover quotient group

methods and the basic Todd-Coxeter procedure, respectively. The final section describes a technique (strongly related to the Todd-Coxeter procedure) for locating subgroups of small finite index. The reader is referred to the *Handbook* for a discussion of the Knuth-Bendix algorithm and its applications.

The reader is referred to [LyS77] for a presentation of the theory of fp-groups, including standard terminology.

30.1 Free Groups and Words

Let $X = \{x_1, \ldots, x_n\}$ be a finite non-empty set. Let X^{-1} be another set whose elements are in one-to-one correspondence with X; call its elements $x_1^{-1}, \ldots, x_n^{-1}$. A *word* over $X \cup X^{-1}$ is a sequence of r elements, each chosen from $X \cup X^{-1}$ with repetitions allowed. The concatenation of two such words u and v is called the *product* uv. Two words w_1 and w_2 are equivalent if w_1 may be obtained from w_2 through the insertion or deletion of a finite number of terms of the form xx^{-1}, $x \in X \cup X^{-1}$. This equivalence of words induces an equivalence relation on the set of words over $X \cup X^{-1}$. The *free group* $F(X)$ *on* X is the set of equivalence classes of words over $X \cup X^{-1}$ with respect to the product defined above. The elements of X are the *generators* of $F(X)$, while the elements of X^{-1} are the *inverses* of the generators. It is easily shown that if X and Y are two sets of cardinality n, then the free groups on X and Y are isomorphic. Consequently, it is usual to refer to this group as the *free group of rank* n.

30.1.1 Constructing the Free Group

The free group of rank n may be created in MAGMA by means of the function **FreeGroup**(n). Thus, the following line constructs the free group *F3* of rank 3:

```
> F3 := FreeGroup(3);
> print F3;
Finitely presented group F3 on 3 generators (free)
```

The three generators of *F3* may be referred to as *F3*.1, *F3*.2, and *F3*.3. Since these generator names can be cumbersome, it is often desirable to give them alternative names that are shorter and perhaps meaningful to the user, such as a, b, and c, by means of a generator assignment:

```
> F3<a, b, c> := FreeGroup(3);
```

The letters on the left are alternative identifier names for the generators, and may be used wherever the standard names $F3.i$ can be used; MAGMA also uses them as the printnames for the generators.

30.1.2 Computing with Words

The multiplicative identity of a free group G is the *empty word*. It is represented by either of the expressions $G!1$ or $\mathbf{Id}(G)$.

Non-trivial words of a free group G may be formed in two ways. Firstly, they can be expressed as the product of generators and/or inverses of G. For example, the following line creates the element $t = ab^{-1}a^{-3}b^2$ of $F3$:

```
> t := a * b^-1 * a^-3 * b^2;
```

Alternatively, a word may be formed by coercing an integer sequence into G, where the positive integer i denotes the i^{th} generator of G and $-i$ denotes its inverse. This method will be explained for the example t above. The first step is to rewrite t as $ab^{-1}a^{-1}a^{-1}a^{-1}bb$, in which every factor is a generator or an inverse of a generator. The sequence $[1, -2, -1, -1, -1, 2, 2]$ corresponds to this product, factors mapping to sequence entries. Then t can be constructed by coercing this sequence into $F3$:

```
> t := F3![1, -2, -1, -1, -1, 2, 2];
> print t;
a * b^-1 * a^-3 * b^2
```

Given a word $w \in G$, the function $\mathbf{Eltseq}(w)$ returns the sequence as described above. The expression $\#w$ is used to find the *length* of a word w, that is, the length of $\mathbf{Eltseq}(w)$. For example:

```
> print Eltseq(t);
[ 1, -2, -1, -1, -1, 2, 2 ]
> print #t;
7
```

For details of element conjugates and commutators, see Section 29.2.

Words can be compared with the usual relational operators: **eq** or **ne** for equality-testing; and **lt**, **le**, **gt**, or **ge** for testing relative order.

Table 30.1 lists several functions which modify or compare words.

The elements of a free group are ordered first by length and then lexicographically. The lexicographic ordering of words is determined by ordering

Table 30.1. Functions on words

MAGMA	Meaning
Eltseq(u)	Sequence of integers representing word u in terms of its letters
Random(G, m, n)	A random word of G of length between m and n inclusive
Subword(u, f, n)	Subword of word u comprising the n letters commencing at the f^{th} letter of u
Match(u, v, f)	Returns **true** if v is a subword of u starting at position $l \geq f$; if **true**, also returns l
Substitute(u, f, n, v)	Word obtained by replacing with the word v the length–n subword of u starting at position f
RotateWord(u, n)	Word u rotated n letters to the right, or $-n$ letters to the left if n negative
ExponentSum(u, x)	Sum of exponents of generator x in word u
GeneratorNumber(w)	If w is the identity of G, returns 0, else returns i if w begins with $G.i$, or $-i$ if w begins with $(G.i)^{-1}$
Eliminate(u, x, v)	Word obtained by replacing every occurrence of generator x in word u by word v, and every x^{-1} by v^{-1}
Eliminate(U, x, v)	Set of words obtained by replacing every occurrence of generator x in u by word v, and every x^{-1} by v^{-1}, for each u in set of words U

the generators and their inverses as follows: The first generator precedes its inverse, which precedes the second generator, which precedes its inverse, and so on. For example:

```
> print t gt c;
true
> print t lt c*a^6;
true
```

In the examples, t is greater than c, because t is longer than c. However, t is less than ac^6, because at the second place in these equal-length words, where they begin to differ, t's factor b^{-1} is less than the other factor c.

When words of finitely-presented groups are compared using relational operators, it is only the free reductions of the words that are compared. For instance, if the group has a relation $a^2 = b^3$, MAGMA will regard ba^4 as equal to $ba^{-1}a^5$ but not equal to b^7.

30.2 Quotients and Construction of Finitely-Presented Groups

Let F be a free group with generating set X. If w_1 and w_2 are words belonging to F, the equation $w_1 = w_2$ is called a *relation* on F; let R be a finite set of r relations of the form $w_{i_1} = w_{i_2}$ on F. Now let N be the normal closure of the set of words $\{w_{i_1} w_{i_2}^{-1} : i = 1, \ldots, r\}$. Then the quotient group $G = F/N$ is the *finitely-presented group* with generators X and relations R. The subgroup N is called the *relation normal subgroup* for G.

The construction of an fp-group G is effected in two stages. Firstly, the appropriate free group F is defined and the relations are given as words in F. Then the quotient of F by the relation normal subgroup N is formed, using the **quo**-constructor.

30.2.1 Relations

Let G be fp-group. If w_1 and w_2 are two words in the generators of G, the relation $w_1 = w_2$ is represented in MAGMA as w_1=w_2. Note that the equality symbol here is `=`, rather than `:=` which is the symbol for assignment. The set of relations that may be formed from the generators of G, form a magma that is uniquely determined by G. Operationally, a relation may be thought of as a special type of sequence of length 2. Thus, if r is the relation w_1=w_2, then either $r[1]$ or **LHS**(r) will return w_1 while either $r[2]$ or **RHS**(r) will return w_2. The left and right sides of a relation may be redefined using the usual sequence mutation operator. These operations are illustrated in the context of the free group F of rank 2.

```
> F<a, b> := FreeGroup(2);
> r := a*b*a = b*a*b;
> print r;
a * b * a = b * a * b
> print LHS(r);
a * b * a
> r[2] := a;
> print r;
a * b * a = a
```

It should be noted that creating a relation in this manner for a group does not effect its definition. Relations created on the generators of G are normally used to define quotients of G as shown in the next section.

30.2.2 The quo-constructor

As is the case with any group, the **quo**-constructor, when applied to an fp-group G, forms the quotient of G with respect to a normal subgroup. However, because of the central role played by the quotient construction in the theory of fp-groups, the **quo**-constructor has slightly different semantics for the category of fp-groups. Recall that it has the following general form:

quo< group | elements **>**

The elements appearing on the right hand side define a subgroup whose normal closure N yields the desired quotient group G/N. In the case of an fp-group, the constructor is more appropriately viewed as having the form

quo< group | relations **>**

where the *relations*-clause defines the desired relation normal subgroup. That is, the normal generators for any quotient group are always viewed as relations. Before giving details of the syntax for the *relations*-clause, the use of the quotient constructor for fp-groups is illustrated by defining the one-relator group

$$G = \langle a, b \mid a * b * a = b * a * b \rangle$$

.

```
> F<x, y> := FreeGroup(2);
> r := x*y*x = y*x*y;
> G := quo< F | r >;
Finitely presented group Q on 2 generators
Relations
    G.1 * G.2 * G.1 = G.2 * G.1 * G.2
```

If there is no particular need to explicitly define the relation r, it can appear directly in the **quo**-constructor:

```
> G := quo< F | x*y*x = y*x*y >;
```

At the time G is assigned, new names should be assigned to its generators, as they are distinct from the generators X of F, being the images of the elements of X under the natural surjection $\phi : F \to G$. So the complete specification of G goes as follows:

```
> F<x, y> := FreeGroup(2);
> G<a,b> := quo< F | x*y*x = y*x*y >;
> print G;
```

```
Finitely presented group G on 2 generators
Relations
    a * b * a = b * a * b
```

Let G be the group referred to on the left side of the **quo**-constructor. The relations-clause on the right side of the **quo**-constructor must be a list of objects which can be expanded into a (normal) generating set for the desired subgroup N. An element of this list may be a simple relation on G. More generally it may be a *relation list*, which is a list of words separated by `=` signs; MAGMA converts relation lists into several simple relations by equating each word in the relation list with the final word in the list. A element may also be a word w in G, in which case it will be converted into the relation $w = 1$. An element of the list may also be a set or sequence of words or relations, or it may be a subgroup of G. In the latter case, the generators of the subgroup are taken as relators. Note that the identity element may be represented by the digit 1 within the relations-clause of the **quo**-constructor.

Consider, for example, the binary tetrahedral group with presentation $\langle p, q \mid (pq)^2 = p^3 = q^3 \rangle$. This is the quotient of the free group of rank 2 by the relation normal subgroup defined by the relations $(pq)^2 = p^3 = q^3$. The group may be created in the following way:

```
> F2<r, s> := FreeGroup(2);
> print F2;
Finitely presented group F2 on 2 generators (free)
> bt<p, q>, phi := quo< F2 | (r*s)^2 = r^3 = s^3 >;
> print bt;
Finitely presented group bt on 2 generators
Relations
    (p * q)^2 = q^3
    p^3 = q^3
> print phi;
Mapping from: GrpFP: F2 to GrpFP
> print Generators(F2), Generators(bt);
{ r, s }
{ p, q }
```

Notice that the generator names for *F2* and *bt* are independent. The original group and the quotient group co-exist independently in MAGMA – so that, in particular, further quotients of *F2* could now be defined.

The use of the MAGMA mechanism for constructing fp-groups, where the number of generators will not be known until run-time, will be illustrated by presenting a short function that constructs the *braid group* B_n on n strings ([CoM72], pp. 62–63). Let $\sigma_1, \ldots, \sigma_{n-1}$ be the generators for B_n. Then the relations for B_n are

$$\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1} \quad (1 \leq i \leq n-2)$$

and

$$\sigma_i\sigma_k = \sigma_k\sigma_i \quad (i \leq k-2)$$

```
Braid := function( n )
    F := FreeGroup(n-1);
    return quo< F |
     {F.i*F.(i+1)*F.i = F.(i+1)*F.i*F.(i+1) : i in [1..n-2]},
     {F.i*F.k = F.k*F.i : i in [1..k-2], k in [3..n-1]} >;
end function;
```

The function *Braid* is applied to construct the braid group on 4 strings.

```
> B4<x,y,z> := Braid(4);
> print B4;
Finitely presented group B4 on 3 generators
Relations
    x * y * x = y * x * y
    y * z * y = z * y * z
    x * z = z * x
```

The quotient of B_n obtained by imposing the relations

$$\sigma_i^2 = 1 \quad (1 \leq i \leq n)$$

is isomorphic to $\mathrm{Sym}(n)$.

```
> S4<a,b,c> := quo< B4 | x^2 = y^2 = z^2 = 1 >;
> print Order(S4);
24
```

This example will be developed further later in the chapter.

30.2.3 The Group Constructor

If the original free group F is not required, then an fp-group G may be created in a single step using the **Group**-constructor:

Group< generators | relations **>**

This construction is expanded to the construction of a free group and its quotient, as before, but provides greater convenience for the user. So the group *bt* and the surjection *phi* could have been defined alternatively as follows:

```
> bt<p, q>, phi := Group< p, q | (p*q)^2 = p^3 = q^3 >;
```

The **Group**-constructor is only available for quotients of free groups so that the **quo**-constructor must be used for quotients of other fp-groups.

Below is an example in which the required group is defined by a list of relators:

```
> fp2<f, g> := Group< f, g | f^8, g^7, (f*g)^2, (f, g)^9 >;
> print fp2;
Finitely presented group fp2 on 2 generators
Relations
    f^8 = Id(fp2)
    g^7 = Id(fp2)
    (f * g)^2 = Id(fp2)
    (f, g)^9 = Id(fp2)
```

30.2.4 Extensions, Products and Standard Groups

Various standard extensions and products are available for finitely-presented groups. Table 30.2 lists the functions that create them in Magma. Note that the two product functions may be applied to an arbitrary number of groups by supplying the groups in the form of a sequence.

Table 30.2. Extensions and products of fp-groups

Magma	Meaning
Darstellungsgruppe(G)	A maximal central extension of G, created as an fp-group
DirectProduct(H, K)	Direct product of H and K
DirectProduct(Q)	Direct product of groups in sequence Q
FreeProduct(H, K)	Free product of H and K
FreeProduct(Q)	Free product of groups in sequence Q

Section 29.1.3 gives details of the functions that construct standard finitely-presented groups such as cyclic and dihedral groups. These functions will be illustrated by showing that the order of the Schur multiplier of Alt(6) is 6:

```
> a6 := AlternatingGroup(GrpFP, 6);
> print a6;
Finitely presented group a6 on 2 generators
```

```
Relations
    a6.1^4 = Id(a6)
    a6.2^3 = Id(a6)
    (a6.1 * a6.2)^5 = Id(a6)
    (a6.2, a6.1)^2 = Id(a6)
    (a6.2 * a6.1^-2 * a6.2 * a6.1^2)^2 = Id(a6)
> d := Darstellungsgruppe(a6);
> print Order(d) / Order(a6);
6
```

30.2.5 Presentations for Concrete Groups

Let G be a finite permutation or matrix group. Provided that the order of G is not too large, the function **FPGroup**(G) may be used to construct a presentation for G on its defining generators. The cutoff point for this function will depend on local memory resources, but one million is a typical upper limit. For larger permutation or matrix groups, a presentation on an expanded generating set (actually a strong generating set) may be obtained by setting the parameter **StrongGenerators** to **true**:

```
> M12 := PermutationGroup< 12 | (1,2,3,4,5,6,7,8,9,10,11),
>   (1,9)(2,6)(4,5)(7,8),
>   (1,10)(2,5)(3,7)(4,8)(6,9)(11,12)>;
> print Order(M12);
95040
> F12<x,y,z> := FPGroup(G);
> print F12;
Finitely presented group F12 on 3 generators
Relations
    y^2 = Id(F12)
    z^2 = Id(F12)
    (x^-1 * y)^3 = Id(F12)
    (x^-1 * z)^3 = Id(F12)
    y * z * y * x^-2 * y * z * y * x * z * x = Id(F12)
    x^-11 = Id(F12)
```

The function **FPGroup**(G) may also be applied to groups belonging to the abstract group categories **GrpPC** and **GrpAb**. In this case, it returns an isomorphic group in the **GrpFP** category.

Table 30.3. Finitely-presented group information

MAGMA	Meaning
Generators(G)	Set containing the defining generators for G
Ngens(G)	Number of defining generators for G
Relations(G)	Sequence containing the defining relations for G
GeneratingWords(G, H)	Given a subgroup H of group G, return generators for H as words in the generators of G (if known)

30.2.6 Access Functions for Finitely Presented Groups

Table 30.3 summarizes the basic access functions which are particular to finitely-presented groups.

30.3 Special Quotients

Considerable progress towards an understanding of the structure of an fp-group may often be achieved through the examination of various kinds of quotients. Very effective algorithms exist for computing abelian, p-group and nilpotent quotients. Progress has also been made towards computing soluble quotients of moderate size.

30.3.1 Abelian Quotients

Usually the first step in investigating a group defined by a presentation is to look at its abelian quotients. MAGMA provides two functions for this purpose. The function **AbelianQuotient**(G) constructs the derived quotient group G/G' of G, i.e., the largest abelian quotient of G. The quotient is constructed as a magma in the category of abelian groups, **GrpAb**. The function returns the natural surjection $\phi : G \to A$ as a second return value. Alternatively, the function **AbelianQuotientInvariants**(G) returns the invariant factors and torsion-free rank of G/G' in the form of a sequence. The first r terms of the sequence contain the invariant factors $[d_1, \ldots, d_r$, where $d_i | d_{i+1}$ for all $i < r$, while a further s zeros give the torsion-free rank. Thus, each of the s zeros represents an infinite cyclic factor.

As a simple example, consider the presentation

$$\langle x, y, z, t \mid (xyz)^6, t^2 = (xz)^2, (xy^3zt^2)^2, (yt^2)^2 = x^2z^3, (xyz)^4(yt)^2 \rangle$$

as discussed in [Joh90], p. 75:

```
> G<x,y,z,t> := Group< x,y,z,t | (x*y*z)^6, t^2 = (x*z)^2,
>   (x*y^3*z*t^2)^2, (y*t^2)^2 = x^2*z^3,
>   (x*y*z)^4*(y*t)^2 >;
> A := AbelianQuotient(G);
> print A;
Abelian Group isomorphic to Z/2 + Z/6 + Z
Defined on 3 generators
Relations:
    2*A.1 = 0
    6*A.2 = 0
```

It can be seen that the abelian quotient of G is isomorphic to the infinite abelian group $Z/2Z \oplus Z/6Z \oplus Z$. Therefore G itself is infinite.

As a slightly more sophisticated example consider the problem of finding the structure of the Fibonacci group $F(2,7)$:

To perform this example online, type `load "I96c30e1";`

```
> F27<a, b> := Group<a, b |
>   a^2 * b^-2 * a^-1 * b^-2 * (a^-1 * b^-1)^2,
>   a * b * a * b^2 * a * b * a * b^-1 * (a * b^2)^2>;
> print AbelianQuotientInvariants(F27);
[ 29 ]
```

The output shows that the quotient is cyclic, of order 29. The next step is to investigate the index of obvious subgroups:

```
> print Index(F27, sub< F27 | a >);
1
```

Consequently, *F27* is generated by a (or b) and so must be cyclic. Hence, *F27* must be cyclic of order 29. This is the procedure followed by George Havas when he first determined the order of *F27*.

30.3.2 p-Quotients

The lower p-central series of a group G is the chain of normal subgroups

$$\Phi_1(G) \rhd \Phi_2(G) \rhd \Phi_3(G) \rhd \ldots,$$

where

$$\Phi_1(G) = G, \qquad \Phi_{i+1}(G) = [G, \Phi_i(G)]\Phi_i(G)^p$$

for $i \geq 1$. The p-quotient algorithm constructs quotients of G with respect to successive terms of this series. If G has a finite p-quotient then, providing that its size does not exceed available resources, this quotient may be constructed using the p-quotient algorithm.

Given an fp-group G, a prime p and a non-negative integer c, the function **pQuotient**(G, p, c) returns two values: the largest p-quotient P of G having lower exponent-p class at most $\min(c, 63)$, as a polycyclic group (category **GrpPC**); and the natural surjection $\phi : G \to P$. The parameter **Print** may be used to provide information about the progress of the algorithm; its default value of 0 specifies no output, and the values 1–3 specify progressively more detailed output.

The relevant primes for **pQuotient** may be discovered by examining the abelian quotient of G. For example, consider the Fibonacci groups $F(9)$. From the structure of its abelian quotient, $F(9)$ has nontrivial p-quotients for $p = 2$ and $p = 19$:

```
> F9< x1,x2,x3,x4,x5,x6,x7,x8,x9 > :=
>    Group< x1, x2, x3, x4, x5, x6, x7, x8, x9 |
>    x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6, x5*x6=x7,
>    x6*x7=x8, x7*x8=x9, x8*x9=x1, x9*x1=x2 >;
> A := AbelianQuotient(F9);
> print A;
Abelian Group isomorphic to Z/2 + Z/38
Defined on 2 generators
Relations:
    2*A.1 = 0
    38*A.2 = 0

> p2 := pQuotient(F9, 2, 10 : Print := 1);
Lower exponent-2 central series for F9
Group: F9 to lower exponent-2 central class 1 has order 2^2
Group: F9 to lower exponent-2 central class 2 has order 2^3
Group completed. Lower exponent-2 central class = 2,
Order = 2^3

> p19 := pQuotient(F9, 19, 10 : Print := 1);
Lower exponent-19 central series for F9
Group: F9 to lower exponent-19 central class 1 has order 19^1
Group completed. Lower exponent-19 central class = 1,
Order = 19^1
```

Hence the largest 2-quotient of $F(9)$ has order 8 and the largest 19-quotient has order 19. Consequently, the maximal nilpotent quotient has order 152.

The following example constructs the largest 2-quotient of class 4 for the group $\langle a, b \mid [b, a, a], (aba)^4 \rangle$:

```
> pQ := pQuotient(Group<a, b | (b,a,a), (a*b*a)^4>, 2, 4:
>   Print := 1);
Lower exponent-2 central series for $
Group: $ to lower exponent-2 central class 1 has order 2^2
Group: $ to lower exponent-2 central class 2 has order 2^5
Group: $ to lower exponent-2 central class 3 has order 2^8
Group: $ to lower exponent-2 central class 4 has order 2^11

> print pQ;
GrpPC : pQ of order 2048 = 2^11
PC-Relations:
    pQ.1^2 = pQ.4,
    pQ.2^2 = pQ.5,
    pQ.3^2 = pQ.7 * pQ.10,
    pQ.4^2 = pQ.8,
    pQ.5^2 = pQ.10 * pQ.11,
    pQ.6^2 = pQ.10,
    pQ.7^2 = pQ.10,
    pQ.8^2 = pQ.11,
    pQ.2^pQ.1 = pQ.2 * pQ.3,
    pQ.3^pQ.2 = pQ.3 * pQ.6,
    pQ.4^pQ.2 = pQ.4 * pQ.7,
    pQ.5^pQ.1 = pQ.5 * pQ.6 * pQ.7 * pQ.10,
    pQ.5^pQ.3 = pQ.5 * pQ.9 * pQ.10,
    pQ.6^pQ.2 = pQ.6 * pQ.9,
    pQ.7^pQ.2 = pQ.7 * pQ.10,
    pQ.8^pQ.2 = pQ.8 * pQ.10
```

An exponent law may be imposed on the group to which **pQuotient** is applied by means of the **Exponent** parameter. If this is set to a positive integer value m, rather than its default value of 0, then the exponent law $x^m = 1$ will be enforced. This is demonstrated below by showing that the order of the largest finite 4-generator group having exponent 4 is 2^{422}:

```
> F4 := FreeGroup(4);
> B44 := pQuotient( F4, 2, 10: Exponent := 4, Print := 1);
Lower exponent-2 central series for F4
Group: F4 to lower exponent-2 central class 1 has order 2^4
Group: F4 to lower exponent-2 central class 2 has order 2^14
Group: F4 to lower exponent-2 central class 3 has order 2^34
```

```
Group: F4 to lower exponent-2 central class 4 has order 2^89
Group: F4 to lower exponent-2 central class 5 has order 2^188
Group: F4 to lower exponent-2 central class 6 has order 2^272
Group: F4 to lower exponent-2 central class 7 has order 2^352
Group: F4 to lower exponent-2 central class 8 has order 2^392
Group: F4 to lower exponent-2 central class 9 has order 2^412
Group: F4 to lower exponent-2 central class 10 has order 2^422
Group completed. Lower exponent-2 central class = 10, Order = 2^422
```

There is also a process version of the p-quotient algorithm, which allows the user to control the individual steps closely. Details may be found in the *Handbook*.

30.3.3 Other Quotients

A soluble quotient algorithm developed by Alice Niemeyer is also available, as explained in the *Handbook*. However, before the algorithm can be applied, the user must determine the orders of the terms in a certain series. The nilpotent quotient algorithm developed by Werner Nickel will be installed during 1997. This algorithm is capable of handling infinite quotients.

30.4 Subgroups

30.4.1 The sub-constructor

A subgroup H of an fp-group G may be defined either by giving a set of generating words or by prescribing the action of G on the right cosets of H. The **sub**-constructor is used in the usual manner to define H in terms of generators (see Chapter 4). The action of G on the cosets of H gives rise to a homomorphism $\phi : G \to \mathrm{Sym}(n)$, where $n = [G : H]$. Hence, a subgroup defined by such an action is created using the **sub**-constructor with ϕ appearing on the right side of the constructor (in place of the generators).

This version of **sub**-constructor is illustrated using the braid group B_4.

```
> B4<x,y,z> := Braid(4);
> P := PermutationGroup< 4 | (1,2), (2,3), (3,4) >;
> phi := hom< B4 -> P | [ P.i : i in [1..3] ] >;
> H4 := sub< B4 | phi >;
> H4;
Finitely presented group H4
Index in group B4 is 4 = 2^2
Subgroup of group B4 defined by coset table
```

It is possible to define a subgroup giving only normal generators, that is, a set of elements whose normal closure gives the required subgroup. Such a subgroup is defined using the **ncl**-constructor whose syntax, apart from the **ncl** tag, is identical to that of the **sub**-constructor. Note that for this constructor to work, MAGMA must be able to construct the normal closure N, a necessary condition for which is N having finite index in G. Note that the resulting subgroup will be defined by a coset table. The **ncl**-constructor will be illustrated by constructing the derived subgroup in the infinite group G below.

```
> G<a,b,c> := Group< a,b,c | a^3, b^4, c^5, (a*b)^2,
>                            (a*b*c)^2, (b*c)^2 >;
> H := ncl< G | (a,b), (a,c), (b,c) >;
> print H;
Finitely presented group H
Index in group G is 2
Subgroup of group G defined by coset table
```

The convention throughout MAGMA] is that submagmas of a magma A should be created as fully self-contained objects in the same category as M. This convention is broken in the case of subgroups of fp-groups since many calculations involving a subgroup H defined only by generating words, are possible in situations where it would be impractical to compute a presentation for H. Thus, in the category of fp-groups, whenever a subgroup is created, it will almost never have defining relations associated with it. In general, if the user wishes to covert a subgroup of an fp-group into an fp-group in its own right, then it is necessary to explicitly apply a rewriting algorithm (see Section 30.6).

30.4.2 The Todd-Coxeter Algorithm

The major tool for computing with subgroups of an fp-group is the Todd-Coxeter procedure. Although sometimes referred to as an algorithm, since it does not necessarily terminate, it is more correctly referred to as a procedure or method. Given a group G and a subgroup H of G, it attempts by trial and error to construct the action of G on the (right) cosets of H. The cosets are traditionally identified with the integers $1, \ldots, n$, where coset 1 always corresponds to the given subgroup H. This method may be used when the group G is infinite, provided that the index of H in G is finite. However, the time taken by the method cannot be predicted, even in the case in which G is the trivial group. In some sense this is yet another quotient algorithm since it may be regarded as a method for constructing a permutation representation of G.

The output of the procedure is a mapping giving the action of G on the (right) cosets of the subgroup H. MAGMA follows customary terminology in referring to this mapping as the *coset table* for H in G. As a by-product, if the Todd-Coxeter terminates, it gives the index of H in G.

A number of functions directly invoke the Todd-Coxeter procedure and differ mainly in the information returned. These functions include **ToddCoxeter**, **Index**, **CosetSpace** and **CosetAction**. In this section, the discussion will focus on **ToddCoxeter** and **Index**. Both of these functions take the same arguments and parameters, and return the index of H in G, or 0 if the enumeration fails to complete. The function **ToddCoxeter** returns three additional values: the coset table, the maximum number of cosets, and the total number of cosets. Several parameters allow the user to control aspects of the coset enumeration as will be explained below.

For example, the following code enumerates the cosets of the subgroup $H8723 = \langle a^2, a^{-1}b\rangle$ in $G8723 = \langle a, b \mid a^8, b^7, (ab)^2, (a^{-1}b)^3\rangle$:

To perform this example online, type `load "I96c30e2";`

```
> G8723<a, b> := Group< a, b |
>   a^8, b^7, (a * b)^2, (a^-1 * b)^3 >;
> H8723 := sub< G8723 | a^2, a^-1 * b >;
> print Index(G8723, H8723);
448
```

From the output, the number of cosets of *H8723* in *G8723* is 448.

A zero value returned by a function invoking the Todd-Coxeter method indicates that the enumeration has failed to complete using a given amount of memory.*No mathematical conclusion can be drawn from this outcome.* It cannot be concluded that the index is infinite, though this may well be the case. In these circumstances, different parameter values (see below) or more sophisticated techniques are required to settle the issue. It is most important to note that the default limit on the memory limit for coset enumeration is quite small and that for all serious or hard enumerations, the user will need to explicitly set his/her own limit as described below.

A large number of different strategies may be employed by the Todd-Coxeter method. The default strategy may be unsuitable for many particular enumerations, in that the method may exhaust the resources before completing. It is therefore necessary for the user to become familiar with some of the parameters which manipulate the progress of the function. The most important of these is **CosetLimit**, which gives an upper limit on the number of cosets that may be defined at any stage of the process. (In the case of hard enumerations, it is frequently true that the method will introduce a

substantially larger number of cosets than the index of H in G.) By default, **CosetLimit** is approximately one million divided by the number of generators. However, when enumerating the cosets of a subgroup of large index, or when attempting to perform a hard enumeration, it may be necessary to increase this value. Alternatively, in other circumstances it may be desirable to lower this limit.

The example below illustrates the effect of **CosetLimit**. It will be seen that for the groups *G8723* and *H8723*, the enumeration fails when the parameter is set to 1000 or 2000, but succeeds for 3000:

```
> index := ToddCoxeter(G8723, H8723: CosetLimit := 1000);
> print index;
0
> index := ToddCoxeter(G8723, H8723: CosetLimit := 2000);
> print index;
0
> index := ToddCoxeter(G8723, H8723: CosetLimit := 3000);
> print index;
448
```

Another important parameter is **Print**, which controls the level of informative printing during the execution of the function. It takes a small non-negative integer as its value (default 0). The value 0 indicates no printing; the value 1 merely causes statistics to be printed upon termination; the value 2 produces a message giving the current maximum and the total cosets to be printed after every m cosets, where the frequency m may be set using the **Grain** parameter (default 10 000). Observation of trends in these values frequently indicates whether or not the enumeration is likely to complete. For example, the following (slightly edited) output indicates the information printed when **Print** is set to 2 and **Grain** is set to 500:

```
> G8723<a, b> := Group< a, b |
>   a^8, b^7, (a * b)^2, (a^-1 * b)^3 >;
> H8723 := sub< G8723 | a^2, a^-1 * b >;
> index := ToddCoxeter(G8723, H8723: Print := 2,
>                                    Grain := 500);
 Maximum table size = 499829
Fill Factor = 7, CT Factor = 0, RT Factor = 0
RD alive=       500, Max=       500, Total=       547
RD alive=      1000, Max=      1000, Total=      1142
RD alive=      1500, Max=      1500, Total=      1745
RD alive=      2000, Max=      2000, Total=      2417
RC alive=      1500, Max=      2176, Total=      2626
RC alive=      1000, Max=      2176, Total=      2626
```

```
RC alive=       500, Max=      2176, Total=      2626
Index = 448, Max = 2176, Total = 2626
Time = 0.168 seconds
```

In the above output, the value `RD alive` gives the number of cosets active at the time the message is printed, `Max` is the maximum number of cosets defined at the same time, and `Total` is the cumulative count of the number of cosets defined.

The parameter **Hard** is a relative measure of how difficult the user expects the coset enumeration to be. If **Hard** is changed to **true** from its default value of **false**, MAGMA will process relators in the coset table style rather than in the relator style. For instance, the following code repeats the above enumeration with **Hard** set to **true**:

```
> index := ToddCoxeter(G8723, H8723: Hard := true,
>                        Print := 2, Grain := 500);
 Maximum table size = 499829
Fill Factor = 7, CT Factor = 1000, RT Factor = 1
CD alive=       500, Max=       500, Total=       500
Index = 448, Max = 861, Total = 877
Time = 0.163 seconds
```

It can be seen that this strategy reduces the maximum number of cosets defined from 2176 to 861. If an enumeration of the cosets for a subgroup of large index is being attempted, such a reduction will greatly increase the chances of a successful enumeration.

The user may exercise much more precise control over the strategy adopted through use of the parameter **Strategy**, the details of which may be found in the *Handbook*.

30.4.3 Coset Spaces

The *(right) indexed coset space* V of the subgroup H of the group G is a G-set consisting of the set of integers $\{1, ..., m\}$, where i represents the right coset c_i of H in G. The action of G on this G-set is that induced by the natural G-action

$$T : V \times G \to V$$

where

$$T(c_i, x) = c_k \quad \Longleftrightarrow \quad c_i x = c_k,$$

for $c_i \in V$ and $x \in G$. (The mapping T is known as the *coset table* for H in G). If some of the products $c_i x$ are unknown, the corresponding images under T are undefined, and V is called an *incomplete coset space* for H in G.

The Todd-Coxeter method may be viewed as constructing a coset space. While the **Index** and **ToddCoxeter** functions emphasize the determination of the index of a subgroup H in the group G, the actual coset space is of interest in a number of situations. For example, if the Todd-Coxeter method fails to terminate, an incomplete coset space may be used to build a subgroup that properly contains H. Iterating the process a number of times may result in constructing a subgroup for which the Todd-Coxeter does terminate. Again, the complete coset space yields a permutation representation for G. The function for constructing a coset space (complete or incomplete) is **CosetSpace**(G, H). It has the same parameters as **ToddCoxeter**, and the user will often need to over-ride the default values. Some elementary functions for coset spaces are given in Table 30.4. Further operations may be found in Table 30.7.

Table 30.4. Operations on Coset Spaces

MAGMA	Meaning
`#`V	Cardinality of the coset space V
Action(V)	Mapping (coset table) $T : V \times G \to V$ giving the action of G on the coset space V
$T(i, w)$	Image of coset i and word w with respect to the coset action mapping T
IsComplete(V)	Returns **true** if V is a complete coset space
ExcludedConjugates(V)	Conjugates of the generators of subgroup H by the generators of G that lie outside of H, according to the (complete or incomplete) coset space V of H in G
ExcludedConjugates(T)	As above, but according to the (complete or incomplete) coset table T of H in G

The following example illustrates the use of coset spaces in building subgroups. It gives a function *DerSubgroup* which computes the derived subgroup G' for the fp-group G (assuming that the Todd-Coxeter method can construct the coset space of G' in G):

```
DerSubgroup := function(G)
    /* Initially define S to contain the commutator
       of each pair of generators of G */
    S := { commut : x, y in Generators(G) |
            commut ne Id(G) where commut is (x,y) };
    /* Successively extend S until it is closed under
       conjugation by the generators of G */
    repeat
        V := CosetSpace(G, sub<G | S>);
```

```
        E := ExcludedConjugates(V);
        S := S join E;
    until #E eq 0;
    return sub<G | S>;
end function;
```

The function is now used to construct the derived subgroup of the infinite 2-dimensional crystallographic group $\langle a, b \mid a^3, b^2, (a*b)^6 \rangle$:

```
> G<a, b> := Group< a, b | a^3, b^2, (a*b)^6 >;
> D := DerSubgroup(G);
> print D;
Finitely presented group D on 6 generators
Index in group G is 6 = 2 * 3
Generators as words in group G
    D.1 = (a^-1, b)
    D.2 = (a, b)
    D.3 = (b, a^-1)
    D.4 = a^-2 * b^-1 * a * b * a
    D.5 = a^-1 * b^-1 * a^-1 * b * a^2
    D.6 = (b, a)
```

30.4.4 Subgroup Constructions

Table 30.5. Operations on Subgroups

MAGMA	Meaning
$H\hat{}x$	Conjugate of subgroup H by element x
H **meet** K	Intersection of H and K
Core(G, H)	Largest normal subgroup of G contained in H
MaximalOvergroup(G, H)	Maximal subgroup of G containing H
MinimalOvergroup(G, H)	Subgroup of G which contains H as a maximal subgroup
Normalizer(G, H)	Normalizer of H in G
NormalClosure(G, H)	Normal closure of H in G
SchreierGenerators(G, H)	Schreier generators for H in G
SchreierSystem(G, H)	System of coset representatives for H in G
Transversal(G, H, K)	System of representatives for the double cosets HuK for the subgroups H and K of G

Let H and K be subgroups of the fp-group G. It is possible to compute various subgroups relating to H and K knowing only a set of generating

Table 30.6. Boolean Operations on Subgroups

H **eq** K	u **in** H	H **subset** K
IsConjugate(G, H, K)	**IsNormal**(G, H)	**IsMaximal**(G, H)

words and a coset table for H in G. The major operations that return a subgroup are summarized in Table 30.5, and some boolean operations are listed in Table 30.6.

The example below applies these subgroup constructions in order to construct subgroups of small index in the two-dimensional space group *p4g* having presentation $\langle r, s \mid r^2, s^4, (r, s)^2 \rangle$:

```
> p4g<r, s> := Group< r, s |
>   r^2 = s^4 = (r, s)^2 = 1 >;
> print p4g;
Finitely presented group p4g on 2 generators
Relations
    r^2 = Id(p4g)
    s^4 = Id(p4g)
    (r, s)^2 = Id(p4g)
> h := sub< p4g | (s^-1 * r)^4, s * r >;
> k := sub< p4g | (s^-1 * r)^2, (s * r)^2 >;
> print Index(p4g, h), Index(p4g, k);
8 8
> n := NormalClosure(p4g, h);
> print n;
Finitely presented group n on 5 generators
Index in group p4g is 2
Generators as words in group p4g
    n.1 = (s^-1 * r)^4
    n.2 = s * r
    n.3 = r * s
    n.4 = r^-1 * s * r^2
    n.5 = s^2 * r * s^-1
> m := MinimalOvergroup(p4g, h);
> print m;
Finitely presented group m on 3 generators
Index in group p4g is 4 = 2^2
Generators as words in group p4g
    m.1 = (s^-1 * r)^4
    m.2 = s * r
    m.3 = (r * s)^2
> n := MaximalOvergroup(p4g, k);
```

```
> print n;
Finitely presented group n on 4 generators
Index in group p4g is 2
Generators as words in group p4g
    n.1 = (s^-1 * r)^2
    n.2 = (s * r)^2
    n.3 = r
    n.4 = s^2
> l := h meet k;
> print l;
Finitely presented group l
Index in group p4g is 16 = 2^4
Subgroup of group p4g defined by coset table
> j := h^s;
> print j;
Finitely presented group j on 2 generators
Index in group p4g is 8 = 2^3
Generators as words in group p4g
    j.1 = (s^-2 * r * s)^4
    j.2 = r * s
> print j eq h;
false
> print IsConjugate(p4g, h, j);
true r^-1
> c := Core(p4g, h);
> print Index(p4g, c);
32
```

30.5 Permutation Representations

The action of G on the coset space V of a subgroup H gives a permutation representation of G. This construction yields yet another type of quotient, and may be used to discover non-soluble composition factors of G. Such a representationa may be constructed by means of the function **CosetAction**(G, H), as discussed in Section **??**. However, note that this function only returns the action and the image when G and H are fp-groups, since the kernel of the action is difficult to compute unless the index of the kernel in G is very small. If the user desires the kernel, it must be obtained using **CosetKernel**(G, H) or **Core**(G, H).

For example:

```
> G8723m, G8723L := CosetAction(G8723, H8723);
```

```
> print G8723L, G8723m;
Permutation group G8723L of degree 448
Mapping from: GrpFP: G8723 to GrpPerm: G8723L
```

For more difficult examples, **CosetAction**(G, H) may not be sufficient to construct a permutation representation of G. In this case, the coset space V should be constructed with a parametrized call to **CosetSpace**(G, H), and then one of the functions in Table 30.7 should be used to extract the permutation representation.

Table 30.7. Representations Afforded by Coset Spaces

MAGMA	Meaning
CosetAction(V)	Returns (i) homomorphism of G into a permutation group L given by the action of G on the coset space V of H; (ii) image L
CosetAction(G, H)	Returns (i) homomorphism of G into a permutation group L given by the action of G on the coset space of H; (ii) image L
CosetImage(V)	Image of **CosetAction**(V)
CosetImage(G, H)	Image of **CosetAction**(G, H)
CosetKernel(V)	Kernel of **CosetAction**(V) (only practical when $[G : H]$ is small)
CosetKernel(G, H)	Kernel of **CosetAction**(G, H) (only practical when $[G : H]$ is small)

The difficult step in constructing a permutation representation is finding a suitable subgroup. This will sometimes be known from the manner in which the presentation has been constructed. Another approach is to examine subgroups generated by random words in the hope of finding one with the desired properties. In the final section in this chapter, a systematic method is given for locating all subgroups having index less than or equal to some modest limit.

The following code constructs a permutation representation of degree 1140 000 for the Harada-Norton simple group. A subgroup H having this index is known from the manner in which the presentation was constructed. Because the index of H is so large, the function **CosetAction** would fail, since it would invoke the Todd-Coxeter method with default values for the parameters:

```
> HN<x, a, b, c, d, e, f, g > :=
>       Group< x, a, b, c, d, e, f, g |
>          x^2, a^2, b^2, c^2, d^2, e^2, f^2, g^2,
```

```
>              (x, a), (x, g),
>              (b*c)^3, (b*d)^2, (b*e)^2, (b*f)^2, (b*g)^2,
>              (c*d)^3, (c*e)^2, (c*f)^2, (c*g)^2,
>              (d*e)^3, (d*f)^2, (d*g)^2,
>              (e*f)^3, (e*g)^2,
>              (f*g)^3,
>              (b, x*b*x),
>              (a, e*d*c*b), (a, f)*d*c*b*d*c*d, (a*g)^5,
>              (c*d*e*f, x*b*x), (b, x*c*d*e*f*x),
>              (c*d*e*f, x*c*d*e*f*x) >;
> H := sub< HN | x,b,c,d,e,f,g>;
> cstspc := CosetSpace(HN, H: CosetLimit := 1200000,
>   Hard := true, Print := 1);
Fill Factor = 12, CT Factor = 1000, RT Factor = 1
Index = 1140000, Max = 1141309, Total = 1470174
Time = 132.900 seconds

> G := CosetImage(cstspc);
> print G;
Permutation group G acting on a set of cardinality 1140000
> print Order(G);
546061824000000
> print FactoredOrder(G);
[ <2, 15>, <3, 6>, <5, 6>, <7, 1>, <11, 1>, <19, 1> ]
> time print CompositionFactors(G);
    G
    |  Cyclic(2)
    *
    |  HA
    1
Time: 3941.309
```

30.6 Presentations for Subgroups

Suppose that G is a fp-group whose defining presentation has r generators and s relations. Let H be some subgroup of G having finite index n which has been created by MAGMA. However H was constructed, it will be initially defined either in terms of a set of generating words or in terms of the action of G on its cosets (i.e. by means of a coset table). For some applications, it will be necessary to have H as a fp-group and this means constructing a presentation for it. Such a presentation may be obtained by rewriting the relations stored for G using the Reidemeister-Schreier algorithm. Unfortunately, the Reidemeister-Schreier algorithm constructs a presentation involv-

ing $(n-1)r+1$ generators and $n*s$ relations. Clever techniques based on the use of Tietze transformations have been developed to simplify such presentations. Even so, the use of subgroup rewriting is restricted to subgroups of modest index.

The function **Rewrite**(G, H) returns an fp-group K isomorphic to H and embedded in G. Note that the default rewriting procedure will not, in general, construct the presentation on the original generators for H. By default, **Rewrite** automatically performs simplification on the rewritten presentation, but the user may turn it off by setting the parameter **Simplify** to false. It should be noted that the simplification procedure will often eliminate a large number of the generators of K at the cost of producing long relations. For some applications these long relations are harmful and so controled simplification needs to be performed using the techniques discussed in the next section.

The **Rewrite** function will be illustrated by proving that the group $\langle a, b \mid a^6, b^6, ab^2 = ba^2 \rangle$ is infinite:

```
>G<a, b> := Group<a, b | a^6 = b^6 = 1, a * b^2 = b * a^2>;
> print AbelianQuotientInvariants(G);
[ 6 ]
> H := ncl<G | (a,b)>;
> print Index(G, H);
6
> K := Rewrite(G, H);
> print K;
Finitely presented group K on 2 generators
Generators as words in group G
    K.1 = b * a^-1
    K.2 = a * b * a^-2
Relations
    K.1^2 * K.2 * K.1^2 * K.2^-1 = Id(K)
    K.1 * K.2^2 * K.1^-1 * K.2^2 = Id(K)
> print AbelianQuotientInvariants(K);
[4, 4]
> L := ncl<K | (K.1, K.2)>;
> print Index(K, L);
16
> M := Rewrite(K, L);
> print M;
Finitely presented group M on 3 generators
Generators as words in group K
    M.1 = (K.1, K.2^-1)
    M.2 = (K.2^-1, K.1^-1)
    M.3 = (K.2, K.1^-1)
```

```
Relations
    (M.1^-1, M.2^-1) = Id(M)
    (M.1^-1, M.3^-1) = Id(M)
    (M.2^-1, M.3^-1) = Id(M)
> print AbelianQuotientInvariants(M);
[ 0, 0, 0 ]
```

The subgroup M has index 96 in G and its abelian quotient is infinite. It follows that G is infinite.

The next example constructs a presentation for a subgroup defined by a coset table. The braid group B_n, considered in Section 30.2.2, has a subgroup H_n of index n which is constructed as follows. As noted in Section 30.2.2 there is a homomorphism $\phi : B_n \to S_n$, where S_n is isomorphic to $\mathrm{Sym}(n)$. The subgroup H_n of B_n is the preimage of the stabilizer of 1 in $\mathrm{Sym}(n)$ with respect to the homomorphism $\tau : B_n \to \mathrm{Sym}(n)$.

```
> B4<x,y,z> := Braid(4);
> S4<a,b,c>, phi := quo< B4 | x^2 = y^2 = z^2 = 1 >;
> rho := CosetAction(S4, sub< S4 | b, c >);
> H4 := sub< B4 | phi*rho >;
> H4<l,m,n> := Rewrite(B4, H4);
> print H4;
Finitely presented group H4 on 3 generators
Generators as words in group B4
    l = y
    m = z
    n = x^2
Relations
    l * n * l * n^-1 * l^-1 * n^-1 = Id(H4)
    n^-1 * l * m * l^-1 * n * l * m^-1 * l^-1 = Id(H4)
    l * n^-1 * m * n * l^-1 * n^-1 * m^-1 * n = Id(H4)
    m^-1 * l^-1 * m^-1 * l * m * l * m * l^-1 = Id(H4)
    m^-1 * n^-1 * m^-1 * n * m * n * m * n^-1 = Id(H4)
```

The subgroup H_n plays a key role in the study of braid groups. It is a semidirect product of the free group of rank $n-1$ with the braid group B_{n-1}. (See [Joh90]).

An important special case of subgroup rewriting occurs when the objective is to use the presentation of the subgroup H of the fp-group G in order to determine the structure of the derived quotient group, H/H'. Rather than producing all the relations of H at once and then abelianizing, the function **AbelianQuotient**(G, H) abelianizes each relator as it is produced, thereby possibly saving a large amount of time and memory. Similarly, the function **AbelianQuotientInvariants**(G, H) returns the invariant factors and

torsion-free rank of H/H'. These versions of the functions are frequently useful when attempting to prove that G is infinite. Even when G has a finite abelian quotient, it may have subgroups of small index possessing infinite abelian quotients. For example:

```
> G<x,z> := Group< x, z | z^3*x*z^3*x^-1, z^5*x^2*z^2*x^2 >;
> H := sub< G | x, z*x*z, z*x^-1*z*x*z^-1>;
> print AbelianQuotientInvariants(H);
[ 2, 6, 0 ]
```

The zero at the end of the sequence of invariants indicates that the abelian quotient has an infinite summand.

30.7 Simplifying a Presentation

It frequently happens that a presentation contains redundant generators and/or relations. This is usually the case if the presentation has been constructed by means of some rewriting method such as the Reidemeister-Schreier algorithm. The function **Simplify**(G) attempts to eliminate generators and to shorten relators by locating substrings that correspond to the left or right hand side of a relation. It returns a new group K, isomorphic to G, whose defining presentation is the simplified presentation for G.

For example, the Fibonacci group $F(n)$ is generated by the set $\{x_1, \ldots, x_n\}$ with the defining relations $x_i x_{i+1} = x_{i+2}$, for $i \in \{1, \ldots, n\}$, where the subscripts are taken modulo n. It is easily seen that any Fibonacci group $F(n)$ can be given as a 2-generator group. The code below invokes **Simplify** to obtain the 2-generator presentation for $F(7)$:

```
> F7<x1,x2,x3,x4,x5,x6,x7> := Group< x1,x2,x3,x4,x5,x6,x7 |
>   x1*x2=x3, x2*x3=x4, x3*x4=x5, x4*x5=x6,
>   x5*x6=x7, x6*x7=x1, x7*x1=x2 >;
> K<a,b> := Simplify(F7);
> print K;
Finitely presented group K on 2 generators
Relations
    a^2 * b^-1 * a * b^-1 * a * b^2 * a * b = Id(K)
    a^2 * b^-1 * a^2 * b^-1 * a * b^-2 * a^-1 * b^-1 = Id(K)
```

The **Simplify** function has a parameter **Iterations** which allows the user to bound the number of times the main elimination loop is executed. (By default, there is no limit.) Usually, one generator is eliminated for each iteration of the loop. There is also a process version, allowing the user to

direct the simplification algorithm. The reader is referred to the *Handbook* for details.

30.8 Locating Subgroups of Finite Index

It is frequently desirable to know all subgroups of an fp-group G which have index less than or equal to some given (small) bound. For instance, if it is known that G has a non-trivial subgroup, then G cannot be trivial. MAGMA provides two related functions for constructing subgroups of low index: **LowIndexSubgroups** simply returns a sequence of subgroups; and **LowIndexProcess** constructs a process from which the user may extract as many low index subgroups as desired.

30.8.1 Subgroups of Low Index

The functions **LowIndexSubgroups**(G, R) returns a sequence of subgroups of the group G satisfying certain bound conditions described by R. The argument R may either be an integer, giving an upper bound on the index, or a 2-tuple of integers, giving a lower bound and an upper bound. The terms of the sequence are arranged in non-decreasing order of the subgroup index, and each entry of the sequence is a representative for a distinct conjugacy class of subgroups.

For example, suppose the user wishes to calculate all conjugacy classes of subgroups of the infinite triangle group $\langle x, y \mid x^2, y^2, (xy)^7 \rangle$ having index less than or equal to 12. The appropriate statements are:

```
> G<x, y> := Group<x, y | x^2, y^3, (x*y)^7 >;
> L := LowIndexSubgroups(G, 12);
[
    Finitely presented group on 2 generators
    Index in group G is 1
    Generators as words in group G
        $.1 = x
        $.2 = y,

    Finitely presented group on 3 generators
    Index in group G is 7
    Generators as words in group G
        $.1 = x
        $.2 = y * x * y^-1
        $.3 = y^-1 * x * y^-1 * x * y * x * y,
```

```
    Finitely presented group on 3 generators
    Index in group G is 7
    Generators as words in group G
        $.1 = x
        $.2 = y * x * y^-1
        $.3 = y^-1 * x * y * x * y^-1 * x * y,

    Finitely presented group on 2 generators
    Index in group G is 8 = 2^3
    Generators as words in group G
        $.1 = y
        $.2 = x * y * x * y * x * y^-1 * x,

    Finitely presented group on 2 generators
    Index in group G is 9 = 3^2
    Generators as words in group G
        $.1 = x
        $.2 = y * x * y * x * y * x * y^-1
]
```

If the user wishes to impose a lower bound l as well as an upper bound u on the size of the index, then R should be given in the form of a tuple `< `l, u` >`. For example, the following statement determines the number of conjugacy classes of subgroups of G having index exactly 50:

```
> L := LowIndexSubgroups(G, <50, 50>);
> print #L;
333
> print L[10];
Finitely presented group on 4 generators
Index in group G is 50 = 2 * 5^2
Generators as words in group G
    $.1 = x
    $.2 = y * x * y * x * y^-1 * x * y^-1 * x * y * x
          * y * x * y^-1
    $.3 = y^-1 * x * y * x * y^-1 * x * y * x * y^-1
          * x * y^-1 * x * y * x * y
    $.4 = y * x * y * x * y * x * y^-1 * x * y^-1 * x
          * y * x * y * x * y^-1 * x * y * x * y
```

LowIndexSubgroups has several parameters. The most important of them are **Limit**, **Subgroup**, and **Print**. If the user assigns a positive integer to **Limit**, the function will terminate after finding n conjugacy classes in the given index range. Otherwise, there is no theoretical limit to the number of

classes MAGMA finds. If a subgroup H of G is assigned to the parameter **Subgroup**, MAGMA will restrict its search for subgroups of appropriate index to those subgroup classes in which at least one subgroup contains H. The default value of **Subgroup** is the trivial subgroup, so that the function normally returns all the subgroups in the index range.

The **Print** parameter allows the user to control the amount of output that MAGMA gives when **LowIndexSubgroups** is invoked. By default (value 0), no intermediate printing is given. If **Print** is assigned 1, 2, or 3, MAGMA gives output progressively as it calculates the low index subgroups. For the value 1, MAGMA prints for each class the index, the length, and a set of generators for the class representative. For the value 2, it also prints the permutation representation of G on the right cosets of the class representative, and for the value 3 it also prints generators for the normalizer N of the class representative and a system of right coset representatives for N in G.

As an illustration of the use of these parameters, consider the problem of finding five subgroups of G which contain *H8723*. Certainly if a subgroup contains *H8723* its index cannot exceed the index of *H8723*, and so 448, the index of *H8723*, will do well for an upper bound. The following line shows how to calculate the subgroups, requesting a maximum of 5 subgroups by means of the **Limit** parameter, and obtaining a small amount of output immediately after each class is constructed:

```
> LH8723 := LowIndexSubgroups(G8723, 448: Print := 1,
>   Limit := 5, Subgroup := H8723);
Subgroup class 1        Index 7 Length 7
Subgroup generators :-
{ a^2, a^-1 * b, b * a^-1 * b^-2 }

Subgroup class 2        Index 7 Length 7
Subgroup generators :-
{ a^2, a * b^-2 * a^-1 * b * a^-1, a^-1 * b }

Subgroup class 3        Index 14        Length 7
Subgroup generators :-
{ a^2, b^2 * a^-2 * b * a^-1, a^-1 * b }

Subgroup class 4        Index 14        Length 7
Subgroup generators :-
{ a^2, (b^2 * a^-1 * b^-2)^2, a^-1 * b }

Subgroup class 5        Index 28        Length 28
Subgroup generators :-
{ (b^3 * a^-1 * b^-3)^4, a^2, b * a^-3 * b * a^-1,
b^2 * a^2 * b * a^-1 * b^-2 * a^-2 * b^-2,
```

```
(b^2 * a^-1 * b^-2)^4, a^-1 * b }

> print forall{s:s in LH8723 |
>   s subset G8723 and H8723 subset s};
true
```

30.8.2 The Low Index Subgroups Process

LowIndexProcess(G, R) operates similarly to **LowIndexSubgroups**, and has the same parameters, except that after MAGMA calculates each subgroup class it stops and returns control to the user, allowing the user to decide whether or not to find another subgroup class. The function returns a *process* P, not a subgroup class, and other intrinsics are used to manipulate this process.

For instance, the following line creates a process for the situation described in the example above:

```
> PH8723 := LowIndexProcess(G8723, 448: Subgroup := H8723);
> print PH8723;
Low Index Subgroups Process
```

The function **ExtractGroup**(P) is required in order to obtain the current conjugacy class from the process. In the statement below, the identifier *nextC* is assigned the new conjugacy class representative:

```
> nextC := ExtractGroup(PH8723); print nextC;
Finitely presented group nextC on 3 generators
Index in group G8723 is 7
Generators as words in group G8723
    nextC.1 = a^2
    nextC.2 = a^-1 * b
    nextC.3 = b * a^-1 * b^-2
```

Notice that the low index process does not return the subgroup of index 1 first; it saves it to be the last class.

The function **ExtractGenerators**(P) returns the set of generators of this current class:

```
> gens := ExtractGenerators(PH8723); print gens;
{ a^2, a^-1 * b, b * a^-1 * b^-2 }
```

ExtractGroup and **ExtractGenerators** only give information about the current conjugacy class. If the user decides to advance along the process

to the next class, the procedure **NextSubgroup**($\tilde{}P$) must be invoked. The tilde symbol ~ is necessary since the procedure modifies the process P:

```
> NextSubgroup(~PH8723);
> nextC := ExtractGroup(PH8723); print nextC;
Finitely presented group nextC on 3 generators
Index in group G8723 is 7
Generators as words in group G8723
    nextC.1 = a^2
    nextC.2 = a^-1 * b
    nextC.3 = a * b^-2 * a^-1 * b * a^-1
```

These operations should be continued until the user has found all the desired conjugacy classes of low index subgroups.

If all of the classes are extracted, then the process becomes empty. The following example demonstrates this on a very small group:

```
> G3 := SymmetricGroup(GrpFP, 3);
> P3 := LowIndexProcess(G3,7);
> print ExtractGroup(P3);
Finitely presented group on 1 generator
Index in group G3 is 2
Generators as words in group G3
    $.1 = G3.1
> NextSubgroup(~P3);
> print ExtractGroup(P3);
Finitely presented group on 1 generator
Index in group G3 is 3
Generators as words in group G3
    $.1 = G3.2
> NextSubgroup(~P3);
> print ExtractGroup(P3);
Finitely presented group on 1 generator
Index in group G3 is 6 = 2 * 3
Generators as words in group G3
    $.1 = Id(G3)
> NextSubgroup(~P3);
> print ExtractGroup(P3);

>> print ExtractGroup(P3);
                   ^
Runtime error in 'ExtractGroup': Argument 1 is not non-empty
```

To avoid this eventual error message, the user may prefer to test the process each time it is modified by means of the function **IsEmpty**(P). This function returns **true** if there are no more classes left. **IsEmpty**(P) is particularly useful if the examination of the process forms part of a user-defined function/procedure, rather than being undertaken interactively.

30.9 Finite Groups

There are two important problems relating to finite fp-groups. The first, naturally, is to decide whether or not a particular fp-group G is finite. Secondly, if a group is known to be finite, it is often desirable to construct a faithful representation for it in some group category having an element normal form so that the structure of the group may be investigated in detail.

30.9.1 Determining the Order

Proving that a group is finite is often quite difficult. The function **Order**(G) tries to deduce the order of G by reducing the problem to enumerating cosets of some group over its trivial subgroup. If it is unable to determine the order, it returns zero. **Order** takes the same parameters as **Index**. The user should be aware that by default, **Order** the memory allocation for the coset enumeration is quite small and for larger or more difficult finite groups, the user may have to reset the parameter **CosetLimit** to a much higher value. More likely, the user will need to adopt a more sophisticated approach.

The standard approach to this problem involves constructing "by hand" a sequence of subgroups

$$G \geq H_1 \geq H_2 \ldots \geq H_r,$$

where H_r is the trivial subgroup, and showing that each subgroup has finite index in its larger neighbour. In fortunate circumstances, it may be possible to identify a subgroup of known order from the presentation defining G. Otherwise, the subgroups in the chain are constructed in various ways. For example, if G is not perfect, the first subgroup will usually be the kernel of a homomorphism corresponding to a quotient obtained using one of the techniques discussed in Section 30.3.

Throughout this chapter, various groups have been shown to be infinite by demonstrating that they possess an infinite section. Another technique for proving that a group is infinite involves trying to construct a homomorphism onto a known infinite group. Producing such a proof is very much an art and many familiar groups such as the Fibonacci group $F(2,9)$ ([Joh90]) resisted attempts to establish their infinite order for many years.

30.9.2 Constructing a Representation

If G is known to be finite, there are various approaches to constructing a group K in a category with an element normal form, that is isomorphic to G. If G is known to be a p-group, the p-quotient algorithm may be used to construct a pc-group K. If G is known to be soluble, the soluble quotient algorithm may be used in a similar manner. Otherwise, it will be necessary to construct a permutation representation for G over a core-free subgroup. In a future release of MAGMA, it will be possible to represent the multiplication table of G by means of a Cayley graph.

31. Abelian Groups

Abelian groups are groups in which all the elements commute with one another. Because of this, it is customary to write expressions within an abelian group additively. MAGMA uses $a+b$, $-a$ and 0 in its implementation of abelian groups, whereas it uses $a*b$, $a\hat{}-1$ and 1 in all other group categories.

31.1 Constructing the Free Abelian Group

The free abelian group F of rank n is the abelian group that has n generators, with no relations on them. The function for creating this group is **FreeAbelianGroup**(n). For example:

```
> F4 := FreeAbelianGroup(4);
> print F4;
AbelianGroup isomorphic to 4 Z
Defined on 4 generators (free)
```

The category of abelian groups is **GrpAb**.

The i^{th} generator of *F4* can be obtained as *F4*.i. However, it is generally easier to give special names to the generators using the angle bracket notation:

```
> F4<w, x, y, z> := FreeAbelianGroup(4);
```

The letters w, x, y, z behave both as alternative identifier names and as printnames for the generators.

31.2 Computing with Abelian Group Elements

The identity element of an abelian group A is the additive identity. The expressions $A!0$ or **Id**(A) return this element, and **IsId**(a) tests whether the element a is the identity element:

```
> print Id(F4);
0
> print IsId(F4!0);
true
```

A general element or word a in an abelian group A with n generators has the form $a = a_1e_1 + \cdots + a_re_r$, where the a_i are integers and the e_i are the generators. There are two main ways to create a in MAGMA, either as an expression in the generators, using + and * signs, or as an integer sequence $[a_1, \ldots, a_n]$ that is coerced into A. For instance, the following statements create the element $g = 4w - 5x + z$ of *F4* in each way, and test that the results are the same:

```
> g := 4*w - 5*x + z;
> print g;
4*w - 5*x + z
> print F4![4,-5,0,1] eq g;
true
```

If A has only one generator, e_1, then $A!a_1$ may be used as an alternative to $A![a_1]$:

```
> print FreeAbelianGroup(1)!8;
8
```

The function **ElementToSequence**(a) or **Eltseq**(a) returns the sequence $[a_1, \ldots, a_n]$ corresponding to the element $a = a_1e_1 + \cdots + a_re_r$:

```
> print ElementToSequence(g);
[ 4, -5, 0, 1 ]
```

Addition is the main arithmetic operation on abelian group elements. Therefore the operators + and − are available. It is also possible to find the sum of m copies of an element a, using the expression m*a. (If m is negative, this expression returns the sum of $|m|$ copies of $-a$.) For instance:

```
> print 5*g, -5*g;
20*w - 25*x + 5*z
-20*w + 25*x - 5*z
```

31.3 Subgroups and Finitely-Presented Groups

The **sub**-constructor may be used to create a subgroup of an abelian group. For example, the subgroup of *F4* generated by g and $y + z$ may be created as follows:

```
> F4s := sub< F4 | g, y+z >;
> print F4s;
AbelianGroup isomorphic to 2 Z
Defined on 2 generators (free)
```

The **quo**-constructor is used to create quotients. These quotients are called finitely-presented abelian groups, because they are presented as abelian groups in terms of a finite number of generators and relations. The relations are listed on the right side of the constructor, either in the explicit form $r_1 = r_2 = \ldots r_k$, where the r_i are elements of the group, or as individual elements r_i (interpreted as the relator $r_i = 0$). Compare the syntax for finitely-presented groups, explained in Section 30.2. For instance, the quotient *F4q* of *F4* defined by the relations $28w - 35x + 7z = 0$ and $w - 7z = 2y$ may be created as follows:

```
> F4q<s,t,u,v> := quo<F4 |
>     28*w - 35*x + 7*z, w - 7*z = 2*y>;
> print F4q;
AbelianGroup isomorphic to Z_7 + 2 Z
Defined on 4 generators
Relations:
    s - 2*u - 7*v = 0
    35*t - 56*u - 203*v = 0
```

As usual, the **quo** constructor can also return the natural homomorphism as its second return value.

The assignment statement above gives the identifier names s, t, u, v to the four generators of *F4q*. However, these are not their printnames. When the generators are printed, the output is an expression in s, t, u, v, not simply the generator name:

```
> print t;
5*s + t - 10*u - 35*v
> print t eq (5*s + t - 10*u - 35*v);
true
```

Here is a difference between the behaviour of abelian groups (category **GrpAb**) and general finitely-presented groups (category **GrpFP**). It arises

because elements of finitely-presented groups cannot be tested for equality whereas abelian group elements can be. Thus finitely-presented groups do not have a canonical form but abelian group elements do. Therefore every element of an abelian group is always printed in the same way. In the group *F4q*, the printed forms of the generators are as follows:

```
> print Generators(F4q);
{
    35*s - 35*t - 12*u - 35*v,
    5*s + t - 10*u - 35*v,
    u,
    s - 2*u - 6*v
}
```

The function **Relations** returns a sequence containing the relations of the group. However, the generators in this sequence are printed in the default form of `$.`i, since they are regarded as belonging not to the group itself but to the free abelian group with the same number of generators. For example:

```
> R := Relations(F4q);
> print R;
[ $.1 - 2*$.3 - 7*$.4 = 0, 35*$.2 - 56*$.3 - 203*$.4 = 0 ]
> print Parent(LHS(R[1]));
Abelian Group isomorphic to 4 Z
Defined on 4 generators (free)
```

If the group is free, then by definition it has no relations, so this sequence will be empty:

```
> print Relations(F4);
[]
```

Since finitely-presented abelian groups are often required, there is a special way of constructing them without explicitly constructing a free abelian group first. The constructor to use is

AbelianGroup< $e_1, \ldots, e_i$ | relations in the e_i **>**

where the e_i are the generators. For instance, the group below, which is isomorphic to *F4q*, is created without explicitly creating *F4* first:

```
> F4qB<s, t, u, v> := AbelianGroup< s, t, u, v |
    28*s - 35*t + 7*v, s - 7*v = 2*u >;
> print F4qB;
AbelianGroup isomorphic to Z_7 + 2 Z
```

```
Defined on 4 generators
Relations:
    s - 2*u - 7*v = 0
    35*t - 56*u - 203*v = 0
```

The **AbelianGroup** constructor only forms the quotient of a free abelian group. The **quo**-constructor must be used for constructing the quotient of an abelian group that is not free. For instance:

```
> F4qBq<h,i,j,k> := quo< F4qB | 8*s + t = 2*v = u - 9*t >;
> print F4qBq;
AbelianGroup isomorphic to Z_5341
Defined on 4 generators
Relations:
    h + 5046*k = 0
    i + 2358*k = 0
    j + 5197*k = 0
    5341*k = 0
```

31.4 Structure of Abelian Groups

Let C_{n_i} be the cyclic group of order n_i and C_∞ be the infinite cyclic group. Then every finitely-generated abelian group A is isomorphic to exactly one group of the form

$$C_{n_1} \oplus C_{n_2} \oplus \cdots \oplus C_{n_r} \oplus C_\infty \oplus \cdots \oplus C_\infty,$$

where there are $t \geq 0$ copies of C_∞ and the order n_i divides n_{i+1} for $1 \leq i < r$. The sequence $[n_1, \ldots, n_r]$ is called the sequence of torsion invariants of A, and the integer t is called the torsion-free rank of A. Conversely, each such direct sum is isomorphic to a finitely-generated abelian group. Therefore finitely-generated abelian groups are completely characterized by their sequence of torsion invariants and their torsion-free rank t.

Given a finitely-generated abelian group A, **TorsionInvariants**(A) returns the sequence of torsion invariants, and **TorsionFreeRank**(A) returns t. A related function, **Invariants**(A), returns a sequence containing the torsion invariants followed by t copies of zero, where the zeros represent the infinite cyclic groups in the direct sum. For example:

```
> A<a,b,c,d,e,f,g,h,i> := AbelianGroup< a,b,c,d,e,f,g,h,i |
>     2*a, 6*b, 6*c, 6*d, 30*e, 60*f >;
> print A;
```

```
Abelian Group isomorphic to
Z_2 + Z_6 + Z_6 + Z_6 + Z_30 + Z_60 + 3 Z
Defined on 9 generators
Relations:
    2*a = 0
    6*b = 0
    6*c = 0
    6*d = 0
    30*e = 0
    60*f = 0
> print TorsionInvariants(A);
[ 2, 6, 6, 6, 30, 60 ]
> print TorsionFreeRank(A);
3
> print Invariants(A);
[ 2, 6, 6, 6, 30, 60, 0, 0, 0 ]
```

Two important subgroups of A are the torsion subgroup and the torsion-free subgroup. In general, a torsion group is one in which all elements are torsion elements, and a torsion-free group is one in which no non-zero element is a torsion element. (An element a is a torsion element if there exists an integer k such that $ka = 0$.) The functions **TorsionSubgroup**(A) and **TorsionFreeSubgroup**(A) return the largest possible subgroups of A of these kinds, namely $C_{n_1} \oplus C_{n_2} \oplus \cdots \oplus C_{n_r}$ and $C_\infty \oplus \cdots \oplus C_\infty$ (i.e., the free group on t generators). For instance:

```
> print TorsionSubgroup(A);
Abelian Group isomorphic to
Z_2 + Z_6 + Z_6 + Z_6 + Z_30 + Z_60
Defined on 6 generators
Relations:
    2*$.1 = 0
    6*$.2 = 0
    6*$.3 = 0
    6*$.4 = 0
    30*$.5 = 0
    60*$.6 = 0
> print TorsionFreeSubgroup(A);
Abelian Group isomorphic to 3 Z
Defined on 3 generators (free)
```

The functions above explain how to decompose a finitely-generated abelian group A as a direct sum of cyclic groups with respect to torsion properties. Another way to decompose A as a direct sum of cyclic groups is to write it in the form

$$C_{p_1^{\alpha_1}} \oplus C_{p_2^{\alpha_2}} \oplus \cdots \oplus C_{p_s^{\alpha_s}} \oplus C_\infty \oplus \cdots \oplus C_\infty,$$

where the p_i are (not necessarily distinct) primes, and there are once again $t \geq 0$ copies of C_∞. The sequence $[p_1, \ldots, p_s]$ is returned by the function **PrimaryInvariants**(A). For example:

```
> print PrimaryInvariants(A);
[ 2, 2, 2, 2, 2, 4, 3, 3, 3, 3, 3, 5, 5 ]
```

The invariants in this sequence that are powers of a given prime p are called the p-primary invariants of A. The function **pPrimaryInvariants**(A, p) returns these invariants in a sequence, and **pPrimaryComponent**(A, p) returns the subgroup of A corresponding to them. For example:

```
> print pPrimaryInvariants(A, 2);
[ 2, 2, 2, 2, 2, 4 ]
> print pPrimaryComponent(A, 5);
Abelian Group isomorphic to Z_5 + Z_5
Defined on 2 generators
Relations:
    5*$.1 = 0
    5*$.2 = 0
```

Given the invariants of a finitely-generated abelian group, it is possible to construct the corresponding group from them, using a version of the function **AbelianGroup**. The invariants must be given as an sequence of non-negative integers, where a zero denotes an infinite cyclic component. For instance, the following group is isomorphic to A:

```
> print AbelianGroup([2, 6, 6, 6, 30, 60, 0, 0, 0]);
Abelian Group isomorphic to
Z_2 + Z_6 + Z_6 + Z_6 + Z_30 + Z_60 + 3 Z
Defined on 9 generators
Relations:
    2*$.1 = 0
    6*$.2 = 0
    6*$.3 = 0
    6*$.4 = 0
    30*$.5 = 0
    60*$.6 = 0
```

It is also possible to use the function **AbelianGroup** to construct an abelian group in a group category other than **GrpAb**. The function should be called in the form **AbelianGroup**(C, Q), where C is the desired category and Q is the sequence of invariants. In these other categories (except for **GrpFP**), none of the invariants may be zero. For instance:

```
> print AbelianGroup(GrpPerm, [ 2, 6, 6, 6, 30, 60 ]);
Permutation group acting on a set of cardinality 110
Order = 777600 = 2^7 * 3^5 * 5^2
```

Table 31.1 (p. 634) summarizes these structure functions.

Table 31.1. Structure of abelian groups

MAGMA	Meaning
Invariants(A)	Invariants of abelian group A, as sequence of the orders of the cyclic factors (entry 0 indicates C_∞)
TorsionFreeRank(A)	Torsion-free rank of A
TorsionFreeSubgroup(A)	Torsion-free subgroup of A
TorsionInvariants(A)	Torsion invariants of A
TorsionSubgroup(A)	Torsion subgroup of A
PrimaryInvariants(A)	Primary invariants of A
pPrimaryInvariants(A, p)	p-primary invariants of A
pPrimaryComponent(A, p)	p-primary component of A
AbelianGroup(Q)	Abelian group in category **GrpAb** with invariant sequence Q (entry 0 indicates C_∞)
AbelianGroup(C, Q)	Abelian group in category C given by invariant sequence Q

31.5 Further Operations and Access Functions

Chapter 29 explains many operations and access functions on groups and elements that are common to several of the group categories, including **GrpAb**. They include **Order**, **Index**, the **ncl** constructor and the coset functions. See that chapter for more information. Note, however, that some of the computations involved in evaluating these functions are trivial in the case of abelian groups. For example, if A is abelian then **Centre**(A) will always equal A.

One of these generic functions does have a different name in the abelian group category, for an obvious reason: **DirectProduct** is called **DirectSum**. For instance:

```
> DS := DirectSum(A, F4q); print DS;
Abelian Group isomorphic to Z_2 + Z_6 + Z_6 + Z_6 + Z_30 +
    Z_420 + 5 Z
Defined on 11 generators
```

```
Relations:
    2*$.1 = 0
    6*$.2 = 0
    6*$.3 = 0
    6*$.4 = 0
    30*$.5 = 0
    420*$.6 = 0
```

Finally, it is possible to use MAGMA to obtain a random element of an abelian group A. If A is finite, then **Random**(A) will do the task. Otherwise, it is necessary to create a *process* to generate randomly chosen elements from A, by means of the function **RandomProcess**(A). (This function has two parameters, but they will not be explained here; refer to the *Handbook* or the online help system.) Once a random process P for A has been created, pseudo-random elements of A are obtained by calling **Random**(P) repeatedly. For example:

```
> P := RandomProcess(DS);
> print P;
Random element process
> print Random(P);
DS.1 + 2*DS.2 + 4*DS.3 + 3*DS.4 + 3*DS.5 + 179*DS.6 +
180*DS.7 + 479*DS.8 + 803*DS.9 + 791*DS.10 + 298*DS.11
> print Random(P);
DS.1 + 4*DS.2 + 2*DS.3 + 5*DS.4 + 26*DS.5 + 143*DS.6 +
143*DS.7 + 364*DS.8 + 627*DS.9 + 609*DS.10 + 222*DS.11
> print Random(P);
4*DS.3 + 4*DS.4 + 8*DS.5 + 2*DS.6 + 2*DS.7 + 7*DS.8 +
10*DS.9 + 10*DS.10 + 4*DS.11
```

32. Permutation Groups

The development of algorithms for computing the structure, representations and extensions of finite permutation groups has been one of the great achievements of Computational Algebra. Almost all of these algorithms are available in MAGMA, and work is continuing on the development of new or improved algorithms.

The MAGMA approach to a permutation group emphasizes its actions. Actions on G-invariant subsets and partitions of the points on which G acts yield reductions to smaller groups. More generally, many permutation groups arise as automorphism groups of some geometric or combinatorial configuration, where interest focuses on how the automorphism group acts on different sets associated with the configuration. In order to be able to consider the action of a single permutation group on many different sets simultaneously, MAGMA makes use of the notion of a *G-set*, whereby a permutation group defined on a set X may have actions specified on other sets.

This chapter concentrates on methods specific to permutation groups, since those functions that realize general group constructions (such as the computation of centralizers) were described in Chapter 29. After some material on constructing permutation groups, the three following sections are all concerned with actions: on G-invariant subsets of the natural G-set, on G-invariant partitions of the natural G-set, and on a general G-set. Next follows a section on primitive groups that presents the techniques available for analyzing a group according to the O'Nan-Scott Theorem. The chapter concludes with a very brief account of the methods available for constructing a base and strong generating set (BSGS), the standard computational representation of the set of elements for a permutation group. *Note that the discussion is simplified: not every function is listed, the optional parameters for some functions are ignored, and the suitability of certain functions for large groups is only briefly mentioned.* Details are available in the *Handbook*, especially techniques for working with very large groups.

The reader is referred to [DiM96] for background theory and terminology.

32.1 Group Actions, Permutation Groups and G-Sets

Let G be a group and X be a set. An *action* of G on X is a mapping $\phi : X \times G \to X$ such that

$$((x, g)\phi, h)\phi = (x, gh)\phi \qquad \text{and} \qquad (x, 1_G)\phi = x$$

for all $x \in X$ and all $g, h \in G$. It is usual to write x^g for $(x, g)\phi$ when there is no risk of confusion.

Let $\mathrm{Sym}(X)$ denote the group of all bijections $X \to X$. A *permutation group* on the set X is a subgroup of $\mathrm{Sym}(X)$. Such a group G has a natural action ϕ on X, whereby $(x, g)\phi$ is the image of $x \in X$ under the permutation $g \in G$. If H is an arbitrary group, a *permutation representation* of H on X is a homomorphism $\rho : H \to \mathrm{Sym}(X)$. It is not difficult to show that there is a one-to-one correspondence between actions of the group H on the set X and the permutation representations of H on the set X. (See [NST94] for details.)

Following [NST94], a set X together with an action of some group on it will be termed a *G-set*, and its elements will be called *points*. Frequently, the set X will be referred to as a G-set for the group G, where the action of G on X is implicit. In the computational context, the G-set X on which the permutation group G is originally defined is called the *natural G-set* for the group G, and the action of G on X is called the *natural action* of G. Note that the group G also has an induced action on the G-closure of any *derived G-set* of X, such as a G-invariant subset of X.

In Magma, every natural G-set must be finite. The cardinality n of the natural G-set X of a permutation group G is called the *degree* of G. Usually, X will be $\{1, 2, \ldots, n\}$, known as the *standard G-set*, but X may be a set of strings, or any other legitimate Magma set. Objects associated with G are represented externally in terms of the points of X. If X is not the standard G-set, then internal computations are performed as if G were the isomorphic permutation group defined on the standard G-set with the same cardinality as X, although this is transparent to the user.

32.2 The Symmetric Group and Permutations

32.2.1 Creating the Symmetric Group

In Magma, the user has the choice of creating a permutation group G that acts either on the standard G-set $\{1, 2, \ldots, n\}$, or on a general (finite) set.

(In most of the examples in this chapter, the groups will be defined on the standard G-set.) In either case, this G-set is called the natural G-set of G, and is returned by the function **GSet**(G).

The group Sym(n) consisting of all bijections of the set $\{1, 2, \ldots, n\}$ is called the *symmetric group on n elements*, and is created by the MAGMA function **Sym**(n). For example, the assignment statement below assigns the symmetric group on 5 elements to the identifier *s5*:

```
> s5 := Sym(5);
> print s5;
Symmetric group s5 acting on a set of cardinality 5
Order = 120 = 2^3 * 3 * 5
> print GSet(s5);
GSet{ 1 .. 5 }
```

To create a symmetric group acting on a non-standard G-set, the appropriate function is **Sym**(T), where T is an enumerated set or indexed set. The group is constructed with a G-set corresponding to T. For instance, the following line constructs the symmetric group acting on the set {"apple", "pear", "fig"}:

To perform this example online, type `load "I96c32e1";`

```
> fruits := Sym({"apple", "pear", "fig"});
> print fruits;
Symmetric group fruits acting on a set of cardinality 3
Order = 6 = 2 * 3
> print GSet(fruits);
GSet{ fig, apple, pear }
```

32.2.2 Representation of Permutations

Traditionally, mathematicians represent permutations using two different notations: as a *mapping* and as a *product of disjoint cycles*. For example, the element $a \in S_5$ which fixes the point 4 and interchanges the pairs 1,3 and 2,5 may be written either as the mapping $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 5 & 1 & 4 & 2 \end{pmatrix}$ or as the product of disjoint cycles $(1, 3)(2, 5)$. MAGMA offers both forms of notation: the mapping form, represented as a sequence of the n distinct elements of the G-set, in some permutation; and the disjoint cycle form, written in the familiar cycle notation (except that cycles of length one must be omitted). Whichever notation is employed, it must be preceded by the group and a !

symbol, so that it can be coerced appropriately. For example, the element above may be created in MAGMA in either of the following ways:

```
> a := s5 ! [3, 5, 1, 4, 2];
> // or
> a := s5 ! (1, 3)(2, 5);
```

Note that the representation of a permutation is always in terms of the elements of its natural G-set, even if that is not the standard G-set. For instance, the following assignment creates the permutation that exchanges `"pear"` and `"fig"`:

```
> fruitexchange := fruits ! ("pear", "fig");
```

The user may choose freely between the mapping and cycle forms. The cycle form tends to be used frequently, since it is compact and it displays the structure of the permutation. On the other hand, the mapping form can be useful for building permutations since the sequence constructor can be used; for an example, see the function *ShuffleGroup* given on p. 173.

Permutations are always printed in cycle form:

```
> print a;
(1, 3)(2, 5)
> print s5 ! [1, 2, 5, 3, 4];
(3, 5, 4)
> print fruitexchange;
(fig, pear)
```

The identity permutation is represented by $\mathbf{Id}(G)$ or $G!1$, where G is the name of the permutation group.

32.2.3 Arithmetic on Permutations

The multiplication operator for permutations in MAGMA is `*`, and `^` denotes exponentiation. MAGMA multiplies permutations left-to-right, as the following example shows:

```
> b := s5!(1, 2, 3, 4);
> print a*b, b*a, b^2, b^-1;
(1, 4)(2, 5, 3)
(1, 5, 2)(3, 4)
(1, 3)(2, 4)
(1, 4, 3, 2)
```

For details of other generic arithmetic operations in groups, including element order, element conjugates, and commutators, see Section 29.2.

The function **IsEven**(g) returns **true** if g is an even permutation. For instance:

```
> print IsEven(s5!(1, 5)(2, 4, 3)), IsEven(s5!(1, 5)(2, 4));
false true
```

IsOdd(g) behaves analogously.

The function **CycleStructure**(g) may be used for determining the structure of the cycles of a permutation g. It returns a sequence of pairs in which the first component of each tuple is a cycle length and the second component is the number of cycles with that length. For example, the output below indicates that the argument has one cycle of length 5, two cycles of length 2, and 4 fixed points:

```
> print CycleStructure(Sym(13) ! (2,1)(9,4,7,3,12)(8,10));
[ <5, 1>, <2, 2>, <1, 4> ]
```

Table 32.1. Invariants for permutations

MAGMA	Meaning
CycleStructure(g)	Sequence of 2-tuples of form `<` i, n_i `>`, where $n_i > 0$, indicating that g has n_i cycles of length i
Fix(g)	$\{a \in X : a^g = a\}$
Support(g)	$\{a \in X : a^g \neq a\}$
Degree(g)	Cardinality of the support of g, i.e., the number of points moved by g
IsEven(g)	**true** if g is an even permutation
IsOdd(g)	**true** if g is an odd permutation

These and other invariants for permutations are summarized in Table 32.1. In this table the permutation g is assumed to be an element of a group with natural G-set X.

32.3 General Permutation Group Constructions

32.3.1 Subgroups and the Permutation Group Constructor

A permutation group H on the set X is a subgroup of the symmetric group $\mathrm{Sym}(X)$, and is commonly described in terms of generating permutations. In MAGMA, H may be created using either of two constructors: the **sub**-constructor or the special **PermutationGroup** constructor.

If the **sub**-constructor is used, then the symmetric group should be placed on the left side and the generators on the right side. For example, consider the subgroup of the symmetric group on $\{1,\ldots,6\}$ generated by $\{(1,2,5,6),(3,4),(1,6)(2,5)\}$. It may be created using the submagma constructor as follows:

```
> s6 := Sym(6);
> s6subgp := sub<s6 | (1, 2, 5, 6), (3, 4), (1, 6)(2, 5)>;
> print s6subgp;
Permutation group s6subgp acting on a set of cardinality 6
    (1, 2, 5, 6)
    (3, 4)
    (1, 6)(2, 5)
```

Alternatively, if no reference need be made to the parent symmetric group, then the **PermutationGroup** constructor is suitable. It has two versions:

PermutationGroup< n | generators >
PermutationGroup< T | generators >

where the first version is used if the group has the standard G-set $\{1,\ldots,n\}$, and the second version is used if the G-set of the group corresponds to the enumerated or indexed set T. For instance:

```
> gp := PermutationGroup<6 | (1,2,5,6), (3,4), (1,6)(2,5)>;
> print gp eq s6subgp;
true

> fr := {"apple", "pear", "fig"};
> print PermutationGroup< fr | ("pear", "fig") >;
Permutation group acting on a set of cardinality 3
    (fig, pear)
```

Of course, the **sub**-constructor may be used to create a subgroup of any group, not just that of the symmetric group. For instance, it may be used

to define the subgroup of *s6subgp* generated by $(1,6)(2,5)$ and the third generator of *s6subgp*:

```
> print sub< s6subgp | (1, 5)(2, 6), s6subgp.3 >;
Permutation group acting on a set of cardinality 6
    (1, 5)(2, 6)
    (1, 6)(2, 5)
```

32.3.2 Standard Groups

Certain permutation groups arise so frequently that standard functions are provided for their creation. Some of these functions may be found in Section 29.1.3. In addition, the functions **Sym**(n) and **Alt**(n) provide succinct ways to create the symmetric and alternating groups of degree n.

The following line constructs the cyclic group of order 15 as a permutation group:

```
> a9 := CyclicGroup(GrpPerm, 15);
> print a9;
Permutation group a9 acting on a set of cardinality 15
Order = 15 = 3 * 5
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

32.3.3 Projective and Affine Groups

The functions listed in Table 32.2 construct various affine and projective groups associated with an n-dimensional vector space V over a finite field K of cardinality q. For some functions, there are restrictions on the values of the arguments; each of the projective group functions require n to be at least 2, for instance. Note also that in the case of the unitary groups, $\mathrm{PGU}(n,q)$ and $\mathrm{PSU}(n,q)$, the group is actually defined over the finite field $K = \mathrm{GF}(q^2)$. With the exception of the projective Suzuki group (which takes as its single argument an odd power of 2) these functions take any of the following three patterns of arguments: n and q; n and K; or V only. In the table only the first form is shown.

Each function returns two values: the designated group G acting on the standard G-set $X = \{1, \ldots, N\}$ (where N is the degree of the group) and an indexed set P of vectors of V that define the objects in V that correspond to X. For example, if G is an affine group, then $P[i]$ is the vector corresponding to $i \in X$. Similarly, if G is a projective group, then the vector $P[i]$ generates the one-dimensional subspace of V corresponding to $i \in X$.

The standard abbreviations for these groups are available, although the greek letters Γ, Σ and Ω must be spelt out. Hence, **PSL**(n, q) may be used in place of **ProjectiveSpecialLinearGroup**(n, q)), and **PGammaU**(n, q) in place of **ProjectiveGammaUnitaryGroup**(n, q).

Table 32.2. Projective and affine groups

MAGMA	Meaning
AffineGammaLinearGroup(n, q)	$A\Gamma L(n, q)$
AffineSigmaLinearGroup(n, q)	$A\Sigma L(n, q)$
AffineGeneralLinearGroup(n, q)	$AGL(n, q)$
AffineSpecialLinearGroup(n, q)	$ASL(n, q)$
ProjectiveGammaLinearGroup(n, q)	$P\Gamma L(n, q)$, $n \geq 2$
ProjectiveSigmaLinearGroup(n, q)	$P\Sigma L(n, q)$, $n \geq 2$
ProjectiveGeneralLinearGroup(n, q)	$PGL(n, q)$, $n \geq 2$
ProjectiveSpecialLinearGroup(n, q)	$PSL(n, q)$, $n \geq 2$
ProjectiveGammaUnitaryGroup(n, q)	$P\Gamma U(n, q)$, $n \geq 2$
ProjectiveSigmaUnitaryGroup(n, q)	$P\Sigma U(n, q)$, $n \geq 2$
ProjectiveGeneralUnitaryGroup(n, q)	$PGU(n, q)$, $n \geq 2$
ProjectiveSpecialUnitaryGroup(n, q)	$PSU(n, q)$, $n \geq 2$
ProjectiveSigmaSymplecticGroup(n, q)	$P\Sigma Sp(n, q)$, $n \geq 4$ and even
ProjectiveSymplecticGroup(n, q)	$PSp(n, q)$, $n \geq 4$ and even
ProjectiveOmegaPlusGroup(n, q)	$P\Omega^+(n, q)$, $n \geq 4$ and even
ProjectiveOmegaMinusGroup(n, q)	$P\Omega^-(n, q)$, $n \geq 4$ and even
ProjectiveOmegaGroup(n, q)	$P\Omega(n, q)$, $n \geq 5$ and odd
ProjectiveSuzukiGroup(q)	$PSz(q)$, q an odd power of 2

For example:

```
> K<w> := GF(3^2);
> p33, pts := ProjectiveGammaUnitaryGroup(3, 3);
> print Degree(p33);
28
> print pts;
{@
    (  1   0   0), (  1 w^7 w^3), (  1 w^6 w^7),
    (  1   w   2), (  1 w^5   w), (  1 w^5 w^3),
    (  1   1 w^7), (  1 w^2 w^5), (  1   1   1),
    (  1 w^7   w), (  1 w^3   2), (  1   2 w^5),
    (  1   2 w^7), (  1   1 w^5), (  1   w   w),
    (  1 w^7   2), (  1 w^3 w^3), (  1 w^6 w^5),
    (  1 w^2   1), (  1 w^3   w), (  1   0 w^6),
    (  0   0   1), (  1   w w^3), (  1 w^6   1),
```

```
    (  1 w^5   2), (  1 w^2 w^7), (  1   0 w^2),
    (  1   2   1)
@}
> print Order(p33);
12096
```

32.3.4 Product Constructions

Table 32.3. Product constructions for permutation groups

MAGMA	Meaning
DirectProduct(H, K)	Direct product of H and K
DirectProduct(Q)	Direct product of sequence Q of permutation groups
TensorProduct(H, K)	Wreath product with product action of H and K
TensorProduct(Q)	Wreath product with product action of sequence Q of permutation groups
WreathProduct(H, K)	Wreath product with normal action of H and K
WreathProduct(Q)	Wreath product with normal action of the of sequence Q of permutation groups

The standard direct product and wreath product constructions are available in MAGMA. Table 32.3 lists the relevant functions.

For example, the statement below prints the direct product of Sym(8) and Alt(4):

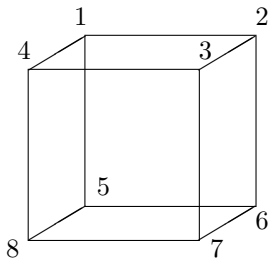
```
> print DirectProduct(Sym(8), Alt(4));
Permutation group acting on a set of cardinality 12
    (1, 2, 3, 4, 5, 6, 7, 8)
    (1, 2)
    (9, 10)(11, 12)
    (9, 10, 11)
```

and the following statements print the 3-fold iterated wreath product of the cyclic group of order 3:

```
> Z3 := CyclicGroup(3);
> W3 := DirectProduct([Z3 : i in [1..3]]);
> print W3;
Permutation group W3 acting on a set of cardinality 27
    (1, 10, 19)(2, 11, 20)(3, 12, 21)(4, 13, 22)(5, 14, 23)
```

```
        (6, 15, 24)(7, 16, 25)(8, 17, 26)(9, 18, 27)
    (1, 4, 7)(2, 5, 8)(3, 6, 9)
    (1, 2, 3)
> print Order(W3);
1594323
```

32.4 Example: The Group of Symmetries of the Cube



Since the group of symmetries of the cube is a convenient example for the next several sections, it will be introduced here. It will be taken in its representation as a group *cube* of permutations of the vertices, numbered 1 to 8 (see figure). This group is the subgroup of Sym(8) generated by $a = (1,2,3,4)(5,6,7,8), b = (2,4,5)(3,8,6), c = (1,5)(2,6)(3,7)(4,8)$, so it can be created in MAGMA as follows:

To perform this example online, type `load "I96c32e2";`

```
> cube<a, b, c> := PermutationGroup< 8 |
>   (1, 2, 3, 4)(5, 6, 7, 8), (2, 4, 5)(3, 8, 6),
>   (1, 5)(2, 6)(3, 7)(4, 8) >;
> print cube;
Permutation group cube acting on a set of cardinality 8
    (1, 2, 3, 4)(5, 6, 7, 8)
    (2, 4, 5)(3, 8, 6)
    (1, 5)(2, 6)(3, 7)(4, 8)
```

32.5 Basic Operations with Permutation Groups

32.5.1 Computing Images

Let X denote the natural G-set of a permutation group G. If x is a point of X, the image of x under the permutation $g \in G$ is usually written in the exponential notation x^g. Consequently, in MAGMA, $x\hat{}g$ denotes the image of x under the permutation g. An alternative notation is **Image**(g, x). For example, the following calculation shows that the point 4 maps to the point 5 when acted on by the permutation ac in *cube*:

```
> print 4^(a*c);
5
```

The ^ operator or **Image** function may also be used to compute the image of a set or sequence of points. For example, the line below computes the image of ac on the sequence of points $[1, 2, 5]$. The resulting sequence shows that 1, 2, 5 map to 6, 7, 2, respectively:

```
> print [1, 2, 5]^(a*c);
[ 6, 7, 2 ]
```

Table 32.4 summarizes the notation for computing images.

Table 32.4. Images under permutations

MAGMA	Meaning
$x\hat{}g$, **Image**(g, x)	Image of point x under permutation g
$S\hat{}g$, **Image**(g, S)	Image of set of points S under permutation g
$Q\hat{}g$, **Image**(g, Q)	Image of sequence of points Q under permutation g

32.5.2 Accessing a Permutation Group

Table 32.5 presents the basic access functions for permutation groups. To interpret these functions correctly, it is important to understand how the natural G-set X for G, the set underlying X, the support of G and its fixed-point set are related. Given a permutation group G with natural G-set X, the function **Set**(X) returns the set underlying X, which is the set on which G is defined. (The G-set and the underlying set will not usually be distinguished here, but since a G-set is a set with structure, it is necessary to apply the

Table 32.5. The permutation representation

MAGMA	Meaning
GSet(G)	Natural G-set X for G
Set(X)	Enumerated set underlying G-set X
Degree(G)	Cardinality of X
Support(G)	$\{a \in X : a^x \neq a$, for some $x \in G\}$
Fix(G)	$\{a \in X : a^x = a$, for all $x \in G\}$
Generators(G)	Set of the defining generators of G
Ngens(G)	Number of defining generators of G
Set(G)	Set of all the elements of a (small) group G
FormalSet(G)	Formal set of all the elements of G
Generic(G)	Symmetric group in which G is naturally embedded

Set functor to it in order to access the underlying set as an enumerated set.) The underlying set may be partitioned into two disjoint subsets: the *support* of G, which is the set of points that are moved by at least one element of G; and the fixed-point set of G, which is the set of points that are not moved by any element of G. The relationship is illustrated with the following group:

```
> G := PermutationGroup< 6 | (1,2,4), (4,5,6), (2,4,5) >;
> X := GSet(G);
> print X;
GSet{ 1 .. 6 }
> om := Set(X);
> print om;
{ 1 .. 6 }
> print Support(G);
{ 1, 2, 4, 5, 6 }
> print Fix(G);
{ 3 }
> print om eq ($1 join $2);
true
```

32.6 Orbits and Transitivity

In this section, the action of G on its natural G-set X will be investigated. At the same time it is convenient to consider the action of G on two derived G-sets: the set of all k-subsets of X, and the set of all sequences of k distinct elements of X.

32.6.1 Transitivity and Regularity

A permutation group G on X is said to be *transitive* if for any pair of distinct points $x, y \in X$, there exists some $g \in G$ such that $x^g = y$. The Boolean function **IsTransitive**(G) returns **true** if G is transitive. For example, the group *cube* defined in Section 32.4 is transitive, since for any given corner of the cube there exists a permutation in *cube* which maps it to any other given corner:

```
> print IsTransitive(cube);
true
```

but the subgroup h of *cube* defined below is intransitive:

```
> h := sub<cube | a^2, b>;
> print h;
Permutation group h acting on a set of cardinality 8
    (1, 3)(2, 4)(5, 7)(6, 8)
    (2, 4, 5)(3, 8, 6)
> IsTransitive(h);
false
```

More generally, given a positive integer k, a permutation group G acting on X is *k-ply transitive* if for any two sequences R and S of k distinct points of X, there exists some $g \in G$ such that $R^g = S$. The maximum value d of k such that G is k-ply transitive is called the *degree of transitivity* of G, and is returned by the function **Transitivity**(G), The function **IsTransitive**(G, k) returns **true** for all $k \leq d$.

As an example, the projective general linear group $\mathrm{PGL}(2, q)$ is 3-ply transitive:

```
> print Transitivity(PGL(2, 9));
3
> print IsTransitive(PGL(2, 9), 2); //  2 le 3   is true
true
```

However, given two arbitrary pairs of corners of a cube, it is not always possible to find a permutation of the vertices that maps one pair to the other, because symmetries preserve distance. Therefore the degree of transitivity of the group *cube* is 1:

```
> print Transitivity(cube);
1
> print IsTransitive(cube, 1) and not IsTransitive(cube, 2);
true
```

Table 32.6. Transitivity and regularity functions

MAGMA	Meaning
IsTransitive(G)	**true** if G is transitive
Transitivity(G)	Degree of transitivity of G
IsTransitive(G, k)	**true** if G acts k-transitively on X
IsSharplyTransitive(G, k)	**true** if G acts sharply k-transitively on X
IsFrobenius(G)	**true** if G is a Frobenius group
IsRegular(G)	**true** if G acts regularly on X
IsSemiregular(G, S)	**true** if G acts semiregularly on the union of orbits S

Table 32.6 lists functions providing information about the transitivity and regularity properties of a permutation group G in its action on its natural G-set X.

32.6.2 Orbits and Stabilizers

The orbit of a point $x \in X$ under a group G is the set x^G of all the images of x. In MAGMA it may be calculated using either $x\hat{}G$ or **Orbit**(G, x), and it is returned as a G-set. For instance, the following lines calculate some orbits of the group *bgp* generated by the element b of *cube*:

```
> bgp := sub< cube | b >;
> print 1^bgp, 2^bgp;
GSet{ 1 }
GSet{ 2, 4, 5 }
```

The output shows that 1 is in an orbit by itself, and that 2 is in an orbit containing 4 and 5 as well.

Just as in the case of images, an analogous notation is available for computing the orbit of a set or sequence. For instance:

```
> print {1, 2}^cube;
GSet{
    { 1, 5 },
    { 2, 6 },
    { 4, 8 },
    { 7, 8 },
    { 5, 6 },
    { 1, 4 },
    { 6, 7 },
```

```
    { 5, 8 },
    { 1, 2 },
    { 3, 7 },
    { 2, 3 },
    { 3, 4 }
}
```

The function **Orbits**(G) returns all the orbits of G, as a sequence of G-sets such that the i^{th} G-set contains the points in the i^{th} orbit. For example:

```
> bgporb := Orbits(bgp);
> print bgporb;
[
    GSet{ 1 },
    GSet{ 2, 4, 5 },
    GSet{ 3, 6, 8 },
    GSet{ 7 }
]
```

Closely related to the idea of an orbit is that of a *stabilizer*. If $x \in X$, the stabilizer G_x of x in G is the subgroup of G consisting of those permutations of G that fix x. The function **Stabilizer**(G, x) returns this subgroup. A standard theorem in permutation group theory states that there is a one-to-one correspondence between the cosets of G_x and the orbits of x. Thus, if $x \in X$ and $g, h \in G$, then $x^g = x^h$ if and only if g and h belong to the same right coset of G_x in G.

The functions and operators pertaining to the orbits and stabilizers that arise from the action of a permutation group G on its natural G-set X are summarized in Table 32.7.

If H is a subgroup of G, then the function **Transversal**(G, H) may be used to analyze the cosets of H in G. Its principal return value is an indexed set T of cardinality $[G : H]$ such that $T[i]$ is a right coset representative for the i^{th} coset of H in G. Its second return value is a mapping $\phi : G \to T$ such that $\phi(g) = T[i]$ if and only if $g \in H \cdot T[i]$. This mapping may be used to find the canonical coset representative for the coset in which a given $g \in G$ lies.

For example, since the orbit of 7 has length 8, the group *cube* can be partitioned into 8 cosets, each of which sends 7 to a different point. The following statements produce canonical coset representatives as an indexed set t:

```
> stab7 := Stabilizer(cube, 7);
> print stab7;
Permutation group stab7 acting on a set of cardinality 8
```

Table 32.7. Orbits and stabilizers

MAGMA	Meaning
$a\hat{}G$, **Orbit**(G, x)	Orbit of point x under G, as a G-set
$S\hat{}G$, **Orbit**(G, S)	Orbit of set S of points under G, as a G-set of subsets of X
$Q\hat{}G$, **Orbit**(G, Q)	Orbit of sequence Q of points under G, as a G-set of sequences of points from X
Orbits(G)	Sequence Q of G-sets, where $Q[i]$ contains the points in the i^{th} orbit of G
IsOrbit(G, S)	**true** if subset S of X is a union of orbits of G (i.e., if S is invariant under G)
Stabilizer(G, x)	Stabilizer in G of point x
Stabilizer(G, S)	Stabilizer in G of set S
Stabilizer(G, Q)	Stabilizer in G of sequence Q
Transversal(G, H)	Given group G and subgroup H, returns (i) indexed set T of elements of G forming a right transversal for G over H (ii) transversal mapping $\phi : G \to T$ such that $\phi(g) = T[i]$ iff $g \in H \cdot T[i]$

```
Order = 6 = 2 * 3
    (2, 4, 5)(3, 8, 6)
    (3, 6)(4, 5)
> t, m := Transversal(cube, stab7);
{@
    Id(cube),
    (1, 2, 6, 5)(3, 7, 8, 4),
    (1, 5)(2, 8)(3, 7)(4, 6),
    (1, 3, 6)(4, 7, 5),
    (1, 4, 3, 7, 6, 5)(2, 8),
    (1, 6)(4, 7),
    (1, 8)(2, 7),
    (1, 7)(2, 8)(3, 5)(4, 6)
@}
Mapping from: GrpPerm: cube to SetIndx: t
```

Notice that the first element of t is a member of *stab7*, so it is the representative for the coset containing the elements of the subgroup *stab7*. By inspection, the second coset sends 7 to 8, the third coset sends 7 to 3, and so on. The following lines demonstrate this in the case of the element c, by means of the transversal mapping m:

```
> print m(c);
(1, 5)(2, 8)(3, 7)(4, 6)
> print Index(t, m(c));
```

```
3
> print 7^c eq $1;
true
```

Table 32.8. Conjugacy of G-set elements

MAGMA	Meaning
IsConjugate(G, x, y)	**true** if $x, y \in X$ are conjugate in G; if **true**, also returns a conjugating element
IsConjugate(G, R, S)	**true** if k-subsets R and S of X are conjugate in G; if **true**, also returns a conjugating element
IsConjugate(G, P, Q)	**true** if sequences P and Q, each containing k elements of X, are conjugate in G; if **true**, also returns a conjugating element

It is sometimes useful to know if two elements $x, y \in X$ belong to the same orbit of G and, if so, to find an element $g \in G$ such that $x^g = y$. The function **IsConjugate**(G, x, y) does precisely this, as illustrated in the code below. More generally, if Y is any G-set for G, then $x, y \in Y$ are said to be *conjugate* in G if there exists an element $g \in G$ such that $x^g = y$. The functions that determine conjugacy of G-set elements for some G-sets related to the natural G-set for G are given in Table 32.8. For example:

```
> cj, g := IsConjugate(stab7, 3, 5);
> print cj;
false
> cj, g := IsConjugate(stab7, 3, 6);
> print cj;
true
> print g;
(2, 5, 4)(3, 6, 8)
> print 3 ^ g;
6
> cj, g := IsConjugate(cube, {2,4}, {3,6});
> print cj;
true
> print g;
(1, 7)(2, 3)(4, 6)(5, 8)
> print {2,4} ^ g;
{ 3, 6 }
```

Table 32.9. The orbits action homomorphism

MAGMA	Meaning
OrbitAction(G, x)	Constructs permutation representation L of G obtained by restricting action of G to x^G. Returns (i) natural homomorphism $f : G \to L$ (ii) induced group L (iii) kernel of the action.
OrbitImage(G, x)	Induced group L which is the image of the above homomorphism
OrbitKernel(G, x)	Kernel of the above homomorphism
OrbitAction(G, S)	Constructs permutation representation L of G obtained by restricting action of G to union of G-orbits of elements of S. See above for return values.
OrbitImage(G, S)	Induced group L which is the image of the above homomorphism
OrbitKernel(G, S)	Kernel of the above homomorphism

32.6.3 The Induced Action on an Orbit

The action of a permutation group G on any G-invariant subset Y of X (that is, a union of orbits) defines a homomorphism of G onto a permutation group H acting on Y. It is often desirable to investigate permutation groups L obtained by restricting the action of G to an orbit or union of orbits. The function **OrbitAction**(G, x) is provided for this purpose. It returns three values: the natural homomorphism from G to L, the induced group L, and the kernel of the action. The functions **OrbitImage**(G, H) and **OrbitKernel**(G, H) return the second and third of these values respectively.

For example, the group *bgp* generated by $b \in$ *cube* is intransitive. The following multiple assignment indicates how to restrict its action to the orbit containing 2:

```
> bgp := sub< cube | b >;
> bhom2, bL2, bK2 := OrbitAction(bgp, 2);
> print bhom2, bL2, bK2;
Mapping from: GrpPerm: bgp to GrpPerm: bL2
Permutation group bL2 acting on a set of cardinality 3
Order = 3
    (1, 2, 3)
Permutation group bK2 acting on a set of cardinality 8
Order = 1
    Id(bK2)
```

Notice that MAGMA considers the induced group *bL2* to be acting on the set $\{1, 2, 3\}$, whereas it originates from an action on the orbit $\{2, 4, 5\}$. The

correspondence between these sets may be found by applying *bhom2* to a sequence consisting of the original points:

```
> print bhom2([2, 4, 5]);
[ 1, 2, 3 ]
```

Therefore 2 maps to 1, 4 maps to 2, and 5 maps to 3.

The functions pertaining to the orbit action are summarized in Table 32.9. As the table indicates, the homomorphism which maps to the representation obtained by restricting the action of G to a union of orbits may also be constructed.

32.7 Systems of Imprimitivity

This section is concerned with G-invariant partitions of X.

32.7.1 Blocks and Block Systems

Suppose that G acts transitively on X. A non-empty subset B of X is a *block* for G if for each $g \in G$ either $B^g = B$ or $B^g \cap B = \phi$. The singleton sets $\{x\}$ for each $x \in X$ are blocks for G, known as the *trivial blocks*. Suppose B is a block for G, and let Y equal $\{B^g : g \in G\}$. Then the sets in Y form a partition of X, and each element of Y is a block for G. The G-invariant partition Y is called a *block system* or *system of imprimitivity* for G, and G has a natural action on Y. In MAGMA, a block system is represented as a G-set whose elements are subsets of X.

If G does not possess a non-trivial system of imprimitivity on X it is said to be *primitive*; otherwise it is said to be *imprimitive*. The function **IsPrimitive**(G) tests for this property. For example, the group *cube* is imprimitive:

```
> print IsPrimitive(cube);
false
```

If S is a subset of X, then **IsBlock**(G, S) returns **true** if S is a block. For example, $\{2, 5\}$ is not a block for *cube*:

```
> print IsBlock(cube, {2, 5});
false
```

The function **MinimalPartition**(G) returns a minimal G-invariant partition (block system), where 'minimal' means that the partition is non-trivial and that there is no other block system whose blocks are proper subsets of these blocks. **MaximalPartition**(G) works similarly.

For example, in the group *cube*, the four space diagonals constitute a minimal partition, and the two regular tetrahedra composed of face diagonals form a maximal partition:

```
> print MinimalPartition(cube);
GSet{
    { 2, 8 },
    { 3, 5 },
    { 4, 6 },
    { 1, 7 }
}
> print MaximalPartition(cube);
GSet{
    { 1, 3, 6, 8 },
    { 2, 4, 5, 7 }
}
```

The function **MinimalPartition**(G) has one parameter, **Block**. (There are no parameters for **MaximalPartition**.) The default value of **Block** is an empty set, but if **Block** is assigned a set of points then the function will return a minimal partition satisfying the condition that all the elements of **Block** are in the same block. For example, it was shown above that $\{2, 5\}$ is not a block in the group *cube*. The following assignment constructs a block system for *cube* in which one block contains $\{2, 5\}$ and is the smallest such block:

```
> block25 := MinimalPartition(cube: Block := {2, 5});
> print block25;
GSet{
    { 1, 3, 6, 8 },
    { 2, 4, 5, 7 }
}
```

In this partition, 2 and 5 are together in the second block.

The function **MinimalPartitions**(G) constructs all the minimal block systems of G, as a sequence of G-sets. Any system of imprimitivity can be written as a union of blocks of one of the minimal systems. For example, the output below shows that *cube* has two minimal block systems:

To perform this example online, type `load "I96c32e3";`

```
> blkscube := MinimalPartitions(cube);
> print blkscube;
[
    GSet{
        { 2, 8 },
        { 3, 5 },
        { 4, 6 },
        { 1, 7 }
    },
    GSet{
        { 1, 3, 6, 8 },
        { 2, 4, 5, 7 }
    }
]
```

Each G-set describes a system of imprimitivity. In this example, the systems happen to be the minimal and maximal partition already constructed.

Table 32.10. Block functions

MAGMA	Meaning
IsPrimitive(G)	**true** if G is primitive
IsBlock(G, S)	**true** if set of points S is a block for G
MinimalPartition(G)	A minimal block system for G; if a set of points is assigned to the parameter **Block**, then one of the blocks must contain that set
MaximalPartition(G)	A maximal block system for G
MinimalPartitions(G)	Sequence containing the minimal block systems of G

The functions pertaining to blocks and block systems are summarized in Table 32.10.

32.7.2 The Induced Action on a Block System

Given any block system Y, **BlocksAction**(G, Y) constructs the homomorphism of G onto the group obtained by the action of G on the blocks of Y. As is the case with the homomorphism functions discussed above, this function returns three values: the natural homomorphism $G \to L$, the induced group L, and the kernel of the action. The functions for obtaining the blocks action are summarized in Table 32.11.

Table 32.11. The blocks action homomorphism

MAGMA	Meaning
BlocksAction(G, Y)	Constructs permutation representation L of G induced by action of G on blocks of Y. Returns (i) natural homomorphism $f : G \to L$ (ii) induced group L (iii) kernel of the action
BlocksImage(G, Y)	Image L of the above homomorphism
BlocksKernel(G, Y)	Kernel of the above homomorphism

For example, when **BlocksAction** is applied to the first system of *blkscube* (defined above), the induced group *cubeL2* is an order-24 subgroup of S_4, so it must be the whole of S_4. The four objects on which *cubeL2* acts are the four blocks in the first minimal system of imprimitivity:

To perform this example online, type `load "I96c32e4";`

```
> cubef2, cubeL2, cubeK2 := BlocksAction(cube, blkscube[1]);
> print cubef2, cubeL2, cubeK2;
Mapping from: GrpPerm: cube to GrpPerm: cubeL2
Permutation group cubeL2 acting on a set of cardinality 4
Order = 24 = 2^3 * 3
    (1, 2, 3, 4)
    (2, 4, 3)
    (3, 4)
Permutation group cubeK2 acting on a set of cardinality 8
Order = 2
    (1, 7)(2, 8)(3, 5)(4, 6)
```

The correspondence between the block system and the G-set of the induced group may be calculated as follows:

```
> print [ < bl, cubef2(bl) > : bl in blkscube[1] ];
[
    <{ 2, 8 }, { 2 }>,
    <{ 3, 5 }, { 3 }>,
    <{ 4, 6 }, { 4 }>,
    <{ 1, 7 }, { 1 }>
]
```

The output above shows that the block $\{2, 8\}$ corresponds to point 2 in the *cubeL2*, the block $\{3, 5\}$ corresponds to point 3 in *cubeL2*, and so on.

32.7.3 Example: A Reduction Algorithm

The following code illustrates the role of actions on orbits and block systems in permutation group algorithms. It implements an unpublished algorithm due to Bill Kantor for constructing the p-core of a permutation group G. Whereas the complete algorithm is designed to find a maximal normal p-subgroup of G, the version presented here determines whether or not G contains a non-trivial elementary abelian normal subgroup, and, if so, constructs it. The algorithm applies **OrbitAction** to reduce to a transitive group T, then **BlocksAction** to reduce to a primitive group P. If the kernel K of the homomorphism $\tau : T \to P$ is non-trivial then the algorithm is called recursively on K. If this does not yield an elementary abelian normal subgroup, the algorithm is applied recursively to $C_G(K)$.

```
pSubgroup := func< G, p |
  sub< G | {x ^ (Order(x) div p): x in Generators(G)} > >;

TransitiveReduction := function(G)
/*
Returns: (i) natural homomorphism phi: G -> H of G onto a
transitive constituent H, (ii) H, (iii) kernel of phi.
*/
    if IsTransitive(G) then
        return IdentityHomomorphism(G), G, sub<G |>;
    else
        return OrbitAction(G, Rep(Support(G)));
    end if;
end function;

PrimitiveReduction := function(G)
/*
Returns: (i) natural homomorphism tau: G -> H of
transitive group G onto a primitive component H,
(ii) H, (iii) kernel of tau.
*/
    S := MaximalPartition(G);
    if IsPrimitive(G) then
        return IdentityHomomorphism(G), G, sub<G |>;
    else
        return BlocksAction(G, S);
    end if;
end function;

AbelianNormalSubgroup := function(G)
/*
```

```
Returns a non-trivial elementary abelian normal subgroup
of G if one exists, otherwise the trivial subgroup.
*/
    if IsTrivial(G) then
        return G;
    end if;
    phi, T, KT := TransitiveReduction(G);
    tau, P, KP := PrimitiveReduction(T);
    K := KP @@ phi;
    if IsTrivial(K) then
        return (EARNS(P) @@ tau) @@ phi;
    else
        Z := Centre(K);
        if not IsTrivial(Z) then
            return pSubgroup(Z, FactoredOrder(Z)[1][1]);
        end if;
        A := $$(K);
        if IsTrivial(A) then
            A := $$(Centralizer(G, K));
            if IsTrivial(A) then
                return A;
            end if;
        end if;
        Z := Centre(NormalClosure(G, A));
        return pSubgroup(Z, FactoredOrder(Z)[1][1]);
    end if;
end function;
```

32.8 General G-sets

32.8.1 Construction of a G-set

It is convenient to distinguish three types of G-set for a permutation group G. As noted at the beginning of the chapter, the set on which G is defined is termed the *natural G-set* and the action of G on X is called the *natural action* of G.

A *general G-set* for the group G is an arbitrary set Y togther with a mapping $\phi : Y \times G \to Y$ satisfying

$$((y, g)\phi, h)\phi = (y, gh)\phi \qquad \text{and} \qquad (y, 1_G)\phi = y$$

for all $y \in Y$ and all $g, h \in G$. Now, let A be some set. A *derived set* of A is defined (recursively) as follows:

– A subset of A is a derived set;

– A set of k-subsets of A, k-sequences of A, or ordered partitions of A is a derived set;

– A subset of a Cartesian product of derived sets of A is a derived set.

The natural action of G on X induces a natural action on the G-closure Y of any derived set of X. Such a set Y is also a G-set. For example, a subset of X is a G-set for G if and only if it is a union of orbits for G. A derived set of the natural G-set X for G will be called a *derived G-set* for G.

For a permutation group G defined on X, MAGMA allows the user to define an arbitrary number of additional G-sets. This mechanism enables the user to work with different actions of G without explicitly having to form the homomorphic image of G with respect to each action. Operations may be specified in terms of G.

In MAGMA, a general G-set may be constructed using the function **GSet**(G, Y, f), where f is a mapping from the Cartesian product of Y and G to Y. If Y' is the closure of Y under f (usually $Y = Y'$), then f must be an action of G on Y', in the sense defined in Section 32.1. The G-set will be constructed on Y'.

To illustrate the use of this function, let V be an n-dimensional vector space defined over GF(2) and let G be a permutation group of degree 2^n defined on the G-set $\{1, 2, \ldots, 2^n\}$. The action of G on V is obtained by establishing a bijection between V and X such that $v \in V$ is mapped to $1 + N(V)$, where $N(v)$ is the integer obtained by taking the components of v to be the digits of the base-2 representation of an integer. The function *VSpaceAction* below produces the required G-set, and returns the corresponding action.

```
VSpaceAction := function(G, V);
    Z := IntegerRing();
    d := Dimension(V);

    VecToInt := func< v |
      1 + &+[ Z!v[i]*(2^(i-1)) : i in [1..d] ] >;
    IntToVec := func< n, V |
      V ! (q cat [ 0 : i in [1..(d-#q)] ])
          where q is IntegerToSequence(n-1, 2) >;

    Y := Set(V);
    Y_Act := map< car<Y, G> -> Y |
      t :-> IntToVec(VecToInt(t[1])^t[2], V) >;
    W := GSet(G, Y, Y_Act);
    return Action(G, W);
```

```
end function;
```

For instance, let V be the 3-dimensional vector space over GF(2) and G be PSL$(2, 7)$:

```
> G := PSL(2, 7);
> V := VectorSpace(GF(2), 3);
> f := VSpaceAction(G, V);
> H := Image(f);
> print H;
Permutation group H acting on a set of cardinality 8
Order = 168 = 2^3 * 3 * 7
    ((0 1 0), (1 0 1), (0 1 1))((1 1 0), (0 0 1), (1 1 1))
    ((0 0 0), (1 1 1), (1 0 0))((1 1 0), (0 0 1), (1 0 1))
> x := Random(G); print x;
(1, 2, 6, 8, 4, 5, 3)
> print [ < v, v^(x@f)> : v in V ];
[
    <(0 0 0), (1 0 0)>,
    <(1 0 0), (1 0 1)>,
    <(0 1 0), (0 0 0)>,
    <(1 1 0), (0 0 1)>,
    <(0 0 1), (0 1 0)>,
    <(1 0 1), (1 1 1)>,
    <(0 1 1), (0 1 1)>,
    <(1 1 1), (1 1 0)>
]
> y := Random(H); print y;
((0 0 0), (1 0 0), (0 0 1), (0 1 0), (1 0 1), (0 1 1),
(1 1 0))
> print [ < a, a^(y@@f)> : a in GSet(G) ];
[ <1, 2>, <2, 5>, <3, 6>, <4, 1>, <5, 3>, <6, 7>, <7, 4>,
<8, 8> ]
```

A derived G-set may be created using the function **GSet**(G, S), where S is a derived set of the natural G-set of G. This function forms the closure Y of S under G, and takes the action of G on Y to be that induced by the action of G on X. In the example below, **GSet**(G, S) will be used to construct a representation of the group PSL$(3, 4)$ on flags (point-line pairs), starting with its action on projective points. In order to construct the flags, it is first necessary to find a line. If H is the stabilizer of a point α in PSL$(3, 4)$ in its action on projective points, then a line consists of α together with the points in any non-trivial orbit of $O_2(G)$.

```
> G := ProjectiveSpecialLinearGroup(3, 4);
```

```
> O2 := pCore(Stabilizer(G, 1), 2);
> print Order(O2);
16
> flag := < 1, Orbit(O2, 2) >;
> print flag;
<1, GSet{ 2, 5, 15, 17 }>
> flags := GSet(G, Orbit(G, flag));
> print #flags;
105
```

Table 32.12. Access functions for G-sets

MAGMA	Meaning
Action(Y)	Mapping giving action on G-set Y
Degree(G, Y)	Cardinality of G-set Y
Group(Y)	Group for G-set Y
Set(Y)	Set underlying G-set Y
Support(Y)	Support of G-set Y
Labelling(G)	Indexed set giving internal mapping of natural G-set of G onto standard G-set $\{1, \ldots, n\}$

The elementary access functions for G-sets are listed in Table 32.12.

32.8.2 Orbits and Stabilizers for General G-Sets

The functions for computing images, orbits and stabilizers and related objects that were presented earlier in this chapter have a generalized form in which the second argument is the required G-set. As usual, suppose that G is a permutation group defined on X, Y is a G-set of G, and $y \in Y$. Then **Image**(g, Y, y) constructs the image of y under $g \in G$, **Orbit**(G, Y, y) forms the orbit of y, **Orbits**(G, Y) constructs all G-orbits on Y, and **Stabilizer**(G, Y, y) finds the subgroup of G that fixes y. These and related functions are listed in Table 32.13.

Some of these functions are illustrated using the action of PSL$(3, 4)$ on flags constructed above:

```
> print Image(G.2, flag);
<8, { 1, 3, 9, 12 }>
> x := <21, { 2, 7, 8, 11 }>;
> y := <11, { 2, 7, 8, 21 }>;
> print IsConjugate(G, flags, x, y);
```

Table 32.13. Orbits and stabilizers on G-sets

Image(g, Y, y)	**Orbit**(G, Y, y)	**Orbits**(G, Y, y)
Fix(g, Y)	**Fix**(G, Y)	**Stabilizer**(G, Y, y)
IsPrimitive(G, Y)	**Transitivity**(G, Y)	**IsConjugate**(G, Y, y, z)
IsTransitive(G, Y)	**IsTransitive**(G, Y, k)	**IsSharplyTransitive**(G, Y, k)
IsRegular(G, Y)	**IsSemiregular**(G, Y)	**IsSemiregular**(G, Y, S)

```
true
(1, 17, 5)(3, 12, 10)(4, 16, 20)(6, 19, 13)
      (8, 21, 11)(9, 18, 14)
> print #Orbit(G, flags, flag);
105
> print IsPrimitive(G, flags);
false
```

32.8.3 The Induced Action on a G-set

Table 32.14. Homomorphism induced by an action on a G-set

MAGMA	Meaning
Action(G, Y)	Homomorphism of G onto G^Y
ActionImage(G, Y)	Image of **Action**(G, Y)
ActionKernel(G, Y)	Kernel of **Action**(G, Y)
IsFaithful(G, Y)	**true** if G acts faithfully on Y

If G is a permutation group defined on X and Y is a G-set for G, the function **Action**(G, Y) constructs the homomorphism $\phi : G \to G^Y$, where G^Y denotes the representation of G given by its action on Y. It also returns the image and kernel of the homomorphism. The functions for the induced action are summarized in Table 32.14.

Continuing with the action of PSL$(3, 4)$ on flags, the following lines construct the image FG of G under this action explicitly. Then the stabilizer of $< 21, \{2, 7, 8, 11\} >$ is constructed in both G and FG, and the two stabilizers are compared:

```
> phi, FG, ker := Action(G, flags);
> print FG;
Permutation group FG acting on a set of cardinality 105
Order = 20160 = 2^6 * 3^2 * 5 * 7
```

```
> x := <21, { 2, 7, 8, 11 }>;
> Gstab := Stabilizer(G, x[1]) meet Stabilizer(G, x[2]);
> print Degree(Gstab), Order(Gstab);
21 192
> FGstab := Stabilizer(FG, x);
> print Degree(FGstab), Order(FGstab);
105 192
> print FGstab@@phi eq Gstab;
true
```

32.9 Primitive Groups and the O'Nan-Scott Theorem

Given an arbitrary permutation group, the **OrbitAction** and **BlocksAction** functions enable the user to construct a chain of normal subgroups such that the quotients of successive terms are primitive.

MAGMA contains tools which provide a detailed description of a primitive permutation group according to the structural description provided by the O'Nan-Scott theorem [DiM96]. In this section, only a few of the available tools will be described; the interested reader should consult the *Handbook* for a more complete description.

32.9.1 Recognition of the Alternating and Symmetric Groups

It is frequently desirable to be able to recognize that a permutation group G defined on X contains $\mathrm{Alt}(X)$, without computing the order of G. The function **IsAltsym**(G) returns **true** if G contains $\mathrm{Alt}(X)$, and the functions **IsAlternating**(G) and **IsSymmetric**(G) are minor variants that return **true** if G is $\mathrm{Sym}(X)$ or $\mathrm{Alt}(X)$ respectively. If G does contain $\mathrm{Alt}(X)$, these tests will usually detect the fact very rapidly, even for groups having degrees in the millions. For example:

```
> G := Sym(1000000);
> H := sub< G | Random(G), Random(G) >;
> time print IsAltsym(H);
true
Time: 6.939
```

These functions operate by examining a number of random elements of G in an attempt to determine whether G is primitive, and, if so, whether G contains elements that are not contained in a primitive group other than $\mathrm{Sym}(X)$ and $\mathrm{Alt}(X)$. If this attempt does not succeed after a certain amount

of effort, a Schreier algorithm is invoked to compute the order of G, so as to settle the issue definitively.

32.9.2 Primitive Groups with Abelian Socle

Primitive groups divide naturally into two classes, depending upon whether or not their socle is abelian. If the primitive group G possesses an abelian socle, then it acts regularly and is unique. The function **EARNS**(G) returns the unique elementary abelian regular normal subgroup N of G, if such a subgroup exists, or the trivial group if not. The stabilizer G_x of any point x in G is a complement for N, and G/N is isomorphic to G_x. Consequently, there is a homomorphism $\phi : G \to G_x$. This homomorphism, its image and its kernel are constructed by the functions **AffineAction**(G), **AffineImage**(G), and **AffineKernel**(G). (Of course, **AffineKernel**(G) returns the same subgroup as **EARNS**(G).) For example:

```
> G := PermutationGroup< 16 |
>   (1,15,7,5,12)(2,9,13,14,8)(3,6,10,11,4),
>   (1,7)(2,11)(3,12)(4,13)(5,10)(8,14),
>   (1,16)(2,3)(4,5)(6,7)(8,9)(10,11)(12,13)(14,15) >;
> print Order(G);
11520
> N := EARNS(G);
> print N;
Permutation group N acting on a set of cardinality 16
Order = 16 = 2^4
    (1, 2)(3, 16)(4, 7)(5, 6)(8, 11)(9, 10)(12, 15)(13, 14)
    (1, 7)(2, 4)(3, 5)(6, 16)(8, 14)(9, 15)(10, 12)(11, 13)
    (1, 13)(2, 14)(3, 15)(4, 8)(5, 9)(6, 10)(7, 11)(12, 16)
    (1, 16)(2, 3)(4, 5)(6, 7)(8, 9)(10, 11)(12, 13)(14, 15)
> print IsRegular(N);
true
> f, H := AffineAction(G);
> print H;
> print FactoredOrder(H);
[ <2, 4>, <3, 2>, <5, 1> ]
> print CompositionFactors(H);
    G
    |  Cyclic(2)
    *
    |  Alternating(6)
    1
> print exists{T : T in NormalSubgroups(H) | T`order eq 2};
false
```

The final statement establishes that H cannot be $Z_2 \times \mathrm{Alt}(6)$ and so must be Sym(6). It may be concluded that G consists of Sym(6) extended by the elementary abelian group of order 16.

32.9.3 Primitive Groups with Non-Abelian Socle

Suppose that G is a primitive group with non-abelian socle N. Then N is a direct product of isomorphic simple groups so that $N = S_1 \times S_2 \times \ldots S_r$. The function **Socle**$(G)$ returns the socle of G. It is well known that a primitive group has at most two minimal normal subgroups and these may be computed using the function **MinimalNormalSubgroups**(G).

A number of related functions allow the user to analyze the socle in different ways: the function **SocleFactors**(G) returns the sequence $[S_1, S_2, \ldots S_r]$; **SocleFactor**(G) returns a single factor of N; and **SocleSeries**(G) returns a composition series for the socle N of G. The group G acts on $S_1, S_2, \ldots S_r$ by conjugation. The group thereby induced is returned by **SocleImage**(G), while the kernel of this action is given by **SocleKernel**(G).

These functions are illustrated by considering the tensor product of Sym(5) and Alt(5) with product action. Recall that product action results in a primitive group.

```
> G := TensorProduct(Sym(5), Alt(5));
> print Order(G);
1492992000000
> print FactoredOrder(G);
[ <2, 17>, <3, 6>, <5, 6> ]
> N := Socle(g); print N;
Permutation group N acting on a set of cardinality 3125
Order = 777600000 = 2^10 * 3^5 * 5^5
> Fac := SocleFactor(G); print Fac;
Permutation group Fac acting on a set of cardinality 3125
Order = 60 = 2^2 * 3 * 5
> Facs := SocleFactors(G);
> print Facs;
[
    Permutation group acting on a set of cardinality 3125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 3125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 3125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 3125
    Order = 60 = 2^2 * 3 * 5,
```

```
    Permutation group acting on a set of cardinality 3125
    Order = 60 = 2^2 * 3 * 5
]
> NSer := SocleSeries(G);
> print NSer;
[
    Permutation group acting on a set of cardinality 3125
    Order = 60 = 2^2 * 3 * 5,
    Permutation group acting on a set of cardinality 3125
    Order = 3600 = 2^4 * 3^2 * 5^2,
    Permutation group acting on a set of cardinality 3125
    Order = 216000 = 2^6 * 3^3 * 5^3,
    Permutation group acting on a set of cardinality 3125
    Order = 12960000 = 2^8 * 3^4 * 5^4,
    Permutation group acting on a set of cardinality 3125
    Order = 777600000 = 2^10 * 3^5 * 5^5
]
> NIm := SocleImage(G); print NIm;
Permutation group NIm acting on a set of cardinality 5
Order = 60 = 2^2 * 3 * 5
    (1, 2, 4)
    (1, 3, 5)
> NKer := SocleKernel(g); print NKer;
Permutation group NKer acting on a set of cardinality 3125
Order = 2^15 * 3^5 * 5^5
> M, tau := GModule(G, SocKer, N);
> print M;
GModule M of dimension 5 over GF(2)
> NKerSer := [ S@@tau : S in CompositionSeries(M) ];
> print NKerSer;
[
    Permutation group acting on a set of cardinality 3125
    Order = 2^14 * 3^5 * 5^5,
    Permutation group SocKer acting on a set of
    cardinality 3125
    Order = 2^15 * 3^5 * 5^5
]
```

From the working above, the structure of the group is clearly revealed. The socle N consists of five copies of Alt(5), each acting on 3125 points. The group G acts as Alt(5) on the set of socle factors while the kernel of this action contains the socle N of G as a subgroup of index 2^5. This section is elementary abelian and so can be regarded as a G-module. This G-module splits into a one-dimensional submodule and a four dimensional submodule.

Pulling these submodules back into G completes the determination of a chief series for G.

32.10 Base and Strong Generating Sets

The material in this short section is concerned with the construction of a special representation of a permutation group that is central to most permutation group algorithms. In most situations the user need not be concerned about this representation, but its construction may dominate the execution time for many algorithms. Knowledge of the material presented here can lead to significantly faster computation in many circumstances. However, this treatment is only superficial, and the serious user will need to study the ideas at much greater depth elsewhere.

Computing structural information for a permutation group G acting on X requires, in most cases, a representation of the set of elements of G. MAGMA represents this set by means of a *base and strong generating set*, or *BSGS*, for G. A *base* B for G is a sequence of distinct points from X with the property that the identity is the only element of G that fixes B pointwise. A base B of length n determines a sequence of subgroups $G^{(i)}$, $0 \leq i \leq n$, where $G^{(i)}$ is the stabilizer of the first $i-1$ points of B. Given a base B for G, a subset S of G is said to be a *strong generating set* for G if $G^{(i)} = S \cap G^{(i)}$, for $i = 1, \ldots, n$.

Almost all of the structure algorithms for permutation groups presuppose the availability of a BSGS for the group in question. Once it has been computed for a particular group, MAGMA will save it for possible future use. Since the computation of a BSGS may be very expensive, a number of different algorithms have been developed; their family is known generically as *Schreier-Sims algorithms*. The best choice of algorithm usually depends upon what is known about the group. If the user does not intervene, MAGMA will automatically choose which Schreier algorithm to invoke. However, for very large groups, it is often possible for the user to reduce the computation time greatly, through the selection of a Schreier algorithm appropriate to the group under consideration.

The procedure **BSGS**(G) constructs a base and strong generating set for the group G using the default algorithm choices.

The procedure **SimsSchreier**(G) constructs a BSGS for G using the standard Schreier-Sims algorithm. This algorithm is mainly appropriate for groups of very small degree (less than 100).

The procedure **SolubleSchreier**(G) is designed to take advantage of the special structure of soluble groups, and is the recommended algorithm for the BSGS if G is known to be soluble.

The procedure **RandomSchreier**(G) constructs a BSGS for G using a random choice for potential strong generators rather than systematically constructing all Schreier generators. *There is no guarantee that the algorithm will have constructed a correct BSGS upon completion, so the user may wish to test the result using* **Verify** *(see below).* The stopping criterion is controlled by two independent parameters taking positive integer values. The first, **Max** (default 100), specifies that at most **Max** random elements are to be tried as potential strong generators; the second, **Run** (default 20), specifies that the algorithm is to terminate after a run of **Run** successive random elements has failed to enlarge the strong generating set. **RandomSchreier** is particularly useful when the order of G is known in advance. The user should first set the order of G using **AssertAttribute**$(G, \texttt{"Order"}, n)$ and then run **RandomSchreier**. If an order is set for G, the random algorithm will continue until a group of that order has been constructed.

The procedure **ToddCoxeterSchreier**(G) applies an algorithm developed by Leon and Sims. It employs a Todd-Coxeter method to select new strong generators and prove correctness. This is the algorithm of choice if the random Schreier is not applicable and the degree lies between 100 and 10 000.

The procedure **Verify**(G) verifies the correctness of a BSGS that has been constructed by the random Schreier. It was developed by Brownie, Cannon and Sims and is based on ideas used by Sims in his construction of the Lyons and O'Nan simple groups. This algorithm is designed for groups having degree in excess of 100 000.

For example, the code below applies **RandomSchreier**(G) to a large wreath product, whose order is known by construction:

```
> G := WreathProduct(Sym(42), Alt(8));
> AssertAttribute(G, "Order",
>   Factorial(42)^8 * (Factorial(8) div 2));
> RandomSchreier(G);
```

33. Matrix Groups

Techniques for computing with matrix groups are less well developed than those for permutation groups. While a matrix group may be created over any ring R satisfying the condition that MAGMA} is able to invert elements of $Mat_n(R)$, the types of computation that are possible depend heavily on the coefficient ring, the dimension of the group and its cardinality (especially whether finite or infinite). In order to get a rough understanding of the possibilities, it is important to distinguish three computationally distinct classes of matrix groups.

The first class comprises finite matrix groups G defined over a field or euclidean ring, where G satisfies the following conditions:

- G has moderate degree;
- G has a short orbit Δ in its action on the natural module and Δ can be located by MAGMA.

(By short orbit is meant an orbit having cardinality at most 1,000,000). Note that G does not have to act faithfully on Δ. For this class of groups, a stabilizer chain representation similar to that used for permutation groups is employed (the *BSGS representation*). A wide range of structural computations may be performed on members of this class of group.

The second class consists of matrix groups defined over finite fields. A computational approach to the analysis of such groups based on Aschbacher's classification of the maximal subgroups of $\mathrm{GL}(n, q)$ is under development by Holt, Leedham-Green, O'Brien, Praeger, P. Neumann and others. This machinery has the capability of determining the order and composition factors of many groups in this class having degree up to 200.

The third class comprises those matrix groups for which it is not possible to construct a Structure Normal Form (SNF). These groups include all infinite matrix groups and finite groups of large degree that do not fall into the first class. While element operations are available for this class of groups, no operations that depend upon testing membership of subgroups are supported.

33.1 The General Linear Group and Its Elements

33.1.1 Creating the General Linear Group

A matrix group over the ring R is a subgroup of the general linear group $\mathrm{GL}(n, R)$ (the group of all invertible $n \times n$ matrices with entries in the ring R). Thus, $\mathrm{GL}(n, R)$ will be the generic group for all $n \times n$ matrix groups over R. This group will be created by the function **GeneralLinearGroup**(n, R), usually abbreviated **GL**(n, R). Note that this function only creates a shell structure: it does not construct actual generators for $\mathrm{GL}(n, R)$. If some subsequent operation requiring the generators is invoked, they will be automatically constructed only in the case where R is a finite field $GF(q)$. For all other coefficient rings it is necessary for the user to supply his or her own generators. The category for subgroups of $\mathrm{GL}(n, R)$ is **GrpMat**.

For example, the group $G5$ below is the general linear group of degree 3 over the finite field of five elements:

```
> gf5 := GF(5);
> G5 := GeneralLinearGroup(3, gf5);
> print G5;
GL(3, GF(5, 1))
```

At this point, *G5* is a "shell structure" that contains the parameters of general linear group but no generators. In the example below, printing the order has the effect of creating generators so that so the **print**-statement gives more information than it did previously:

```
> print #G5;
1488000
> print G5;
MatrixGroup(3, GF(5)) of order 1488000 = 2^7 * 3 * 5^3 * 31
Generators:
    [2 0 0]
    [0 1 0]
    [0 0 1]

    [4 0 1]
    [4 0 0]
    [0 4 0]
```

The possible categories of coefficient ring for matrix groups in Magma include finite fields, number fields, function fields, p-adic fields, the ring of integers $\mathbb{Z}$, quotient rings $\mathbb{Z}/m\mathbb{Z}$ and univariate polynomial rings over a field.

33.1.2 Representation of Matrices

An element of a matrix group may be created by building a sequence containing its entries listed in row-major order, and coercing this sequence into the group. This is the same method for matrix creation as is used in the case of matrix spaces and matrix algebras. For example:

```
> g := G5 ! [3, 1, 2, 2, 4, 4, 2, 0, 3];
> print g;
[3 1 2]
[2 4 4]
[2 0 3]
```

Here MAGMA has automatically coerced the integer terms of the sequence into elements of the finite field *gf5*. The function for decomposing a matrix g into the corresponding sequence of ring elements, is **ElementToSequence**(m) or **Eltseq**(m):

```
> print Eltseq(g);
[ 3, 1, 2, 2, 4, 4, 2, 0, 3 ]
```

The identity matrix of a group G may be constructed as **Identity**(G), or $G!1$.

33.1.3 Arithmetic on Matrices

The standard multiplicative operators apply to elements of matrix groups: $*$ and `^`. An inverses is found by exponentiation to the power -1.

```
> h := G5 ! [3, 0, 0, 0, 4, 2, 0, 2, 3];
> print g * h, h * g;
[4 3 3]
[1 4 0]
[1 1 4]

[4 3 1]
[2 1 2]
[0 3 2]

> print g ^ 10;
[0 3 0]
[2 0 0]
[2 4 1]
```

```
> print h ^ -1;
[2 0 0]
[0 1 1]
[0 1 3]
```

It is possible to access entry i, j or row i of a matrix group element g, using the syntax $g[i, j]$ or $g[i]$. For instance:

```
> print g[2];
(2 4 4)
> print g[2,3];
4
```

However, it is not permitted to modify a matrix by redefining entries (as is done with sequences), since, in general, the result will not be an element of the group. If the user needs to perform such an operation, the matrix should be first converted to a sequence using **Eltseq**, the sequence modified as required and then coerced back into the group.

Multiplication involving matrices is possible when the two matrices are elements of (or coercible into) a common matrix group, and also in some other cases when the coefficient rings are compatible and the number of columns of the first matrix equals the number of rows of the second matrix. If the other matrix is from a matrix ring then it will be possible. If the first matrix is from a matrix space or suitably-sized vector space (or module) and the second is from a matrix group it will be possible; this is an instance of the group *acting* on the space.

33.1.4 The Order of a Matrix

The order of an element g of a matrix group is returned by the function **Order**(g). There are several points to note about the MAGMA facilities for determining the order of a matrix. Firstly, if g has infinite order, the function will not recognize this fact and may never return.

The analysis of matrix groups over finite fields makes heavy use of being able to locate elements having certain orders. For this case, MAGMA employs a very efficient algorithm due to Leedham-Green which is capable of computing the order of elements in groups having degree up to 1000. Since the factorization of the orders may be very difficult in large groups, an alternate version is available for when the factored order is required: **FactoredOrder**(g). This exploits the fact that the manner in which the order is determined produces a partial factorization. The example below sets up the orthogonal group

$\Omega^+(200, 4)$, produces a random element of the group and computes its order, both as an integer and as a factorization sequence.

```
> G := OmegaPlus(200, 4);
> RP := RandomProcess(G);
> x := Random(RP);
> time Order(x);
1286039129078980091566889421260550476237553615698059806510
Time: 27.649
> time FactoredOrder(x);
[ <2, 1>, <3, 2>, <5, 1>, <7, 1>, <11, 1>, <17, 1>,
<31, 1>, <3855260977, 1>, <64082150767423457, 1>,
<14253432751031263273727 69, 1>
]
Time: 26.689
```

Since the Leedham-Green algorithm requires factorization of large integers, the order algorithm may sometimes fail or be very expensive. For some applications, it suffices to obtain a multiple of the order. The boolean-valued parameter **Proof** for functions **Order** and **FactoredOrder** allows the user to specify that difficult integer factorizations are not to be attempted so that the value returned may be a multiple of the true order of g. In this case, a second boolean value is returned: **true** if the order is known to be correct and **false** if it may be a multiple of the true order.

The *projective order* of an element g of a matrix group is the least positive integer n such that $g^n = \lambda I$, where I is the identity matrix. Functions **ProjectiveOrder** and **FactoredProjectiveOrder** return the projective order or factored projective order, respectively. The scalar λ is returned as a second return value by both functions.

33.1.5 Matrix Invariants and Canonical Forms

Many of the operations provided for elements of matrix rings may also be applied to the elements of matrix groups (see Table 33.1 (p. 676)). The various canonical forms may be computed over a field, while **MinimalPolynomial** works for any field and $\mathbb{Z}$. Operations such as **Adjoint** and **Transpose**, when applied to an element of the group G, return their result as an element of the matrix ring in which G is naturally embedded. See also Table 28.4 (p. 536) and Table 28.5 (p. 539).

Table 33.1. Properties of a matrix regarded as a linear transformation

Adjoint(g)	**JordanForm**(g)
CharacteristicPolynomia(g)	**MinimalPolynomial**(g)
Determinant(g)	**PrimaryRationalForm**(g)
Eigenspace(g, a)	**PrimaryInvariantFactors**(g)
Eigenvalues(g)	**RationalForm**(g)
HessenbergForm(g)	**Transpose**(g)
InvariantFactors(g)	**Trace**(g)

33.2 General Matrix Group Constructions

A matrix group is always viewed by MAGMA as being a subgroup of $\mathrm{GL}(n, R)$. The formal way of defining a matrix group is therefore to construct it explicitly as a subgroup of $\mathrm{GL}(n, R)$. As usual, a constructor is provided that allows the two steps to be performed by a single statement.

33.2.1 Constructing Subgroups

Given a subgroup G of $\mathrm{GL}(n, R)$, the **sub**-constructor applied to G has the form

sub< G | generator specification **>**

where the generators for the subgroup are given as a list whose terms may be any of the following: element of G, element of $Mat_n(R)$ belonging to G, sequence defining a matrix of G, a subgroup of G, or a set or sequence or any of the above.

For instance, consider the group $G6$ of 2×2 matrices over $\mathbb{Z}_6$ generated by the matrices

$$\begin{pmatrix} 5 & 1 \\ 5 & 0 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 5 & 5 \\ 1 & 2 \end{pmatrix}$$

```
> Z6 := ResidueClassRing(6);
> GL2Z6 := GL(2, Z6);
> G6 := sub< GL2Z6 | [5, 1, 5, 0 ], [5, 5, 1, 2] >;
> print G6;
MatrixGroup(2, Residue class ring of integers modulo 6)
Generators:
    [5 1]
    [5 0]
```

```
[5 5]
[1 2]
```

The ability to include subgroups in the generator list has many uses. Suppose, for example, the user wishes to get an idea as to what kinds of subgroup of GL(3, 3) are generated by a pair of distinct Sylow 3-subrgoups. The following code looks at the orders of 20 such random subgroups.

```
> G := GL(3, 3);
> S3 := SylowSubgroup(G, 3);
> print [ #sub< G | S3, S3^Random(G) > : i in [1..20] ];
[ 5616, 5616, 5616, 216, 5616, 5616, 5616, 5616, 5616,
5616, 5616, 5616, 5616, 5616, 5616, 216, 5616, 5616,
216, 5616 ]
```

Thus, it seems that only two subgroups arise this way: SL(3, 3) and a subgroup of order 216. In fact, the subgroup obtained is probably controlled by the intersection of the two Sylow subgroups:

```
> print [ < #sub< G | S3, S3^x >, #(S3 meet S3^x) >
>               where x is Random(G) : i in [1..20] ];
[ <5616, 3>, <5616, 3>, <5616, 1>, <5616, 1>, <5616, 3>,
<5616, 3>, <5616, 1>, <5616, 3>, <216, 9>, <216, 9>,
<5616, 3>, <216, 9>, <5616, 3>, <5616, 1>, <5616, 1>,
<5616, 1>, <5616, 3>, <5616, 1>, <216, 9>, <5616, 1> ]
```

The above output produces evidence for this conjecture: an interection of order 1 or 3, generates SL(3, 3), while an intersection of order 9 produces the subgroup of order 216.

The dihedral group D_n of order $2n$ has a 3-dimensional complex representation. The following pair of matrices generate D_n in this representation:

$$\begin{pmatrix} \cos\frac{2\pi}{n} & \sin\frac{2\pi}{n} & 0 \\ -\sin\frac{2\pi}{n} & \cos\frac{2\pi}{n} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The function *dihedral* constructs D_n in this representation.

```
dihedral := function(n)
   cos := func< r | (x+x^-1)/2 where x := RootOfUnity(r) >;
   sin := func< r | -RootOfUnity(4)*(x-x^-1)/2
                                 where x := RootOfUnity(r) >;
```

```
    K<z> := CyclotomicField(LCM(4, n));
    GL := GeneralLinearGroup(3, K);
    a :=   GL ! [ cos(n), sin(n), 0,
                 -sin(n), cos(n), 0,
                       0,      0, 1 ];
    b :=   GL ! [  -1,   0, 0,
                    0,   1, 0,
                    0,   0, 1 ];
    return sub< GL | a, b>;
end function;
```

33.2.2 The Matrix Group Constructor

The two steps in constructing a matrix group, i.e. first defining $\mathrm{GL}(n, R)$ and then specifying the desired subgroup, may be performed using a single constructor: **MatrixGroup** constructor:

MatrixGroup< n, R | generator specification **>**

which directly builds the desired subgroup of $\mathrm{GL}(n, R)$ from the user's generators.

For instance, the group $G6$ of 2×2 matrices over $\mathbb{Z}_6$ introduced above may defined in the alternative manner:

```
> Z6 := ResidueClassRing(6);
> G6 := MatrixGroup< 2, Z6 | [5, 1, 5, 0 ], [5, 5, 1, 2] >;
```

Sometimes generators for a matrix group are constructed in a sequence of operations which may result in intermediate matrices being singular. Since these matrices are not elements of $\mathrm{GL}(n, R)$, such generators may need to be constructed as elements of $Mat_n R$). The **sub**-constructor, applied to a matrix group, will happily coerce non-singular elements of $Mat_n R$) into G. This is illustrated by constructing the symmetric group Sym(6) as a 6-dimensional matrix group over $\mathbb{Z}$ generated by permutation matrices corresponding to the cycles $(1, 2)$ and $(1, 2, 3, 4, 5, 6)$.

```
> Z := IntegerRing();
> M := MatrixAlgebra(Z, 6);
> e := func<i, j | MatrixUnit(M, i, j) >;
> a := e(1,2) + e(2,1) + e(3,3) + e(4,4) + e(5,5) + e(6,6);
> b := e(1,2) + e(2,3) + e(3,4) + e(4,5) + e(5,6) + e(6,1);
> G := MatrixGroup< 6, Z | a, b >;
> print #G;
720
```

The function **Generic**(G): performs the inverse operation. Given a group G, it returns the generic group of G, that is, the group $\mathrm{GL}(n, R)$ where n and R are chosen such that G is a subgroup of this group. For example:

```
> print Generic(G6) eq GL2Z6;
true
```

33.2.3 Normal Subgroups and Quotients

The constructors for normal subgroups and quotients are **ncl** and **quo**. At the time of writing the quotient group is returned as a permutation group though this may change at a future date. For example, consider the group generated by the matrices

$$\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

with coefficients in the residue class ring $\mathbb{Z}/24\mathbb{Z}$.

```
> R := ResidueClassRing(24);
> G := MatrixGroup< 2, R | [1,1,0,1], [0,1,-1,0] >;
> H := ncl< G | [-1, 0, -6, -1] >;
> print H;
MatrixGroup(2, Residue class ring of integers modulo 24)
of order 2^6
Generators:
    [23  0]
    [18 23]

    [ 7 18]
    [ 0  7]

    [17  0]
    [ 0 17]
> Q, phi := quo< G | [-1, 0, -6, -1] >;
> print Q;
Permutation group Q acting on a set of cardinality 144
Order = 144 = 2^4 * 3^2
> NQ := NormalSubgroups(Q);
> NG := [ n`subgroup@@phi : n in NQ ];
> print [ Order(n) : n in NG ];
[ 64, 128, 192, 256, 384, 512, 768, 768, 768, 1536, 1536,
2304, 3072, 4608, 9216 ]
```

MAGMA is able to calculate the normal closure whenever it can find the order of the group. It can calculate the quotient only if it can compute the coset table for the group over the normal subgroup relative to its defining generators. In this case, the permutation group that is the quotient comes from the columns of the coset table.

33.2.4 Families of Classical Groups

Let K be a finite field. Functions are provided for constructing a group G belonging to any of the four families of classical groups defined over K in the sense that a concise generating set will be constructed for G. In almost all cases, the group will be defined on two generators. Recall, that **GeneralLinearGroup**(n, R), or its abbreviated form, **GL**(n, R), returns the general linear group of degree n over the ring R as a shell structure. In the case where R is a finite field, generators for $\mathrm{GL}(n, R)$ will be created if an operation requiring them is invoked. If R is any ring other than a finite field, MAGMA will not automatically construct generators for a group created using the function **GL(n,R)**. The function has the alternative forms **GL**(n, q) and **GL**(V), where V is an n-dimensional vector space over K.

The subgroup of **GL**(n, K) consisting of matrices of determinant 1, the *special linear group*, is returned by the function **SpecialLinearGroup** or its abbreviated form **SL**. This function takes the same arguments as **GL** giving the three possible forms **SL**(n, K), **SL**(n, q) and **SL**(V).

Similar functions are provided for the symplectic, unitary and orthogonal families as summarized in Table 33.2 (p. 681). In each case, the three possible argument patterns, (n, K), (n, q), and (V) are supported. However, to save space only the (n, q) possibility is presented in the table. It should be noted that there are restrictions on possible dimensions and sometimes the field K. Thus, the symplectic groups are only defined in even dimension at least 4. The unitary groups $U(n, q)$ are actually defined over the field $K = GF(q^2)$. Since there are two distinct quadratic forms in even dimension, there are two families of orthogonal groups in even dimension at least 4, while there is a single family in odd dimension at least 5. Also included in this table are the Suzuki groups, a family of 4-dimensional matrix groups defined over the field $K = GF(2^{2n+1})$. The possible forms are **Sz**(q), **Sz**(K), and **Sz**(V), where V is a 4-dimensional vector space over the field $K = GF(2^{2n+1})$.

MAGMA also provides for the creation of projective versions of the above groups as permutation groups. For example, **PGL**(n, q) and **PSL**(n, q) return the projective general and projective special linear groups, while **PGammaL**(n, q) and **PSigmaL**(n, q) return the corresponding automorphism groups. Similar projecive versions are available for the other three

Table 33.2. Classical Groups

MAGMA	Meaning
GeneralLinearGroup(n, q)	$GL(n, q)$, $n \geq 2$
SpecialLinearGroup(n, q)	$SL(n, q)$, $n \geq 2$
GeneralUnitaryGroup(n, q)	$GU(n, q)$, $n \geq 2$
SpecialUnitaryGroup(n, q)	$SU(n, q)$, $n \geq 2$
SymplecticGroup(n, q)	$Sp(n, q)$, $n \geq 4$ and even
GeneralOrthogonalPlusGroup(n, q)	$GO^+(n, q)$, $n \geq 4$ and even
GeneralOrthogonalMinus(n, q)	$GO^-(n, q)$, $n \geq 4$ and even
GeneralOrthogonalGroup(n, q)	$GO(n, q)$, $n \geq 5$ and odd
SpecialOrthogonalPlusGroup(n, q)	$SO^+(n, q)$, $n \geq 4$ and even
SpecialOrthogonalMinusGroup(n, q)	$SO^-(n, q)$, $n \geq 4$ and even
SpecialOrthogonalGroup(n, q)	$SO(n, q)$, $n \geq 5$ and odd
OmegaPlusGroup(n, q)	$\Omega^+(n, q)$, $n \geq 4$ and even
OmegaMinusGroup(n, q)	$\Omega^-(n, q)$, $n \geq 4$ and even
OmegaGroup(n, q)	$\Omega(n, q)$, $n \geq 5$ and odd
SuzukiGroup(q)	$Sz(q)$, q an odd power of 2

classical families and the Suzuki groups. All of these functions take the same arguments as the non-projective versions.

For example:

```
> F8<u> := FiniteField(8);
> Sp := SymplecticGroup(6, F8);
> print Sp;
MatrixGroup(6, GF(2^3))
Generators:
    [  u   0   0   0   0   0]
    [  0   1   0   0   0   0]
    [  0   0   u   0   0   0]
    [  0   0   0   u   0   0]
    [  0   0   0   0   1   0]
    [  0   0   0   0   0 u^6]

    [  0   1   1   1   0   0]
    [  1   0   0   0   0   0]
    [  0   1   0   0   0   0]
    [  0   0   1   0   1   0]
    [  0   0   0   0   0   1]
    [  0   0   1   0   0   0]
> PSp := ProjectiveSymplecticGroup(6, F8);
> print PSp;
```

```
Permutation group PSp acting on a set of cardinality 37449
```

33.2.5 Product Constructions

Table 33.3. Other matrix group constructions

MAGMA	Meaning
DirectProduct(G, H)	Direct product of matrix groups G and H of degrees m and n, as a matrix group of degree $m + n$
DirectProduct(Q)	Direct product of sequence Q of matrix groups
TensorProduct(G, H)	Tensor product $G \otimes H$ of matrix group G and permutation group H, as a matrix group
WreathProduct(G, H)	Wreath product $G \wr H$ of matrix group G and permutation group H, as a matrix group

Table 33.3 lists some standard product constructions available for matrix groups. Note that functions **TensorProduct** and **WreathProduct** take a permutation group rather then a matrix group as their the second argument. For example:

```
> F4<w> := FiniteField(4);
> Gsu := SU(3, F4);
> print Gsu;
MatrixGroup(3, GF(2^2))
Generators:
    [  1   w   w]
    [  0   1 w^2]
    [  0   0   1]

    [  w   1   1]
    [  1   1   0]
    [  1   0   0]
> TP := TensorProduct(Gsu, Sym(3));
> print Degree(TP), Order(TP);
27 6718464
```

33.3 Access Functions for Matrix Groups and Elements

Table 33.4 summarizes the functions giving facts about a matrix group.

Table 33.4. General facts about a matrix group

MAGMA	Meaning
Degree(G)	Degree n of a group G of $n \times n$ matrices
CoefficientRing(G), **BaseRing**(G)	Coefficient ring or base ring R of G
Generic(G)	Given G of degree n and coefficient ring R, return general linear group $\mathrm{GL}(n, R)$ in which G is naturally embedded

33.4 Matrix Rings and Matrix Groups

Since a matrix group is embedded in the group of units of the corresponding matrix ring, an invertible matrix x may be regarded both as an element of $\mathrm{GL}(n, R)$ and as an element of $Mat(n, R)$. Since an object in MAGMA must have a unique parent, x must have one or the other as its parent. In some situations it is necessary to move between the group context and the ring context. To convert between a matrix group and the corresponding matrix ring, the following functions may be used:

```
group2ring := func< G |
    MatrixRing< CoefficientRing(G), Degree(G) |
      Generators(G) > >;

ring2group := func< R |
    MatrixGroup< Degree(R), CoefficientRing(R) |
      Generators(R) > >;
```

The following example employs a finite subgroup $SX3$ of $\mathrm{GL}(3, \mathbb{Z})$.

```
> SX3 := MatrixGroup(3, Integers() |
>            [-1, 0, 0,  1, 1, 1,  0, 0,-1],
>            [ 0, 0,-1, -1, 0, 0,  0,-1, 0] >;
> rgSX3 := group2ring(SX3);
> print rgSX3 : Maximal;
Matrix Algebra of degree 3 with 2 generators
over Integer Ring
Generators:
[ 0  0 -1]
[-1  0  0]
[ 0 -1  0]

[-1  0  0]
```

```
[ 1  1  1]
[ 0  0 -1]
> print #SX3;
48
> print IsFinite(rgSX3);
false
```

The elements of the ring and the group will not be the same, because the ring will contain non-invertible elements. However, if an element is in common to both, it may be coerced into the other structure using the ! operator:

```
> gpelt := SX3 ! [1, 1, 1, 0, -1, 0, 0, 0, -1];
> rgelt := rgSX3 ! gpelt; print rgelt;
[ 1  1  1]
[ 0 -1  0]
[ 0  0 -1]
> print gpelt eq SX3 ! rgelt;
true
> rgelt2 := rgSX3 ! [0, 1, 0, 0, 3, 2, 0, 0, 1];
> print SX3 ! rgelt2;

>> print SX3 ! rgelt2;
             ^
Runtime error in '!': Matrix is not invertible
LHS: GrpMat
RHS: AlgMatElt
```

33.5 Finite Groups of Small Degree

In general, most group structure algorithms require a structure normal form (SNF). This and the following section are concerned with the case where it is possible to construct to construct the particular SNF known as *base and strong generating set (BSGS)* for a matrix group. This SNF requires that G be finite and be defined over an euclidean ring R. Given a BSGS representation of G, most of the group structure algorithms described in Chapter 29 may be used to compute detailed structural information about the group.

33.5.1 Base and Strong Generating Set

A *base* for G is a sequence $B = [b_1, b_2, \ldots, b_k]$ of vectors or subspaces taken in the natural R-module M for G such that the only element of G that fixes

B pointwise is the identity. Set $G^{(0)} = G$, and $G^{(i)} = G_{b_1,\ldots,b_{i-1}}$ for $i \geq 1$. The subgroup chain giving the required representation of G consists of the stabilizers

$$G = G^{(0)} \geq G^{(1)} \geq G^{(1)} \geq \ldots \geq G^{(k)} = 1.$$

The construction of a BSGS depends upon being able to choose an object b_i (vector or subspace) in the natural module M such that the orbit Δ_i of b_i under $G^{(i-1)}$ is not excessively long, i.e. generally having length less than a million. Usually, once suitable choices have been made for the first and second base points, b_1 and b_2, then it is easy to find suitable candidates for the remainder.

A BSGS for a matrix group is constructed by some version of the Schreier-Sims algorithm. Any calculation requiring a BSGS will automatically trigger an attempt to construct it, unless one is already known. As the degree of the group increases, the cost of computing a BSGS quickly becomes excessive. Thus, in some circumstances, it is better for the user to control the manner in which a BSGS is constructed. The standard method available for constructing a BSGS for a matrix group is the Todd-Coxeter Schreier algorithm which may be invoked by means of the procedure **ToddCoxeterSchreier**(G).

The random Schreier algorithm attempts to construct a BSGS by means of a randomized procedure. Without additional information, this version of the algorithm cannot prove that it has found a correct BSGS for G. The algorithm terminates when m randomly chosen elements of G have failed to enlarge the group defined by the current BSGS. The random Schreier may be invoked by the procedure **RandomSchreier**(G). The default value of m (40) may be changed by the assigning the desired value to the parameter **Run**. The random Schreier algorithm provides a cheap way of obtaining a lower bound on the order of G. Consider the group $\mathrm{SL}(5, 8)$. The procedure **RandomSchreier** will be used to find a probable BSGS, and therefore a probable value for the order of G. The user can obtain some idea of the progress of the algorithm by setting a *verbose flag*, using the procedure **SetVerbose**:

```
> SetVerbose("RandomSchreier", 1);
> G := SL(5, 8);
> time RandomSchreier(G);
New strong generator after 1.040 seconds and 11 failures
Current 'order' is 2^3 * 3^2 * 5 * 13 * 31 * 151
New strong generator after 1.670 seconds and 1 failures
Current 'order' is 2^4 * 3^2 * 5 * 7 * 13 * 31 * 151
New strong generator after 3.350 seconds and 6 failures
Current 'order' is 2^9 * 3^2 * 5 * 13 * 31 * 73 * 151
New strong generator after 4.550 seconds and 1 failures
Current 'order' is 2^9 * 3^4 * 5 * 7 * 13 * 31 * 73 * 151
New strong generator after 6.900 seconds and 0 failures
Current 'order' is 2^18 * 3^4 * 5 * 13 * 31 * 73 * 151
```

```
New strong generator after 8.820 seconds and 3 failures
Current 'order' is 2^19 * 3^4 * 5 * 7 * 13 * 31 * 73 * 151
New strong generator after 11.571 seconds and 2 failures
Current 'order' is 2^30 * 3^4 * 5 * 13 * 31 * 73 * 151
New strong generator after 14.041 seconds and 0 failures
Current 'order' is 2^30 * 3^4 * 5 * 7 * 13 * 31 * 73 * 151
New strong generator after 16.621 seconds and 0 failures
Current 'order' is 2^30 * 3^4 * 5 * 7^2 * 13 * 31 * 73 * 151
New strong generator after 19.211 seconds and 1 failures
Current 'order' is 2^30 * 3^4 * 5 * 7^3 * 13 * 31 * 73 * 151
New strong generator after 22.312 seconds and 0 failures
Current 'order' is 2^30 * 3^4 * 5 * 7^4 * 13 * 31 * 73 * 151
Random Schreier finished in 23.122 seconds
76 random generators were tested
Time: 24.372
> print HasAttribute(G, "Order");
true 4638226007491010887680
```

The Order attribute is now set, with the value 4638 226 007 491 010 887 680, so this is the probable order of G. However, if the function **Order** is now invoked then MAGMA will try to verify this value, effectively by calling **Verify**(G) to engage the Todd-Coxeter Schreier method.

If the user knows the order n of G, it should be communicated to MAGMA using the statement **AssertAttribute**$(G,$ **"Order"**$, n)$. Alternatively, if the order n is given as a factorization sequence Q, then Q may be supplied in place of n as the third argument. When the order of G is stored, the random Schreier algorithm will continue until a BSGS is constructed for a group of order n. Thus, the result is guaranteed to be correct in this situation. This is the fastest known way of computing a BSGS. If the order of G has not been supplied but the user believes that the BSGS is correct, this information can be asserted by means of the procedure **AssertAttribute**$(G,$ **"IsVerified"**, **true**$)$. In this situation, if the BSGS is not correct, results depending upon it mat be incorrect or MAGMA may crash in an unexpected way during subsequent computations involving G.

A probable BSGS constructed by the **RandomSchreier** may be checked using the function **Verify**(G). If the BSGS is incomplete, **Verify** will cause it to be completed by applying the Todd-Coxeter Schreier algorithm. Consequently, the use of **RandomSchreier** followed by **Verify** is generally no faster than having called **ToddCoxeterSchreier**(G) at the outset.

33.6 The Natural R-module

Every matrix group G of degree n over a ring R has a natural action on its underlying R-module $R^{(n)}$. This section is concerned with this action.

33.6.1 Action on the Natural R-module

The function **RSpace**(G) returns the natural R-module for a subgroup of $\mathrm{GL}(n, R)$. Note that it returns an R-module, not an $R[G]$-module. If R is a field, the function will return a vector space. For example, consider the Mathieu group M_{11} of order 7920 in a 5-dimensional representation over the field $K = \mathrm{GF}(3)$.

```
> K := GF(3);
> G := MatrixGroup< 5, K |
>      [ 2,1,2,1,2,  2,0,0,0,2,  0,2,0,0,0,
>        0,1,2,0,1,  1,0,2,2,1 ],
>      [ 2,1,0,2,1,  1,2,0,2,2,  1,1,2,1,1,
>        0,2,0,1,1,  1,1,2,2,2 ] >;
> print G;
MatrixGroup(5, GF(3))
Generators:
    [2 1 2 1 2]
    [2 0 0 0 2]
    [0 2 0 0 0]
    [0 1 2 0 1]
    [1 0 2 2 1]

    [2 1 0 2 1]
    [1 2 0 2 2]
    [1 1 2 1 1]
    [0 2 0 1 1]
    [1 1 2 2 2]
> V := RSpace(G);
> print V;
Full Vector space of degree 5 over GF(3)
```

A matrix group acts on its natural module by multiplication on the right. Either the multiplication operator ($*$) or the exponentiation operator (`^`) may be used to compute images with respect to this action.

```
> v := V![2, 0, 0, 1, 2];
> print v ^ (G.1);
(0 0 1 0 1)
> print v * (G.1);
(0 0 1 0 1)
```

The same operators may be used to calculate the image of a set, sequence or subspace. For instance:

```
> print sub<V|v, V.4> ^ (G.1);
Vector space of degree 5, dimension 2 over GF(3)
Echelonized basis:
(0 1 0 0 2)
(0 0 1 0 1)

> H := sub< G | G.1^2 * G.2 >;
> print Order(H);
6
> print v ^ H;
{
    (1 2 0 2 0),
    (1 0 0 0 2),
    (2 0 0 1 2)
}
```

The reader may wish to compare this section with the permutation group operations in Section 32.6.

33.6.2 Orbits and Stabilizers

For the remainder of this section it will be assumed that the matrix group G is defined over an euclidean ring R. Let M be the natural R-module for the group G. The function **Orbit**(G, T) or $T\hat{}G$ computes the orbit of the object T under the action of G, where T may be either a vector of M, a set of vectors, a sequence of vectors or a subspace of M. Note that if the orbit of T is infinite, the function will not return. An alternative function, **OrbitBounded**(G, T, b) allows the user to specify a limit b on the size of the orbit constructed. If the orbit of T under G has length less than or equal to b, the function returns **true** and the orbit. Otherwise, it simply returns **false**.

```
> v := V![2, 0, 0, 1, 2];
> w := V![0, 2, 1, 1, 2];
> print #Orbit(G, v);
22
> print #Orbit(G, w);
220
> print #Orbit(G, {v, w});
1980
> print #Orbit(G, [v, w]);
1980
> print #Orbit(G, sub< M | v, w >);
330
```

If S is a set whose elements are either vectors, sets of vectors, sequences of vectors, subspaces, sets of subspaces or sequences of subspaces, the function **OrbitClosure**(G, S) constructs the smallest G-invariant subset that contains S. In other words, **OrbitClosure** returns the union of orbits of the elements of S.

In the case in which M is both finite and relatively small, the function **Orbits**(G) may be used to partition its vectors into orbits, while the function **LineOrbits**(G) will compute the orbits of G in its action on the one-dimensional subspaces of M:

```
> orbs := Orbits(G);
> print [ #orb : orb in orbs ]; // sum of them = size of V
[ 1, 220, 22 ]
> lorbs := LineOrbits(G);
> print [ #orb : orb in lorbs ];
[ 110, 11 ]
```

The function **Stabilizer**(G, T) constructs the stabilizer of the object T under the action of G, where T may be either a vector of M, a sequence of vectors or a subspace of M.

```
> Gv := Stabilizer(G, v);
> print #Gv;
360
Sw := Stabilizer(G, w);
> print #Sw;
36
> Svw := Stabilizer(G, [v, w] );
> print #Svw;
4
> Ss := Stabilizer(G, sub< V | v,w >);
> print Ss;
MatrixGroup(5, GF(3)) of order 24 = 2^3 * 3
Generators:
    [2 1 2 2 1]
    [2 0 2 1 1]
    [1 1 1 2 0]
    [2 2 1 2 2]
    [1 1 2 2 2]

    [0 0 0 2 2]
    [0 1 0 0 0]
    [1 2 2 2 0]
    [2 0 0 0 2]
```

```
[0 0 0 0 1]

[0 2 1 2 1]
[0 1 0 0 0]
[0 0 1 0 0]
[0 1 0 2 0]
[1 2 2 2 0]
```

33.6.3 The Induced Action on an Orbit

Assume that G is a finite group with natural module M for which it is possible to construct a SNF. Let T be, as usual, a vector, set of vectors, sequence of vectors or a subspace of M. Let U be the smallest G-invariant set containing T. Then the natural action of G on the elements of U induces a permutation representation ϕ of G: $\phi : G \to L$. Functions to compute this action or merely its image or kernel are listed in Table 33.5.

Table 33.5. The orbits action homomorphism

MAGMA	Meaning
OrbitAction(G, S)	Constructs permutation representation L of G corresponding to the action of G on the orbit closure of the set S. Returns (i) natural homomorphism $\phi : G \to L$, (ii) induced group L, (iii) kernel of the action
OrbitImage(G, S)	Induced group L which is the image of the above homomorphism
OrbitKernel(G, S)	Kernel of the above homomorphism
OrbitActionBounded(G, S, b)	Constructs permutation representation L of G corresponding to the action of G on the orbit closure C of the set S provided that the size of C does not exceed the positive integer b. Returns (i) **true** is successful together with (ii) natural homomorphism $\phi : G \to L$, (iii) induced group L, (iv) kernel of the action. If $\|C\|$ exceeds b, the value false is returned
OrbitImageBounded(G, S, b)	Induced group L which is the image of the above homomorphism
OrbitKernelBounded(G, S, b)	Kernel of the above homomorphism

The subgroup G of $\mathrm{GL}(3, 7)$ generated by the matrices

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 6 & 4 \\ 3 & 0 & 6 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 3 & 0 & 0 \\ 3 & 4 & 4 \\ 6 & 4 & 0 \end{pmatrix}$$

has order 98 784. The **OrbitAction** function will be used to find the composition factors of G by reducing to the computation of factors in permutation groups.

```
> K := GF(7);
> G := MatrixGroup< 3, K | [ 1,0,0, 1,6,4, 3,0,6 ],
>                          [ 3,0,0, 3,4,4, 6,4,0 ] >;
> print OrderG);
98784
> M := RSpace(G);
> v := M![1,0,0];
> O := Orbit(G, v);
> print #O;
6
> phi, Im, Ker := OrbitAction(G, O);
> print Im;
Permutation group Im acting on a set of cardinality 6
Order = 6 = 2 * 3
    (1, 6)(2, 5)(3, 4)
    (1, 3, 2, 6, 4, 5)
> print Ker;
MatrixGroup(3, GF(7)) of order 16464 = 2^4 * 3 * 7^3
Generators:
    [1 0 0]
    [1 6 4]
    [3 0 6]

    [1 0 0]
    [5 1 1]
    [5 3 4]
> U := sub< M | [0,1,0] >;
> P := Orbit(Ker, U);
> print #P;
56
> phi1, Im1, Ker1 := OrbitAction(Ker, U);
> print Order(Im1);
16464
> print Order(Ker1);
1
> print CompositionFactors(Im1);
    G
    |  A(1, 7)               = L(2, 7)
    *
    |  Cyclic(2)
    *
```

```
|  Cyclic(7)
*
|  Cyclic(7)
1
```

33.7 The Natural $K[G]$-module

Let G be a matrix group defined over a field K and let M be the natural K[G]-module for G. This section is concerned with the action of G on proper submodules and quotient modules of M.

33.8 Submodules

A matrix group G is said to act irreducibly on its natural module M, if M has no (G-invariant) submodules. The function **IsIrreducible**(G) returns **true** if G acts irreducibly on M. If M is reducible under the action of G, then **IsIrreducible** returns **false** together with a proper submodule of M.

```
> K<w> := GF(4);
> G := MatrixGroup< 6, K |
>     [ 1,0,w,0,0,0,   0,0,0,0,w^2,0,   w^2,0,w^2,0,0,0,
>       0,0,0,w^2,0,1, 0,w,0,0,0,0,     0,0,0,w,0,w ],
>     [ 1,0,1,0,0,0,   0,w,0,0,0,0,     0,0,1,0,0,0,
>       0,0,0,1,0,1,   0,w^2,0,0,w^2,0, 0,0,0,0,0,1 ] >;
> print G;
MatrixGroup(6, GF(2^2))
Generators:
    [  1   0   w   0   0   0]
    [  0   0   0   0 w^2   0]
    [w^2   0 w^2   0   0   0]
    [  0   0   0 w^2   0   1]
    [  0   w   0   0   0   0]
    [  0   0   0   w   0   w]

    [  1   0   1   0   0   0]
    [  0   w   0   0   0   0]
    [  0   0   1   0   0   0]
    [  0   0   0   1   0   1]
    [  0 w^2   0   0 w^2   0]
    [  0   0   0   0   0   1]
```

```
> b, N := IsIrreducible(G);
> print b;
false
> print N:Maximal;
GModule N of dimension 2 over GF(2^2)
Generators of acting algebra:

[w^2   1]
[  w   w]

[  1   1]
[  0   1]
> G1 := QuotientModuleImage(G, N);
> print G1;
MatrixGroup(4, GF(2^2))
Generators:
    [  1   0   w   0]
    [  0   0   0 w^2]
    [w^2   0 w^2   0]
    [  0   w   0   0]

    [  1   0   1   0]
    [  0   w   0   0]
    [  0   0   1   0]
    [  0 w^2   0 w^2]
> print #G2;
10800
> G2 := SubmoduleImage(G, N);
> print G2;
MatrixGroup(2, GF(2^2))
Generators:
    [w^2   1]
    [  w   w]

    [  1   1]
    [  0   1]
> print #G1;
180
> b, U := IsIrreducible(G1);
> b;
false
> print U;
GModule U of dimension 2 over GF(2^2)
> b, V :=  IsIrreducible(G2);
```

```
> print b;
true
> G11 := SubmoduleImage(G1, U);
> #G11;
180
> G12 := QuotientModuleImage(G1, U);
> print #G12;
60
> print IsIrreducible(G11);
true
> print IsIrreducible(G12);
true
```

The function **GModule**(G) creates the natural $K[G]$-module M for the group G. Now the functions described in Chapter 29 for computing with $K[G]$-modules may be applied to analyze the structure of M.

33.9 Actions Induced from a Submodule

33.10 Matrix Groups of Large Degree over a Finite Field

The facilities described earlier in this chapter for computing structural information for finite matrix groups, depend upon being able to construct a base and strong generating set. However, there are many examples of groups of moderately small degree defined over finite fields where this is not practical. A major project, involving a large group of mathematicians and directed by Charles Leedham-Green, is concerned with the development of an alternative approach to investigating matrix groups defined over finite fields which are outside the range for which BSGS techniques apply. The basis of this approach is the Aschbacher classification of the maximal subgroups of $\mathrm{GL}(n, q)$. At the time of writing much work remains to be done so that the functions presented in this section represent an early stage of the project. Interested readers should consult a current *Handbook* for up-to-date details.

33.10.1 Recognizing Classical Groups

Let G be a subgroup of $\mathrm{GL}(n, q)$. The functions in Table 33.6 employ Las Vegas algorithms to determine whether, modulo scalars, G lies between a classical simple group and its automorphism group. The functions whose names end with the word **Type** do precisely this while the remaining functions either test whether G is a specific group or contains a specific classical group.

Table 33.6. Las Vegas recognition of classical groups

MAGMA	Meaning
IsClassicalGroup(G)	**true** if $S \leq G \leq Aut(S)$, where S is a classical group. (See text for details)
IsLinearGroup(G)	**true** if $\mathrm{SL}(d,q) \leq G \leq \mathrm{GL}(d,q)$
IsSpecialLinearGroup(G)	**true** if G is the special linear group $\mathrm{SL}(n,q)$
IsSymplecticType(G)	**true** if G preserves a symplectic form up to scalar multiplication by a constant. In this case the form is also returned
IsSymplecticGroup(G)	**true** if G is the symplectic group $\mathrm{Sp}(n,q)$
IsOrthogonalType(G)	**true** if G preserves an orthogonal form (quadratic form for even characteristic) up to scalar multiplication by a constant. In this case the form is also returned
IsOrthogonalGroup(G)	**true** if either $\mathrm{SO}^+(n,q) \leq G \leq \mathrm{GO}^+(n,q)$ or $\mathrm{SO}^-(n,q) \leq G \leq \mathrm{GO}^-(n,q)$ for n even, or $\mathrm{SO}(n,q) \leq G \leq \mathrm{GO}(n,q)$ for n odd
IsSpecialOrthogonalGroup(G)	**true** if G is either of the special orthogonal groups $\mathrm{SO}^+(n,q)$ or $\mathrm{SO}^-(n,q)$ for n even, or $\mathrm{SO}(n,q)$ for n odd.
IsUnitaryType(G)	**true** if G preserves a unitary form up to scalar multiplication by a constant. In this case the form is also returned
IsUnitaryGroup(G)	**true** if $\mathrm{SU}(n,q) \leq G \leq \mathrm{GU}(n,q)$
IsSpecialUnitaryGroup(G)	**true** if G is the special unitary group $\mathrm{SU}(n,q)$

If any of these functions returns **true**, the answer is guaranteed to be correct. However, if the value **false** is returned, the most that can be asserted is that G is *probably* not the group in question.

The function **IsClassicalGroup** combines the effect of several of the functions listed in Table 33.6. If it successfully recognizes G as lying between some classical simple group S and its automorphism group, it returns **true** together with a string naming the family to which S belongs and in the case of the symplectic, orthogonal and unitary cases, the form that is fixed by the group.

33.10.2 The Aschbacher Families

According to a theorem of Aschbacher, (M. Aschbacher, [1]On the maximal subgroups of the finite classical groups, *Invent. Math.* 76 (1984) 469-514.), a matrix group G acting on the finite dimensional KG-module V over a

[1] [1]

finite field K satisfies at least one of the following conditions (which we have simplified slightly for brevity):

– (i) G acts reducibly on V;

– (ii) G acts semilinearly over an extension field of K;

– (iii) G acts imprimitively on V;

– (iv) G preserves a nontrivial tensor-product decomposition of V;

– (v) G has a normal subgroup N, acting absolutely irreducibly on V, which is an extra-special p-group or 2-group of symplectic type;

– (vi) G preserves a nontrivial symmetric tensor-product decomposition of V;

– (vii) G acts (modulo scalars) linearly over a proper subfield of K;

– (viii) G contains the special linear group, or one of the classical groups in its natural action over K;

– (ix) G is almost simple.

The philosophy behind the functions described here is to attempt to decide that G lies in at least one of the above categories, and to calculate the associated isomorphism or decomposition explicitly. Groups in Category (i) can be recognised easily by means of the Meataxe functions described in the chapter on R-modules. Considerable progress has also been made in the recognition of groups in Category (viii).

Groups which acts irreducibly but not absolutely irreducibly on V fall theoretically into Category (ii), and furthermore act linearly over an extension field of K. In fact, absolute irreducibility can be tested using the built-in MAGMA functions and, by redefining their field to be an extension field L of K and reducing, they can be rewritten as absolutely irreducible groups of smaller dimension, but over L instead of K. We can therefore concentrate on absolutely irreducible matrix groups.

33.11 Invariants for Finite Groups

In this section G is assumed to be a finite matrix group defined over a field K or a permutation group. MAGMA incorporates powerful new algorithms for computing both ordinary and modular invariants of such finite groups.

Table 33.7 lists the functions in MAGMA for computing within invariant rings. The functions taking a group G can be supplied a matrix group for G,

or a permutation group G followed by (i.e. as the immediately next argument) the desired coefficient field. Using a permutation group speeds up enormously the computation of invariants of a fixed degree since only orbits of monomials under the group action need be computed.

Table 33.7. Invariant Theory Constructions

MAGMA	Meaning
IsInvariant(f, g)	True iff polynomial f is an invariant of group element g
IsInvariant(f, G)	True iff polynomial f is an invariant of group G
MolienSeries(G)	The Molien series of group G, as a rational function
MonomialsOfDegree(P, d)	A indexed set containing all monomials of degree d from polynomial ring P
InvariantsOfDegree(G, d)	A K-basis of the invariants of degree d of G
InvariantsOfDegree(G, d, k)	k linearly independent invariants of degree d of G
PrimaryInvariants(G)	A sequence of primary invariants of G
SecondaryInvariants(G)	A sequence of secondary invariants of G
InvariantRing(G)	Invariant ring of G

In the first example, primary and secondary invariants for a 2-dimensional complex representation of the cyclic group G of order 4 will be computed. Firstly, the Molien series of G is constructed to give information about the possible degrees of the primary invariants. The function **PrimaryInvariants** produces primary invariants of G of degrees 2 and 4. Then the function **SecondaryInvariants** produces two secondary invariants of degrees 0 and 4 with respect to these primary invariants. Note that previous methods for computing primary invariants would usually produce the two invariants $x1^4$ and $x2^4$ of degrees 4 and 4, leading to the secondary invariants $1, x1 * x2, x1^2 * x2^2, x1^3 * x2^3$ of degrees 0, 2, 4, 6.

```
> K<i> := CyclotomicField(4);
> G := MatrixGroup<2, K | [i,0, 0,-i]>;
> M<t> := MolienSeries(G);
> print M;
(t^4 + 1)/(t^6 - t^4 - t^2 + 1)
> P<t> := PowerSeriesRing(IntegerRing());
> print P ! M;
1 + t^2 + 3*t^4 + 3*t^6 + 5*t^8 + 5*t^10 + 7*t^12 +
```

```
    7*t^14 + 9*t^16 + 9*t^18 + O(t^20)
> print PrimaryInvariants(G);
[
    x1*x2,
    x1^4 + x2^4
]
> print SecondaryInvariants(G);
[
    1,
    x1^4
]
```

In this second example, we compute primary and secondary invariants for the regular permutation representation of the cyclic group G of order 4 over the finite field of size 2. Since the characteristic of the field divides the order of the group, we are in the modular case, and there is no Molien series. There are primary invariants of degrees 1, 2, 2, 4, and secondary invariants of degrees 0, 3, 3, 4, 5 with respect to these primary invariants. The whole process takes a small fraction of a second even though the computation of the secondary invariants involves a non-trivial Gröbner basis elimination computation on a module of degree 16—the computation is dramatically improved by using the Hilbert-driven Buchberger algorithm.

```
> K := GF(2);
> G := CyclicGroup(4);
> time PrimaryInvariants(G, K);
[
    x1 + x2 + x3 + x4,
    x1*x2 + x1*x4 + x2*x3 + x3*x4,
    x1*x3 + x2*x4,
    x1*x2*x3*x4
]
Time: 0.019
> time SecondaryInvariants(G, K);
[
    1,
    x1*x2*x3 + x1*x2*x4 + x1*x3*x4 + x2*x3*x4,
    x1^2*x3 + x1^2*x4 + x1*x2^2 + x1*x3^2 + x2^2*x4 +
        x2*x3^2 + x2*x4^2 + x3*x4^2,
    x1^2*x3^2 + x1^2*x3*x4 + x1*x2^2*x4 + x1*x2*x3^2 +
        x2^2*x4^2 + x2*x3*x4^2,
    x1^3*x3*x4 + x1^2*x2^2*x3 + x1^2*x2^2*x4 +
        x1^2*x2*x3^2 + x1^2*x2*x3*x4 + x1^2*x2*x4^2 +
        x1^2*x3^2*x4 + x1^2*x3*x4^2 + x1*x2^3*x4 +
        x1*x2^2*x3^2 + x1*x2^2*x3*x4 + x1*x2^2*x4^2 +
```

```
        x1*x2*x3^3 + x1*x2*x3^2*x4 + x1*x2*x3*x4^2 +
        x1*x3^2*x4^2 + x2^2*x3^2*x4 + x2^2*x3*x4^2 +
        x2*x3^2*x4^2 + x2*x3*x4^3
]
Time: 0.060
```

Part VIII

Algebraic Geometry

34. Elliptic Curves

This chapter contains descriptions for functions pertaining to arithmetic with elliptic curves. The category to which elliptic curves belong is called **CurveEll**; points on curves are of type **CurveEllPt**; and there is also a special category for Kodaira symbols **SymKod**.

This module is currently under construction; the main functions that are present deal with elementary arithmetic over fields, and with more sophisticated questions over the rational field. The latter include functions for minimal models, local information (Tate's algorithm), and the computation of the Mordell-Weil group. These facilities are based on implementations by John Cremona. Refer to his book [Cre92] for details.

34.1 Some Algebraic Geometry

It is not easy to define elliptic curves informally without some knowledge of algebraic geometry. One formal definition is that an elliptic curve over a field is a plane, non-singular cubic curve, of genus 1, with a rational point. Most of the terms in this definition can be made intuitively clear. A plane curve over a field F is given by a polynomial equation $P(X, Y) = 0$ with coefficients in the field; a cubic curve is a curve for which the polynomial P has degree three, and a rational point means, in this context, that there exists a point (x, y) with $x, y \in F$ such that $P(x, y) = 0$. It is somewhat more difficult to explain what genus 1 means. Moreover, in algebraic geometry points are allowed to be at infinity, and therefore it is necessary to cater for that by using projective coordinates $(x : y : z)$ rather than affine coordinates (x, y) for the plane. A projective point in the plane over the field F is not merely a triple (x, y, z) of elements of F, but it is an equivalence class of such triples, where two triples (x, y, z) and $(x', y'z')$ are equivalent if and only if there exists a non-zero element $u \in F$ such that $ux = x'$, $uy = y'$, and $uz = z'$. The triple $(0, 0, 0)$ is not a valid projective point. The way to think of these projective points is that there are those corresponding to 'ordinary' points, namely those with $z \neq 0$, and 'extraordinary' points ('at infinity'), namely those with $z = 0$. For the ordinary points the usual choice of representative has $z = 1$, which gets

back to pairs (x, y) distinguishing them. At infinity, there are those points for which $y \neq 0$, with representatives $(x, 1, 0)$, and those for which $y = 0$ as well, represented by $(1, 0, 0)$. (Perhaps it is clear, from all this dividing out by non-zero elements, how convenient it is to assume the coordinates are from a field!)

A plane cubic curve over F in projective coordinates is given by a homogeneous cubic polynomial equation $C(X, Y, Z) = 0$ with coefficients in F; homogeneous cubic means that every term has total degree 3 (in the variables X, Y, Z). Non-singular now just means that at no point of F are the three partial derivatives of C (with respect to X, Y, Z) zero simultaneously.

Algebraic geometers like to classify objects up to 'birational equivalence'; that is, to consider things equal if there exists a 1–1 transformation between them with maps defined by quotients of polynomials with coefficients in the field F. It turns out that non-singular plane cubic curves of genus 1 with a rational point have a canonical form. That is, there is a birational transformation putting such a curve into that standard form. The canonical form is usually referred to as the Weierstrass form, and it looks like this:

$$Y^2Z = X^3 + aXZ^2 + bZ^3.$$

So any elliptic curve E can be brought into Weierstrass form by a change of projective variables; from now on, elliptic curves will be defined for us by such a Weierstrass form. Moreover, the 'distinguished' rational point on such an elliptic curve will always be $O = (0 : 1 : 0)$. This is only strictly true if the characteristic of the field is not 2 or 3; in those cases the canonical form is slightly more complicated (because it is impossible to 'complete the square', or cube, in the field):

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3,$$

which is called the extended Weierstrass form.

The most important fact for computational purposes is the following: the collection of all points $(x : y : z)$ in the projective plane over F (which is non-empty since it contains O) forms an abelian group (indicated by $E(F)$) of points on the curve. Moreover, the 'addition' in this group is very simple, and can be given explicitly in terms of the coefficients of the points; the point O is the zero element of the group, and the inverse of a point $(x : y : z)$ is simply $(x : -y : z)$. Geometrically, the group law is given by a tangent and chord construction.

The most interesting cases for us are those where the field of definition is either a finite field, or the field of rational numbers. In the first case, the number of points over a finite field must be finite. In the second case this is not true, but the groups are not completely arbitrary.

34.2 Elliptic Curves over Arbitrary Fields

34.2.1 Creation and Basic Quantities

An elliptic curve is defined by giving a sequence of its Weierstrass coefficients; these coefficients must be elements from a field, which will be the field of definition of the curve, or from $\mathbb{Z}$, in which case the field of definition will be $\mathbb{Q}$. The sequence of Weierstrass coefficients may have length 2 or 5, corresponding to the normal and extended Weierstrass forms respectively. These coefficients must define a non-singular curve; singularity gives an error. The function **IsEllipticCurve** can be used to determine if the coefficients define a non-singular curve; if so, this curve is returned as the second value.

```
> F<f> := FiniteField(25);
> E := EllipticCurve( [f^2, -1] );

>> E := EllipticCurve( [f^2, -1] );
                       ^
Runtime error in 'EllipticCurve': Curve is not non-singular
> print IsEllipticCurve( [f^2, -1] );
false
> E := EllipticCurve( [f, -1] );
> ok, E2 := IsEllipticCurve( [f, -1] );
> print ok;
true
> print E2 eq E;
true
> print E;
Elliptic Curve defined by y^2 = x^3 + f*x + 4 over GF(5^2)
```

Now E is an elliptic curve over $F = \mathrm{GF}(25)$. Table 34.1 lists the functions for retrieving various quantities associated with a curve. Note that the discriminant is always non-zero for a valid (i.e. non-singular) elliptic curve.

34.2.2 Points on Elliptic Curves

Points on a curve can be created either by using an element constructor or by coercing a sequence of coefficients; in both cases 2 or 3 coefficients must be specified; in the former case $z = 1$ will be assumed. The point will be normalized if necessary:

```
> P := elt< E | f^2, f^5 >;
> Q := E ! [ 4, 2*f^2, 2 ];
```

Table 34.1. Basic quantities associated with an elliptic curve

MAGMA	Meaning
CoefficientRing(E), **BaseRing**(E)	The field of definition of the elliptic curve E
aInvariants(E), **Coefficients**(E)	The sequence $[a_1, a_2, a_3, a_4, a_6]$ of extended Weierstrass coefficients of E
bInvariants(E)	The sequence $[b_2, b_4, b_6, b_8]$ of b-invariants of E
cInvariants(E)	The sequence $[c_4, c_6]$ of c-invariants of E
Discriminant(E)	The discriminant of E
jInvariant(E)	The j-invariant of E

```
> O := E ! 0;
> print P, Q, O;
(f^2, f^5, 1) (2, f^2, 1) (0, 1, 0)
```

The special point O can be defined either using **Identity**, or by coercing 0 as in the example.

The symbol + is used for the group law on the curve.

```
> R := P+Q;
> print R, R-P eq Q, P+O, O+P eq P, IsIdentity(Q);
(f^10, f^8, 1) true (f^2, f^5, 1) true false
```

Let us simply enumerate all points in $E(F)$. For every possible x we determine whether $x^3 + fx + 4$ is a square in the field; if so we take its square root y, and include $(x : y : 1)$. Afterwards we also include all negates $(x : -y : 1)$, and then the identity O (which is its own negative, of course).

```
> Pts := { E ! [x, Sqrt(P)] : x in F | IsSquare(P)
>   where P is x^3+f*x+4 };
> Pts join:= { -P : P in Pts };
> Pts join:= { O };
> print #Pts;
28
```

So our curve has 28 points defined over F. Let us print the order of all elements.

```
> ord := function(P)
>    Q := P;
>    n := 1;
>    O := Identity(Parent(P));
```

```
>    while Q ne O do
>        Q := P+Q;
>        n +:= 1;
>    end while;
>    return n;
> end function;
> for P in Pts do
>     print P, ord(P);
> end for;
(f^15, 4, 1) 14
(f^20, f^14, 1) 28
(2, f^14, 1) 7
(f^15, 1, 1) 14
(f^4, f^5, 1) 28
(f^5, f^13, 1) 28
(f^10, f^8, 1) 14
(f^2, f^17, 1) 14
(f^5, f, 1) 28
(f^7, f^17, 1) 28
(f^3, f^15, 1) 7
(f, 0, 1) 2
(f^3, f^3, 1) 7
(f^13, f^21, 1) 28
(f^4, f^17, 1) 28
(f^11, f^22, 1) 7
(0, 2, 1) 28
(f^11, f^10, 1) 7
(f^20, f^2, 1) 28
(f^10, f^20, 1) 14
(f^13, f^9, 1) 28
(f^21, f^14, 1) 4
(f^2, f^5, 1) 14
(0, 1, 0) 1
(2, f^2, 1) 7
(f^7, f^5, 1) 28
(f^21, f^2, 1) 4
(0, 3, 1) 28
>
> print 14 * (E ! [0,3,1]);
(f, 0, 1)
```

We see that the points defined over GF(5) do not form a subgroup (because the curve is not defined over GF(5)). Finding points over a subfield can be done with a simple check:

```
> print { P : P in Pts | P[1] in GF(5) and P[2] in GF(5) };
{ (0, 2, 1), (0, 3, 1), (0, 1, 0) }
```

but to find points over an extension field requires **Lift**:

```
> M<m> := ext< F | 2 >;
> Ee := Lift(E, M);
> ePts := { Ee ! [x, Sqrt(P)] : x in M | IsSquare(P)
>   where P is x^3+f*x+4 };
> ePts join:= { -q : q in Epts };
> ePts join:= { Ee ! 0 };
> print #ePts;
672
> print exists{ P : P in ePts | ord(P) eq 672 };
false
```

Observe that the curve defines a non-cyclic group of order 672, which has $E(F)$ as a subgroup.

Lift(E, F) creates a new elliptic curve by coercing the coefficients of the curve E into the field F. If this results in a singular elliptic curve, then an error occurs. There is also a three-argument version of **Lift**, where the third argument consists of a map specifying the embedding of the original field of definition into the extension.

34.3 Elliptic Curves over The Rationals

34.3.1 Minimal Models

Two elliptic curves E and E' are *isomorphic* if there is a change of coordinates of a particular form which transforms the equation of E into the equation of E'. This form is:

$$\begin{aligned} x &= u^2x' + r \\ y &= u^3y' + su^2x' + t \end{aligned}$$

where $r, s, t \in K$ and $u \in K^*$ for some field K. If the coefficients of the equations also lie in K, then the curves are said to be *isomorphic over* K. For example, the curves $y^2 = x^3 + 17$ and $y^2 = x^3 - 17$ are isomorphic over $\mathbb{C}$ with parameters $\langle r, s, t, u\rangle = \langle 0, 0, 0, i\rangle$, but they are not isomorphic over $\mathbb{Q}$.

The effect of such a transformation on the coefficients is:

$$\begin{aligned} ua_1' &= a_1 + 2s \\ u^2a_2' &= a_2 - sa_1 + 3r - s^2 \\ u^3a_3' &= a_3 + ra_1 + 2t \\ u^4a_4' &= a_4 - sa_3 + 2ra_2 - (rs+t)a_1 + 3r^2 - 2st \\ u^6a_6' &= a_6 + ra_4 - ta_3 + r^2a_2 - rta_1 + r^3 - t^2 \end{aligned}$$

whence $u^{12}\Delta' = \Delta$ and $j' = j$.

It can be seen from the above that suitable choice of u transforms any elliptic curve with coefficients in $\mathbb{Q}$ into an elliptic curve with coefficients in $\mathbb{Z}$. The resulting discriminant is then an integer. The function **IntegralModel**(E) returns such a curve:

```
> EE := EllipticCurve([ 1/2, 1, -1/2, 1/2, 0 ]);
> print EE;
Elliptic Curve defined by y^2 + 1/2*x*y - 1/2*y = x^3 + x^2
+ 1/2*x over Rational Field
> print Discriminant(EE);
-19/128
> IE := IntegralModel(EE);
> print IE;
Elliptic Curve defined by y^2 + x*y - 4*y = x^3 + 4*x^2 +
8*x over Rational Field
> print Discriminant(IE);
-608
```

Amongst all such curves over $\mathbb{Z}$, those for which $|\Delta|$ is minimal are called *global minimal models*. By careful choice of r, s and t a global minimal model can be found for which $a_1, a_3 \in \{0, 1\}$ and $a_2 \in \{-1, 0, 1\}$; this is called the *reduced minimal model* (henceforth just *minimal model*), and it is unique. Comparing minimal models thus gives an easy way to tell if two curves are isomorphic over $\mathbb{Q}$.

The function **MinimalModel**(E) returns the minimal model M of E, together with the maps $E \to M$ and $M \to E$. The function **IsIsomorphic**(E, F) returns true if the two curves are isomorphic over $\mathbb{Q}$. If so, it returns the mapping between them as its second return value.

```
> ME, f1, f2 := MinimalModel(IE);
> print ME;
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + 1 over
Rational Field
> iso, map := IsIsomorphic(ME, IE);
> print iso;
true
```

```
> P := elt<ME | 1, -3>;
> print map(P);
(0, 0, 1)
> print f1($1) eq P;
true
>
> E1 := EllipticCurve([ 0, 17 ]);
> E2 := EllipticCurve([ 0, -17 ]);
> print IsIsomorphic(E1, E2);
false
```

Table 34.2. Functions for models

MAGMA	Meaning
IntegralModel(E)	Given an elliptic curve E with coefficients in $\mathbb{Q}$, returns an isomorphic curve with coefficients in $\mathbb{Z}$
MinimalModel(E)	Returns (i) the minimal model M of E (ii) the map $E \to M$ (iii) the map $M \to E$
IsIsomorphic(E, F)	**true** if curves E and F are isomorphic; if so, the map $E \to F$ is returned as the second return value

34.3.2 Local Information

One method of computing the minimal model of a curve is to apply *Tate's algorithm*. Given an elliptic curve E and a prime p, Tate's algorithm produces the following information: e_p, the exponent of p in the discriminant of the minimal model M of E; f_p, the exponent of p in the conductor of E; c_p, the Tamagawa number of E at p; and k_p, the Kodaira symbol classifying the type of reduction of E at p. The algorithm is not described here - see [Cre92] for details.

The Tamagawa number $c_p = [E(\mathbb{Q}_p) : E^0(\mathbb{Q}_p)]$, where $E^0(\mathbb{Q}_p)$ is the subgroup of the group $E(\mathbb{Q}_p)$ of p-adic points of E whose reduction modulo p is non-singular. The Kodaira symbol k_p classifies the type of reduction of E at p as one of: I_0 for good reduction - in this case p does not divide the discriminant of M; $I_n (n > 0)$ for bad multiplicative reduction; and one of $I_n^*, II, II^*, III, III^*, IV$ or IV^* for bad additive reduction.

Table 34.3 lists the functions available for accessing the local information of a curve. Note in particular that **BadPrimes**(E) returns only the primes dividing the discriminant of M, not E; so while it will be safe to lift M into $GF(p)$, it might not be possible for E:

Table 34.3. Local information

MAGMA	Meaning
BadPrimes(E)	Sequence of primes dividing the discriminant of M, the minimal model of the elliptic curve E
LocalInformation(E, p)	Local information tuple <p, e_p, f_p, c_p, k_p> of M at the prime p, where the elements of the tuple are as defined above
LocalInformation(E)	Sequence of local information tuples of M for each bad prime
Conductor(E)	Conductor of M
TamagawaNumber(E, p)	Tamagawa number of M at p
TamagawaNumbers(E)	Sequence of Tamagawa numbers of M for each bad prime
KodairaSymbol(E, p)	Kodaira symbol of M at p
KodairaSymbols(E)	Sequence of Kodaira symbols of M for each bad prime

```
> F := EllipticCurve([ 0, 17*5^6 ]);
> MF := MinimalModel(F);
> print BadPrimes(F);
[ 2, 3, 17 ]
> print Lift(F, GF(5));

>> print Lift(F, GF(5));
            ^
Runtime error in 'Lift': Curve is not non-singular
> print Lift(MF, GF(5));
Elliptic Curve defined by y^2 = x^3 + 2 over GF(5)
```

Kodaira symbols belong to the special category **SymKod**. They can be obtained from the local information of a curve, or created by calling **KodairaSymbol**(s), where s is one of the strings: `"I0"`; `"I`n`"` or `"I`n`*"` for $n > 0$; `"II"`; `"III"`; `"IV"`; `"II*"`; `"III*"`; or `"IV*"`. The special symbols `"In"` and `"In*"` may also be used; they compare equal to the corresponding symbols with n replaced by any positive integer.

```
> S := KodairaSymbols(ME);
> print S;
[ I5, I1 ]
> In := KodairaSymbol("In");
> print In;
In
> print S[1] eq In;
```

```
true
> print In eq S[2];
true
> print S[1] eq S[2];
false
```

34.3.3 The Mordell-Weil Group

As mentioned at the beginning of the chapter, the points on an elliptic curve form an abelian group - the Mordell-Weil group. If the curve has coefficients in $\mathbb{Q}$, then MAGMA has various functions to calculate information about this group. These are summarized in Table 34.4.

Table 34.4. Group functions

MAGMA	Meaning
TorsionSubgroup(E)	(i) abelian group isomorphic to the torsion subgroup of the group G of rational points of E (ii) mapping from this group into E
MordellWeilRank(E), **Rank**(E)	The computed rank of G
MordellWeilRankBounds(E), **RankBounds**(E)	The computed lower and upper bounds on the rank of G
MordellWeilGroup(E)	(i) abelian group isomorphic to G (ii) mapping from this group into E
Generators(E)	Sequence $[P_1, \ldots, P_k]$ of generators of G
Order(P)	The order of P in G; if P is of infinite order, 0 is returned

The Mordell-Weil group of a curve over $\mathbb{Q}$ is the product of a finite torsion part T and a free abelian group F of rank $r \geq 0$ ($F \cong \mathbb{Z}^r$). By a theorem of Mazur, the structure of T is either C_k for $1 \leq k \leq 10$, C_{12}, or $C_{2k} \times C_2$ for $1 \leq k \leq 4$. The function **TorsionSubgroup**(E) returns an abelian group isomorphic to T and a mapping from this group into E.

```
> T,h := TorsionSubgroup(ME);
> print T;
Abelian Group isomorphic to Z/5
Defined on 1 generator
Relations:
    5*T.1 = 0
> print [ h(t) : t in T ];
[ (0, 1, 0), (-1, 1, 1), (1, -3, 1), (1, 1, 1), (-1, -1, 1) ]
```

```
> print [ Order(P) : P in $1 ];
[ 1, 5, 5, 5, 5 ]
> T,h := TorsionSubgroup(E1);
> print T;
Abelian Group of order 1
> print h(T.0);
(0, 1, 0)
> P := E1 ! [ 2, 5 ];
> print Order(P);
0
> print Order(E1 ! 0);
1
```

Computing the torsion subgroup is fairly straightforward, and, while it may take some time for large examples, it is the easiest part of the Mordell-Weil group to compute. In contrast, the procedure for calculating the rank of the group (two-descent) is not guaranteed to return the correct result; it will, however, provide bounds on the rank, and with luck they will be equal. The function **MordellWeilRank**(E) (or just **Rank**(E)) returns the lower bound on the rank; **MordellWeilRankBounds**(E) (or **RankBounds**(E)) returns both the lower and upper bounds. Note that computing the rank is much faster than computing the full group, so these functions should be used if they suffice.

```
> E := EllipticCurve([ 0, 0, 1, -7, 6 ]);
> time print MordellWeilRank(E);
3
Time: 0.039
> F := EllipticCurve([ 0, 36861504658225, 0,
>   1807580157674409809510400, 0 ]);
> time print Rank(F);
13
Time: 131.449
> F := EllipticCurve([ -17^2, 0 ]);
> print RankBounds(F);
0 2
```

The function **MordellWeilGroup**(E) returns an abelian group isomorphic to the Mordell-Weil group and a mapping from this group into E. This is by far the slowest function so far, as it requires searching for the generators. These can be got by using the mapping on the generators of the group, or from the intrinsic **Generators**(E).

```
> E := EllipticCurve([ 0, 0, 1, -7, 6 ]);
```

```
> time G,h := MordellWeilGroup(E);
Time: 0.439
> print G;
Abelian Group isomorphic to Z + Z + Z
Defined on 3 generators (free)
> print Generators(E);
[ (0, 2, 1), (-1, 3, 1), (-2, 3, 1) ]
> print [ h(g) : g in Generators(G) ];
[ (-2, 3, 1), (-1, 3, 1), (0, 2, 1) ]
```

The search for generators requires the concept of the *height* of a point. If $P = (x, y)$ is a point on E, then the *naive height* of P is $h(P) = \log \max\{|m|, |n|\}$, where $x = \frac{m}{n}$. The *canonical height* $\hat{h}(P)$ is the sum of the *local heights* $\hat{h}_p(P)$ at each prime p and ∞ also. If E has good reduction at p, and p does not divide n, then $\hat{h}_p(P)$ will be 0. The functions dealing with heights are summarized in Table 34.5.

Table 34.5. Height related functions

MAGMA	Meaning
NaiveHeight(P)	$\log \max\{\|m\|, \|n\|\}$, where $\frac{m}{n}$ is the x-coordinate of P
LocalHeight(P, p)	Local height $\hat{h}_p(P)$ of P at the prime p; to specify $p = \infty$ use $p = 0$
Height(P)	The canonical height $\hat{h}(P) = \Sigma \hat{h}_p(P)$ where the sum is over all primes p and ∞
HeightPairing(P, Q)	Height pairing $\hat{h}(P, Q) = \frac{1}{2}(\hat{h}(P + Q) - \hat{h}(P) - \hat{h}(Q))$
Regulator(E)	Regulator of E

```
> P := Generators(E)[1];
> print P;
(0, 2, 1)
> print NaiveHeight(P);
 0.E-92
> print LocalHeight(P, 0);        /* infinity */
0.99090633315308762618
> print [ LocalHeight(P, p) : p in BadPrimes(E) ];
[  0.E-92 ]
> print &+$1 + $2;
0.99090633315308762618
> print Height(P);
0.99090633315308762618
```

The process of computing the rank provides the generators of a subgroup of G, the Mordell-Weil group of E. The canonical heights of these points provide a bound on the canonical heights of the generators of G, and a result of Silverman bounds the difference between a point's naive and canonical heights (the function **SilvermanBound**(E) can be used to get this bound). So we get a bound on the naive heights of the generators of G and simply search up to this bound. This bound gets quite large quickly, and for most curves this computations is infeasible; if the rank provides sufficient information it should be used instead.

One other quantity of interest is the *regulator* of E. If the generators of the non-torsion part of G are $P_1, \ldots, P_r$, then the regulator of E is $R(E) = |det(\hat{h}(P_i, P_j))|$ where $\hat{h}(P, Q) = \frac{1}{2}(\hat{h}(P + Q) - \hat{h}(P) - \hat{h}(Q))$ is the height pairing of P and Q. The functions **Regulator**(E) and **HeightPairing**(P, Q) can be used to compute these values.

```
> S := Generators(E);
> A := MatrixAlgebra(RealField(), #S);
> m := A ! [ HeightPairing(P, Q) : P,Q in S ];
> print Determinant(m);
0.41714355875838417559
> print Regulator(E);
0.41714355875838621257
```

Part IX

Geometric and Combinatorial Structures

35. Enumerative Combinatorics

This chapter presents some of the tools provided by MAGMA for enumerative combinatorics. As facilities in this area were still under development at the time of writing, the user should consult the online help system for an up-to-date list of the facilities.

35.1 Elementary Counting Functions

Table 35.1. Counting functions

MAGMA	Meaning
Factorial(n)	$n!$
NumberOfPermutations(n, k)	Number of permutations of n distinct objects, taken k at a time
Binomial(n, k)	Binomial coefficient $\binom{n}{k}$
Multinomial(n, Q)	Given integer n and $Q = [r_1, \ldots, r_k]$, where $n = r_1 + \cdots + r_k$, returns multinomial coefficient $\frac{n!}{r_1! \ldots r_k!}$
Fibonacci(n)	n^{th} Fibonacci number, where $F_0 = 0, F_1 = 1$ (returns $(-1)^{n-1} F_{-n}$ if $n < 0$)
StirlingFirst(m, n)	Stirling number of the first kind, $\left[{m \atop n}\right]$
StirlingSecond(m, n)	Stirling number of the second kind, $\left\{{m \atop n}\right\}$
EulerianNumber(n, r)	Number $E(n, r)$ of permutations p of $\{1, \ldots, n\}$ having exactly k ascents (i.e., places where $p_i < p_{i+1}$)
HarmonicNumber(n)	n^{th} harmonic number
BernoulliNumber(n)	n^{th} Bernoulli number B_n, as a rational
BernoulliApproximation(n)	Approximation to B_n, as a finite-precision free real
BernoulliPolynomial(n)	n^{th} Bernoulli polynomial

The elementary counting functions of enumerative combinatorics available in MAGMA are listed in Table 35.1. Some brief explanation of the less familiar functions is given below. The reader is referred to [Rio58] for a full account of the definition and properties of these functions.

The Stirling number of the first kind, $\left[{m \atop n}\right]$ or $s(m,n)$, is returned as the value of **StirlingFirst**(m,n) and may be interpreted as $(-1)^{m-n}$ times the number of ways of arranging m objects into n cycles. The Stirling number of the second kind, $\left\{{m \atop n}\right\}$ or $S(m,n)$, is returned by **StirlingSecond**(m,n) and gives the number of ways of partitioning a set of m objects into n non-empty subsets.

The Bernoulli number B_n, a rational number, is calculated by the function **BernoulliNumber**(n). A real approximation (calculated to the current precision) is given by **BernoulliApproximation**(n). For example:

```
> bern50 := BernoulliNumber(50);
> print bern50;
495057205241079648212477525/66

> bern50app := BernoulliApproximation(50);
> print bern50app;
7500866746076964366855720.07275390625
> print Round(bern50app * 66);
495057205241079648212477525
```

BernoulliPolynomial(n) returns the Bernoulli polynomial $B_n(x)$, which is defined as $B_n(x) = \sum_{k=0}^{n} \binom{n}{k} B_k x^{n-k}$, where B_n is the n^{th} Bernoulli number. For instance:

```
> p<x> := BernoulliPolynomial(3);
> print p;
x^3 - 3/2*x^2 + 1/2*x
> print &+[Binomial(3,k) * BernoulliNumber(k) * x^(3-k):
>     k in [0..3]];
x^3 - 3/2*x^2 + 1/2*x
```

35.2 Subsets of a Finite Set

Let S be an enumerated set of cardinality n. The total number of subsets of S, including the empty set and S itself, is 2^n. The function **Subsets**(S) returns all of these subsets, as an enumerated set of enumerated sets. For example:

Table 35.2. Subsets of a finite set

MAGMA	Meaning
Subsets(S)	Set of all subsets of S
Subsets(S, k)	Set of all k-subsets of S
Multisets(S, k)	Set of all k-multisets formed from elements of S
Subsequences(S, k)	Set of all k-sequences formed from elements of S
Permutations(S, k)	Set of all permutations (as sequences) of elements of S, taken k at a time
Permutations(S)	Set of all permutations (as sequences) of elements of S

```
> print Subsets({1..3});
{
    { 1 },
    {},
    { 1, 3 },
    { 2 },
    { 3 },
    { 1, 2, 3 },
    { 2, 3 },
    { 1, 2 }
}
```

The function **Subsets**(S) should be distinguished from **PowerSet**(S). Both functions return the same mathematical object, that is, the *power set of* S. However, **Subsets** returns the subsets explicitly, whereas **PowerSet** merely returns them as a formal object.

Now, let k be an non-negative integer. A k-subset of S is a subset of S having cardinality k, that is, an unordered set of size k whose elements are distinct elements of S. The function **Subsets**(S, k) returns all the k-subsets of S, as an enumerated set of enumerated sets.

Three other concepts resembling the notion of subset-ness may be obtained by allowing the subsets to be ordered or to have repeated elements. The function **Multisets**(S, k) returns an enumerated set containing all the multisets of size k (counting multiplicity) whose elements are in S. Similarly, **Subsequences**(S, k) returns an enumerated set containing all the sequences (i.e., ordered multisets) of length k whose elements are in S. Finally, **Permutations**(S, k) returns an enumerated set containing all the distinct-element sequences (i.e., ordered sets) of length k whose elements are in S. There is a special permutation function, **Permutations**(S), which may be used when k equals the cardinality of S. For all of these functions, if k is greater than n then an empty set will be returned.

Table 35.2 summarizes the subset functions. In the table, a k-subset is a subset with cardinality k, a k-multiset is a multiset with cardinality k (counting multiplicity), and a k-sequence is a sequence of length k (i.e., cardinality k, counting multiplicity).

If the user wishes to know only the total number of subsets of any of these types, the functions in Table 35.1 should be used. The number of plain k-subsets is **Binomial**(n), the number of k-multisets is **Binomial**$(n + k - 1, k)$, the number of k-sequences is $n\hat{}k$, and the number of k-permutations is **NumberOfPermutations**(n, k) (or **Factorial**(n) if $k = n$).

For example:

```
> S := {"N", "S", "E", "W"};
> print Subsets(S, 3);
{
    { E, W, N },
    { W, S, N },
    { E, S, N },
    { E, W, S }
}
> print Permutations({"a", "e", "i", "o", "u"}, 2);
{
    [ e, a ], [ u, i ], [ i, a ], [ i, e ], [ a, o ],
    [ u, a ], [ a, u ], [ u, e ], [ o, u ], [ e, o ],
    [ e, u ], [ o, i ], [ a, i ], [ i, o ], [ i, u ],
    [ u, o ], [ o, a ], [ e, i ], [ o, e ], [ a, e ]
}
```

35.3 Partitions

Table 35.3. Partitions

MAGMA	Meaning
Partitions(n)	The unrestricted partitions of n
NumberOfPartitions(n)	The number of unrestricted partitions of n
RestrictedPartitions(n, T)	The partitions of n, restricted to elements of set T

Table 35.3 lists the functions concerned with partitions.

A partition of an integer n is a representation of it as the sum of positive integers, ignoring order. **Partitions**(n) returns a sequence containing all the partitions of n, each partition being given as a non-increasing sequence of positive integers.

```
> print Partitions(6);
[
    [ 6 ],
    [ 5, 1 ],
    [ 4, 2 ],
    [ 4, 1, 1 ],
    [ 3, 3 ],
    [ 3, 2, 1 ],
    [ 3, 1, 1, 1 ],
    [ 2, 2, 2 ],
    [ 2, 2, 1, 1 ],
    [ 2, 1, 1, 1, 1 ],
    [ 1, 1, 1, 1, 1, 1 ]
]
```

The function **NumberOfPartitions**(n), returns the number of these partitions:

```
> print NumberOfPartitions(6), NumberOfPartitions(600);
11 458004788008144308553622
```

The function **RestrictedPartitions**(n, T) is similar to **Partitions**(n), except that it returns only those partitions whose components all lie in the set T of positive integers. For example, in Australia, the coins of value less than one dollar have the values 5, 10, 20 and 50 cents. Therefore, the possible ways of giving change for a 50 cent piece are those listed in the output below:

```
> print RestrictedPartitions(50, {5, 10, 20});
[
    [ 20, 20, 10 ],
    [ 20, 20, 5, 5 ],
    [ 20, 10, 10, 10 ],
    [ 20, 10, 10, 5, 5 ],
    [ 20, 10, 5, 5, 5, 5 ],
    [ 20, 5, 5, 5, 5, 5, 5 ],
    [ 10, 10, 10, 10, 10 ],
    [ 10, 10, 10, 10, 5, 5 ],
    [ 10, 10, 10, 5, 5, 5, 5 ],
    [ 10, 10, 5, 5, 5, 5, 5, 5 ],
    [ 10, 5, 5, 5, 5, 5, 5, 5, 5 ],
```

```
      [ 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 ]
]
```

35.4 Example: Bell Numbers

The Bell number (or exponential number) B_n may be defined as the number of partitions of a set of n objects into non-empty subsets. ($B_0 = 1$, since an empty set may be partitioned uniquely into zero non-empty subsets.) There are several formulae for the Bell numbers (see [LiW92], pp. 105–107 and [Ber71], pp. 42–44), some of which will be implemented below. Note that these formulae/implementations are not equally efficient, but collectively the examples demonstrate several techniques in enumerative combinatorics: counting functions, summation of sequences, recursive functions and recursive sequences, generating functions, and approximations to infinite series.

The first formula is based on the Stirling numbers of the second kind:

$$B_0 = 1; \quad B_n = \sum_{k=1}^{n} \left\{ {n \atop k} \right\} \quad \text{for } n \geq 1.$$

It is easily implemented by constructing a sequence containing the Stirling numbers and then summing its terms using the reduction operator.

```
> Bell := func< n | n eq 0 select 1 else
>     &+[ StirlingSecond(n, k) : k in [1..n] ] >;

> time print Bell(3), Bell(15);
5 1382958545
Time: 0.030
> time print Bell(100);
4758539127676483365879076884138720782636366968682561146661\
6
3346375591144978924426226727240442177563069535578825607 51
Time: 4.000
```

The next formula is a recursive summation:

$$B_0 = 1; \quad B_n = \sum_{k=0}^{n-1} \binom{n-1}{k} B_k \quad \text{for } n \geq 1.$$

There are two approaches to the implementation of this formula. The simplest approach is to implement it as a recursive function.

```
> Bell := func< n | n eq 0 select 1 else
```

```
>     &+[ Binomial(n-1, k) * $$(k) : k in [0..n-1] ] >;

> time print Bell(3), Bell(15);
5 1382958545
Time: 16.271
```

This implementation is inefficient since whenever the function is called for a particular value of n, every B_k for $k \leq n$ has to be calculated, and each of these calculations, in turn, involves finding all the B_i for $i \leq k$. Since the intermediate values of B_k are not remembered by the system, they must be computed repeatedly. Therefore, it is preferable to calculate and store all B_k for $k \leq N$, where N is a pre-determined upper limit, using a recursively-defined sequence. This technique is followed in the code below. Since MAGMA's sequences are indexed beginning at 1, the $k = 0$ part of the summation requires special handling, and B_0 will not be available from the sequence:

```
> Bellseq := func< N |
>   [ n eq 1 select 1 else
>     (1 + &+[ Binomial(n-1, k) * Q[k] : k in [1..n-1] ]
>     where Q is Self() )
>       : n in [1..N] ] >;

> time print Bellseq(15);
[ 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570,
4213597, 27644437, 190899322, 1382958545 ]
Time: 0.030
> time b100 := Bellseq(100);
Time: 4.440
> print b100[100];
47585391276764833658790768841387207826363669686825611466616\
33463755911449789244262267272404421775630695355788256075 1
```

Note that in the recursive sequence above, Q is defined as the outer sequence, by means of the **where**-construction. This handles a difficult programming issue: the **Self**() and **Self**(i) functions refer to the *innermost* sequence, so **Self**(k) cannot be used in the inner sequence to refer to the outer sequence. Instead, $Q[k]$ is used.

The B_n may also be evaluated as an approximation to an infinite series of real numbers:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}.$$

Section 26.8.3 gives details of the MAGMA functions for such summations; the implementation below uses **PositiveSum**, since all the terms in the series are positive. *The value to which the precision is set only gives fully accurate*

answers for moderately small n; some numerical analysis would be required to find a better value. Nonetheless, it will serve to demonstrate the principles involved.

```
> Bell := function(n)
>     R := RealField();
>     AssertAttribute(FldPr, "Precision", 2*n);
>     m := map< IntegerRing() -> R |
>         k :-> k^n / Factorial(k) >;
>     return Round(PositiveSum(m, 0) / Exp(1));
> end function;

> time print Bell(3), Bell(15);
5 1382958545
Time: 0.440
> time print Bell(100);
4758539127676483365879076884138720782636366968682561146661 6\
3346375591144978924426226727240442177563069535578825607 51
Time: 34.823
```

A final approach to the calculation of Bell numbers is to use a *generating function*. The following generating function may be used to obtain Bell numbers:

$$\sum_{n=0}^{\infty} \frac{B_n}{n!} t^n = e^{e^t - 1}$$

and the corresponding MAGMA code is:

```
> Bell := function(n)
>     Q := RationalField();
>     RR<t> := PowerSeriesRing(Q : Precision := n+1);
>     p := Exp(Exp(t) - 1);
>     return Coefficient(p, n) * Factorial(n);
> end function;

> time print Bell(3), Bell(15);
5 1382958545
Time: 0.040
> time print Bell(100);
4758539127676483365879076884138720782636366968682561146661 6\
3346375591144978924426226727240442177563069535578825607 51
Time: 35.163
```

36. Overview of Incidence Structures

The following chapters explain the major constructions and operations for the fundamental finite incidence structures in MAGMA. These structures are:

- General incidence structures that do not obey special axioms (category **Inc**);
- Undirected and directed graphs (categories **GrphUnd** and **GrphDir**);
- Near-linear spaces and linear spaces (categories **IncNsp** and **IncLsp**);
- Designs (category **Dsgn**);
- Projective and affine planes (categories **PlaneProj** and **PlaneAff**).

Because of the strong relationships that exist between the different kinds of incidence structures, MAGMA provides many transfer functions to allow easy translation between these categories. Moreover, it is possible to transfer between incidence structures and *codes*.

MAGMA's approach to incidence structures places emphasis on methods for creating the structures and on the calculation of automorphism groups. The latter, of course, includes the ability to test isomorphism.

36.1 Points and Blocks of Incidence Structures

A feature of incidence structures that distinguishes them from the algebraic structures supported by MAGMA is that they are genuine *multi-sorted algebras*, having more than one *carrier set*. For example, each graph has a vertex-set and an edge-set; each design has a point-set and a block-set; and each plane has a point-set and a line-set. It is these "sets" that act as parents for the objects of the incidence structure (points, blocks, and so on), not the incidence structure itself. Since they are not ordinary enumerated sets, they belong to special categories.

In general, every function or constructor that returns an incidence structure as its principal value also returns the point-set and the block-set (or vertex-set and edge-set, etc.) as its second and third return values. When assigning an incidence structure to an identifier, the user may wish to assign these carrier sets at the same time:

I, *P*, *B* := incstr constructor;

For example, the following multiple assignment constructs the affine plane of order 2, together with its point-set and line-set:

```
> pl, ptset, lnset := AffinePlane(2);
> pl;
Affine Plane AG(2, 2)
> ptset;
Point-set of Affine Plane AG(2, 2)
> Random(ptset);
( 0, 1 )
> lnset;
Line-set of Affine Plane AG(2, 2)
> Random(lnset);
< 1 : 1 : 0 >
```

Many of the access operations for I involve P or B, since they are the parents of the basic objects in the structure. For example, #P and #B are the number of points and the number of blocks, and the i^{th} point and i^{th} block are $P.i$ and $B.i$. There are special categories for individual points and blocks.

In the theory of incidence structures, one speaks of a point as being (not) *incident with* a block. Less formally, one speaks of a point as being (not) in a block. The two usages are equivalent, provided that repeated blocks are distinguished in the case of non-simple incidence structures. Because of this, a block may be regarded as a set of points, and a limited number of set-style operations are supported for blocks in MAGMA, but a block is distinguished from a mere enumerated set of points. For example:

```
> pts := Set(ptset); // or Points(pl)
> pts;
{ ( 0, 0 ), ( 1, 0 ), ( 0, 1 ), ( 1, 1 ) }
> lns := Set(lnset); // or Lines(pl)
> lns;
{
    < 1 : 0 : 0 >,
    < 1 : 1 : 0 >,
```

```
    < 1 : 0 : 1 >,
    < 1 : 1 : 1 >,
    < 0 : 1 : 0 >,
    < 0 : 1 : 1 >
}
> ln := Random(lns); ln;
< 0 : 1 : 0 >
> Set(ln);
{ ( 0, 0 ), ( 1, 0 ) }
```

The function **IncidenceMatrix**(I) returns the incidence matrix m of I, as an integer matrix with $\#P$ rows and $\#B$ columns. The matrix entry m_{ij} is 1 if point i is incident with block j, else 0. For example:

```
> incmat := IncidenceMatrix(pl);
> incmat;
[1 1 0 0 1 0]
[0 0 1 1 1 0]
[1 0 0 1 0 1]
[0 1 1 0 0 1]
> lnset.4 eq {ptset.2, ptset.3};
true
```

36.2 Equality and Isomorphism

Two incidence structures I and J belonging to the same category are *equal* if the sets of points and blocks over which they are defined are equal. The expression I **eq** J returns **true** if I equals J, and I **ne** J returns **true** if I does not equal J.

An *isomorphism* from an incidence structure $I = (P_I, B_I)$ to another incidence structure $J = (P_J, B_J)$ is a mapping from I to J, such that the point-set P_I and block-set B_I are mapped to P_J and B_J respectively, and the incidence relation is preserved. The function **IsIsomorphic**(I, J) returns **true** if I and J are isomorphic. If they are isomorphic, it also returns the isomorphism as a mapping from I to J. For example:

```
> pl, ptset, lnset := AffinePlane(2);
> incmat := IncidenceMatrix(pl);
> SwapRows(~incmat, 1, 3);
> mypl := AffinePlane< 4 | incmat >;
> mypl : Maximal;
Affine Plane of order 2
```

```
Points: {@ 1, 2, 3, 4 @}
Lines:
    {1, 3},
    {3, 4},
    {2, 4},
    {1, 2},
    {2, 3},
    {1, 4}
> pl eq mypl;
false
> y, m := IsIsomorphic(pl, mypl);
> y;
true
> m;
Mapping from: PlaneAff: pl to PlaneAff: mypl
```

The function **IsIsomorphic** has a parameter **AutomorphismGroups** which can take any of the string values `"None"`, `"Left"`, `"Right"` (the default), or `"Both"`. It specifies which, if any, of the automorphism groups of I and J (see below) are to be constructed first. For certain examples, the isomorphism test progresses faster when this parameter is changed from its default value.

36.3 Automorphism Groups and Subgroups

An automorphism of an incidence structure $I = (P, B)$ is an isomorphism of I onto itself, that is, a mapping such that P and B map to themselves and the incidence relation is preserved.

36.3.1 Action on Points

The function **AutomorphismGroup**(I) returns the group of automorphisms of I, as a permutation group. MAGMA selects a suitable support for representing the automorphisms as permutations. If there are no repeated blocks then the support is taken to be P; otherwise, the support is $P \cup B$. However, because of the awkwardness of printing permutations on complicated objects, the group that is returned is a permutation group $G = \mathrm{Aut}(I)$ on the natural support $\Omega = \{1, 2, \ldots, n\}$, where n is the cardinality of the support and $i \in \Omega$ corresponds to the i^{th} element of the support.

The automorphism group G has natural actions on both the point-set and block-set. These actions are provided through use of the G-set mechanism. As well as returning the automorphism group G as its principal value,

AutomorphismGroup returns a G-set giving the action of G on the points and a G-set giving the action of G on the blocks. Actions of G on the structure may be computed using the G-set versions of orbits, stabilizers, and so on, as described in Chapter 32.

It is often convenient to represent an automorphism of I as a mapping from I to I rather than a permutation. For this reason, a fourth and a fifth value are returned by **AutomorphismGroup**: the parent M for all automorphisms of I when represented as maps; and a transfer map t from G to M. If $g \in G$ is an automorphism represented as a permutation, then the corresponding map is $t(g)$ or g@t. The parent M and transfer map t can also be created without calling **AutomorphismGroup**(I) explicitly by the function **Aut**(I) which returns M and t. In this case, the domain of t is the full symmetric group of the appropriate degree so an automorphism of I can applied to by t.

The example below constructs the automorphism group and associated objects for the affine plane of order 2. It demonstrates the computation of images, orbits and stabilizers, and how to convert the representation of an automorphism of the plane from an element of G into a mapping:

```
> pl, ptset, lnset := AffinePlane(2);
> G, gspt, gsln, pm, t := AutomorphismGroup(pl);
> G;
Permutation group G acting on a set of cardinality 4
Order = 24 = 2^3 * 3
    (2, 4)
    (2, 3)
    (1, 2)(3, 4)
> gspt, gsln;
GSet{ ( 0, 0 ), ( 1, 0 ), ( 0, 1 ), ( 1, 1 ) }
GSet{
    < 1 : 0 : 0 >,
    < 1 : 1 : 0 >,
    < 1 : 0 : 1 >,
    < 1 : 1 : 1 >,
    < 0 : 1 : 0 >,
    < 0 : 1 : 1 >
}
> pm;
Set of all automorphisms of Affine Plane AG(2, 2)
> t;
Mapping from: GrpPerm: G to PowMap: pm

> g := Random(G); g;
(1, 4)(2, 3)
```

```
> Image(g, gspt, ptset.2);
( 0, 1 )
> Image(g, gsln, lnset.1);
< 1 : 0 : 1 >
> Orbit(G, gsln, lnset.1) eq gsln;
true
> Stabilizer(G, gsln, lnset.2);
Permutation group acting on a set of cardinality 4
    (2, 3)
    (1, 4)(2, 3)

> au := t(g); au;
Mapping from: PlaneAff: pl to PlaneAff: pl
> au(lnset.5);
< 0 : 1 : 1 >
> au(ptset.2) eq ptset.3;
true
```

36.3.2 Action on Blocks

The function **BlockGroup**(I), **EdgeGroup**(I), or **LineGroup**(I) (depending on the nomenclature in the category of I) returns the automorphism group of I in its action on the blocks of I, as a permutation group acting on a set whose cardinality is the number of blocks of I. The function also returns the G-set of the blocks. It may be more convenient to use this function rather than **AutomorphismGroup** when the emphasis is on the blocks rather than the points. For example, the line group of the plane pl is the automorphism group of pl in its action on the 6 lines of pl:

```
> LG := LineGroup(pl);
> LG;
Permutation group LG acting on a set of cardinality 6
Order = 24 = 2^3 * 3
    (2, 5)(4, 6)
    (1, 5)(3, 6)
    (1, 3)(2, 4)
```

36.3.3 Subgroups of the Automorphism Group

Let I be any incidence structure except for a graph or digraph. If I is very large, then the computation of the automorphism group G of I may be too difficult to accomplish in a reasonable time. For such situations, MAGMA

offers functions that compute certain subgroups of G. These functions are particularly helpful when the user wishes to know whether the whole group G is non-trivial.

The function **AutomorphismGroupStabilizer**(I, k) calculates the subgroup of the automorphism group G that fixes the first k base points, and the function **AutomorphismSubgroup**(I) calculates a subgroup of G generated by one element. Each of them returns five values: the subgroup itself; and the four auxiliary values as returned by **AutomorphismGroup**(I).

37. Graphs

A graph consists of vertices joined by edges. *Simple* graphs are those in which no pair of vertices is joined by an edge more than once and no edge is a loop back to the same vertex. In *directed* graphs (*digraphs*), each edge has a direction from one vertex to the other; in *undirected* graphs, the edge has no direction.

Any simple graph, whether directed or undirected, may be constructed in MAGMA, but digraphs will not be discussed here. In this chapter, therefore, the term *graph* will denote an undirected simple graph.

37.1 Constructing Graphs

MAGMA represents an [undirected simple] graph by stating, for each vertex, the vertices to which it is joined by an edge. For example:

```
> print PolygonGraph(6);

Graph
Vertex  Neighbours

1       2 6 ;
2       1 3 ;
3       2 4 ;
4       3 5 ;
5       4 6 ;
6       1 5 ;
```

The output shows that each vertex in this graph has two neighbours. For instance, vertex 3 has neighbours 2 and 4, meaning that there are edges between 3 and 2 and between 3 and 4. Pairs of vertices joined by an edge are said to be *adjacent*.

Table 37.1. Standard graphs

MAGMA	Meaning
CompleteGraph(p)	Complete graph K_p on p vertices
BipartiteGraph(m, n)	Complete bipartite graph $K_{m,n}$, i.e., the graph of $m+n$ vertices such that m of the vertices are joined to all of the other n vertices, and vice versa
MultipartiteGraph(Q)	Complete multipartite graph $K_{m_1,\ldots,m_r}$, where $Q = [m_1, \ldots, m_r]$, i.e., the graph of $m_1 + \cdots + m_r$ vertices such that the vertices in each partite set of m_i vertices are all joined to all the other vertices
EmptyGraph(p)	Graph on p vertices and with no edges
KCubeGraph(k)	Graph of the k-dimensional cube, Q_k
PathGraph(p)	Path graph on p vertices, i.e., the graph whose only edges are $\{v_i, v_{i+1}\}$ where $1 \leq i < p$
PolygonGraph(p)	Polygon graph on p vertices, i.e., path graph with extra edge $\{v_1, v_p\}$
RandomGraph(p, r)	A random graph on p vertices, with probability r that any given pair of vertices is adjacent
RandomTree(p)	A random tree (connected acyclic graph) on p vertices

The example above is of the graph corresponding to a hexagon, with its six vertices and six edges. It was constructed with one of MAGMA's functions for standard graphs, **PolygonGraph**. These functions are listed in Table 37.1.

There are several ways of constructing graphs in MAGMA without using the standard functions. The constructor for building a general graph with p vertices is

Graph< p | specification of edges **>**

There are several ways of giving the edge specification on the right side of this constructor. The most important methods are to give a list of sets of pairs of integers, indicating pairs of adjacent vertices, or to give a sequence of p sets of integers, indicating the neighbours of each vertex in order. For example, the graph on 5 vertices with edges between 1 and 2, 1 and 5, 2 and 3, and 2 and 5, can be constructed by stating the edges:

```
> G := Graph< 5 | {1, 2}, {1, 5}, {2, 3}, {2, 5} >;
```

or by stating the neighbours of the vertices:

```
> G := Graph< 5 | [ {2, 5}, {1, 3, 5}, {2}, {}, {1, 2} ] >;
```

In either case, the following graph results:

```
> print G;

Graph
Vertex  Neighbours

1       2 5 ;
2       1 3 5 ;
3       2 ;
4       ;
5       1 2 ;
```

In the neighbour-sequence method, it is permissible to specify an edge between v_i and v_j by saying that v_i is a neighbour of v_j without explicitly saying also that v_j is a neighbour of v_i:

```
> print G eq Graph< 5 | [ {2, 5}, {3, 5}, {}, {}, {} ] >;
true
```

The edge-list and neighbour-sequence methods may be used together in an edge specification, and graphs of the same order p may also be included. MAGMA will combine all the edge information to find the edge specification for the graph being constructed.

37.2 Sets of Vertices and Edges

Each graph G has a corresponding set of vertices and set of edges. In MAGMA these sets can be thought of in two senses. The functions **VertexSet**(G) and **EdgeSet**(G) return sets V and E in the special categories **VertSet** and **EdgeSet**. These special kinds of sets are also the second and third return values of the **Graph** constructor:

```
> G, V, E := Graph< 5 | {1, 2}, {1, 5}, {2, 3}, {2, 5} >;
> print V, E;
The vertex-set of graph G
The edge-set of graph G
> print Category(V), Category(E);
VertSet EdgeSet
```

Secondly, the functions **Vertices**(G) and **Edges**(G) return enumerated sets whose universes are V and E respectively:

```
> Vs := Vertices(G); print Vs;
```

Table 37.2. Vertex and edge functions for graphs

MAGMA	Meaning
u **adj** v	**true** if vertices u and v are adjacent (joined by an edge)
u **notadj** v	**true** if vertices u and v are not adjacent
e **adj** f	**true** if edges e and f share a common vertex
e **notadj** f	**true** if edges e and f do not share a common vertex
u **in** e	**true** if vertex u is an endpoint of edge e
u **notin** e	**true** if vertex u is not an endpoint of edge e
EndVertices(e)	Set of the two end-vertices of edge e
IncidentEdges(u)	Set of edges incident with vertex u
Neighbours(u)	Set of vertices adjacent to vertex u
Degree(u)	Number of edges incident to vertex u
DegreeSequence(G)	Sequence D of sets of vertices of G such that the degree of each vertex in the set $D[i]$ is i
Maxdeg(G)	Returns (i) maximum degree of vertices of G (ii) a vertex with that degree
Mindeg(G)	Returns (i) minimum degree of vertices of G (ii) a vertex with that degree
Alldeg(G, n)	Set of vertices of G with degree equal to n
IsRegular(G)	**true** if all vertices of G have the same degree

```
{ 1, 2, 3, 4, 5 }
> Es := Edges(G); print Es;
{ {2, 3}, {1, 2}, {1, 5}, {2, 5} }
> print Universe(Vs), Universe(Es);
The vertex-set of graph G
The edge-set of graph G
```

There are several ways to refer to an individual vertex or edge. The i^{th} vertex is $G.i$ or $V!i$. The edge between vertices u and v is $E!\{u, v\}$, or $E!\{i, j\}$ where u is the i^{th} vertex and v is the j^{th} vertex. The number of vertices of a graph is **Order**(G) or $\#V$, and the number of edges is **Size**(G) or $\#E$.

To revert from a vertex-set, edge-set, vertex or edge to the parent, the function **ParentGraph** is used.

Some other vertex and edge functions for graphs are shown in Table 37.2 (p. 738).

37.3 Creating New Graphs

Once a graph G has been constructed, other graphs may be built from it. All of the construction methods described below return three values: the new graph, its vertex-set and its edge-set.

37.3.1 Addition and Subtraction

One method of creating a new graph from an existing graph G is to add or subtract edges, or to subtract vertices. To add an edge means to make a pair of vertices adjacent, and to subtract an edge means to remove the edge between a pair of vertices. The syntax for these operations is $G + \{u, v\}$ and $G - e$. To subtract a vertex means to remove a vertex, thus causing a renumbering of the vertices unless the vertex being removed is the highest-numbered one, and of course to remove the edges involving the vertex. The syntax for this is $G - v$. For example:

```
> print G + {G.3, G.4}; // add an edge

Graph
Vertex  Neighbours

1       2 5 ;
2       1 3 5 ;
3       2 4 ;
4       3 ;
5       1 2 ;

> print G - G.2; // remove a vertex

Graph
Vertex  Neighbours

1       4 ;
2       ;
3       ;
4       1 ;
```

A set of edges may be added or subtracted in the same way as a single edge, and a set of vertices may be subtracted in the same way as a single vertex. Therefore care must be taken not to confuse a set of two vertices with an edge. In particular, $G - \{u, v\}$ is G with vertices u and v removed, whereas $G - E!\{u, v\}$ is G with the edge $\{u, v\}$ removed:

```
> print G - {G.1, G.5}; // remove two vertices

Graph
Vertex  Neighbours

1       2 ;
2       1 ;
3       ;

> print G - E!{G.1, G.5}; // remove an edge

Graph
Vertex  Neighbours

1       2 ;
2       1 3 5 ;
3       2 ;
4       ;
5       2 ;
```

37.3.2 Subgraph

The **sub** constructor for graphs has a vertex version and an edge version:

sub< G | list of vertices **>**
sub< G | list of edges **>**

The vertex version creates the subgraph of graph G containing the listed vertices (renumbered as necessary), preserving whatever edges are between vertices in the list. The edge version creates the subgraph containing all the vertices of G but only those edges that are listed. The lists can contain sets of vertices/edges as well as individual vertices/edges.

For instance, the following statements create and print the subgraph of the 3-dimensional cube graph generated by vertices 3, 4, 7 and 8:

```
> K := KCubeGraph(3);
> Ksub := sub< K | K.3, K.4, K.7, K.8 >;
> print Ksub;

Graph
Vertex  Neighbours

1       2 3 ;
```

```
2       1 4 ;
3       1 4 ;
4       2 3 ;
```

This graph is the same as the graph of the 2-dimensional cube (i.e., the square):

```
> print Ksub eq KCubeGraph(2);
true
```

Therefore in the original 3-dimensional cube graph, the four vertices must be the corners of one of the cube's faces.

An example of **sub** in the edge version is the creation of the subgraph of G (defined above) with edges involving vertex 2. Here v **in** e tests whether a vertex v is one of the two endpoints of an edge e:

```
> print sub< G | {edge: edge in E | G.2 in edge} >;

Graph
Vertex  Neighbours

1       2 ;
2       1 3 5 ;
3       2 ;
4       ;
5       2 ;
```

The final example for **sub** illustrates the constructor using a larger graph. Operations on integer sets are used so as to supply the vertices easily:

```
> M := MultipartiteGraph([10, 5, 20]);
> Msub := sub< M |
>   M.1, ChangeUniverse({13..30}, VertexSet(M)) >;
> print Msub;

Graph
Vertex  Neighbours

1       2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ;
2       1 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ;
3       1 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ;
4       1 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ;
5       1 2 3 4 ;
6       1 2 3 4 ;
```

```
  ...
19      1 2 3 4 ;
```

Every subgraph is returned as a graph in its own right, but coercion may be used to relate the vertices and edges of the subgraph to those of the main graph. For instance, in *Msub* the vertices numbered 2 and 5 correspond to those called 13 and 16 in M:

```
> print VertexSet(M) ! Msub.2;
13
> print EdgeSet(M) ! {Msub.2, Msub.5};
{13, 16}
```

37.3.3 Quotient Graph

A quotient graph Q is constructed by identifying certain vertices of a graph G. The vertices of G are partitioned into a set of disjoint subsets $P_1, \ldots, P_r$ of the vertices whose union is the whole vertex-set of G. The vertices of Q correspond to the subsets P_i, and two vertices of Q are adjacent if some vertex in the subset corresponding to one is adjacent to some vertex in the other subset. The MAGMA constructor for the quotient graph is

quo< $G \mid \{P_1, \ldots, P_r\}$ **>**

For example, in the cube graph K constructed above, the vertices 3, 4, 7 and 8, which are the corners of a face, may be identified:

```
> Q := quo< K | {{3, 4, 7, 8}, {1}, {2}, {5}, {6}} >;
> print Q;

Graph
Vertex  Neighbours

1       2 3 4 ;
2       1 3 5 ;
3       1 2 4 5 ;
4       1 3 5 ;
5       2 3 4 ;
```

From the output, vertex 3 of Q corresponds to the subset of cardinality 4 given in the partition, vertices 4 and 5 correspond to vertices 5 and 6 in K, and vertices 1 and 2 have the same numbers as their counterparts. The function **Contract**(S), in Table 37.3 (p. 743), would also construct this graph.

37.3.4 Some Standard Constructions

Table 37.3. Standard related graphs

MAGMA	Meaning
Complement(G)	Graph whose edges are exactly those between the non-adjacent vertices of G
Contract(u, v)	Given vertices u and v of G, returns the graph which is G with u and v identified
Contract(e)	Given an edge $e = \{u, v\}$ of G, returns **Contract**(u, v)
Contract(S)	Given a set S of vertices of G, returns the graph which is G with all these vertices identified
InsertVertex(e)	Given an edge e of G, returns the graph which is G with a new degree-2 vertex inserted in e
InsertVertex(T)	Given an set T of edges of G, returns the graph which is G with a new degree-2 vertex inserted in each edge in T
LineGraph(G)	Line graph of the non-empty graph G
Switch(u)	Given a vertex u of G, returns the graph which is G but with the neighbours of u becoming its non-neighbours
Switch(S)	Given a set S of vertices of G, returns the graph which is G with **Switch**(u) applied to it for each vertex u in S

Table 37.3 (p. 743) and Table 37.4 (p. 744) list some other ways of constructing graphs from existing graphs G and H in MAGMA.

In the union and product functions given in Table 37.4 (p. 744), G and H denote graphs with vertex-sets V_G and V_H and edge-sets E_G and E_H. These functions manipulate the graphs as if their vertices are *disjoint*, so that, for example, the **Union** function when applied to two copies of the same graph yields a graph with twice as many vertices:

```
> G := Graph< 5 | {1, 2}, {1, 5}, {2, 3}, {2, 5} >;
> print Union(G, G);

Graph
Vertex  Neighbours

1       2 5 ;
2       1 3 5 ;
3       2 ;
4       ;
5       1 2 ;
6       7 10 ;
7       6 8 10 ;
```

Table 37.4. Graph unions and products

MAGMA	Meaning
Union(G, H), G **join** H	Graph with vertex-set $V_G \cup V_H$ and edge-set $E_G \cup E_H$
EdgeUnion(G, H)	Given G and H with the same order n, return the graph with order n whose edge-set is $E_G \cup E_H$, by identifying the vertices of G and H having corresponding vertex number
CompleteUnion(G, H)	Graph that is **Union**(G, H) together with edges $\{u, v\}$, for all $u \in V_G$ and $v \in V_H$
CartesianProduct(G, H)	Graph with vertex-set $V_G \times V_H$ and such that 2 vertices are adjacent when either the first components are equal and the second are adjacent, or vice versa
LexProduct(G, H)	Graph with vertex-set $V_G \times V_H$ and such that vertices $\{u_1, u_2\}$ and $\{v_1, v_2\}$ are adjacent when either u_1 and v_1 are adjacent, or $u_1 = v_1$ and u_2 and v_2 are adjacent
TensorProduct(G, H)	Graph with vertex-set $V_G \times V_H$ and such that 2 vertices are adjacent when the first components are adjacent and the second components are adjacent
$G\hat{}n$	Graph with vertex-set V_G, and such that u and v are adjacent when **Distance**$(u, v) \leq n$

```
8       7 ;
9       ;
10      6 7 ;
```

37.4 Testing Properties of a Graph

Table 37.5 (p. 745) lists several of the functions that test properties of graphs.

37.5 Incidence, Adjacency and Distance Matrices

Several matrices are closely associated with each graph G, including those returned by the functions **IncidenceMatrix**(G), **AdjacencyMatrix**(G) and **DistanceMatrix**(G).

If G has p vertices and q edges, then the incidence matrix is a $p \times q$ matrix M of 1s and 0s. Each column corresponds to an edge, and contains exactly

Table 37.5. Some tests on graphs

MAGMA	Meaning
IsBipartite(G)	**true** if G is bipartite
IsComplete(G)	**true** if G is complete
IsPath(G)	**true** if G is a path graph
IsPolygon(G)	**true** if G is a polygon graph
IsTree(G)	**true** if G is connected and has no cycles
IsForest(G)	**true** if G has no cycles (i.e., is the disjoint union of trees)
IsEmpty(G)	**true** if G has no edges

two 1s, in the rows whose numbers are the index numbers of the vertices lying on that edge. Returning to the earlier example:

```
> G := Graph< 5 | {1, 2}, {1, 5}, {2, 3}, {2, 5} >; print G;

Graph
Vertex  Neighbours

1       2 5 ;
2       1 3 5 ;
3       2 ;
4       ;
5       1 2 ;

> print IncidenceMatrix(G);
[1 1 0 0]
[1 0 1 1]
[0 0 1 0]
[0 0 0 0]
[0 1 0 1]
```

The columns of the incidence matrix show that G has four edges, joining vertices 1 and 2, 1 and 5, 2 and 3, and 2 and 5.

It is also possible to go in the opposite direction, constructing a graph from an incidence matrix M, with the function **IncidenceGraph**(M).

The adjacency matrix contains 1s and 0s only, like the incidence matrix, but its dimension is $p \times p$. It lists neighbours of vertices, like the printed form of a graph. The 1s in row i give the vertices that are neighbours of vertex i. For example:

```
> print AdjacencyMatrix(G);
```

```
[0 1 0 0 1]
[1 0 1 0 1]
[0 1 0 0 0]
[0 0 0 0 0]
[1 1 0 0 0]
```

The distance matrix is also a $p \times p$ symmetric matrix of integers. The entry at row i, column j gives the distance between vertices i and j of G. For instance:

```
> print DistanceMatrix(G);
[ 0  1  2 -1  1]
[ 1  0  1 -1  1]
[ 2  1  0 -1  2]
[-1 -1 -1  0 -1]
[ 1  1  2 -1  0]
```

37.6 Distance and Connectedness

Table 37.6. Distance, paths, connectedness

Magma	Meaning
Distance(u, v)	Length of a shortest path between u and v, or -1 if they are not connected
Geodesic(u, v)	Sequence of vertices defining a shortest path between u and v, or the empty sequence if they are not connected
Ball(u, n)	Set of vertices at distance $\leq n$ from u.
Sphere(u, n)	Set of vertices at distance n from u.
Diameter(G)	Length of a longest path in G, or -1 if G is not connected
DiameterPath(G)	Sequence of vertices defining a longest path in G, or the empty sequence if G is not connected
IsConnected(G)	**true** if between any pair of vertices of G there is a path
Component(u)	Subgraph corresponding to the connected component of the graph containing vertex u
Components(G)	Connected components of G as a sequence of subsets of vertices
Girth(G)	Length of a shortest cycle of G
GirthCycle(G)	Sequence of vertices defining a shortest cycle of G
IsEulerian(G)	**true** if there is a cycle visiting each vertex of G exactly once (i.e., if all vertices have even degree)

Table 37.6 (p. 746) explains functions related to the ideas of distance, paths, cycles and connectedness.

37.7 Vertex Subsets and Colourings

Two types of subsets of vertices are particularly important in graph theory. An *independent set* of a graph is a subset of vertices such that no two of the vertices are adjacent. The subgraph generated by an independent set is empty, that is, it has no edges. By contrast, a *clique* of a graph is a subset of vertices such that all the vertices are adjacent to one another. Thus the subgraph generated by a clique is complete.

There are three functions associated with each of these types of sets of vertices. **CliqueNumber**(G) and **IndependenceNumber**(G) return the maximum number of elements that a clique or independent set of graph G can have. **LargestClique**(G) and **LargestIndependent**(G) return an example. For instance, returning to the large tripartite set M defined above, we have:

```
> M := MultipartiteGraph([10, 5, 20]);
> print CliqueNumber(M);
3
> print LargestClique(M);
{ 10, 15, 35 }
> print IndependenceNumber(M);
20
> print LargestIndependent(M);
{ 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35}
```

The function **Clique**(G, n) returns a clique of cardinality at least n. If n is larger than the clique number, an empty set is returned. **Independent**(G, n) works similarly.

Several problems in graph theory can be expressed in terms of colouring the vertices or edges of a graph, according to some rule. A frequent task is to determine the minimum number of different colours required such that no two adjacent vertices have the same colour. **ChromaticNumber**(G) returns this minimum number. If instead the task is to colour the edges such that no two adjacent edges have the same colour, then **ChromaticIndex**(G) can be used to find the minimum number of colours required. For example, consider the graph of the cube:

```
> K := KCubeGraph(3);
```

```
> print ChromaticNumber(K), ChromaticIndex(K);
2 3
```

The vertices of the cube can be coloured with as few as 2 colours, by giving one colour to the vertices on a diagonal of the upper face and on the other diagonal of the lower face, and the other colour to the remaining vertices. As for the edges of the cube, they cannot be coloured with 2 colours since the maximum degree is greater than 2, but can be coloured with 3 colours, one for each of the 3 classes of parallel sides.

37.8 Automorphism Group of a Graph

The automorphism group of a graph G is the group of permutations on the vertices of G resulting in an identical graph. (Equality of graphs in this sense is a stronger property than isomorphism.) The function for constructing the automorphism group of G is **AutomorphismGroup**(G). (It has several parameters, which will not be explained here.)

```
> M := MultipartiteGraph([10, 5, 20]);
> A := AutomorphismGroup(M);
> print A;
Permutation group A acting on a set of cardinality 35
Order = 2^29 * 3^13 * 5^7 * 7^3 * 11 * 13 * 17 * 19
    (14, 15)
    (13, 14)
    (12, 13)
    (11, 12)
    (9, 10)
    (8, 9)
    (7, 8)
    (6, 7)
    (5, 6)
    (4, 5)
    (3, 4)
    (2, 3)
    (1, 2)
    (34, 35)
    (33, 34)
  ...
    (16, 17)
> print CompositionFactors(A);
    G
    |  Cyclic(2)
```

```
    *
    |  Alternating(20)
    *
    |  Cyclic(2)
    *
    |  Alternating(10)
    *
    |  Cyclic(2)
    *
    |  Alternating(5)
    1
```

The ^ operator is used for finding the action of an automorphism on a vertex:

```
> print M.9, A.5;
9 (9, 10)
> print M.9 ^ A.5;
10
```

The action of an automorphism on a set or sequence of vertices or edges can also be calculated with the ^ operator.

Another automorphism action which may be calculated is the action on the graph or a subgraph of it. (When an automorphism acts on a subgraph, the computation is done with respect to the main graph.) However, since every element of the automorphism group acts on the graph or subgraph to return an isomorphic graph, the results are not of great interest; each of these isomorphic graphs, converted to a graph in its own right, is the same as the others:

```
> Msub := sub< M |
>   M.1, ChangeUniverse({13..30}, VertexSet(M)) >;
> print Msub eq Msub ^ A.Random(1, Ngens(A));
true
> print Msub ^ A;
GSet{
Graph
Vertex  Neighbours

1       2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ;
2       1 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 ;
  ...
19      1 2 3 4 ;
 }
```

```
> print #(M ^ A), M in (M ^ A);
1 true
```

Once the automorphism group A has been constructed, it may be used in any permutation group operations as desired. Additionally, there are several graph functions, such as **IsPrimitive**(G), that apply to properties of the automorphism group A of graph G but do not require A to be constructed explicitly. Table 37.7 (p. 750) lists some of them.

Table 37.7. Automorphism group

MAGMA	Meaning
AutomorphismGroup(G)	Returns (i) the automorphism group A of G (ii) the canonical graph of G, if parameter **Canonical** is **true** or $\#G < 500$
EdgeGroup(G)	Automorphism group of G in its action on the edges
$x\hat{}a$, $x\hat{}S$	Image of x under action of automorphism a or subgroup S of A, where x is a vertex, edge, or set or sequence of vertices or edges of G
$H\hat{}x$	Image of graph H under action of x, where x is an element or subgroup of A (including A itself)
IsVertexTransitive(G), **IsTransitive**(G)	**true** if automorphism group of G is transitive
IsEdgeTransitive(G)	**true** if edge group of G is transitive
IsPrimitive(G)	**true** if automorphism group of G is primitive
IsSymmetric(G)	**true** if automorphism group of G is symmetric
OrbitsPartition(G)	Partition of the vertex-set of G corresponding to orbits of automorphism group A of G
Stabilizer(S, x)	Stabilizer of x in the subgroup S of A, where x is a vertex, edge, or set or sequence of vertices or edges
IsDistanceTransitive(G)	**true** if for all vertices u, v, w, t in connected graph G such that $d(u, v) = d(w, t)$, there exists $a \in A$ such that $u^a = w$ and $v^a = t$
IsDistanceRegular(G)	**true** if G is distance regular
IntersectionArray(G)	Intersection array of distance regular graph G

As an example, consider the 4-dimensional cube graph, which contains eight 3-dimensional cube graphs. MAGMA can be used to find their vertices easily. The method is to find the orbit of a set of vertices which are known to generate a 3-dimensional cube graph. The orbit will be all those sets of vertices generating graphs isomorphic to this graph:

```
> q4 := KCubeGraph(4);
```

```
> s := { VertexSet(q4) | 1, 2, 3, 4, 5, 6, 7, 8 };
> print IsIsomorphic(sub< q4 | s >, KCubeGraph(3));
true
> A4 := AutomorphismGroup(q4);
> print s ^ A4;
GSet{
    { 1, 2, 3, 4, 9, 10, 11, 12 },
    { 1, 2, 3, 4, 5, 6, 7, 8 },
    { 2, 4, 6, 8, 10, 12, 14, 16 },
    { 1, 2, 5, 6, 9, 10, 13, 14 },
    { 1, 3, 5, 7, 9, 11, 13, 15 },
    { 3, 4, 7, 8, 11, 12, 15, 16 },
    { 9, 10, 11, 12, 13, 14, 15, 16 },
    { 5, 6, 7, 8, 13, 14, 15, 16 }
}
```

The output lists the eight sets of vertices for 3-dimensional cube graphs.

37.8.1 Edge Group

The edge group is a variant of the automorphism group that considers the action on the edges instead of on the vertices. It is returned by the function **EdgeGroup**(G). Just as the degree of the automorphism group is the number of vertices, the degree of the edge group is the number of edges. For example:

```
> EG := EdgeGroup(q3); print EG;
Permutation group EG acting on a set of cardinality 12
    ({1, 3}, {1, 5})({5, 7}, {3, 7})({5, 6}, {3, 4})({2, 6},
        {2, 4})({4, 8}, {6, 8})
    ({1, 3}, {1, 2})({5, 7}, {5, 6})({2, 6}, {3, 7})({2, 4},
        {3, 4})({6, 8}, {7, 8})
    ({1, 3}, {2, 4})({5, 7}, {6, 8})({1, 5}, {2, 6})({3, 7},
        {4, 8})
```

The edge group may be considered explicitly as the action of the automorphism group on the G-set of the edges:

```
> A3 := AutomorphismGroup(q3);
> print ActionImage(A3, GSet(A3, Edges(q3))) eq EG;
true
```

The function **IsEdgeTransitive**(G) tests whether the edge group of G is transitive.

37.8.2 Canonical Graph and Isomorphic Graphs

For every graph G, there is a well-defined labelling of its vertices that is called the canonical labelling. The function **CanonicalGraph**(G) returns the graph labelled in this way. The canonical graph is isomorphic to G. For example:

```
> C := CanonicalGraph(M); print C;

Graph
Vertex  Neighbours

1       21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 ;
  ...
20      21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 ;
21      1 2 3 4 5 6 7 8 9 10 ... 17 18 19 20 31 32 33 34 35 ;
  ...
30      1 2 3 4 5 6 7 8 9 10 ... 17 18 19 20 31 32 33 34 35 ;
31      1 2 3 4 5 6 7 8 9 10 ... 17 18 19 20 21 22 23 24 ... 30 ;
  ...
35      1 2 3 4 5 6 7 8 9 10 ... 17 18 19 20 21 22 23 24 ... 30 ;
```

The canonical graph may also be obtained as the second return value of **AutomorphismGroup**(G), if G has less than 500 vertices or the parameter **Canonical** is set to **true**.

MAGMA can test whether two graphs G and H are isomorphic by testing their canonical graphs for equality. The function **IsIsomorphic**(G, H) fulfils this task in one step. It returns **true** or **false**, according to whether G and H are isomorphic, and if they are isomorphic then it also returns the isomorphism, as a mapping from the vertex-set of G to the vertex-set of H. For example:

```
> PG4 := PolygonGraph(4);
> q2 := KCubeGraph(2);
> print PG4 eq q2;
false
> ii, m := IsIsomorphic(PG4, q2);
> print ii;
true
> print m;
Mapping from: The vertex-set of graph PG4
to The vertex-set of graph q2
```

38. Designs

MAGMA supports both general incidence structures (category **Inc**) and three families of incidence structures satisfying stronger conditions, namely near-linear spaces (**IncNsp**), linear spaces (**IncLsp**), and t-designs (**Dsgn**). This chapter concentrates on the facilities for designs, but explains how to construct structures in the other categories as well.

For an introduction to the theory of designs, the reader should consult [AsK92].

38.1 The Categories of Incidence Structures

An *incidence structure* is a triple (P, B, I) consisting of a set P of *points*, a set B of *blocks*, and an incidence relation I which is a subset of $P \times B$. The MAGMA system adopts the usual approach to incidence structures, in which the blocks are subsets of P, and I describes which points are in which block. The only difficulty with this approach is the problem of representing repeated blocks, i.e., two or more blocks consisting of the same set of points; in such cases MAGMA considers such blocks to be distinct, although their points are the same. An incidence structure with no repeated blocks is called *simple*. An incidence structure is said to be *uniform* with *blocksize* k if it has at least one block and all blocks contain exactly k points. If $t \geq 0$, then an incidence structure is said to be *t-balanced* if there exists $\lambda \geq 1$ such that each t-subset of the point set is contained in exactly λ blocks of the structure.

It is usual, when discussing near-linear spaces, to use the term *line* in place of the term *block*. A *near-linear space* is an incidence structure in which every line contains at least two points and any two points lie in at most one block. A *linear space* is a near-linear space in which any two points lie in exactly one line.

A *t-(v, k, λ) design* (or *t-design*) D is a simple incidence structure having v points, that is uniform with blocksize k and t-balanced for some λ. Note that a design must contain at least one block. A *Steiner design* is a t-design

with $\lambda = 1$. A *symmetric design* is a t-design where $t \geq 2$ and with the same number of blocks as points.

38.2 Creating Incidence Structures

Constructors are provided in MAGMA for the definition of magmas in each of the four categories of incidence structures. The category is given by the word on the left of the constructor: **IncidenceStructure**, **NearLinearSpace**, **LinearSpace**, or **Design**, designating the categories **Inc**, **IncNsp**, **IncLsp** and **Dsgn** respectively. Within the angle bracket delimiters, the set of points is defined on the left side of the | symbol, while the set of blocks is specified on the right side of the | symbol:

constructor-name< point-set description | block-set description >

For designs only, the value of t must also be given on the left side, before the points are specified:

Design< t, point-set specification | block-set specification >

In all the categories except the general category **Inc**, the set of blocks is subject to certain restrictions, as explained below. The constructor returns three values: the incidence structure (in whichever category was requested); the point-set; and the block-set (also known as the line-set for the categories **IncNsp** and **IncLsp**).

The ways in which the points and blocks may be described in the constructor are as follows. The v points may be described in two ways:

- As an indexed set P of cardinality v, giving the set of points directly;
- As the integer v, in which case the point-set is taken to be $\{1, \ldots, v\}$.

The b blocks of the structure may be given in three ways:

- As a $v \times b$ matrix of zeros and ones (as an element of a matrix space over any coefficient ring), which is interpreted as an incidence matrix for the structure;
- As a set of codewords of a linear code with length v, in which case the block set is the set of supports of the codewords;
- As a comma-separated list (in the order of the blocks) in which each item may be a subset of P, a block of an existing incidence structure, or a sequence or indexed set of either of these.

For example, the following statement assigns to S the incidence structure with points $2, 5, 6$ and blocks $\{5, 6\}$, $\{\}$, and $\{5\}$. The magma S is then printed, using the **Maximal** printing option so that the points and blocks are displayed:

```
> S := IncidenceStructure< {@ 2,5,6 @} | {5,6}, {}, {5} >;
> S : Maximal;
Incidence Structure on 3 points with 3 blocks
Points: {@ 2, 5, 6 @}
Blocks:
    {5, 6},
    {},
    {5}
```

Suppose now that the incidence structure $S2$ has the same points as S and its blocks are the blocks of S together with $\{2\}$ and $\{5, 6\}$ (again). The new structure may be constructed with the aid of the functions **Points**(S) and **Blocks**(S), which return the points and blocks of S as indexed sets:

```
> S2 := IncidenceStructure< Points(S) |
>   Blocks(S), {2}, {5,6} >;
> S2 : Maximal;
Incidence Structure on 3 points with 5 blocks
Points: {@ 2, 5, 6 @}
Blocks:
    {5, 6},
    {},
    {5},
    {2},
    {5, 6}
> IsSimple(S2);
false
```

The output in the final line verifies that $S2$ has a repeated block.

As a final example of a general incidence structure, the following statement constructs the incidence structure $S3$ with points $\{@1, \ldots, 6@\}$ and blocks $\{1, \ldots, 6\}$, $\{3\}$, $\{6\}$, and also assigns the point-set and block-set to P and B. Since the point-set is of the form $\{@1, \ldots, v@\}$, it is sufficient to give the size of the point-set, rather than the set itself, on the left side of the constructor:

```
> S3, P, B := IncidenceStructure< 6 | {1..6}, {3}, {6} >;
> S3 : Maximal;
Incidence Structure on 6 points with 3 blocks
```

```
Points: {@ 1, 2, 3, 4, 5, 6 @}
Blocks:
    {1, 2, 3, 4, 5, 6},
    {3},
    {6}
> P, B;
Point-set of Incidence Structure on 6 points with 3 blocks
Block-set of Incidence Structure on 6 points with 3 blocks
```

For incidence structures in the categories **IncNsp** and **IncLsp**, each line (block) must contain at least two points. In a near-linear space, any two points must lie in at most one line; in a linear space, any two points must lie in exactly one line. MAGMA will test these conditions when the space is constructed, and will give an error message if they do not hold. For example:

```
> LS := LinearSpace< 4 | {1,3,4}, {1,2}, {2,3} >;

>> LS := LinearSpace< 4 | {1,3,4}, {1,2}, {2,3} >;
                      ^
Runtime error in LinearSpace< ... >:
Resulting structure is not a linear space
> NLS := NearLinearSpace< 4 | {1,3,4}, {1,2}, {2,3} >;
> NLS : Maximal;
Near-Linear Space on 4 points with 3 lines
Points: {@ 1, 2, 3, 4 @}
Lines:
    {1, 3, 4},
    {1, 2},
    {2, 3}
> LS := LinearSpace< 4 | {1,3,4}, {1,2}, {2,3}, {2,4} >;
> LS : Maximal;
Linear Space on 4 points with 4 lines
Points: {@ 1, 2, 3, 4 @}
Lines:
    {1, 3, 4},
    {1, 2},
    {2, 3},
    {2, 4}
```

The next example demonstrates the construction of a near-linear space in terms of an incidence matrix; the same principle applies for other incidence structures. The columns of the matrix define the lines or blocks of the structure, as sets of points:

```
> M := RMatrixSpace(IntegerRing(), 5, 6);
```

```
> m := M ! [ 0,1,0,0,1,0, 1,1,1,0,0,0, 0,0,1,1,0,1,
>   1,0,0,1,1,0, 0,1,0,0,0,1 ];
> m;
[0 1 0 0 1 0]
[1 1 1 0 0 0]
[0 0 1 1 0 1]
[1 0 0 1 1 0]
[0 1 0 0 0 1]
> Nm := NearLinearSpace< 5 | m >;
> Nm : Maximal;
Near-Linear Space on 5 points with 6 lines
Points: {@ 1, 2, 3, 4, 5 @}
Lines:
    {2, 4},
    {1, 2, 5},
    {2, 3},
    {3, 4},
    {1, 4},
    {3, 5}
```

The function **IncidenceMatrix**(I) returns the incidence matrix of an incidence structure I:

```
> IncidenceMatrix(Nm) eq m;
true
```

For t-designs (incidence structures in the category **Dsgn**), three conditions must hold: no block may be repeated; each block must have the same blocksize k; and every set of points of cardinality t must lie in the same number λ of blocks. The following example illustrates a way to construct a design using the function **Subsets**(T, k), which returns the set of all k-subsets of a set S. In the example, the return value of this function has to be converted into a sequence, so that the blocks will be ordered:

```
> D := Design< 3, 5 | Setseq(Subsets({1..5}, 3)) >;
> D : Maximal;
3-(5, 3, 1) Design with 10 blocks
Points: {@ 1, 2, 3, 4, 5 @}
Blocks:
    {1, 3, 5},
    {2, 3, 4},
    {1, 4, 5},
    {2, 4, 5},
    {1, 2, 5},
    {1, 2, 3},
```

```
    {1, 2, 4},
    {3, 4, 5},
    {2, 3, 5},
    {1, 3, 4}
> IncidenceMatrix(D);
[1 0 1 0 1 1 1 0 0 1]
[0 1 0 1 1 1 1 0 1 0]
[1 1 0 0 0 1 0 1 1 1]
[0 1 1 1 0 0 1 1 0 1]
[1 0 1 1 1 0 0 1 1 0]
> IsComplete(D);
true
```

The design above is a 3-$(5, 3, 1)$ design, indicating that t is 3, the number of points v is 5, the blocksize k is 3, and λ is 1. In fact, it is the *complete* $(5, 3, 1)$ design.

As a second example, let H be the Hamming code over GF(3) with $r = 3$. The incidence structure formed from the supports of the words of H with weight 5 is a 2-design:

```
> H := HammingCode(GF(3), 3);
> H : Minimal;
[13, 10, 3] Hamming code (r = 3) over GF(3)
> W5 := Words(H, 5);
> DD := Design< 2, 13 | W5 >;
> DD;
2-(13, 5, 90) Design with 702 blocks
```

When an incidence structure in a category other than **Inc** is being built, it is often the case that the user knows from earlier work or from theory that the blocks satisfy the conditions for the category. MAGMA's default action is to check that the conditions are fulfilled, but this can take a great deal of time if the structure is large. For this reason, a parameter **Check** is provided for the constructors **NearLinearSpace**, **LinearSpace**, and **Design**, so that the user can turn off the checking when it would be superfluous. If **Check** is assigned **false** instead of the default value of **true**, then MAGMA does not check that the conditions required by the category are correct; it is then the user's responsibility to ensure that they are. For example, if *CG2x* is the extended binary Golay code, then the supports of the words of weight 12 in *CG2x* form a 5-design. The **time** statements below demonstrate that the construction of the design is much faster with checking turned off:

```
> CG2x := GolayCode(GF(2), true);
> wds := Words(CG2x, 12);
```

```
> #wds;
2576
> time DG := Design<5, Length(CG2x) | wds : Check := false>;
Time: 1.410
> time DG := Design<5, Length(CG2x) | wds>;
Time: 219.781
> DG;
5-(24, 12, 48) Design with 2576 blocks
```

38.3 Points and Blocks

Table 38.1. Point-set and block-set functions

MAGMA	Meaning
PointSet(I)	Point-set P of incidence structure I
BlockSet(I)	Block-set B of incidence structure I
Points(I)	Indexed set of points of I
Blocks(I)	Indexed set of blocks of I
NumberOfPoints(I), #P	Number of points of I
NumberOfBlocks(I), #B	Number of blocks of I
Support(I)	Indexed set of underlying points of I, in their original category
Rep(P)	Representative point of point-set P
Random(P)	Random point of point-set P
Rep(B)	Representative block of block-set B
Random(B)	Random block of block-set B
Point$(I, i), P.i$	i^{th} point of incidence structure I with point-set P
Block$(I, i), B.i$	i^{th} block of incidence structure I with block-set B
$P!x$	Point corresponding to element x of indexed set used to create I with point-set P
PointDegrees(I)	Sequence whose i^{th} term is the number of blocks containing point $P.i$
BlockDegrees(I), **BlockSizes**(I)	Sequence whose i^{th} term is the number of points in block $B.i$
Covalence(I, S)	Given subset S of point-set of I, return number of blocks of I containing S
IncidenceMatrix(I)	Incidence matrix of I

Table 38.2. Functions for individual blocks

MAGMA	Meaning
Set(b)	Set of points in block b
Support(b)	Set of underlying points in block b, in their original category
Rep(b)	Representative point in block b
Random(b)	Random point in block b
p **in** b, p **notin** b	**true** if point p is [not] in block b
S **subset** b, S **notsubset** b	**true** if set S of points is [not] in block b
PointDegree(I, p)	Number of blocks of I containing point p
BlockDegree(I, b), **BlockSize**(I, b), #b	Number of points contained in block b in the incidence structure I
ConnectionNumber(I, p, b)	Number of blocks joining p to b in I
IsBlock(I, S)	**true** if set or block S represents a block of I; if **true**, also returns one such block
Block(I, p, q), **Line**(I, p, q)	A block of I containing points p and q, if it exists (in a linear space, if $p \neq q$ then the line exists uniquely)
IsParallelClass(I, b, c)	**true** if there is a parallel class of I containing blocks b and c; if **true**, also returns one such class

Let P and B denote the point-set and block-set of an incidence structure I (in any category). As was mentioned above, P and B are given as the second and third return values of all constructors and functions that return incidence structures. They may also be accessed directly using the functions **PointSet**(I) and **BlockSet**(I). P and B belong to the special categories **IncPtSet** and **IncBlkSet**. They are not true MAGMA sets, but special structures which act as the parents for points and blocks respectively. For example, let DG be the design calculated above from the extended binary Golay code:

```
> P := PointSet(DG);
> B := BlockSet(DG);
> P, B;
Point-set of 5-(24, 12, 48) Design with 2576 blocks
Block-set of 5-(24, 12, 48) Design with 2576 blocks
> Category(P), Category(B);
IncPtSet IncBlkSet
```

There is another pair of functions, **Points**(I) and **Blocks**(I), returning the points and blocks of I explicitly as indexed sets. (The indexing reflects

the ordering of the points and the blocks.) The user should take care not to confuse these functions with **PointSet**(I) and **BlockSet**(I). The universes of these sets are P and B, and their elements are in the categories **IncPt** and **IncBlk**. For example:

```
> pts := Points(DG);
> pts;
{@ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24 @}
> blks := Blocks(DG);
> #blks;
2576
> Universe(pts) eq P and Universe(blks) eq B;
true
> Category(Random(pts)), Category(Random(blks));
IncPt IncBlk
```

One consequence of the fact that special categories are used for points and blocks is that repeated blocks can be handled correctly by the system. If the blocks of a non-simple incidence structure are printed, the result seems to be an indexed set containing more than one copy of the same element. However, the blocks are internally distinct; it is only their printed forms that are the same. For example:

```
> Irpt := IncidenceStructure< 4 | {1,3}, {3,4}, {3,4} >;
> BL := Blocks(Irpt); BL;
{@ {1, 3}, {3, 4}, {3, 4} @}
> BL[2] eq BL[3];
false
```

Although the category of each block is **IncBlk**, not **SetEnum**, blocks may be treated as normal sets for most purposes. For example, the operators **in**, **join**, **meet** and **subset** are available. However, in some circumstances a block has to be converted into a true enumerated set of points, using the function **Set**:

```
> BL2set := Set(BL[2]);
> Category(BL2set);
SetEnum
> BL2set;
{ 3, 4 }
> Universe(BL2set) eq PointSet(Irpt);
true
```

A third way of regarding points and blocks is in terms of the universe of the set of points used in the creation of I (often the integer ring). The function **Support**(I) may be applied to the whole structure I to return the indexed set of the points in the original universe. Similarly, **Support**(b), given an individual block b, returns the set of the points in the block, in the original universe. For example:

```
> supp := Support(DG);
> supp;
{@ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24 @}
> Universe(supp);
Integer Ring

> bl5 := B.5;
> bl5;
{5, 6, 7, 8, 11, 13, 16, 18, 19, 20, 22, 23}
> Support(bl5);
{ 5, 6, 7, 8, 11, 13, 16, 18, 19, 20, 22, 23 }
> Universe($1);
Integer Ring
```

It should be noted that if the set of points is other than $\{@1, \ldots, v@\}$ then there is a distinction between the i^{th} point and the point called i (if indeed such a point exists). For example, in the structure below, the point called 2 is the first point, and the second point is the point called 5:

```
> S, P, B := IncidenceStructure< {@ 2,5,6 @} |
>   {5,6}, {}, {5} >;
> pt := P!2; pt;
2
> Index(Points(S), pt);
1
> P.2;
5
```

Table 38.1 lists operators and functions connected with the points and blocks of an incidence structure I, and Table 38.2 lists operations which apply to an individual block b. They include functions for random and representative elements of the point-set, the block-set, and a block, and for the number of points in a block and the number of blocks containing a point. Most of these are straightforward to apply, once the user understands the distinction between the point-set and the indexed set of points (and similarly for blocks), and the way that MAGMA uses special categories for points and blocks.

38.4 Elementary Invariants of a Design

Table 38.3. Elementary invariants of a design

MAGMA	Meaning
Parameters(D)	Parameters of t-(v, b, r, k, λ) design D, as a record with fieldnames **t**, **v**, **b**, **r**, **k**, **lambda**
ReplicationNumber(D)	Replication number r $(t > 0)$
BlockDegree(D), **BlockSize**(D)	Number k of points in each block of D
Covalence(D, s)	Number λ_s of blocks containing an arbitrary s-subset of the points of D
IntersectionNumber(D, i, j)	Block intersection number λ_i^j $(i + j \leq t)$
Order(D)	Order λ_1^1 of D $(t \geq 2)$
PascalTriangle(D)	'Pascal triangle' of D

The conditions satisfied by the blocks of a design D imply the existence of certain invariants for D. The functions in Table 38.3 return these values.

The fundamental parameters for a design D are: t; v, the number of points; b, the number of blocks; r, the replication number, which is defined as the number of blocks containing any point of D, where $t > 0$; k, the blocksize; and λ. Some of these parameters may be obtained individually from particular functions, but they are also available collectively from **Parameters**(D). It returns a record whose fieldnames are the names of the parameters. For example:

```
> DD := Design< 2, 13 | Words(HammingCode(GF(3), 3), 5) >;
> DD;
2-(13, 5, 90) Design with 702 blocks
> pp := Parameters(DD);
> pp;
rec<recformat<t, v, b, r, k, lambda> | t := 2, v := 13,
b := 702, r := 270, k := 5, lambda := 90>
> pp`r;
270
> pp`lambda;
90
```

Some of the functions in the table involve the block intersection number λ_i^j. This quantity is defined for integers i and j such that $i + j \leq t$, and it equals the number of blocks of D that contain an i-set but are disjoint from a j-set. For example, **PascalTriangle**(D) returns a 'Pascal triangle' whose

rows are of the form $\lambda_0^{i-1}, \lambda_1^{i-2}, \ldots, \lambda_{i-1}^0$. The triangle has $k+1$ rows if D is a Steiner design, or $t+1$ rows otherwise. The return value of the function is a sequence whose terms are the rows, also stored as sequences. However, when this triangle is printed, it is displayed in a formatted style:

```
> CG2x := GolayCode(GF(2), true);
> DG := Design<5, Length(CG2x) | Words(CG2x, 12) :
>   Check := false >;
> pt := PascalTriangle(DG);
> pt[4];
[ 280, 336, 336, 280 ]
> pt;
                2576
             1288  1288
          616   672   616
       280   336   336   280
    120   160   176   160   120
 48    72    88    88    72    48
```

38.5 Incidence Structures with Particular Properties

Table 38.4 summarizes the functions that test whether a given incidence structure, of any category, has a certain property. The table also lists boolean functions that may be applied to designs or (near-)linear spaces only.

Two of the functions, **IsBalanced**(I, t) and **IsDesign**(I, t), have a parameter **Al** so that the user can specify which algorithm should be followed when testing if I is balanced with respect to t. The default value of **Al** is `"NoOrbits"`, meaning a 'brute force' test. The alternative algorithm is `"Orbits"`, which uses the orbits of t-sets under the automorphism group of I; this is much faster for some cases, but slower for others. For example:

```
> CG2x := GolayCode(GF(2), true);
> incDG8 := IncidenceStructure< Length(CG2x) |
>   Words(CG2x, 8) >;
> time IsBalanced(incDG8, 4);
true 5
Time: 16.241
> // redefine incDG8 for a new test
> incDG8 := IncidenceStructure< Length(CG2x) |
>   Words(CG2x, 8) >;
> time IsBalanced(incDG8, 4 : Al := "Orbits");
true 5
```

Table 38.4. Testing properties of an incidence structure

MAGMA	Meaning
IsSimple(I)	**true** if incidence structure I has no repeated blocks
IsUniform(I)	**true** if I has at least one block and the blocksize k is constant; if **true**, also returns k
IsTrivial(I)	**true** if I is uniform and each k-subset of points appears (at least once) as a block
IsSelfDual(I)	**true** if I is isomorphic to its dual
IsBalanced(I, t)	**true** if I is balanced with respect to t; if **true**, also returns λ. Parameter **Al** specifies algorithm.
IsResolvable(I)	**true** if there exists a resolution of I (a partition of the blocks of D into parallel classes); if **true**, also returns a resolution
IsNearLinearSpace(I)	**true** if I is a near-linear space
IsLinearSpace(I)	**true** if I is a linear space
IsDesign(I, t)	**true** if I is a t-design; if **true**, also returns λ. Parameter **Al** specifies algorithm.
IsPointRegular(L)	**true** if (near-)linear space L is point regular; if **true**, also returns the point regularity
IsLineRegular(L)	**true** if (near-)linear space L is line regular; if **true**, also returns the line regularity
IsComplete(D)	**true** if design D is trivial
IsSymmetric(D)	**true** if design D satisfies $t \geq 2$ and $v = b$
IsSteiner(D, t)	**true** if design D is a Steiner t-design (i.e., $\lambda = 1$)

```
Time: 21.882
```

Notice that Table 38.4 includes functions for testing whether an incidence structure I is a design, near-linear space or linear space. If such a property does hold for I, it may be converted to a structure in the appropriate category using one of the functions in Table 38.5. The reason for performing such a conversion is so that operations provided for that category may be applied to the structure. For example:

```
> IsDesign(incDG8, 4);
true 5
> IsDesign(incDG8, 5);
true 1
> DG8 := Design(incDG8, 5);
> DG8;
5-(24, 8, 1) Design with 759 blocks
> PascalTriangle(DG8);
                759
```

Table 38.5. Changing the category of an incidence structure

MAGMA	Meaning
IncidenceStructure(I)	Given I (in any incidence structure category), return I in category **Inc**
NearLinearSpace(I)	Given near-linear space I (in any incidence structure category), return I in category **IncNsp**
LinearSpace(I)	Given linear space I (in any incidence structure category), return I in category **IncLsp**
Design(I, t)	Given t-design I (in any incidence structure category), return I as a t-design in category **Dsgn**

```
                506 253
              330 176  77
            210 120  56  21
          130  80  40  16   5
        78  52  28  12   4   1
      46  32  20   8   4   0   1
    30  16  16   4   4   0   0   1
30   0  16   0   4   0   0   0   1
```

38.6 Producing New Incidence Structures

Table 38.6. Operations on incidence structures

MAGMA	Meaning
Complement(I)	Complement of incidence structure I
Dual(I)	Dual of I
Contraction(I, p)	$(P - \{p\}, \{b - \{p\} : b \in B \mid p \in b\})$
Contraction(I, b)	$(b, \{b \cap c : c \in B \mid c \neq b\})$
Residual(I, p)	$(P - \{p\}, \{x : x \in B \mid p \notin x\})$
Residual(I, b)	$(P - b, B - \{b\})$
Simplify(I)	I with repeated blocks removed (leaving one copy of each block)
Sum(Q)	Given sequence Q of incidence structures over same point-set, return structure with this point-set and whose block-set is the union of the blocks
Union(I_1, I_2)	Given $I_1 = (P_1, B_1)$ and $I_2 = (P_2, B_2)$, with P_1 and P_2 disjoint, return $(P_1 \cup P_2, B_1 \cup B_2)$
Restriction(L, S)	Restriction of (near-)linear space L to set S of points

Table 38.6 lists functions that may be used to produce new incidence structures from a given incidence structure I with point-set P and block-set B, where p or b denote a point or block of I. The notation (π, β) denotes the resulting incidence structure, with point-set π and block-set β.

For example, the Fano plane may be considered as a 2-$(7, 3, 1)$ design. Since it is symmetric, its dual is a design with the same parameters. However, the **Dual** function does not return a design with the optimal t, since to do so in general might be very expensive computationally; that value must be found 'by hand':

```
> fano := Design< 2, 7 | {1,2,3}, {1,4,5}, {1,6,7},
>   {2,4,7}, {2,5,6}, {3,5,7}, {3,4,6} >;
> fano;
2-(7, 3, 1) Design with 7 blocks
> IsSymmetric(fano);
true
> du := Dual(fano);
> du;
0-(7, 3, 7) Design with 7 blocks
> du2 := Design(du, 2);
> du2;
2-(7, 3, 1) Design with 7 blocks
```

38.7 Standard Constructions for Designs

Table 38.7. Witt and Hadamard designs

Magma	Meaning
WittDesign(12)	5-$(12, 6, 1)$ Witt design (small Mathieu design)
WittDesign(24)	5-$(24, 8, 1)$ Witt design (large Mathieu design)
IsHadamard(M)	**true** if matrix ring element M (over any ring) is a Hadamard matrix
IsHadamardEquivalent(H, J)	**true** if Hadamard matrices H and J are Hadamard equivalent
HadamardRowDesign(H, i)	Hadamard 3-design corresponding to row i of H, where H is an $n \times n$ Hadamard matrix with $n \geq 4$
HadamardColumnDesign(H, i)	Hadamard 3-design corresponding to i^{th} column i of H, where H is as above

Two of the most famous examples of designs are the Witt (or Mathieu) designs and the Hadamard designs. MAGMA's functions for constructing these are listed in Table 38.7. **WittDesign**(n) accepts two values for its argument, $n = 12$ and $n = 24$:

```
> W12 := WittDesign(12); W12;
5-(12, 6, 1) Design with 132 blocks
> W24 := WittDesign(24); W24;
5-(24, 8, 1) Design with 759 blocks
```

The Hadamard functions relate to the construction of a 3-design from a Hadamard matrix, which is a square matrix with orthogonal rows and containing only ± 1 as its entries. Thus an $m \times m$ matrix M is a Hadamard matrix if all entries are ± 1 and $MM^t = mI_m$, where I_m is the $m \times m$ identitiy matrix. For example:

```
> H := MatrixRing(IntegerRing(), 8) !
>   [ 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,  1, -1,  1, -1,  1,
>     1, -1, -1,  1,  1, -1,  1, -1] ;
> IsHadamard(H);
true
> HRD := HadamardRowDesign(H, 5);
> HRD : Maximal;
3-(8, 4, 1) Design with 14 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8 @}
Blocks:
    {1, 3, 5, 8},
    {1, 3, 6, 7},
    {1, 4, 5, 7},
    {1, 4, 6, 8},
    {1, 2, 3, 4},
    {1, 2, 7, 8},
    {1, 2, 5, 6},
    {2, 4, 6, 7},
    {2, 4, 5, 8},
    {2, 3, 6, 8},
    {2, 3, 5, 7},
    {5, 6, 7, 8},
    {3, 4, 5, 6},
```

```
    {3, 4, 7, 8}
> HRD eq HadamardColumnDesign(H, 2);
true
```

38.8 Difference Sets and Their Development

Table 38.8. Difference sets and their developments

MAGMA	Meaning
DifferenceSet(p, t)	Difference set of type given by string t and corresponding to prime p, where t is `"Q"`, `"H6"`, `"T"`, `"B"`, `"B0"`, `"O"`, `"O0"`, or `"W4"` and the choice of t imposes conditions on p
SingerDifferenceSet(n, q)	Singer difference set corresponding to a hyperplane of PG(n, q)
IsDifferenceSet(B)	**true** if B is a difference set over $\mathbb{Z}/m\mathbb{Z}$ or an iterable finite group; if **true**, also returns λ, the number of times each non-identity group/ring element appears as a 'difference' of elements of B
Development(B)	Given set B which is a difference set relative to A, where A is $\mathbb{Z}/m\mathbb{Z}$ or a finite group, return the symmetric design with point set A and whose blocks consist of the sets obtained by translating B by each element of A in turn
Development(T)	Given difference family $T = \{B_1, \ldots, B_l\}$ where the B_i are difference sets relative to A [as above], return the incidence structure with point set A and whose i^{th} block is the set $\{B_1 \cup \cdots \cup B_l\}$ translated by the i^{th} element of A

Let G be a group of order v. A (v, k, λ)-*difference set* for G is a k-subset D of G such that the multiset $\{* \, gh^{-1} : g, h \in D \mid g \neq h \, *\}$ contains every non-identity element of G exactly λ times. Difference sets may be used to generate incidence structures by means of the process known as *development*.

The MAGMA functions for difference sets and their developments are shown in Table 38.8. In the case of the function **DifferenceSet**(p, t), the possible values for the string t have the interpretations as given by Marshall Hall in [Hal86], pp. 141–142. A difference set produced by this function or by the function **SingerDifferenceSet** will have a residue class ring as its universe, in which case the set has the additive form $\{g-h : g, h \in D | g \neq h\}$. For the functions that operate on an existing (or putative) difference set, the group G may be represented either additively by a residue class ring

or abelian group, or multiplicatively by a finite group in any other group category in which iteration is possible (i.e., other than **GrpFP**).

In the following example, the difference set D constructed is of type "H6" with $p = 43$. The multiset operations are used to verify that D is a difference set with $\lambda = 10$, and then a design is constructed from D:

```
> D := DifferenceSet(43, "H6");
> D;
{ 1, 2, 3, 32, 4, 33, 5, 35, 37, 8, 39, 11, 41,
12, 42, 16, 19, 20, 21, 22, 27 }
> Universe(D);
Residue class ring of integers modulo 43
> _, lambda := IsDifferenceSet(D);
> lambda;
10

> // check
> diffs := {* g - h : g, h in D | g ne h *};
> diffs;
{* 29^^10, 1^^10, 30^^10, 2^^10, 31^^10, 3^^10, 32^^10,
33^^10, 4^^10, 5^^10, 34^^10, 6^^10, 35^^10, 36^^10,
7^^10, 8^^10, 37^^10, 9^^10, 38^^10, 39^^10, 10^^10,
40^^10, 11^^10, 41^^10, 12^^10, 42^^10, 13^^10, 14^^10,
15^^10, 16^^10, 17^^10, 18^^10, 19^^10, 20^^10, 21^^10,
22^^10, 23^^10, 24^^10, 25^^10, 26^^10, 27^^10, 28^^10 *}
> #Set(diffs) eq (43 - 1);
true

> dv := Development(D);
> dv;
2-(43, 21, 10) Design with 43 blocks
```

Difference sets may also be constructed using group theory; the following example is adapted from [LiW92], pp. 198, 345, 348. Suppose that D is any normalized $(21, 5, 1)$-difference set in $\mathbb{Z}/21\mathbb{Z}$, if such a difference set exists. It follows from the Multiplier Theorem for difference sets that D must satisfy $2D = D$, that is, D equals the set consisting of the elements of D multiplied by 2. Therefore D must be a union of orbits of the mapping $x \mapsto 2x$ in $\mathbb{Z}/21\mathbb{Z}$, and so all possible difference sets D may be constructed simply by collecting together orbits of appropriate sizes. In MAGMA, the orbits may be calculated as the orbits of a permutation group generated by the permutation-representation of this mapping:

```
> z21 := Set(ResidueClassRing(21));
```

```
> G := PermutationGroup< z21 | [2*a : a in z21] >;
> orbs := Orbits(G);
> orbs;
[
    GSet{ 0 },
    GSet{ 13, 5, 17, 19, 20, 10 },
    GSet{ 11, 1, 2, 4, 16, 8 },
    GSet{ 15, 18, 9 },
    GSet{ 12, 3, 6 },
    GSet{ 14, 7 }
]
```

From the sizes of the orbits, the only possible difference sets could be $D1$, the union of the fourth and sixth orbits, or $D2$, the union of the fifth and sixth orbits. It turns out that $D1$ is a difference set, and since $D2$ consists of the negatives of $D1$, all the difference sets with the given parameters are equivalent:

```
> D1 := orbs[4] join orbs[6];
> D1;
{ 14, 15, 18, 7, 9 }
> IsDifferenceSet(D1);
true 1
> D2 := orbs[5] join orbs[6];
> D2 eq { -x : x in D1 };
true
```

The blocks of the development of $D1$ form a projective plane of order 4:

```
> dv := Development(D1);
> dv;
2-(21, 5, 1) Design with 21 blocks
> proj := ProjectivePlane(dv);
> proj : Maximal;
Projective plane of order 4
Points: {@ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20 @}
Lines:
    {7, 9, 14, 15, 18},
    {8, 10, 15, 16, 19},
    {9, 11, 16, 17, 20},
    {0, 10, 12, 17, 18},
    {1, 11, 13, 18, 19},
    {2, 12, 14, 19, 20},
    {0, 3, 13, 15, 20},
```

```
{0, 1, 4, 14, 16},
{1, 2, 5, 15, 17},
{2, 3, 6, 16, 18},
{3, 4, 7, 17, 19},
{4, 5, 8, 18, 20},
{0, 5, 6, 9, 19},
{1, 6, 7, 10, 20},
{0, 2, 7, 8, 11},
{1, 3, 8, 9, 12},
{2, 4, 9, 10, 13},
{3, 5, 10, 11, 14},
{4, 6, 11, 12, 15},
{5, 7, 12, 13, 16},
{6, 8, 13, 14, 17}
```

38.9 Equality and Isomorphism

Two incidence structures I and J are deemed to be equal in MAGMA, as tested by the operators **eq** and **ne**, if the sets of points over which they were defined are the same, and the blocks are the same. The function **IsIsomorphic**(I, J) returns whether I and J are isomorphic, which is a weaker property than equality. If the result is **true**, a second value is returned giving the isomorphism as a mapping from I to J. (Section 36.2 gives details of a parameter to this function.)

For example, let $B1$ and $B2$ be the two 2-$(16, 6, 2)$ designs defined below. It will be shown that they are isomorphic. This example is taken from [AsK92], p. 5.

The points of $B1$ are the 16 pairs (i, j), for $1 \leq i, j \leq 4$. Each pair (i, j) defines a block $B_{i,j}$ by the following rule: the points in $B_{i,j}$ are the pairs $(i, k), 1 \leq k \leq 4$ for $k \neq j$, together with the pairs $(k, j), 1 \leq k \leq 4$ for $k \neq i$. In the MAGMA code below, the function N maps the pairs (i, j) bijectively onto the set of integers $\{1, 2, \ldots, 16\}$.

```
> B1 := Design< 2, 16 |
>     [
>      { N(i, k) : k in [1..4] | k ne j } join
>      { N(k, j) : k in [1..4] | i ne k } :
>                  i, j in [1..4]
>     ]
>     where N := func< i, j | i + 4*(j-1) > >;
>
```

```
> B1 : Maximal;
2-(16, 6, 2) Design with 16 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16 @}
Blocks:
    {2, 3, 4, 5, 9, 13},
    {1, 6, 7, 8, 9, 13},
    {1, 5, 10, 11, 12, 13},
    {1, 5, 9, 14, 15, 16},
    {1, 3, 4, 6, 10, 14},
    {2, 5, 7, 8, 10, 14},
    {2, 6, 9, 11, 12, 14},
    {2, 6, 10, 13, 15, 16},
    {1, 2, 4, 7, 11, 15},
    {3, 5, 6, 8, 11, 15},
    {3, 7, 9, 10, 12, 15},
    {3, 7, 11, 13, 14, 16},
    {1, 2, 3, 8, 12, 16},
    {4, 5, 6, 7, 12, 16},
    {4, 8, 9, 10, 11, 16},
    {4, 8, 12, 13, 14, 15}
```

The design $B2$ is constructed from the complete graph K_6 on six vertices. The points are the 15 edges of K_6, together with another point denoted by ∞. The first ten blocks comprise the edges of those subgraphs of K_6 that consist of two disjoint triangles. The remaining six blocks are formed by taking each vertex v of K_6 in turn, and forming the set comprising the edges incident with v and ∞. In the code below the points are renumbered 1 to 16.

```
> inf := 16;
> S := { 1..6 };
> E := SetToIndexedSet(Subsets(S, 2));
> I := func< x | Index(E, x) >;
> triangles := func< T |
>   { I(U) : U in Subsets(T, 2) } join
>   { I(U) : U in Subsets(S diff T, 2) } >;
> claw := func< i |
>   Include({ I({i, j}) : j in [1..6] | i ne j }, inf) >;
> B2 := Design< 2, 16 |
>     [ triangles({1,j,k}) : k in [j+1..6], j in [2..5]  ]
>     cat [ claw(i) : i in [1..6] ] >;
> B2 : Maximal;
2-(16, 6, 2) Design with 16 blocks
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16 @}
```

```
Blocks:
    {1, 6, 7, 9, 10, 12},
    {7, 8, 10, 11, 14, 15},
    {2, 3, 6, 10, 13, 15},
    {4, 5, 9, 10, 13, 14},
    {1, 3, 4, 7, 8, 13},
    {1, 2, 4, 6, 11, 14},
    {1, 3, 5, 9, 11, 15},
    {2, 4, 8, 9, 12, 15},
    {3, 5, 6, 8, 12, 14},
    {2, 5, 7, 11, 12, 13},
    {1, 2, 5, 8, 10, 16},
    {3, 4, 10, 11, 12, 16},
    {1, 12, 13, 14, 15, 16},
    {6, 8, 9, 11, 13, 16},
    {2, 3, 7, 9, 14, 16},
    {4, 5, 6, 7, 15, 16}
```

Now $B1$ and $B2$ may be tested for isomorphism, using **IsIsomorphic**. The second return value of this function is an isomorphism from $B1$ to $B2$.

```
> flag, m := IsIsomorphic(B1, B2);
> flag;
true
> m;
Mapping from: Dsgn: B1 to Dsgn: B2
> [<i, m(i)>: i in Points(B1)];
[ <1, 1>, <2, 6>, <3, 7>, <4, 16>, <5, 12>, <6, 3>, <7, 15>,
<8, 9>, <9, 2>, <10, 5>, <11, 8>, <12, 11>, <13, 13>,
<14, 14>, <15, 10>, <16, 4> ]
```

38.10 Automorphism Group

Table 38.9 lists the major operations associated with the automorphism group of an incidence structure I. Section 36.3 explains these and other operations in more detail, including the multiple values returned by the group functions.

For example:

```
> D := WittDesign(24);
> IsPointTransitive(D) and IsBlockTransitive(D);
true
> A, gspt, gsbl := AutomorphismGroup(D);
```

Table 38.9. Automorphism group functions

MAGMA	Meaning
AutomorphismGroup(I)	If incidence structure I is simple, returns automorphism group of I in its action on the point-set; otherwise, returns automorphism group of I in its action on the union of the point-set and the block-set
PointGroup(I)	Automorphism group of I in its action on the point-set
BlockGroup(I)	Automorphism group of I in its action on the block-set
IsPointTransitive(I)	**true** if automorphism group of I acts transitively on point-set of I
IsBlockTransitive(I)	**true** if automorphism group of I acts transitively on block-set of I

```
> FactoredOrder(A);
[ <2, 10>, <3, 3>, <5, 1>, <7, 1>, <11, 1>, <23, 1> ]
> a := Random(A); a;
(2, 7, 17, 24, 13, 20, 23, 5, 18, 16, 21, 11, 12, 4, 6,
    14, 10, 22, 8, 9, 19, 15, 3)
> p5 := Point(D, 5); p5;
5
> Image(a, gspt, p5);
18
> b6 := Block(D, 6); b6;
{8, 11, 17, 18, 19, 20, 22, 24}
> Image(a, gsbl, b6);
{8, 9, 12, 13, 15, 16, 23, 24}
> A3 := SylowSubgroup(A, 3);
> Orbit(A3, gspt, p5);
GSet{ 12, 23, 5, 6, 18, 20, 21, 22, 11 }
```

Notice that if I is not simple, then **AutomorphismGroup**(I) acts on both the points and the blocks, so the degree of the group is the sum of the number of points and the number of blocks. For example:

```
> S2 := IncidenceStructure< {@ 2,5,6 @} |
>   {5,6}, {}, {5}, {2}, {5,6} >;
> G := AutomorphismGroup(S2);
> G;
Permutation group G acting on a set of cardinality 8
Order = 2
    (4, 8)
```

38.11 Relationship to Graphs and Linear Codes

Table 38.10. Incidence structures, graphs and codes

MAGMA	Meaning
IncidenceStructure(G)	Incidence structure I corresponding to graph G, where blocks of D correspond to edges of G
PointGraph(I)	Point graph G of incidence structure I, where vertices of G are points of I, and vertices $u, v \in G$ are adjacent if both u and v are in a block of I
BlockGraph(I)	Point graph of dual of I
IncidenceGraph(I)	Incidence graph (bipartite) of incidence structure I, with vertex set the union of the point set and block set of I, and with point p being adjacent to block b whenever $p \in b$
LinearCode(I, K)	Given incidence structure I with v points, and finite field K, return linear code of length v generated by characteristic functions of blocks of I considered as vectors of $K^{(v)}$

Incidence structures, graphs and linear codes possess close combinatorial relationships. Table 38.10 lists the functions for transferring between a graph and an incidence structure, and the function for converting an incidence structure into a code. In addition, recall from Section 38.2 that a set of words from a linear code may be used to generate an incidence structure.

For example, the code over GF(2) generated by the Fano plane is isomorphic to the Hamming code with $r = 3$:

```
> fano := Design< 2, 7 | {1,2,3}, {1,4,5}, {1,6,7},
>   {2,4,7}, {2,5,6}, {3,5,7}, {3,4,6} >;
> C := LinearCode(fano, GF(2));
> C;
[7, 4, 3] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1]
[0 1 0 0 1 1 0]
[0 0 1 0 1 0 1]
[0 0 0 1 1 1 1]
> IsIsomorphic(C, HammingCode(GF(2), 3));
true (2, 7, 3, 6)
```

39. Finite Planes

In its facilities for finite geometry, MAGMA supports both projective planes (category **PlaneProj**) and affine planes (**PlaneAff**). Within these categories, there are special operations for classical planes, that is, planes defined over a 2-dimensional vector space (for affine planes) or a 3-dimensional vector space (for projective planes). Note that MAGMA's operations for creating and manipulating planes bear strong resemblance to those for incidence structures.

For an introduction to the theory, the reader should consult [HuP73].

39.1 Constructing Projective and Affine Planes

In MAGMA, there are three main ways to construct a projective or affine plane: from an incidence relation involving points and lines that obeys the appropriate axioms; from a pre-existing incidence structure (in any category) that obeys the appropriate axioms; or from a vector space over a finite field. Some MAGMA operations on planes are only possible if the plane has been constructed explicitly from a vector space; such planes will be called *classical* here.

All these methods of construction return the plane as the principal return value, in the category **PlaneProj** or **PlaneAff** as requested. They also provide the point-set and the line-set of the plane as the second and third return values. The point-set and line-set belong to special categories **PlanePtSet** and **PlaneLnSet**. Although called the point-*set* and line-*set*, they are not true MAGMA sets, but structures which act as the parent structures for points and lines respectively.

39.1.1 Planes from Incidence Relations

A finite plane may be described by v points and b lines, where the lines are sets of points. For a projective plane, the incidence relation between the

points and lines must obey the following criteria: any two distinct points must lie on a unique line, any two lines must meet in a unique point, and there must be four points such that no three are collinear. For an affine plane, the criteria are: any two distinct points must lie on a unique line, given any line l and any point P not on l there must be a unique line m such that P is on m and l and m have no common points, and there must be three non-collinear points.

Any finite projective or affine plane, including a non-Desarguesian plane, may be created in MAGMA by means of a constructor of the following form:

ProjectivePlane< point-set description | line-set description **>**
AffinePlane< point-set description | line-set description **>**

On the left side of the | symbol, the points are described: either explicitly as an indexed set V of v points; or, in terms of the positive integer v, if the points are $V = \{1, \ldots, v\}$. On the right side, the lines are described: as a $v \times b$ matrix of zeros and ones (as an element of a matrix space over any coefficient ring), which is interpreted as an incidence matrix for the plane; as a set of codewords of a linear code with length v, in which case the line set is the set of supports of the codewords; or as a comma-separated list (in the order of the lines) in which each item may be a subset of V, a line of an existing plane, or a sequence or indexed set of either of these.

To take a familiar example, the Fano plane (a projective plane of order 2 on 7 points) may be defined in any of the following ways. In the first method, the points are the elements of the integer-set $\{1, \ldots, 7\}$, and the lines are typed directly by the user:

```
> fp1 := ProjectivePlane< 7 | {1,2,3}, {1,4,5},
>     {1,6,7}, {2,4,7}, {2,5,6}, {3,5,7}, {3,4,6} >;
> fp1 : Maximal;
Projective Plane of order 2
Points: {@ 1, 2, 3, 4, 5, 6, 7 @}
Lines:
    {1, 2, 3},
    {1, 4, 5},
    {1, 6, 7},
    {2, 4, 7},
    {2, 5, 6},
    {3, 5, 7},
    {3, 4, 6}
```

In the second method, the Fano plane is defined by means of an incidence matrix, using the standard set of points once again:

```
> incmat := RMatrixSpace(IntegerRing(), 7, 7) !
>   [1,1,1,0,0,0,0, 1,0,0,1,1,0,0, 1,0,0,0,0,1,1,
>    0,1,0,1,0,0,1, 0,1,0,0,1,1,0, 0,0,1,0,1,0,1,
>    0,0,1,1,0,1,0 ];
> incmat;
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[1 0 0 0 0 1 1]
[0 1 0 1 0 0 1]
[0 1 0 0 1 1 0]
[0 0 1 0 1 0 1]
[0 0 1 1 0 1 0]
> fp2 := ProjectivePlane< 7 | incmat >;
```

In the third and final method, the points are the elements of $\mathbb{Z}/7\mathbb{Z}$ (labelled 0 to 6), and the lines are all triples of the form $\{a, a+1, a+3\}$, where $a \in \mathbb{Z}/7\mathbb{Z}$:

```
> Z7 := ResidueClassRing(7);
> pts := SetToIndexedSet(Set(Z7));
> lns := {@ {a, a+1, a+3} : a in pts @};
> fp3 := ProjectivePlane< pts | lns >;
> fp3 : Maximal;
Projective Plane of order 2
Points: {@ 0, 1, 2, 3, 4, 5, 6 @}
Lines:
    {0, 1, 3},
    {1, 2, 4},
    {2, 3, 5},
    {3, 4, 6},
    {0, 4, 5},
    {1, 5, 6},
    {0, 2, 6}
```

As an example of an affine plane, let C be the full vector space of dimension 4 over GF(2), regarded as a linear code, and consider the order-2 affine plane constructed from the codewords of weight 2. The lines of the resulting structure are the same as the 2-subsets of $\{@1, \ldots, 4@\}$:

```
> C := LinearCode(VectorSpace(GF(2), 4));
> A1 := AffinePlane< 4 | Words(C, 2) >;
> A1 : Maximal;
Affine Plane of order 2
Points: {@ 1, 2, 3, 4 @}
Lines:
    {1, 3},
```

```
    {1, 4},
    {2, 4},
    {2, 3},
    {3, 4},
    {1, 2}

> A2 := AffinePlane< 4 | Setseq(Subsets({1..4}, 2)) >;
> A2 : Maximal;
Affine Plane of order 2
Points: {@ 1, 2, 3, 4 @}
Lines:
    {1, 3},
    {1, 4},
    {2, 4},
    {2, 3},
    {1, 2},
    {3, 4}

> A1 eq A2;
true
```

When a plane is being built using one of these constructors, the user often knows from earlier work or from theory that the lines satisfy the conditions for the category. MAGMA provides a parameter **Check** for the constructors **ProjectivePlane** and **AffinePlane** so that the user can stop the system checking that the conditions are satisfied. If **Check** is assigned **false**, instead of its default value **true**, then MAGMA will not check the conditions, so the user must ensure that they are true. This technique will save a considerable amount of time if the plane is large, and is especially recommended for structures which are loaded into MAGMA from previously generated files.

39.1.2 Planes from Incidence Structures

Suppose that I is an incidence structure, in any of the incidence structure categories. If I has the properties of an affine plane, then it may be converted to an affine plane in the **PlaneAff** category using the function **AffinePlane**(I). Similarly, if I has the properties of a projective plane, then it may be converted to the category **PlaneProj** by means of the function **ProjectivePlane**(I).

For example, a projective plane of order 5 whose points are the elements of $\mathbb{Z}/31\mathbb{Z}$ may be constructed from the development [a kind of design] of a Singer difference set with $n = 2$ and $q = 5$:

```
> sds := SingerDifferenceSet(2, 5); sds;
```

```
{ 0, 1, 3, 14, 26, 10 }
> sdv := Development(sds); sdv;
2-(31, 6, 1) Design with 31 blocks
> spp := ProjectivePlane(sdv);
> Universe(Support(spp));
Residue class ring of integers modulo 31
> spp : Maximal;
Projective Plane of order 5
Points: {@ 0, 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 @}
Lines:
    {0, 1, 3, 10, 14, 26},
    {1, 2, 4, 11, 15, 27},
    {2, 3, 5, 12, 16, 28},
    {3, 4, 6, 13, 17, 29},
    {4, 5, 7, 14, 18, 30},
    {0, 5, 6, 8, 15, 19},
    {1, 6, 7, 9, 16, 20},
    {2, 7, 8, 10, 17, 21},
    {3, 8, 9, 11, 18, 22},
    {4, 9, 10, 12, 19, 23},
    {5, 10, 11, 13, 20, 24},
    {6, 11, 12, 14, 21, 25},
    {7, 12, 13, 15, 22, 26},
    {8, 13, 14, 16, 23, 27},
    {9, 14, 15, 17, 24, 28},
    {10, 15, 16, 18, 25, 29},
    {11, 16, 17, 19, 26, 30},
    {0, 12, 17, 18, 20, 27},
    {1, 13, 18, 19, 21, 28},
    {2, 14, 19, 20, 22, 29},
    {3, 15, 20, 21, 23, 30},
    {0, 4, 16, 21, 22, 24},
    {1, 5, 17, 22, 23, 25},
    {2, 6, 18, 23, 24, 26},
    {3, 7, 19, 24, 25, 27},
    {4, 8, 20, 25, 26, 28},
    {5, 9, 21, 26, 27, 29},
    {6, 10, 22, 27, 28, 30},
    {0, 7, 11, 23, 28, 29},
    {1, 8, 12, 24, 29, 30},
    {0, 2, 9, 13, 25, 30}
```

39.1.3 Classical Planes from Vector Spaces

Every Desarguesian projective or affine plane bears a strong relationship to some vector space over a finite field. MAGMA provides special functions for defining such planes, as explained below, and any plane defined in this way will be described here as a *classical plane*. A few MAGMA operations on planes are only available for classical planes, and will not work for a plane defined otherwise, even if it happens to be Desarguesian.

Let W be a 3-dimensional vector space defined over the field $F = \mathrm{GF}(q)$, where q is a prime power. Then there exists a classical projective plane P of order $n = q$ defined by the one-dimensional and two-dimensional subspaces of W. P may be created in MAGMA by any of the functions **ProjectivePlane**(W), **ProjectivePlane**(F), or **ProjectivePlane**(q). The points and lines of classical projective planes are printed by MAGMA in a special way: the points are given by 3 colon-separated field elements, surrounded by parentheses; the lines are also given by 3 colon-separated field elements, but are surrounded by angle brackets. A line $\langle a : b : c \rangle$ may be interpreted as the line given by the equation $ax + by + cz = 0$. For example, the following lines construct the classical projective plane of order 27, then demonstrate the appearance of points and lines:

```
> F<w> := GF(27); F;
Finite field of size 3^3
> P27, VP27, LP27 := ProjectivePlane(F);
> P27;
Projective Plane PG(2, 27)
> VP27, LP27;
Point-set of Projective Plane PG(2, 27)
Line-set of Projective Plane PG(2, 27)

> pt := Random(VP27);
> pt;
( 1 : w^7 : w^10 )
> ln := Random(LP27);
> ln;
< 1 : w^6 : w^19 >
> Set(ln);
{  ( 1 : w^4 : 1 ), ( 1 : w^9 : w^19 ), ( 1 : w^18 : 2 ),
( 1 : w^8 : w^10 ), ( 0 : 1 : 1 ), ( 1 : w^12 : w ), ( 1 :
w^11 : w^14 ), ( 1 : w^16 : w^8 ), ( 1 : w^7 : 0 ), ( 1 :
w^15 : w^6 ), ( 1 : 1 : w^5 ), ( 1 : w^17 : w^18 ), ( 1 :
w^21 : w^3 ), ( 1 : 2 : w^17 ), ( 1 : w^23 : w^21 ), ( 1 :
w^20 : w^7 ), ( 1 : w^6 : w^22 ), ( 1 : w^24 : w^12 ), ( 1
: w^3 : w^23 ), ( 1 : w^22 : w^15 ), ( 1 : w^5 : w^4 ), (
```

```
1 : w^2 : w^9 ), ( 1 : w^25 : w^11 ), ( 1 : w^10 : w^16 ),
( 1 : 0 : w^20 ), ( 1 : w^19 : w^2 ), ( 1 : w : w^24 ), (
1 : w^14 : w^25 ) }
```

In the case of affine planes, let W be a 2-dimensional vector space defined over the field $F = \mathrm{GF}(q)$, where q is a prime power. Then there exists a classical affine plane P of order $n = q$ defined by the one-dimensional and two-dimensional subspaces of W. P may be created in MAGMA using any of the functions **AffinePlane**(W), **AffinePlane**(F), or **AffinePlane**(q). The points of classical affine planes are printed as 2 comma-separated field elements, surrounded by parentheses, and the lines are printed as 3 colon-separated field elements, surrounded by angle brackets, where $\langle a : b : c \rangle$ may be interpreted as the line given by the equation $ax + by + c = 0$. For example:

```
> A27, VA27, LA27 := AffinePlane(F);
> A27;
Affine Plane AG(2, 27)
> pt := Random(VA27);
> pt;
( w^25, w^7 )
> ln := Random(LA27);
> ln;
< 1 : w^5 : 0 >
> Set(ln);
{  ( w^18, 1 ), ( w^19, w ), ( w^20, w^2 ), ( w^21, w^3 ),
( w^22, w^4 ), ( w^23, w^5 ), ( w^24, w^6 ), ( w^25, w^7),
( 1, w^8 ), ( w, w^9 ), ( w^2, w^10 ), ( w^3, w^11 ),
( w^4, w^12 ), ( w^5, 2 ), ( w^6, w^14 ), ( w^7, w^15),
( w^8, w^16 ), ( w^9, w^17 ), ( w^10, w^18 ), ( w^11, w^19),
( w^12, w^20 ), ( 2, w^21 ), ( w^14, w^22 ), ( w^15, w^23),
( w^16, w^24 ), ( w^17, w^25 ), ( 0, 0 ) }
```

39.2 Equality and Isomorphism

Two planes P and Q are deemed to be equal in MAGMA, as tested by the operators **eq** and **ne**, if the sets of points over which they were defined are the same, and the lines are the same. The function **IsIsomorphic**(P, Q) returns whether or not P and Q are isomorphic. If the result is **true**, a second value is returned giving the isomorphism as a mapping from P to Q. (See Section 36.2 for details of a parameter of this function.) For example:

```
> fp2 eq fp1;
true
```

```
> fp1 eq fp3;
false
> ii, m:= IsIsomorphic(fp1, fp3);
> ii;
true
> m;
Mapping from: PlaneProj: fp1 to PlaneProj: fp3
```

39.3 Points and Lines

39.3.1 General Point and Line Operations

Table 39.1. Point-set and line-set functions

MAGMA	Meaning
IncidenceMatrix(P)	Incidence matrix of P
PointSet(P)	Point-set V of plane P
LineSet(P)	Line-set L of plane P
Points(P)	Indexed set of points of P
Lines(P)	Indexed set of lines of P
NumberOfPoints(P), #V	Number of points of P
NumberOfLines(P), #L	Number of lines of P
Support(P)	Indexed set of underlying points of P, in their original category
Rep(V)	Representative point of point-set V
Random(V)	Random point of point-set V
Rep(L)	Representative line of line-set L
Random(L)	Random line of line-set L
ParentPlane(x)	Plane for which x is the point-set, the line-set, a point or a line
$V.i$	i^{th} point of plane P with point-set V
$L.i$	i^{th} line of plane P with line-set L
$V\,!\,x$	Point corresponding to element x of support of P, where V is point-set of P (if P is classical, x must be a vector)
$L\,!\,S$	Given line-set L of plane P and set or sequence S of collinear points, return line containing S

Table 39.1 (p. 784) lists the functions associated with the point-set V and line-set L of a projective or affine plane P. As was mentioned above, V and L are given as the second and third return values of all constructors

and functions that return planes. They are also available from the functions **PointSet**(P) and **LineSet**(P). V and L belong to the special categories **PlanePtSet** and **PlaneLnSet**. They act as the parent structures for points and lines. For example:

```
> A3, VA3, LA3 := AffinePlane(3);
> A3;
Affine Plane AG(2, 3)
> VA3;
Point-set of Affine Plane AG(2, 3)
> LA3;
Line-set of Affine Plane AG(2, 3)

> fp1 := ProjectivePlane< 7 | {1,2,3}, {1,4,5},
>     {1,6,7}, {2,4,7}, {2,5,6}, {3,5,7}, {3,4,6} >;
> LineSet(fp1);
Line-set of Projective Plane of order 2
```

Individual points and lines in a plane P belong to the categories **PlanePt** and **PlaneLn**. As discussed above, the appearance of points and lines in a plane P depends on whether P is classical and (if it is classical) on whether it is affine or projective. In all cases, however, the functions **Points**(P) and **Lines**(P) return the points and lines of P explicitly as indexed sets, where the indexing reflects the ordering of the points and the lines. The universes of these sets are V and L. For example:

```
> ptsA3 := Points(A3);
> ptsA3;
{@ ( 0, 0 ), ( 1, 0 ), ( 2, 0 ), ( 0, 1 ),
( 1, 1 ), ( 2, 1 ), ( 0, 2 ), ( 1, 2 ), ( 2, 2 ) @}
> Universe(ptsA3) eq VA3;
true
> lnsA3 := Lines(A3);
> lnsA3;
{@
    < 1 : 0 : 0 >,
    < 1 : 1 : 0 >,
    < 1 : 2 : 0 >,
    < 1 : 0 : 1 >,
    < 1 : 1 : 1 >,
    < 1 : 2 : 1 >,
    < 1 : 0 : 2 >,
    < 1 : 1 : 2 >,
    < 1 : 2 : 2 >,
    < 0 : 1 : 0 >,
```

```
    < 0 : 1 : 1 >,
    < 0 : 1 : 2 >
@}
> Universe(lnsA3) eq LA3;
true

> Points(fp1);
{@ 1, 2, 3, 4, 5, 6, 7 @}
> Lines(fp1);
{@
    {1, 2, 3},
    {1, 4, 5},
    {1, 6, 7},
    {2, 4, 7},
    {2, 5, 6},
    {3, 5, 7},
    {3, 4, 6}
@}
```

The function **Support**(P) returns an indexed set containing the underlying objects from which the points of P were created. For a classical plane, the universe of this set will be the vector space corresponding to P. For other planes, the support of P will be as specified in the constructor of the plane (often a subset of the integers). For example:

```
> Support(A3);
{@
    (0 0),
    (1 0),
    (2 0),
    (0 1),
    (1 1),
    (2 1),
    (0 2),
    (1 2),
    (2 2)
@}
> Universe($1);
Full Vector space of degree 2 over GF(3)

> P3, VP3, LP3 := ProjectivePlane(3);
> Support(P3);
{@
    (1 0 0),
    (0 1 0),
```

```
    (0 0 1),
    (1 2 0),
    (0 1 2),
    (1 2 1),
    (1 1 1),
    (1 1 2),
    (1 0 1),
    (1 1 0),
    (0 1 1),
    (1 2 2),
    (1 0 2)
@}
> Universe($1);
Full Vector space of degree 3 over GF(3)

> Support(fp1);
{@ 1, 2, 3, 4, 5, 6, 7 @}
> Universe($1);
Integer Ring
```

The i^{th} point and the i^{th} line of P are returned respectively by $V.i$ and $L.i$, where V is the point-set of P and L is the line-set. (The same results may be obtained by indexing the return values of **Points**(P) and **Lines**(P).) A further way to construct a point is to coerce an element x of the support into V, using the expression $V!x$. For example:

```
> VA3.6;
( 2, 1 )
> LA3.7;
< 1 : 0 : 2 >

> PointSet(fp1) ! 6;
6
> Category($1);
PlanePt
```

The following example illustrates some of the point and line functions, by converting the classically-defined affine plane $\mathrm{AG}_2(3)$ into a general affine plane:

```
> A3 := AffinePlane(3);
> A3;
Affine Plane AG(2, 3)
> pts := Points(A3);
> lns := [ {Index(pts, p) : p in Set(ll)} :
```

```
>     ll in Lines(A3) ];
> myA3 := AffinePlane< #pts | lns >;
> myA3 : Maximal;
Affine Plane of order 3
Points: {@ 1, 2, 3, 4, 5, 6, 7, 8, 9 @}
Lines:
    {1, 4, 7},
    {1, 6, 8},
    {1, 5, 9},
    {3, 6, 9},
    {3, 5, 7},
    {3, 4, 8},
    {2, 5, 8},
    {2, 4, 9},
    {2, 6, 7},
    {1, 2, 3},
    {7, 8, 9},
    {4, 5, 6}
> A3 eq myA3;
false
> IsIsomorphic(A3, myA3);
true
Mapping from: PlaneAff: A3 to PlaneAff: myA3
```

The functions relating to concepts of collinearity, concurrency, and parallelism are given in Table 39.2 (p. 789).

Note the distinction between **Set**(l) and **Support**(l), where l is a line of P. **Set**(l) returns the points of l *as points*, whereas **Support**(l) returns them in their original category. For example:

```
> l := Rep(LA3); l;
< 1 : 0 : 0 >
> Set(l), Support(l);
{ ( 0, 0 ), ( 0, 1 ), ( 0, 2 ) }
{
    (0 0),
    (0 2),
    (0 1)
}

> l := Rep(LP3); l;
< 1 : 0 : 0 >
> Set(l), Support(l);
{ ( 0 : 1 : 0 ), ( 0 : 0 : 1 ),
```

Table 39.2. Concurrency, collinearity, and parallel classes

MAGMA	Meaning
Rep(l)	Representative point from line l
Random(l)	Random point from line l
p **in** l, p **notin** l	**true** if point p is [not] on line l
S **subset** l, S **notsubset** l	**true** if set S of points is [not] on line l
l **meet** m	Unique point common to lines l and m
IsCollinear(P, S)	**true** if points of P in set S are collinear; if **true**, also returns the line which they define
IsConcurrent(P, S)	**true** if lines of P in set S are concurrent; if **true**, also returns the common point
Pencil(P, p)	Set of all lines through point p in plane P
Set(l)	Set of all points on line l
Support(l)	Set of underlying points on line l, in their original category
ContainsQuadrangle(P, S)	**true** if set S of points of P contains a quadrangle
IsParallel(P, l, m)	**true** if lines l and m of affine plane P are parallel
ParallelClass(P, l)	Set of all lines parallel to line l of affine plane P
ParallelClasses(P)	Set of classes of parallel lines partitioning affine plane P

```
( 0 : 1 : 2 ), (0 : 1 : 1 ) }
{
    (0 1 1),
    (0 1 2),
    (0 1 0),
    (0 0 1)
}

> l := Rep(LineSet(fp1)); l;
{1, 2, 3}

> Set(l), Universe(Set(l));
{ 1, 2, 3 }
Point-set of Projective Plane of order 2

> Support(l), Universe(Support(l));
{ 1, 2, 3 }
Integer Ring
```

39.3.2 Operations for Classical Planes

Table 39.3. Functions for classical planes over finite fields

MAGMA	Meaning
VectorSpace(P)	Vector space underlying classical plane P
Field(P)	(Finite) field K over which P is defined
$V\,!\,x$	Point corresponding to element x of vector space used to create P, where V is point-set of P
$V\,!\,[a,b,c]$	Given point-set V of $P = \mathrm{PG}_2(K)$, and sequence $[a,b,c]$ of elements of K, return point $(a:b:c)$ in P
$V\,!\,[a,b]$	Given point-set V of $P = \mathrm{AG}_2(K)$, and sequence $[a,b]$ of elements of K, return point (a,b) in P
Eltseq(p)	Coordinates of point p as sequence of elements of K ($[a,b,c]$ for $\mathrm{PG}_2(K)$; $[a,b]$ for $\mathrm{AG}_2(K)$)
Coordinates(P,p)	Coordinates of point p in plane P as sequence of elements of K ($[a,b,c]$ for $\mathrm{PG}_2(K)$; $[a,b]$ for $\mathrm{AG}_2(K)$)
$p[i]$	i^{th} component of point p, where p is represented as $(a:b:c)$ in $\mathrm{PG}_2(K)$ or (a,b) in $\mathrm{AG}_2(K)$
$L\,!\,[a,b,c]$	Given line-set L of classical plane P defined over K, and sequence $[a,b,c]$ of elements of K, return line $\langle a:b:c\rangle$ in P (i.e., line given by equation $ax+by+cz=0$ for $\mathrm{PG}_2(K)$, or $ax+by+c=0$ for $\mathrm{AG}_2(K)$)
Eltseq(l)	Coordinates of line l as sequence $[a,b,c]$ of elements of K
Coordinates(P,l)	Coordinates of line l in plane P as sequence $[a,b,c]$ of elements of K
$l[i]$	i^{th} component of line l, where l is represented as $\langle a:b:c\rangle$
$L\,!\,[m,b]$	Given line-set L of $P = \mathrm{AG}_2(K)$, and sequence $[m,b]$ of elements of K, return line in P given by equation $y = mx+b$
Slope(l)	Slope of line l in $\mathrm{AG}_2(K)$

Table 39.3 (p. 790) lists additional functions for classical planes. Most of them relate to the coordinates of a point or line, and their relationship to the vector space from which the plane P is constructed.

In order to use these operations correctly in the projective case, it is essential to note that if $P = \mathrm{PG}_2(q)$, then there are $(q-1)$ non-zero vectors corresponding to each point of P, since two vectors u and v are considered equivalent if there exists $a \in \mathrm{GF}(q) \setminus \{0\}$ such that $u = av$. Therefore the coordinates of points of P are always stored and printed in normalized form, such that the leftmost non-zero coordinate is 1. For example:

```
> F<w> := GF(9);
> P9, VP9, LP9 := ProjectivePlane(F);
```

```
> vecspP9 := VectorSpace(P9);
> pt := Random(VP9); pt;
( 1 : w^6 : w^2 )
> { v : v in vecspP9 |
>     not IsZero(v) and VP9!v eq pt };
{
    (w^6   2   1),
    (  1 w^6 w^2),
    (w^2   1   2),
    (w^5 w^3 w^7),
    (w^7 w^5   w),
    (  w w^7 w^3),
    (w^3   w w^5),
    (  2 w^2 w^6)
}
> VP9 ! [w^5, w^3, w^7];
( 1 : w^6 : w^2 )
> Eltseq(VP9 ! [w^5, w^3, w^7]);
[ 1, w^6, w^2 ]
```

However, this issue does not arise for classical affine planes, since the correspondence between vectors and points is one-one, with the obvious mapping of coordinates:

```
> A9, VA9, LA9 := AffinePlane(F);
> vecspA9 := VectorSpace(A9);
> #vecspA9 eq #VA9;
true
> VA9 ! [2, w];
( 2, w )
```

39.4 Properties of a Plane

Table 39.4 (p. 792) lists the functions returning numerical invariants and other properties of a plane P.

The order of a projective plane P is the integer n such that P has n^2+n+1 points and n^2+n+1 lines, and there are $n+1$ points on each line and $n+1$ lines through each point. The order of an affine plane P is the integer n such that A has n^2 points and n^2+n lines, there are n points on each line and $n+1$ lines through each point, and each of the $(n+1)$ parallel classes of P has n lines. For classical finite planes, the order n is equal to the cardinality q of the field with which they were created.

Table 39.4. Properties of a plane

MAGMA	Meaning
Order(P)	Order n of plane P
pRank(P, p)	p-rank of plane P
pRank(P)	p-rank of plane P of order p^t
IsDesarguesian(P)	**true** if plane P is Desarguesian (always **true** for classical planes)
IsSelfDual(P)	**true** if projective plane P is isomorphic to its dual

The function **IsDesarguesian**(P) tests whether the plane P is Desarguesian. This is always the case for classical planes, but may also hold for a plane defined in some other way. For example:

```
> IsDesarguesian(fp1);
true
```

39.5 Subplanes

Given a projective plane P and a set T of points of P such that T contains a quadrangle (i.e., there exist four points of T no three of which are collinear), it is possible to extend T to a set T' in such a way that the points of T', together with the lines of P which contain at least two points from T', form the smallest subplane of P containing the points in T.

Such a subplane can be created in MAGMA using a constructor of the following form:

sub< P | points specification **>**

The points are specified by a comma-separated list of points, sets of points, or sequences of points.

This also applies to the case where P is an affine plane, but here the restriction on T is slightly looser: the set T must includes three non-collinear points.

As an example of the use of the **sub** constructor, consider the plane $P = \mathrm{PG}_2(4)$. The points $(1:0:0), (0:1:0), (0:0:1)$ and $(1:w:1)$, where w is a primitive element of $\mathrm{GF}(4)$, form a quadrangle, and thus can be used to create a subplane of P.

```
> K<w> := GF(4);
```

```
> P, V, L := ProjectivePlane(K);
> S := sub< P | [ V | [1, 0, 0], [0, 1, 0],
>                     [0, 0, 1], [1, w, 1] ] >;
> S: Maximal;
Projective Plane of order 2
Points: {@ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ),
           ( 1 : w : 0 ), ( 1 : 0 : 1 ), ( 1 : w : 1 ),
           ( 0 : 1 : w^2 ) @}
Lines:
    {( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 0 : 1 : w^2 )},
    {( 1 : 0 : 0 ), ( 0 : 0 : 1 ), ( 1 : 0 : 1 )},
    {( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 1 : w : 0 )},
    {( 1 : 0 : 0 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 )},
    {( 0 : 1 : 0 ), ( 1 : 0 : 1 ), ( 1 : w : 1 )},
    {( 0 : 0 : 1 ), ( 1 : w : 0 ), ( 1 : w : 1 )},
    {( 1 : w : 0 ), ( 1 : 0 : 1 ), ( 0 : 1 : w^2 )}
```

For a classical plane P defined over a field K, a subplane can be obtained by taking only those points of P which have all coordinates lying in a subfield F of K.

The function **SubfieldSubplane** provides a simple way of creating this type of subplane in MAGMA. The affine plane $\mathrm{AG}_2(4)$ is used as an example.

```
> A := AffinePlane(4);
> SA := SubfieldSubplane(A, GF(2));
> SA;
Affine Plane AG(2, 2)
> SA subset A;
true
```

The points of a subplane can be coerced into the point-set of the larger plane, and this coercion is automatic for most functions. For example:

```
> SV := PointSet(S);
> p := Random(SV);
> p;
( 0 : 1 : w^2 )
> Parent(p);
Point-set of Projective Plane of order 2
> V!p;
( 0 : 1 : w^2 )
> Parent(V!p);
Point-set of Projective Plane PG(2, 4)
> Pencil(S, p);
```

```
{
    {( 0 : 1 : 0 ), ( 0 : 0 : 1 ), ( 0 : 1 : w^2 )},
    {( 1 : 0 : 0 ), ( 1 : w : 1 ), ( 0 : 1 : w^2 )},
    {( 1 : w : 0 ), ( 1 : 0 : 1 ), ( 0 : 1 : w^2 )}
}
> Pencil(P, p);
{
    < 1 : 0 : 0 >,
    < 0 : 1 : w >,
    < 1 : w^2 : 1 >,
    < 1 : w : w^2 >,
    < 1 : 1 : w >
}
> Universe(Pencil(P, p)) eq L;
true
```

39.6 Constructing New Planes from Existing Planes

Table 39.5. Embedding of affine planes in projective planes

MAGMA	Meaning
Dual(P)	Returns (i) dual of projective plane P (ii) point-set (iii) line-set
AffinePlane(P, l)	Returns (i) affine plane A obtained by removing line l from projective plane P (ii) point-set (iii) line-set (iv) embedding map $A \to P$
ProjectiveEmbedding(A)	Returns (i) projective completion P of affine plane A (ii) point-set (iii) line-set (iv) embedding map $A \to P$

Table 39.5 (p. 794) lists operations that construct a plane from a given plane. One of them is **Dual**(P), available for projective planes only; the other two involve the natural embedding of affine planes in projective planes. Each of these functions returns the new plane as its principal return value, followed by the point-set and the line-set. The two embedding functions also have a fourth return value, the embedding map from the affine plane to the projective plane.

For example:

```
> F<w> := GF(9);
```

```
> P9, VP9, LP9 := ProjectivePlane(F);
> lin := LP9 ! [1, w^3, w^6]; lin;
< 1 : w^3 : w^6 >
> AP9, VAP9, LAP9, m1 := AffinePlane(P9, lin);
> AP9;
Affine Plane of order 9
> m1;
Mapping from: PlaneAff: AP9 to PlaneProj: P9

> pt := Random(VAP9); pt;
( 1 : w^3 : 2 )
> m1(pt);
( 1 : w^3 : 2 )
> ln := Random(LAP9); ln;
{( 1 : w : w ), ( 1 : 1 : w^2 ), ( 1 : 2 : 1 ),
( 1 : w^6 : w^7 ), ( 1 : 0 : w^3 ), ( 1 : w^5 : 2 ),
( 0 : 1 : w^5 ), ( 1 : w^3 : w^6 ), ( 1 : w^2 : 0 )}
> m1(ln);
< 1 : w^2 : w >

> PAP9, VPAP9, LPAP9, m2 := ProjectiveEmbedding(AP9);
> IsIsomorphic(PAP9, P9);
true
Mapping from: PlaneProj: PAP9 to PlaneProj: P9
> m2(pt);
81
> m2(ln);
{5, 6, 38, 39, 48, 55, 63, 66, 79, 90}
```

39.7 Arcs, Conics and Unitals

A *k-arc* of a projective or affine plane P is a set of k points of P, no three of which are collinear. It is said to be *complete* if it cannot be extended to a $(k+1)$-arc by adding another point. In relation to the intersection of a line with an arc A, there are three special kinds of lines: a *tangent* to A meets it exactly once; a *secant* to A meets it exactly twice; and a *passant* or *external line* never meets A. The MAGMA functions associated with these concepts are shown in Table 39.6 (p. 796). For example:

```
> myA3, V, L := AffinePlane< 9 |
>     {1, 4, 7}, {1, 6, 8}, {1, 5, 9}, {3, 6, 9},
>     {3, 5, 7}, {3, 4, 8}, {2, 5, 8}, {2, 4, 9},
>     {2, 6, 7}, {1, 2, 3}, {7, 8, 9}, {4, 5, 6} >;
```

Table 39.6. Operations involving arcs in planes

MAGMA	Meaning
kArc(P, k)	A k-arc for the plane P
CompleteKArc(P, k)	A complete k-arc for plane P (if it exists)
IsArc(P, S)	**true** if set S of points is an arc in plane P
IsComplete(P, A)	**true** if the k-arc A is complete in plane P
Tangent(P, C, p)	Tangent to conic C at point p in plane P
AllTangents(P, A)	Set of all tangents to arc A in plane P
AllSecants(P, A)	Set of all secants to arc A in plane P
ExternalLines(P, A), **AllPassants**(P, A)	Set of all passants to arc A in plane P

```
> arc4 := kArc(myA3, 4);
> arc4;
{ 1, 2, 4, 5 }
> Tangent(myA3, arc4, V.2);
{2, 6, 7}
> AllSecants(myA3, arc4);
{
    {1, 4, 7},
    {1, 5, 9},
    {2, 5, 8},
    {2, 4, 9},
    {1, 2, 3},
    {4, 5, 6}
}
```

In the case of a classical projective plane P over a finite field F (created as such in MAGMA), there are some further functions provided. Table 39.7 (p. 797) lists them. A *conic*, defined by a *quadratic form* over F, may be generated from 5 points in general position (i.e., a 5-arc), provided that the order n of P is greater than 3. If the cardinality of F is a square q^2, then a *unital* U of P is a set of $q^3 + 1$ points having the property that every line meeting two points in U meets exactly $q + 1$ points in U.

For example, consider the projective plane P of order q^2, where $q = 4$. The following lines demonstrate how to construct a conic in P, and the corresponding quadratic form:

```
> q := 4;
> F<w> := GF(q ^ 2);
> P, V, L := ProjectivePlane(F);
> ka := kArc(P, 5);
```

Table 39.7. Conics and unitals in classical planes

MAGMA	Meaning
Conic(P, S)	Unique conic through 5-arc S of a classical projective plane P of order $n > 3$
QuadraticForm(P, S)	Quadratic form defining conic through 5-arc S of a classical projective plane P of order $n > 3$
Knot(P, C)	Given conic C in a projective plane P over GF(q) where q is even, return intersection point of tangents to C in P
Exterior(P, C)	Given conic C in a projective plane P over GF(q) where q is odd, return points that lie on two tangents to C in P
Interior(P, C)	Given conic C in a projective plane P over GF(q) where q is odd, return points that do not lie on any tangents to C in P
IsUnital(P, U)	Given set U of points in a plane P over GF(q^2), return **true** if U is a unital in P
AllTangents(P, U)	Set of tangents to the points of unital U in plane P
UnitalFeet(P, U, p)	Set of points of unital U whose tangents (w.r.t. U) pass through point p in plane P

```
> ka;
{ ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ),
( 1 : w^12 : w^13 ), ( 1 : w : w^9 ) }
> c := Conic(P, ka);
> c;
{  ( 1 : 0 : 0 ), ( 0 : 1 : 0 ), ( 0 : 0 : 1 ),
( 1 : w^12 : w^13 ), ( 1 : w : w^9 ), ( 1 : w^7 : w^10 ),
( 1 : w^4 : 1 ), ( 1 : w^13 : w^4 ), ( 1 : w^6 : w^5 ),
( 1 : w^10 : w^2 ), ( 1 : w^3 : w ), ( 1 : w^5 : w^7 ),
( 1 : w^11 : w^11 ), ( 1 : w^8 : w^3 ), ( 1 : w^2 : w^6 ),
( 1 : w^9 : w^14 ), ( 1 : w^14 : w^8 ) }

> qf<x,y,z> := QuadraticForm(P, ka);
> qf;
w^12*x*y + x*z + y*z
> forall{ pt : pt in c |
>     IsZero(Evaluate(qf, Eltseq(pt)))};
true
```

The set of points in P satisfying the equation $x^{q+1} + y^{q+1} + z^{q+1} = 0$ forms a hermitian unital:

```
> hu := { pt : pt in V |
>     IsZero(pt[1]^(q+1) + pt[2]^(q+1) + pt[3]^(q+1)) };
> #hu eq q^3 + 1;
true
> IsUnital(P, hu);
true
```

Furthermore, a Steiner 2-design D may be constructed from hu by taking the points of D to be the elements of hu and the blocks of D to be those intersections of lines of P with hu which have cardinality $q + 1$:

```
> blks := [blk : lin in L |
>     #blk eq (q+1) where blk is lin meet hu];
> D := Design< 2, SetToIndexedSet(hu) | blks >;
> D;
2-(65, 5, 1) Design with 208 blocks
> IsSteiner(D, 2);
true
```

39.8 Group Actions and Collineation Groups

The facilities for group action and automorphism groups of planes in MAGMA include those facilities provided for all categories of incidence structures, as described in Section 36.3. The only distinctive aspect of these operations as they apply to planes is one of terminology: it is more common to refer to the *collineation group* of a plane than to the automorphism group in its action on the points, so MAGMA understands **CollineationGroup**(P) to mean the same as **AutomorphismGroup**(P).

Table 39.8 (p. 799) lists the collineation group operations for planes, including some special operations. For example:

```
> F<w> := GF(9);
> P9, VP9, LP9 := ProjectivePlane(F);
> G := CollineationGroup(P9);
> FactoredOrder(G);
[ <2, 8>, <3, 6>, <5, 1>, <7, 1>, <13, 1> ]

> lin := LP9 ! [1, w^3, w^6];
> GG := CentralCollineationGroup(lin);
> Order(GG);
648
```

Table 39.8. Collineations of planes

MAGMA	Meaning
CollineationGroup(P), **PointGroup**(P), **AutomorphismGroup**(P)	Collineation group of plane P in its action on the points
LineGroup(P)	Collineation group of P in its action on the lines
IsPointTransitive(P), **IsTransitive**(P)	**true** if collineation group of P acts transitively on points
IsLineTransitive(P)	**true** if collineation group of P acts transitively on lines
CentralCollineationGroup(P, p, l)	Given point p and line l of projective plane P, return group of (p, l)-central collineations of P
CentralCollineationGroup(P, p)	Given point p of projective plane P, return group of central collineations with centre p
CentralCollineationGroup(P, l)	Given line l of projective plane P, return group of central collineations with axis l
IsCentralCollineation(P, g)	**true** if collineation g of projective plane P is a central collineation; if **true**, also returns the centre and axis of g
CollineationGroupStabilizer(P, k)	Subgroup of collineation group of P fixing the first k base points
CollineationSubgroup(P)	Subgroup of collineation group of P generated by one element

All the collineation functions that return groups, except **LineGroup**, are able to return five values: the group G itself; a G-set giving the action of G on the points; a G-set giving the action of G on the lines; the parent M for all automorphisms of the plane when represented as maps; and a transfer map $t : G \to M$. **LineGroup** returns only the line group and the G-set of the lines.

39.9 Relationship to Designs, Graphs and Codes

Table 39.9 (p. 800) lists the functions for transferring between a graph and a plane, and the functions for converting a plane into a design or code. In addition, it will be recalled from Section 38.2 that a set of words from a linear code may be used to generate a plane.

For example, consider the projective plane $P9$ over GF(9). Given a field $K = \mathrm{GF}(p^t)$, a linear code over K may be constructed from $P9$. The dimension of this code is the p-rank of the plane:

Table 39.9. Planes, designs, graphs and codes

MAGMA	Meaning
Design(P)	Design corresponding to points and lines of plane P
PointGraph(P)	Point graph G of plane P, where vertices of G are points of P, and vertices $u, v \in G$ are adjacent if both u and v are in a line of P
LineGraph(P)	Line graph of plane P
IncidenceGraph(P)	Incidence graph (bipartite) of plane P, with vertex set the union of the point set and line set of P, and with point p being adjacent to line l whenever $p \in l$
LinearCode(P, K)	Given plane P with v points, and finite field K, return linear code of length v generated by characteristic functions of lines of P regarded as vectors of $K^{(v)}$

```
> P9 := ProjectivePlane(9);
> C := LinearCode(P9, GF(2));
> C : Minimal;
[91, 90] Linear Code over GF(2)
> pRank(P9, 2) eq Dimension(C);
true

> C := LinearCode(P9, GF(3));
> C : Minimal;
[91, 37] Linear Code over GF(3)
> pRank(P9, 3);
37

> C := LinearCode(P9, GF(9));
> MinimumWeight(C);
10
> #C;
202755595904452569706561330872953769
> C : Minimal;
[91, 37, 10] Cyclic Code over GF(3^2)
> pRank(P9); // same as pRank(P9, 3)
37
```

40. Error-Correcting Codes

The three main families of error-correcting codes supported in MAGMA are:

– Linear codes over finite fields;

– General (non-linear) codes over finite fields;

– Linear codes over finite euclidean rings, such as $\mathbb{Z}/n\mathbb{Z}$.

All of these are *block codes*, that is, codes in which each codeword has the same length. In this book, emphasis is given to linear codes over finite fields, so the reader should consult the *Handbook* for an account of non-linear codes and codes over rings.

The design of the MAGMA machinery for codes stresses the algebraic basis of coding theory. The facilities may be broadly grouped as follows: constructions; elementary operations; weight distribution properties; isomorphism/automorphism questions (monomial action and permutation action); and decoding algorithms. In general, for matters of terminology and notation, MAGMA follows MacWilliams and Sloane [MaS77].

Let R be a finite ring, and let $R^{(n)}$ be the free module of rank n over R. The *Hamming distance* $d(u, v)$ between two vectors u and v of $R^{(n)}$ is defined as the number of coordinate positions in which they differ. An (n, M, d) *code* C *over* R is a set of M vectors from $R^{(n)}$ such that the distance between any two distinct members of C is at least d. An $[n, k, d]$ *linear code* C *over* R is a k-dimensional subspace of $R^{(n)}$ such that the distance between any two distinct members of C is at least d. *In this chapter,* R *will be a finite field* $K = \mathrm{GF}(q)$*, unless stated otherwise.*

40.1 Defining a Linear Code

In MAGMA, linear codes belong to the category **Code**. They are regarded as subspaces of $K^{(n)}$, with additional operations.

40.1.1 Defining a Code from a Vector Space

A linear code may be created by defining the appropriate subspace V of $K^{(n)}$ and applying the transfer function **LinearCode**(V). For example, consider the length-6 code C over $K = \mathrm{GF}(3)$ generated by the vectors 201201, 111212, and 120210. It may be created in MAGMA as follows:

```
> K := GF(3);
> vsp := sub< VectorSpace(K, 6) |
>   [2,0,1,2,0,1], [1,1,1,2,1,2], [1,2,0,2,1,0] >;
> print vsp;
Vector space of degree 6, dimension 3 over GF(3)
Generators:
(2 0 1 2 0 1)
(1 1 1 2 1 2)
(1 2 0 2 1 0)
Echelonized basis:
(1 0 2 0 2 0)
(0 1 2 0 0 1)
(0 0 0 1 1 2)

> C := LinearCode(vsp);
> print C;
[6, 3, 3] Linear Code over GF(3)
Generator matrix:
[1 0 2 0 2 0]
[0 1 2 0 0 1]
[0 0 0 1 1 2]
```

Notice from the output that the generator matrix of C is the (echelonized) basis matrix of the vector space *vsp*. C is a $[6, 3, 3]$ linear code, meaning that the number of components in each codeword is 6, the dimension of the vector space corresponding to the code is 3, and the minimum weight of the codewords is 3. In general, a linear code over a finite field may be described as an $[n, k, d]$ code, where n is the block length, k is the dimension, and d is the minimum weight.

It is possible to circumvent the direct construction of the vector space by means of the **LinearCode** constructor:

LinearCode< K, n | generator specification **>**

It constructs the linear code generated by the vectors in $K^{(n)}$ specified on the right side of the constructor; the generators may be given in any of the forms

suitable for a **sub** constructor as applied to a vector space. The example below shows how to create the code C using this constructor:

```
> C2 := LinearCode< K, 6 |
>   [2,0,1,2,0,1], [1,1,1,2,1,2], [1,2,0,2,1,0] >;
> print C eq C2;
true
```

40.1.2 Defining a Code from a Generator Matrix

Another way to create a linear code in MAGMA is to specify its generator matrix. If A is a rectangular matrix over a finite field K, then **LinearCode**(A) returns the linear code generated by the rows of A (echelonized if necessary). For example, the code C described above could also be defined by creating a matrix whose rows are the generator vectors, and applying **LinearCode** to this matrix:

```
> M := KMatrixSpace(K, 3, 6);
> print M;
Full KMatrixSpace of 3 by 6 matrices over GF(3)
> genmat := M ! [2,0,1,2,0,1, 1,1,1,2,1,2, 1,2,0,2,1,0];

> C3 := LinearCode(genmat);
> print C eq C3;
true
```

40.1.3 Defining a Code from a Geometrical Structure

Finally, linear codes may also be created from planes and incidence structures. Suppose that K is a finite field, and P is a plane with v points. Then **LinearCode**(P, K) returns the linear code of length v generated by the characteristic functions of the lines of P, considered as vectors of the K-space $K^{(v)}$. Similarly, if I is an incidence structure with v points, then **LinearCode**(I, K) returns the linear code of length v generated by the characteristic functions of the blocks of I, considered as vectors of $K^{(v)}$. (Equivalently, the code may be seen as being generated by the rows of the incidence matrix of the original structure.)

For example, consider the binary code given by the projective plane $\mathrm{PG}(2, 8)$:

```
> P := ProjectivePlane(GF(8));
> C := LinearCode(P, GF(2));
```

```
> mw := MinimumWeight(C);
> print C : Minimal;
[73, 28, 9] Cyclic Code over GF(2)
> print pRank(P) eq Dimension(C);
true
> print Binomial(2+2-1, 2) ^ 3 + 1;
28
```

Note that the p-rank of P (which equals 28, the dimension of C) agrees with the theorem of Graham and MacWilliams stating that the p-rank of $\mathrm{PG}(m, q)$ is $\binom{m+p-1}{m}^t + 1$, where $q = p^t$.

40.2 Calculations with Codewords and Vectors

Because of the close relationship between linear codes and vector spaces, codewords in MAGMA are implemented as ordinary vectors. Given a code C of length n over K, a codeword of C may be created by coercing a length-n sequence (of integers or elements of K) into C. However, the parent of the result will not be C, but $K^{(n)}$. *The parent magma of a codeword is not the linear code, but the corresponding generic vector space.* For instance, the codeword 201122 of C may be assigned to c as follows:

```
> c := C ! [2, 0, 1, 1, 2, 2];
> print c;
(2 0 1 1 2 2)

> print Category(c);
ModTupFldElt
> print Parent(c);
Full Vector space of degree 6 over GF(3)
```

An individual component of a codeword may be altered using index notation on the left side of an assignment statement; this operation imitates the effect of sending the codeword along a noisy communications channel. For instance, the fourth component of the codeword c in C can be changed to 2 as follows:

```
> c[4] := 2; print c;
(2 0 1 2 2 2)
```

Although linear codes do not act as parent magmas, membership and subset testing is still provided, by means of the operators **in** and **subset**. For example:

```
> print c in C;
false
```

Coercion of a vector (or a sequence representing a vector) into a code is only allowed if the vector is a member of the code. If the vector is not a codeword, an error message will be given:

```
> print C ! c;

>> print C ! c;
          ^
Runtime error in '!': Result is not in the given structure
```

Table 40.1. Coding-theory operations on vectors

MAGMA	Meaning
InnerProduct(u, v)	Inner product of vectors u and v
Distance(u, v)	Hamming distance between u and v
Weight(u)	Hamming weight of u (number of non-zero entries)
Syndrome(u, C)	Syndrome of u relative to code C
Support(u)	Set of coordinates of u with non-zero entries
Coordinates(C, u)	Coordinates of u relative to basis of C (i.e., coordinates of vector v in information space such that $vG = u$, where G is generator matrix of C)

Table 40.1 lists the functions on vectors that are designed for use in the context of linear codes. See also Section 27.4, for the general-purpose operations on vectors.

The function **Syndrome**(x, C) provides a way to calculate the syndrome of a vector x relative to a linear code C without explicitly having to calculate the parity check matrix of C first. For instance, the fact that the syndrome of c relative to C is non-zero is another demonstration that the new version of c is not in the code C:

```
> print Syndrome(c, C);
(2 2 2)
```

The function **Support**(v) returns the set of coordinate positions for which a vector v has non-zero components. It is useful for such tasks as building designs from codes. For instance, the support of c shows that only its second entry is zero:

```
> print Support(c);
{ 1, 3, 4, 5, 6 }
```

40.3 General Facts about a Linear Code

Table 40.2. General facts about a linear code

MAGMA	Meaning
Alphabet(C),**Field**(C)	Underlying field K for code C
Length(C)	Block length n of $[n, k, d]$ linear code C
Dimension(C)	Dimension k of C
MinimumWeight(C), **MinimumDistance**(C)	Minimum weight d of words in C
StandardForm(C)	Returns (i) a code S equivalent to C but in standard form, (i.e., formed from C with the columns permuted so that the information columns are columns $1, \ldots, k$) (ii) isomorphism $f : C \to S$. There are many such codes, but the same one is returned each time.
GeneratorMatrix(C)	Generator matrix for C
ParityCheckMatrix(C)	Parity check matrix for C
InformationSpace(C)	k-dimensional vector space of information vectors for C
InformationSet(C)	Length-k sequence giving the numbers of the information columns of C (i.e., k linearly independent columns of the generator matrix G, such that G is the identity matrix when restricted to these columns)
AllInformationSets(C)	All possible information sets of C, as a sorted sequence whose terms are length-k sequences of indices of linearly independent columns
SyndromeSpace(C)	$(n - k)$-dimensional vector space of syndrome vectors for C
VectorSpace(C)	Vector subspace whose basis corresponds to the rows of the generator matrix for C
AmbientSpace(C)	Generic vector space $K^{(n)}$
Generic(C)	Generic $[n, n, 1]$ code containing C
Dual(C)	Dual code of C

Table 40.2 (p. 806) lists functions giving general information about a code while Table 40.3 (p. 807) lists tests for properties. In these tables,

Table 40.3. Properties of a linear code

MAGMA	Meaning
IsCyclic(C)	**true** if C is cyclic
IsMDS(C)	**true** if C is maximum-distance separable (i.e., is an $[n, k, n-k+1]$ code)
IsSelfOrthogonal(C), **IsSelfDual**(C)	**true** if C is self-orthogonal
IsWeaklySelfOrthogonal(C), **IsWeaklySelfDual**(C)	**true** if C is weakly self-orthogonal

it should be understood that C is an $[n, k, d]$ linear code over the finite field K. Most of these functions are fairly trivial, with the exception of **MinimumWeight**(C); the advanced facilities for weight enumeration are discussed later in the chapter. There are some additional functions available for cyclic codes, as listed in Table 40.6 (p. 811).

For example, suppose that the binary $[4, 2, 2]$ code generated by 1010 and 0111 is created in MAGMA with the following assignment statement:

```
> lc := LinearCode< GF(2), 4 | [1,0,1,0], [0,1,1,1] >;
```

While this one-statement code definition is attractively compact, it has the minor disadvantage that the algebraic structures auxiliary to *lc* are nameless. The functions listed in the table provide a means of accessing these structures. If, for instance, the user wishes to define the vector $v = 0001$, which is not in *lc* but is in the vector space of which *lc* is a subspace, then the **AmbientSpace**(C) should be used to obtain this vector space:

```
> v := AmbientSpace(lc) ! [0, 0, 0, 1];
```

The access operations are also useful when writing a function whose argument is an arbitrary code. For instance, according to the sphere-packing bound, any q-ary $(n, M, 2t+1)$ code satisfies

$$M\left\{\binom{n}{0}+\binom{n}{1}(q-1)+\cdots+\binom{n}{t}(q-1)^t\right\}\le q^n.$$

If equality is achieved then the code is called *perfect*. The following function *spherepacking* returns the right hand side of this inequality minus the left hand side, for a given code C. It employs the functions **MinimumWeight**, **Length**, and **Field**, as well as the operator #, which returns the cardinality of any given code:

To perform this example online, type `load "I96c40e1";`

```
> spherepacking := function(C)
>   q := #Field(C);
>   n := Length(C);
>   M := #C;
>   t := (MinimumWeight(C) - 1) div 2;
>   return q^n -
>     M*(&+[ Binomial(n, i)*(q-1)^i : i in [0..t] ]);
> end function;
```

For instance, the following applications of *spherepacking* to certain standard linear codes show that $\mathrm{Ham}(3, 2)$ is perfect (as are all the Hamming codes $\mathrm{Ham}(r, q)$), but that the extended binary Golay code is far from attaining the sphere-packing bound:

```
> print spherepacking(HammingCode(GF(2), 3));
0
> print spherepacking(GolayCode(GF(2), true));
7254016
```

The generators of a code C are stored in *echelonized form*, which may differ from the way in which they were given by the user. As usual, $C.i$ returns the i^{th} generator, and **Generators**(C) returns the generators in a set. Another function, **GeneratorMatrix**(C), returns a matrix whose rows are the generators. For example:

```
> g := GeneratorMatrix(lc);
> print g;
[1 0 1 0]
[0 1 1 1]
> h := ParityCheckMatrix(lc);
> print h;
[1 0 1 1]
[0 1 0 1]
> print g * Transpose(h);
[0 0]
[0 0]
```

40.4 Families of Linear Codes

Magma has functions which return many of the well-known linear codes: Hamming, Reed-Muller, Golay, Reed-Solomon, cyclic, alternant, and generalized BCH. For all of these well-known codes, it is much more efficient to

define them using their special functions rather than specifying their generators by hand.

40.4.1 Some Standard Codes

Table 40.4. Some basic linear codes

MAGMA	Meaning
HammingCode(K, r)	$[(q^r-1)/(q-1), (q^r-1)/(q-1)-r, 3]$ Hamming code over K
ReedMullerCode(r, m)	r^{th} order binary Reed-Muller code of length 2^m $(r \leq m)$
GolayCode$(K,$ **true**$)$	Extended Golay code; binary or ternary according as K is GF(2) or GF(3)
GolayCode$(K,$ **false**$)$	Unextended Golay code; binary or ternary according as K is GF(2) or GF(3)
ReedSolomonCode(K, d)	$[q-1, q-d, d]$ Reed-Solomon code over K
ReedSolomonCode(n, d)	As above, where $q = n+1$
ReedSolomonCode(K, d, b)	$[q-1, q-d, d]$ Reed-Solomon code over K, where the primitive element is first raised to the b^{th} power
ReedSolomonCode(n, d, b)	As above, where $q = n+1$
GRSCode(A, V, k)	$[n, k' \leq k]$ generalized Reed-Solomon code over K, where A is a sequence of n distinct elements α_i of K, V is a sequence of n non-zero elements v_i of K, and $k \geq 0$

Table 40.4 (p. 809) summarizes the functions which return some of the basic linear codes. In the table, q denotes the cardinality of the field K.

For example, the extended ternary Golay code is created by the assignment statement:

```
> gol3ext := GolayCode(GF(3), true);
> print gol3ext;
[12, 6, 6] Extended Golay Code over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 1 1 1 1 1]
[0 1 0 0 0 0 1 0 1 2 2 1]
[0 0 1 0 0 0 1 1 0 1 2 2]
[0 0 0 1 0 0 1 2 1 0 1 2]
[0 0 0 0 1 0 1 2 2 1 0 1]
[0 0 0 0 0 1 1 1 2 2 1 0]
```

40.4.2 Cyclic Codes

Table 40.5. Some cyclic codes

MAGMA	Meaning
CyclicCode(u)	$[n,k]$ cyclic code generated by the right cyclic shifts of vector $u \in K^{(n)}$
CyclicCode(n,f)	Length-n cyclic code generated by the right cyclic shifts of the vector corresponding to the polynomial $f \in K[x]$, where f has degree $n-k$ and divides x^n-1
CyclicCode(n,R,K)	Length-n cyclic code over K generated by the least-degree polynomial having the elements of R as roots, where R is a set or sequence of primitive n^{th} roots of unity in an extension of K
BCHCode(K,n,d,b)	q-ary $[n, k \geq n-m(d-1), D \geq d]$ BCH code of designated distance d having generator polynomial $g(x) = \text{lcm}\{m_1(x), \ldots, m_{d-1}(x)\}$, where $m_i(x)$ is the minimum polynomial of α^{b+i-1}, for $i = 1, \ldots, d-1$; here b is a positive integer, α is a primitive n^{th} root of unity in the degree-m extension of K (i.e., $\text{GF}(q^m)$), and n must satisfy $\gcd(n,q) = 1$
BCHCode(K,n,d)	As above but with $b = 1$
QRCode(K,n)	q-ary length-n quadratic residue code with generator polynomial $g_0(x) = \prod(x-\alpha^r)$, where α is a primitive n^{th} root of unity in an extension field L of K, and the product is taken over the quadratic residues in L; here n is an odd prime such that q is a quadratic residue modulo n

Table 40.5 (p. 810) lists the functions for the creation of cyclic codes. Once again, q denotes the cardinality of the field K. The special operations on cyclic codes are given in Table 40.6 (p. 811); they may be applied to any code which is cyclic, not only those generated from the functions in Table 40.5.

For example, the length-7 binary cyclic code generated by the vector 1101000 corresponding to the polynomial $1 + x + x^3$ can be generated in any of the following ways:

```
> P<x> := PolynomialRing(GF(2));
> cyc := CyclicCode(7, 1+x+x^3);
> print cyc;
[7, 4, 3] Cyclic Code over GF(2)
Generator matrix:
[1 0 0 0 1 1 0]
[0 1 0 0 0 1 1]
```

Table 40.6. Operations on cyclic codes

MAGMA	Meaning
GeneratorPolynomial(C)	A generator polynomial for cyclic code C
CheckPolynomial(C)	A check polynomial for cyclic code C
Idempotent(C)	Idempotent for cyclic code C

```
[0 0 1 0 1 1 1]
[0 0 0 1 1 0 1]

> v := VectorSpace(GF(2), 7) ! [1,1,0,1,0,0,0];
> cyc2 := CyclicCode(v);
> print cyc2 eq cyc;
true

> L<w> := GF(2, 3);
> cyc3 := CyclicCode(7, {w, w^2, w^4}, GF(2));
> print cyc3 eq cyc;
true
```

As another example, the length-15 binary BCH code with $b = 1$ which is 2-error-correcting and hence has designated distance 5 is

```
> BCH15 := BCHCode(GF(2), 15, 5);
> print BCH15;
[15, 7, 5] BCH code (d = 5, b = 1) over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 1 0 0 0 1 0 1 1]
[0 1 0 0 0 0 0 1 1 0 0 1 1 1 0]
[0 0 1 0 0 0 0 0 1 1 0 0 1 1 1]
[0 0 0 1 0 0 0 1 0 1 1 1 0 0 0]
[0 0 0 0 1 0 0 0 1 0 1 1 1 0 0]
[0 0 0 0 0 1 0 0 0 1 0 1 1 1 0]
[0 0 0 0 0 0 1 0 0 0 1 0 1 1 1]
```

40.4.3 Alternant and Generalized BCH Codes

MAGMA provides several functions for alternant and generalized BCH codes. Some of them are discussed below; in all these descriptions, $K = \mathrm{GF}(q)$, and $F = \mathrm{GF}(q^m)$ is a degree-m extension of K. For more advanced codes, see the *Handbook*.

Let A and Y be length-n sequences of elements of F, such that the terms of A are distinct and $0 \notin Y$. Then the function **AlternantCode**(A, Y, r) returns the q-ary $[n, k \geq n - mr, d \geq r + 1]$ alternant code $A(A, Y)$.

Let P and G be univariate polynomials over the splitting field of the polynomial $(x^n - 1)$ over K, such that $\deg(P) \leq n - 1$, $\deg(G) \leq n - 1$, and P and G are coprime to $(x^n - 1)$. The function **ChienChoyCode**(P, G, n, q) returns the Chien-Choy generalized BCH code of length n associated with P and G over GF(q), where $\gcd(n, q) = 1$.

Let $G = G(z)$ be a degree-r polynomial over F. Then **GoppaCode**(L, G) returns the q-ary $[n, k \geq n - mr, d \geq r + 1]$ Goppa code $\Gamma(L, G)$, where $L = [\alpha_1, \ldots, \alpha_n]$ is a sequence of n distinct elements of F such that $G(\alpha_i) \neq 0, \forall \alpha_i \in L$.

The function **SrivastavaCode**(A, W, mu, K) returns the Srivastava code of parameters A, W, mu over K, where mu is an integer and A and W are sequences of elements of F such that $0 \notin A$ and the terms of A and W combined are distinct. Similarly, **GeneralizedSrivastavaCode**(A, W, Z, t, K) returns the generalized Srivastava code of parameters A, W, Z, t over K, where $t > 0$ and A, W, Z are sequences of elements of F such that $0 \notin A$, $0 \notin Z$, and the terms of A and W combined are distinct.

The following examples illustrate the construction of a Goppa code and an alternant code:

```
> q := 2^5;
> K<w> := GF(q);
> Pq<z> := PolynomialRing(K);
> G := z^3 + z + 1;
> L := [w^i: i in [0..(q-2)]];
> print GoppaCode(L, G);
[31, 16, 7] Goppa code (r = 3) over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 1 1 0 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 1 1 1 0 0 1 0 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 1]
[0 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 1 1 0 1 0 1 0 1 0 1 1]
[0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 1 0 1 1 1 1 1]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 0 0 1 0 1 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 1 1 0 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 0 1 0 0 1 1]
[0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 0 1 0 0 0 0 0 1 0 1 0 1 0 1 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 1 0 0 0 1 0 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 1 1 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 1 0 1 0 0 1 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 0 0 1 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 1 0 1 1 1 1]
```

```
> K<u> := GF(4); F := ext< K | 3 >;
> n := 15;
> w := PrimitiveElement(F);
> A := [ w^i : i in [0..n-1] ];
> y := [ F!1 : i in [0..n-1] ];
> T := AlternantCode(A, y, 4, K);
> print T;
[15, 5, 7] Alternant code over GF(2^2)
Generator matrix:
[  1   0   0   0   0   u   1   u u^2   1 u^2   u u^2   0   0]
[  0   1   0   0   0   0   u   1   u u^2   1 u^2   u u^2   0]
[  0   0   1   0   0   0   0   u   1   u u^2   1 u^2   u u^2]
[  0   0   0   1   0 u^2   0   u   u   0 u^2 u^2 u^2   1   0]
[  0   0   0   0   1   0 u^2   0   u   u   0 u^2 u^2 u^2   1]
```

40.5 Constructing Codes from Other Codes

There are many ways of constructing a new code by modifying the words of an existing code in some way. One of these is to build a subcode, with the usual **sub** constructor. In addition, MAGMA has functions corresponding to the constructions specific to coding theory. Table 40.7 (p. 814) summarizes these functions; in the table, C should be understood to be an $[n, k, d]$ linear code.

For binary codes, one important characteristic of a codeword is its parity, that is, whether its weight is odd or even. Even-weight codewords are particularly important because any linear combination of such words is also even. Given a binary linear code C, the function **ExpurgateCode**(C) returns the code which contains only those codewords of C which are even. For example, in the code *h4exp* constructed below, every word is even.

```
> h4 := HammingCode(GF(2), 4);
> print h4;
[15, 11, 3] Hamming code (r = 4) over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 1 0 1]
[0 0 0 1 0 0 0 0 0 0 1 0 1 1 1]
[0 0 0 0 1 0 0 0 0 0 1 0 1 0 0]
[0 0 0 0 0 1 0 0 0 0 1 0 0 1 0]
[0 0 0 0 0 0 1 0 0 0 1 0 0 0 1]
[0 0 0 0 0 0 0 1 0 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 1 0 1 0 1 0 1]
[0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
```

Table 40.7. Constructing codes from existing codes

MAGMA	Meaning
AugmentCode(C)	Code consisting of the words of binary code C and the all-ones vector (if it is not already in C)
ExtendCode(C)	Code formed from binary code C by adding an overall parity check (i.e., adding a 0 at the end of every even-weight codeword, and a 1 at the end of every odd-weight codeword)
ExpurgateCode(C)	Code formed from binary code C by deleting all odd-weight codewords
LengthenCode(C)	Code formed from binary code C by augmenting then extending it
DirectSum(C, D)	Direct sum of codes C and D over same field, (i.e., all vectors $u\|v$, where $u \in C, v \in D$)
PlotkinSum(C, D)	Plotkin sum of codes C and D over same field, (i.e., all vectors $u\|u+v$, where $u \in C, v \in D$)
DirectProduct(C, D)	Code generated by Kronecker product of generator matrices of C and D
PunctureCode(C, i)	Given binary code C and $i \in \{1, \ldots, n\}$, forms code with i^{th} component of each codeword deleted
PunctureCode(C, S)	Given binary code C and set $S \subseteq \{1, \ldots, n\}$, forms code with codeword components indexed by S deleted
ShortenCode(C, i)	Given binary code C and $i \in \{1, \ldots, n\}$, forms code containing only those codewords with zero in i^{th} component and with this component deleted from them
ShortenCode(C, S)	Given binary code C and set $S \subseteq \{1, \ldots, n\}$, forms code containing only those codewords with zero in each component indexed by S and with these component deleted from them
SubfieldSubcode(C, S)	Given code C over K and subfield S of K, returns (i) code consisting of those words of C whose components are in S (ii) restriction map
SubfieldSubcode(C)	As above, where S is the prime subfield of K

```
[0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
> h4exp := ExpurgateCode(h4);
> print h4exp;
[15, 4, 4] Linear Code over GF(2)
Generator matrix:
[0 0 0 0 0 0 0 1 0 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 1 0 1 0 1 0 1]
[0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
[0 0 0 0 0 0 0 0 0 0 0 1 1 1 1]
```

Another way of constructing a code in which every word is even is to use all the words of the given code but to modify them so that they are even. The **ExtendCode**(C) function does this, by extending each word with a parity-check digit. If the minimum weight d is odd, the new code will be an $[n+1, k, d+1]$ code. For example:

```
> h4ext := ExtendCode(h4);
> print h4ext;
[16, 11, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1]
[0 0 0 1 0 0 0 0 0 0 1 0 1 1 1 1]
[0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 1]
[0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0]
[0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0]
[0 0 0 0 0 0 0 0 0 1 1 0 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0]
```

Note that the generator matrix for *h4ext* is the same as the one for *h4*, except that a column has been added on the right so as to give each generator codeword even parity.

If a binary linear code C does not contain the all-ones vector then the function **AugmentCode**(C) will return a larger linear code which contains C, by including the all-ones vector with the words of C. Every binary Hamming code contains the all-ones vector of the relevant length already, so this function would merely return *h4* itself if applied to *h4*, but the dual of *h4* will give a more fruitful result:

```
> dualh4 := Dual(h4);
> print dualh4;
[15, 4, 8] Linear Code over GF(2)
Generator matrix:
[1 0 1 0 1 1 0 0 1 1 0 0 1 1 0]
[0 1 1 0 0 1 1 0 0 1 1 1 1 0 0]
[0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> dualh4aug := AugmentCode(dualh4);
> print dualh4aug;
[15, 5, 7] Linear Code over GF(2)
Generator matrix:
```

```
[1 0 0 0 0 1 1 0 0 1 1 0 0 1 1]
[0 1 0 0 1 1 0 0 1 1 0 1 0 0 1]
[0 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> print dualh4 subset dualh4aug;
true
```

The last line above confirms that *dualh4* is a subcode of *dualh4aug*.

The **LengthenCode**(C) function does two of the above operations in one step: it first augments C with the all-ones vector and then extends the new code with a parity-check. This may be tested as follows:

```
> print LengthenCode(dualh4) eq ExtendCode(dualh4aug);
true
```

There are two functions which return a code which is the sum, either direct or Plotkin, of two codes C and D. A general codeword in **DirectSum**(C, D) has the form $u|v$, and a general codeword in **PlotkinSum**(C, D) has the form $u|u+v$, where $u \in C, v \in D$. For example, the construction of a first-order binary Reed-Muller code can be achieved with Plotkin sums. Since the binary repetition codes of lengths 4, 8, and 16 are required in the construction, it is useful to construct a function that returns the binary repetition code of given length n:

To perform this example online, type `load "I96c40e2";`

```
> repet := function(n)
>   allones := [ GF(2)!1 : i in [1..n] ];
>   return LinearCode(sub<VectorSpace(GF(2), n) | allones>);
> end function;
```

The Reed-Muller code is built by forming the Plotkin sum of the binary length-4 even-weight code with the binary length-4 repetition code (which, conveniently, are dual to one another), and then by Plotkin summing the result with the length-8 repetition code, and finally Plotkin summing this result with the length-16 repetition code:

```
> E := PlotkinSum(Dual(repet(4)), repet(4));
> F := PlotkinSum(E, repet(8));
> G := PlotkinSum(F, repet(16));
> print G eq ReedMullerCode(1, 5);
true
```

The output verifies that G is the first-order binary Reed-Muller code with $m = 5$.

There are also some functions which construct new codes according to properties of individual components of codewords. **PunctureCode**(C, i) deletes the i^{th} component of each codeword of C to form a new code. For instance, the extended binary Golay code, punctured in the ninth position, can be created as follows:

```
> punct9 := PunctureCode(GolayCode(GF(2), true), 9);
```

It can be proved that *punct9* is equivalent to the unextended binary Golay code, as is the extended binary Golay code punctured in any other position. However, if they are tested for equality, then the result is **false**, because MAGMA's operator **eq** tests genuine equality, which is a stronger relationship than equivalence. A test of equivalence could be performed with the function **IsIsomorphic**, explained in Section 40.7.2:

```
> gol2unext := GolayCode(GF(2), false);
> print punct9 eq gol2unext;
false

> print IsIsomorphic(punct9, gol2unext);
true
Mapping from: Code: punct9 to Code: gol2unext
```

The result indicates that the codes are isomorphic, and also returns the isomorphism, as a mapping.

The puncturing operation can also be performed on more than one coordinate, so there is an alternative form of the function, **PunctureCode**(C, S), where S is the set of component coordinates to be deleted.

The **ShortenCode**(C, i) function differs from **PunctureCode** in that only the codewords which have a zero in the i^{th} component are selected for the new code. Then these codewords have this component deleted. If a code is to be shortened in several components then the appropriate function is **ShortenCode**(C, S), where S is the set of components. For instance, Ham$(2, 11)$ may be shortened in the first and second positions:

```
> h11 := HammingCode(GF(11), 2);
> print h11;
[12, 10] Hamming code (r = 2) over GF(11)
Generator matrix:
[ 1  0  0  0  0  0  0  0  0  0  9  2]
[ 0  1  0  0  0  0  0  0  0  0  9  1]
```

```
[ 0  0  1  0  0  0  0  0  0  0  7  3]
[ 0  0  0  1  0  0  0  0  0  0  3  7]
[ 0  0  0  0  1  0  0  0  0  0  6  4]
[ 0  0  0  0  0  1  0  0  0  0  1  9]
[ 0  0  0  0  0  0  1  0  0  0  2  8]
[ 0  0  0  0  0  0  0  1  0  0  4  6]
[ 0  0  0  0  0  0  0  0  1  0  8  2]
[ 0  0  0  0  0  0  0  0  0  1  5  5]
> print ShortenCode(h11, {1, 2});
[10, 8] Linear Code over GF(11)
Generator matrix:
[ 1  0  0  0  0  0  0  0  7  3]
[ 0  1  0  0  0  0  0  0  3  7]
[ 0  0  1  0  0  0  0  0  6  4]
[ 0  0  0  1  0  0  0  0  1  9]
[ 0  0  0  0  1  0  0  0  2  8]
[ 0  0  0  0  0  1  0  0  4  6]
[ 0  0  0  0  0  0  1  0  8  2]
[ 0  0  0  0  0  0  0  1  5  5]
```

40.6 The Weight Distribution

The most important invariant of a code C is its *minimum distance*. In a linear code, this corresponds to the *minimum weight* of the codewords of C, where the *(Hamming) weight* of a codeword is defined as the number of non-zero components. Table 40.8 (p. 819) lists some of MAGMA's functions connected with minimum weight and weight distribution in a code. It should be noted that the determination of the minimum weight is, in general, a very expensive computation, although it can be performed without enumerating the words of the code. However, a more efficient algorithm exists for cyclic codes, so in that case significantly larger codes can be handled. For the determination of the complete weight distribution, it is necessary to enumerate all the codewords up to scalar multiples. Hence this calculation is much more expensive than that of finding the minimum weight.

The function **WeightDistribution**(C) may be used to determine the number of codewords of C of each weight. It returns a sequence of tuples, each of which is an integer pair such that the first number is a i^{th} weight and the second number is the number of codewords with that weight. For example:

```
> print WeightDistribution(C);
[ <0, 1>, <3, 6>, <4, 12>, <5, 6>, <6, 2> ]
```

Table 40.8. Weight distribution functions

MAGMA	Meaning
CoveringRadius(C)	Covering radius of C
Diameter(C)	Largest weight of words of C
MinimumWeight(C), **MinimumDistance**(C)	Minimum weight d of words in C
Weight(u)	Weight of vector u
MinimumWords(C)	Set of words of C having minimum weight
Words(C, t)	Set of words of C with weight t
NumberOfWords(C, t)	Number of words of C with weight t
WeightDistribution(C)	Sequence of tuples <w_i, n_i> where n_i is the number of codewords of C with weight w_i ($n_i > 0$)
WeightDistribution(C, u)	Weight distribution of coset $C + u$, giving weight-counts of coset vectors (as a sequence of form described above)
CosetDistanceDistribution(C)	Sequence of tuples <d_i, n_i> where n_i is the number of cosets of C with distance d_i from C ($n_i > 0$)
DualWeightDistribution(C)	Weight distribution of dual of C
MacWilliamsTransform(n, k, q, W)	Supposing a hypothetical $[n, k]$ code C over GF(q), with weight distribution W, return weight distribution of dual of C

```
> print WeightDistribution(h4);
[ <0, 1>, <3, 35>, <4, 105>, <5, 168>, <6, 280>, <7, 435>,
<8, 435>, <9, 280>, <10, 168>, <11, 105>, <12, 35>,
<15, 1> ]
```

The other MAGMA functions associated with weight distribution, including **MacWilliamsTransform**, express the distribution in the same form.

Information about weight distribution may be obtained in a different form from the function **WeightEnumerator**(C). It returns an integer polynomial in two indeterminates, x and y say, such that the coefficient of $x^i y^{n-i}$ is the number of codewords with weight i. There are two methods for naming the indeterminates (i.e., as identifiers and as printnames): either by creating the parent ring over the integers with generator assignment or by *using generator assignment while assigning the polynomial*. Both techniques are demonstrated below, so that the user may choose whichever is preferred:

```
> print WeightEnumerator(C);
```

Table 40.9. Weight enumerator functions

MAGMA	Meaning
WeightEnumerator(C)	Polynomial in two indeterminates in which coefficient of $x^i y^{n-i}$ is number of codewords of C with weight i
WeightEnumerator(C, u)	Weight enumerator of coset $C + u$, giving weight-counts of coset vectors as a polynomial (of form described above)
CompleteWeightEnumerator(C)	Complete weight enumerator of C over GF(q), as a polynomial in q indeterminates
CompleteWeightEnumerator(C, u)	Complete weight enumerator of coset $C + u$
MacWilliamsTransform(n, k, K, W)	Supposing a hypothetical $[n, k]$ code C over K, with complete weight enumerator W, return complete weight enumerator of dual of C

```
$.1^6 + 6*$.1^3*$.2^3 + 12*$.1^2*$.2^4 + 6*$.1*$.2^5
+ 2*$.2^6

> // 1st method
> P<x,y> := PolynomialRing(IntegerRing(), 2);
> print WeightEnumerator(C);
x^6 + 6*x^3*y^3 + 12*x^2*y^4 + 6*x*y^5 + 2*y^6

> // 2nd method
> wtenum<x,y> := WeightEnumerator(C);
> print wtenum;
x^6 + 6*x^3*y^3 + 12*x^2*y^4 + 6*x*y^5 + 2*y^6
```

The weight enumerator functions are listed in Table 40.9 (p. 820).

40.7 Group Actions, Automorphism and Equivalence

40.7.1 Group Actions on Codes

There are two kinds of group action on an $[n, k]$ linear code C over GF(q): permutation action and monomial action. If G is a permutation group of degree n, then the *permutation action* of $g \in G$ on C is the operation whereby

the coordinate positions of codewords are permuted according to the permutation g. If G is a permutation group of degree $n(q-1)$, then the *monomial action* of $g \in G$ on C is the operation whereby not only coordinate positions are permuted, but also the coordinatees are multiplied by a non-zero element of GF(q). (Only some of the elements of G describe monomial permutations.) Note that if C is a binary code ($q = 2$), then monomial action reduces to permutation action.

[In more detail, a monomial permutation of monomial degree n is essentially a permutation s on the Cartesian product of the non-zero elements of K and the set $S = \{1, \ldots, n\}$ which satisfies the following property for all non-zero $\alpha, \beta, \gamma \in K$ and $i, j, \in S$:

$$(\alpha, i)^s = (\beta, j) \text{ implies } (\gamma\alpha, i)^s = (\gamma\beta, j).$$

Note that s is completely determined by its action on the points $(1, i)$ for each i; note also that the actual degree of s is $(q-1)n$. To represent a monomial permutation of monomial degree n, we number the pair (α, i) by $(q-1)(i-1)+\alpha$ and then use a permutation s of degree $(q-1)n$.]

The MAGMA operator ^ or the function **Image** may be used to compute images of codewords under group actions. Suppose that c is a codeword (or indeed any vector) with n coordinates over GF(q), and that g is an element of a permutation group G over the standard support. If the degree of G is n, then the expression $c\hat{}g$ or **Image**(g, c) returns the vector which is the image of c under the permutation action of g. On the other hand, if the degree of G is $n(q-1)$, and g describes a monomial permutation, then $c\hat{}g$ or **Image**(g, c) returns the vector which is the image of c under the monomial action of g. If the degree of G is neither n nor $n(q-1)$, then an error message will be given.

For example, let *gol3ext* be the extended ternary Golay code. The following lines show how to find the images of the sixth generator of *gol3ext* under a certain permutation action and monomial action. It emerges that neither of the images are in the code:

```
> gol3ext := GolayCode(GF(3), true);
> gen6 := gol3ext.6; print gen6;
(0 0 0 0 0 1 1 1 2 2 1 0)
> g1 := Sym(12) ! (5, 6, 7)(1, 10);
> v1 := gen6 ^ g1; // permutation action
> print v1;
(2 0 0 0 1 0 1 1 2 0 1 0)
> g2 := Sym(24) ! (5, 7)(6, 8)(11, 16, 20)(12, 15, 19);
> v2 := gen6 ^ g2; // monomial action
> print v2;
(0 0 0 0 0 1 1 2 2 1 1 0)
> print v1 in gol3ext, v2 in gol3ext;
```

```
false false
```

Group actions on a code C (or vector space) are computed by finding the action on the generators of C and returning the code generated by the resulting images. Both permutation and monomial actions may be computed, as for single codewords. The image of C under a permutation action is *isomorphic* to C, and the image of C under a monomial action is *equivalent* to C. For instance, the code *isoC* created below is isomorphic to *gol3ext*:

```
> isoC := gol3ext ^ g1; print isoC;
[12, 6, 6] Linear Code over GF(3)
Generator matrix:
[1 0 0 0 0 0 0 1 1 1 1 1]
[0 1 0 0 0 0 2 2 2 0 1 1]
[0 0 1 0 0 0 2 1 2 1 2 0]
[0 0 0 1 0 0 2 0 1 2 2 1]
[0 0 0 0 1 0 1 2 0 1 2 1]
[0 0 0 0 0 1 2 2 1 1 0 2]
```

The column-relationship between *isoC* and *gol3ext* is not obvious to the eye, because when MAGMA creates a vector space or code, it echelonizes the generators to find a basis. The following calculations show in more detail the underlying process:

```
> new_gens := [gol3ext.i ^ g1 : i in [1..6]];
> new_code := LinearCode< GF(3), 12 | new_gens >;
> print new_code eq isoC;
true
```

Whenever an object is acted on by a permutation group rather than a single permutation, the result is the G-set of the images of the object under each permutation in the group. Among the possible objects are a vector and a code. For example, consider the dihedral group of degree 12 and order 24 in its action on the first generator of *gol3ext*:

```
> print gol3ext.1 ^ DihedralGroup(12);
GSet{
    (1 0 0 0 0 0 0 1 1 1 1 1),
    (0 0 0 1 1 1 1 1 1 0 0 0),
    (1 1 1 1 1 0 0 0 0 0 0 1),
    (1 1 0 0 0 0 0 0 1 1 1 1),
    (0 0 0 0 1 1 1 1 1 1 0 0),
    (1 1 1 1 1 1 0 0 0 0 0 0),
    (1 1 1 1 0 0 0 0 0 0 1 1),
    (0 0 1 1 1 1 1 1 0 0 0 0),
```

```
    (0 0 0 0 0 0 1 1 1 1 1 1),
    (1 1 1 0 0 0 0 0 0 1 1 1),
    (0 0 0 0 0 1 1 1 1 1 1 0),
    (0 1 1 1 1 1 1 0 0 0 0 0)
}
```

Since the size of this G-set is less than the order of the group, not all of the images are distinct.

For details of other permutation group operations, see Chapter 32. There is one more group action function specially tailored for codes: given an $[n, k]$ code C and a permutation group G of degree n, **Fix**(C, G) returns the subcode of C consisting of the codewords that are fixed by the elements of G.

40.7.2 Equivalent and Isomorphic Codes

Two linear codes in $K^{(n)}$ are *equivalent* if one may be obtained from the other by a monomial action. The function **IsEquivalent**(C, D) tests whether the codes C and D are equivalent. If they are equivalent, it returns the corresponding mapping from C to D as a second value. For example,

```
> C1 := LinearCode<GF(3), 3 | [1,0,1]>;
> C2 := LinearCode<GF(3), 3 | [1,2,0]>;
> i, f := IsEquivalent(C1, C2);
> print i;
true
> print f(C1) eq C2;
true
```

Two linear codes in $K^{(n)}$ are *isomorphic* if one may be obtained from the other by a permutation action, that is, by permuting the columns (coordinate positions). For binary codes, this is the same as equivalence, but for other codes, isomorphism is a stronger relationship than equivalence. Currently, isomorphism testing is only possible in MAGMA for binary codes. The function **IsIsomorphic**(C, D) tests whether the binary codes C and D are isomorphic; if the principal return value is **true**, it returns as a second value a mapping which is an isomorphism from C to D.

As an example, consider the first-order binary Reed-Muller code R with $m = 4$, and the code T formed by the action of $p = (4, 13)(5, 12)(6, 11)$ under R. The following lines demonstrate that R and T are isomorphic, as expected:

```
> R := ReedMullerCode(1, 4); print R;
[16, 5, 8] Reed-Muller Code (r = 1, m = 4) over GF(2)
```

```
Generator matrix:
[1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> p := Sym(16) ! (4, 13)(5, 12)(6, 11);
> T := R ^ p; print T;
[16, 5, 8] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 0 1 0 1 0 1 1 1 1 0]
[0 1 0 0 0 1 1 0 1 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 1 1 1 1 1 0 1 1]
[0 0 0 1 0 0 1 1 0 0 1 1 0 1 1 1]
[0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0]
> print R eq T;
false
> iso, m := IsIsomorphic(R, T);
> print iso, m;
true Mapping from: Code: R to Code: T
```

Note that isomorphic codes need not have the same standard form. The function **StandardForm**(C), explained in Table 40.7 (p. 814), returns a code in standard form that is isomorphic to C. It returns the same code every time, but there are actually many codes satisfying the description. For example:

```
> print StandardForm(R) eq StandardForm(T);
false
```

The functions **IsEquivalent**(C, D) and **IsIsomorphic**(C, D) both have a parameter **AutomorphismGroups** which can take any of the string values `"None"`, `"Left"`, `"Right"` (the default), or `"Both"`. It specifies which, if any, of the automorphism groups of the codes (see below) are to be constructed first. For certain examples, the isomorphism test progresses faster when this parameter is changed from its default value.

40.7.3 Automorphism Groups

MAGMA offers facilities for computing the automorphism groups corresponding to each kind of group action on a code. Given an $[n, k]$ code C over GF(q), the function **MonomialGroup**(C) or **AutomorphismGroup**(C) returns the group of all monomial transformations that preserve C, as a permutation group of degree $n(q-1)$. The function **PermutationGroup**(C) returns the group of coordinate permutations that map the code C into itself,

as a permutation group of degree n. (This group is equal to the monomial automorphism group only if C is binary.) Each of these functions returns three values: the group itself; the parent M for all such automorphisms of C when represented as maps; and a transfer map t from G to M.

For example, let *BCHmod11* be the $[10, 6, 5]$ BCH code over GF(11). The following statements demonstrate the construction of and operations on the monomial group and permutation group of *BCHmod11*:

```
> BCHmod11 := BCHCode(GF(11), 10, 5);
> A, Apm, At := MonomialGroup(BCHmod11);
> print Apm;
Set of all automorphisms of [10, 6, 5] BCH code
(d = 5, b = 1) over GF(11)
> print At;
Mapping from: GrpPerm: A to PowMap: Apm
> print A;
Permutation group A acting on a set of cardinality 100
Order = 200 = 2^3 * 5^2
    (1, 11, 10, 20)(2, 12, 9, 19)(3, 13, 8, 18)(4, 14, 7,
        17)(5, 15, 6, 16)(21, 91, 30, 100)(22, 92, 29,
        99)(23, 93, 28, 98)(24, 94, 27, 97)(25, 95, 26,
        96)(31, 90, 40, 81)(32, 89, 39, 82)(33, 88, 38,
        83)(34, 87, 37, 84)(35, 86, 36, 85)(41, 71, 50,
        80)(42, 72, 49, 79)(43, 73, 48, 78)(44, 74, 47,
        77)(45, 75, 46, 76)(51, 70, 60, 61)(52, 69, 59,
        62)(53, 68, 58, 63)(54, 67, 57, 64)(55, 66, 56,
        65)
    (1, 9, 4, 3, 5)(2, 7, 8, 6, 10)(11, 92, 14, 98, 15,
        100, 19, 97, 13, 96)(12, 94, 18, 95, 20, 99, 17,
        93, 16, 91)(21, 89, 24, 83, 25, 81, 29, 84, 23,
        85)(22, 87, 28, 86, 30, 82, 27, 88, 26, 90)(31,
        72, 34, 78, 35, 80, 39, 77, 33, 76)(32, 74, 38,
        75, 40, 79, 37, 73, 36, 71)(41, 69, 44, 63, 45,
        61, 49, 64, 43, 65)(42, 67, 48, 66, 50, 62, 47,
        68, 46, 70)(51, 52, 54, 58, 55, 60, 59, 57, 53,
        56)
    (1, 10)(2, 9)(3, 8)(4, 7)(5, 6)(11, 91)(12, 92)(13,
        93)(14, 94)(15, 95)(16, 96)(17, 97)(18, 98)(19,
        99)(20, 100)(21, 90)(22, 89)(23, 88)(24, 87)(25,
        86)(26, 85)(27, 84)(28, 83)(29, 82)(30, 81)(31,
        71)(32, 72)(33, 73)(34, 74)(35, 75)(36, 76)(37,
        77)(38, 78)(39, 79)(40, 80)(41, 70)(42, 69)(43,
        68)(44, 67)(45, 66)(46, 65)(47, 64)(48, 63)(49,
        62)(50, 61)
```

```
    (11, 100)(12, 99)(13, 98)(14, 97)(15, 96)(16, 95)(17,
        94)(18, 93)(19, 92)(20, 91)(21, 81)(22, 82)(23,
        83)(24, 84)(25, 85)(26, 86)(27, 87)(28, 88)(29,
        89)(30, 90)(31, 80)(32, 79)(33, 78)(34, 77)(35,
        76)(36, 75)(37, 74)(38, 73)(39, 72)(40, 71)(41,
        61)(42, 62)(43, 63)(44, 64)(45, 65)(46, 66)(47,
        67)(48, 68)(49, 69)(50, 70)(51, 60)(52, 59)(53,
        58)(54, 57)(55, 56)
> print CompositionFactors(A);
    G
    |  Cyclic(2)
    *
    |  Cyclic(2)
    *
    |  Cyclic(5)
    *
    |  Cyclic(2)
    *
    |  Cyclic(5)
    1

> P, Ppm, Pt := PermutationGroup(BCHmod11);
> print P;
Permutation group P acting on a set of cardinality 10
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
> print Ppm;
Set of all automorphisms of [10, 6, 5] BCH code
(d = 5, b = 1) over GF(11)
> print Pt;
Mapping from: GrpPerm: P to PowMap: Apm

> c := Random(BCHmod11); print c;
( 1 10  5  1  5  9  5  0  1  5)
> ga := Random(A); print ga;
(1, 37, 6, 39, 3, 40, 7, 35, 9, 38, 10, 34, 5, 32, 8, 31,
    4, 36, 2, 33)(11, 24, 16, 22, 13, 21, 17, 26, 19, 23,
    20, 27, 15, 29, 18, 30, 14, 25, 12, 28)(41, 97, 46,
    99, 43, 100, 47, 95, 49, 98, 50, 94, 45, 92, 48, 91,
    44, 96, 42, 93)(51, 84, 56, 82, 53, 81, 57, 86, 59,
    83, 60, 87, 55, 89, 58, 90, 54, 85, 52, 88)(61, 77,
    66, 79, 63, 80, 67, 75, 69, 78, 70, 74, 65, 72, 68,
    71, 64, 76, 62, 73)
> print c ^ ga;   // or   Image(ga, c)
( 4  2  7  7  9  7  0  2  3  2)
```

```
> pa := Random(P); print pa;
(1, 8, 5, 2, 9, 6, 3, 10, 7, 4)
> print c ^ pa;
( 1  5  9  5  0  1  5  1 10  5)

> print Orbit(P, c);
GSet{
    ( 0  1  5  1 10  5  1  5  9  5),
    ( 5  9  5  0  1  5  1 10  5  1),
    (10  5  1  5  9  5  0  1  5  1),
    ( 5  1  5  9  5  0  1  5  1 10),
    ( 1  5  9  5  0  1  5  1 10  5),
    ( 5  0  1  5  1 10  5  1  5  9),
    ( 5  1 10  5  1  5  9  5  0  1),
    ( 1 10  5  1  5  9  5  0  1  5),
    ( 1  5  1 10  5  1  5  9  5  0),
    ( 9  5  0  1  5  1 10  5  1  5)
}
> print BCHmod11^ga eq BCHmod11;
true
```

The parent M of the automorphism maps and transfer map t can also be created without calling the **AutomorphismGroup**(C) function explicitly by the function **Aut**(C) which returns M and t. In this case, the domain of t is the full symmetric group of the appropriate degree (for the monomial action) so an automorphism of I can be applied to by t. The function **Aut**(C, T) allows one to select the kind of action: if T is `''Monomial''`, the monomial action is used; if T is `''Permutation''`, the permutation action is used.

40.8 Encoding and Decoding

The original motivation for coding theory was the encoding and decoding of information. This section will demonstrate the concepts involved, and their associated functions.

40.8.1 Encoding

Consider the extended binary Hamming code $C = \widehat{\text{Ham}}(4, 2)$. This code may be created in MAGMA by extending $\text{Ham}(4, 2)$ and putting the result in standard form:

```
> h4 := HammingCode(GF(2), 4);
> C := StandardForm(ExtendCode(h4));
> print C;
[16, 11, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1]
[0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1]
[0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1]
[0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 1]
[0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 1]
[0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0]
[0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 0]
[0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0]
```

Since the code C has length 16 and dimension 11, it encodes messages of 11 binary digits into codewords of 16 binary digits. The minimum weight of C is 4, so it can correct any single error, using nearest-neighbour decoding, and detect any double error.

Suppose now that the message to be encoded is 01101001101. It may be defined in MAGMA as a vector in the information space of C:

```
> infospace := InformationSpace(C);
> message := infospace![0,1,1,0,1,0,0,1,1,0,1];
```

To encode this vector, it must be multiplied by the generator matrix of C:

```
> encoded := message * GeneratorMatrix(C);
> print encoded;
(0 1 1 0 1 0 0 1 1 0 1 1 0 1 1 1)
```

Observe that *encoded* is the message vector followed by five check digits. This is because the code was created in *standard form*, in which the information set corresponds to the first k components of the $[n, k]$ code. This may be verified by typing

```
> print InformationSet(C);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]
```

to obtain the sequence from 1 to 11 inclusive. (The function returns a sequence rather than a set because it is often useful for the information set to be ordered.)

40.8.2 Errors in Transmission

Now imagine that *encoded* is sent along a noisy communications channel, and that consequently an error occurs in the eighth place of the codeword. The following assignment statements will simulate this:

```
> received := encoded;
> received[8] := 0;
```

The task of decoding involves the detection and correction of the error in *received*.

40.8.3 Elementary Syndrome-Decoding Techniques

The syndrome of a vector relative to a code is zero if and only if that vector is a word of the code. A call to the function **Syndrome** will show that the syndrome of *received* is non-zero:

```
> syn := Syndrome(received, C);
> print syn;
(0 0 0 0 1)
```

Therefore *received* is not a codeword. That is, the presence of error(s) in transmission has been detected.

Syndrome decoding rests on the theorem that two vectors have the same syndrome if and only if they are in the same coset of the code. The function **CosetLeaders**(C) returns a sequence containing all the coset leaders of a code C, and also returns a map from the syndrome space to the ambient space of the code. It is the map that is needed in this circumstance, because in practice it returns the coset leader with the given syndrome. The map is applied to the syndrome in the lines below:

```
> cl, syntoleader := CosetLeaders(C);
> errorvec := syn @ syntoleader;
> print errorvec;
(0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
```

The error vector is all zero except for the eighth component, as expected. The received vector must be corrected by subtracting the error vector:

```
> corrected := received - errorvec;
```

Now the decoding process is almost complete, since the decoded message can be extracted as the information part of *corrected*, using the function **Coordinates**(C, u):

```
> decoded := infospace ! Coordinates(C, corrected);
> print decoded eq message;
true
```

The message has been decoded successfully.

The limitations of a code can also be explored in this way. Suppose the received vector has an error in the sixth place as well. In this case, when the error vector is calculated using the same method as above, the answer is wrong; it should be 0000010100000000:

```
> rec2 := received; rec2[6] := 1;

> err2 := Syndrome(rec2, C) @ syntoleader;
> print err2;
(1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0)
```

If the decoding process is continued with this incorrect error vector, then the decoded vector *dec2* will not equal the original message:

```
> corr2 := rec2 - err2;
> dec2 := infospace ! Coordinates(C, corr2);
> print dec2;
(1 1 1 0 1 1 0 0 1 0 0)
```

The problem in this second example occurred because *rec2* was equally spaced (with a distance of two) between two codewords, *corrected* and *corr2*:

```
> print Distance(rec2, corrected);
2
> print Distance(rec2, corr2);
2
```

The decoding procedure regrettably chose the wrong codeword. Here is an example of a double error which C can detect but not correct.

40.8.4 Automated Decoding

The MAGMA function **Decode** performs an automated form of decoding, in the restricted sense of constructing the *corrected* vector from the *received*

vector as demonstrated above. Two algorithms are accessible: syndrome decoding, as explained above; and a Euclidean algorithm, which operates on alternant codes (including BCH, Goppa, and Reed-Solomon codes, as well as those discussed in Section 40.4.3). [The reader should consult the *Handbook* for further decoding methods.] The Euclidean algorithm cannot correct as many errors as the syndrome algorithm can, but in general it is faster, since the syndrome algorithm requires the coset leaders of the code. If the code is alternant, the Euclidean algorithm is used by default, but the user may request the other algorithm by assigning the string `"Syndrome"` to the parameter **Al** instead of the default value `"Euclidean"`. For non-alternant codes, only syndrome decoding is possible, so the parameter **Al** is not relevant.

Given a code C and a vector v in the ambient space of C, **Decode**(C, v) attempts to decode v. If the decoding algorithm succeeds in computing a vector v' as the decoded version of v, then **Decode** returns **true** and v'. *In the Euclidean case it may happen that v' is not in C, so the user should take care.* If the decoding algorithm does not succeed in decoding v, then **Decode** returns **false** and the zero vector. This function can also operate on a sequence Q of vectors: **Decode**(C, Q) returns a sequence of Booleans and a sequence of decoded vectors corresponding to the given sequence.

The statements below illustrate automatic syndrome decoding on a non-alternant code, using the same examples as above. Note that the results are the same as those achieved through manual application of syndrome decoding techniques:

```
> OK, corrD := Decode(C, received);
> print OK;
true
> print corrD eq corrected;
true

> OK2, corr2D := Decode(C, rec2);
> print OK2;
true
> print corr2D eq corr2;
true
```

The final phase in decoding, the computation of the information part of the corrected vector, should be accomplished as demonstrated previously.

The next example demonstrates the application of the Euclidean decoding algorithm with respect to the $[10, 6, 5]$ BCH code over GF(11):

```
> BCHmod11 := BCHCode(GF(11), 10, 5);
> print BCHmod11;
```

```
[10, 6, 5] BCH code (d = 5, b = 1) over GF(11)
Generator matrix:
[ 1  0  0  0  0  0  1  8  5  3]
[ 0  1  0  0  0  0  8 10  4  7]
[ 0  0  1  0  0  0  4  7  8  5]
[ 0  0  0  1  0  0  6  8  4  4]
[ 0  0  0  0  1  0  7  7 10  3]
[ 0  0  0  0  0  1  8  5  3  1]
```

This code contains 11^6 codewords and is a subspace of a vector space containing 11^{10} vectors, so there are 11^4 coset leaders. By using Euclidean decoding rather than syndrome decoding, the construction of these coset leaders can be avoided. (On the other hand, the coset leaders of a code are stored once they are computed, so it is a once-only time constraint.) For example:

```
> BCHenc := Random(BCHmod11);
> print BCHenc;
( 0  0  6  9 10  2 10  7  3 10)
> BCHrec := BCHenc; BCHrec[5] := 1; BCHrec[10] := 7;
> print BCHrec;
( 0  0  6  9  1  2 10  7  3  7)

> // Euclidean decoding
> time OK, BCHcorr := Decode(BCHmod11, BCHrec);
Time: 0.000
> print OK;
true
> print BCHcorr;
( 0  0  6  9 10  2 10  7  3 10)
> print BCHcorr eq BCHenc;
true
> // Syndrome decoding
> time OK, BCHcorr := Decode(BCHmod11, BCHrec :
>   Al := "Syndrome");
Time: 11.501
> print BCHcorr eq BCHenc;
true
```

For a sequence example, consider the Goppa code defined below. In this code, $r = 3$, so the Euclidean algorithm is able to correct $\lfloor \frac{r}{2} \rfloor = 1$ error:

```
> q := 2^5;
> K<w> := GF(q);
> Pq<z> := PolynomialRing(K);
> G := z^3 + z + 1;
```

```
> L := [w^i: i in [0..(q-2)]];
> GC := GoppaCode(L, G);
> print GC : Minimal;
[31, 16, 7] Goppa code (r = 3) over GF(2)

> wds := [ Random(GC) : i in [1..20] ];
> vecs := wds;
> for i in [1..20] do
>     vecs[i, Random(1, 31)] +:= 1;
> end for;
> OK, rec := Decode(GC, vecs);
> print false notin OK;
true
> print rec eq wds;
true
```

40.8.5 Simulation of Message Transmission

Suppose that *str* is a message, represented as a string. The function *send* developed below encodes *str* using an error-correcting code, imitates the transmission of the encoded version along a noisy communications channel, optionally applies a decoder to the received version to obtain a corrected version, and finally converts the corrected version back to a string and returns the result.

For simplicity, the string is encoded character-by-character. Therefore, in order to encode the characters with ASCII codes 1 to 127, a code with at least 127 codewords is required. The code C chosen for the demonstration is a binary BCH code with $n = 15$ and $d = 5$, having 128 codewords, so one of the codewords will not be used.

The function *send* has three arguments: the string *str*; a non-negative number *noise*, such that 0 means perfect transmission, slightly higher values mean that some codeword components may be altered, and 1 means every codeword component will be altered; and a Boolean *decode*, which should be set to **true** if and only if the **Decode** algorithm is to be applied. It returns a string of the same length as *str*, which will resemble it to an extent depending on the values of *noise* and *decode*.

To perform this example online, type `load "I96c40e3";`

```
> send := function(str, noise, decode)
>     C := BCHCode(GF(2), 15, 5);
>     GM := GeneratorMatrix(C);
```

```
>     InfSp := InformationSpace(C);
>     wds := Setseq(Set(InfSp));
>     STC := StringToCode;
>     CTS := CodeToString;
>     // from string to sequence of codewords
>     enc := [ wds[STC(str[i])] * GM :
>      i in [1..#str]];
>
>     // introduce noise
>     perturbate := function(v)
>         for i := 1 to Length(C) do
>             if Random(1000) / 1000 le noise then
>                 v[i] := v[i] + 1;
>             end if;
>         end for;
>         return v;
>     end function;
>     rec := [ perturbate(v) : v in enc ];
>
>     // apply Decode algorithm if requested
>     if decode then
>         _, corr := Decode(C, rec);
>     else
>         corr := rec;
>     end if;
>
>     // convert non-codewords to the codeword for "X"
>     Xwd := wds[STC("X")] * GM;
>     corrB := [ v in C select v else Xwd : v in corr ];
>     // find index-positions for the codewords,
>     dec := [ Index(wds, InfSp!Coordinates(C, wd)) : wd in corrB ];
>     // convert the (illegal) index-positions
>     // of the unused codewords to the index for "x"
>     xindex := STC("x");
>     decB := [ i le 127 select i else xindex : i in dec ];
>
>     return &cat[ CTS(i) : i in decB ];
> end function;
```

For example,

```
> text := "Mathematicians are like Frenchmen: \
> whenever you say something to them, \
> they translate it into their own language, \
> and at once it is something entirely different.
```

```
>          J. W. von Goethe";
> print text;
Mathematicians are like Frenchmen: whenever you say
something to them, they translate it into their own
language, and at once it is something entirely different.
         J. W. von Goethe

> print send(text, 0.01, false);
MathXmaticians aXe likX XreXcXmen: whenXver you say
sXmetXing to theX, they translXte iX into tXeir
ownXlXnguage, and at once XtXiX
somXthingXentirelyXdifXerXnt.
 X       J. W. von Goethe
> print send(text, 0.01, true);
Mathematicians are like Frenchmen: whenever you
sayXsomething to them, they translate it into their own
language, and at once it is something entirely different.
         J. W. von Goethe
> print send(text, 0.1, false);
XXXXXXXXXXXXXXaXXXXXXeXXXXXXXXXXX
XXXXXXXXXXXXXXaXXsXXXtXXXgXXXXXXeXX XXeyXXrXnsXaXX
iXXXXXXXXhXiXXXwXXXXXXXXXXXXXXXXaXXXXcXXXtXiXXXXXXXXiXXXXXX\
XXXXXXXXXXeXeXXX
XXX XXXXXX.XXXXXXXXXoeXXe
> print send(text, 0.1, true);
MatXmatiXians‘areliXe FrencXmen: whenevXX@you
saX7XmethinXto them, theX translate it iN=o shXir own
lnguggeX anD@at once it is something entiXely different.
 X X     J.XW. on GoetXe
```

40.9 Advanced Facilities

In addition to the operations described in this chapter, MAGMA contains further functions for constructing particular families of codes. Access is provided to a database of upper and lower bounds on the minimum distance for an optimal $[n, k]$ code as tabulated by Brouwer and Sloane for codes over GF(2), GF(3) and GF(4).

40.10 Notes on the Algorithms

The algorithms for computing minimum distances and weight distributions were developed by Allan Steel from ideas of Andries Brouwer. Testing equivalence of codes and the computation of automorphism groups are peformed using Jeff Leon's package PERM.

References

[AsK92] Assmus, E. F., Jr. and Key, J. D. (1992). *Designs and Their Codes*. Cambridge: Cambridge University Press.

[BBC93] Batut, C., Bernardi, D., Cohen, H. and Olivier, M. (November 1, 1993). *User's Guide to PARI-GP*. Unpublished. Updated for version 1.38.2.

[BeW93] Becker, T. and Weispfenning, V. (1993). *Gröbner Bases*. New York: Springer Verlag.

[Ber71] Berge, B. (1971). *Principles of Combinatorics*. New York: Academic Press.

[BGK86] Boege, Gebauer, and Kredel (1986). Some Examples for Solving Systems of Algebraic Equations by Calculating Groebner Bases. In: *J. Symbolic Computation (1986)* **1***, 83–98*.

[Boo59] Boone, W.W. (1959). The word problem. *Annals of Math.* , 207–265.

[BoC96] Bosma, W. and Cannon, J. J. (1996). *Handbook of Magma Functions*. Sydney: School of Mathematics and Statistics, University of Sydney.

[BCM94] Bosma, W., Cannon, J. J., Matthews, G. (1994). Programming with algebraic structures: design of the Magma language. In: M. Giesbrecht (ed.), *Proceedings of the 1994 International Symposium on Symbolic and Algebraic Computation, Oxford*, 52–57. Association for Computing Machinery.

[BCP96] Bosma, W., Cannon, J.J., Playoust, C. (1996). The Magma Algebra System I: The user language. Submitted to *J. Symbolic Computation*.

[BCP96] Bosma, W., Cannon, J. J, Playoust, C., Steel, A. (1996). *Solving Problems with Magma*. Sydney: School of Mathematics and Statistics, University of Sydney.

[Bou70] Bourbaki, N. (Nouvelle Édition 1970). *Algèbre I: Chapitres 1 à 3 (Éléments de Mathématique)*. Paris: Hermann.

[BuC89] Butler, G. and Cannon, J. J. (1989). Cayley version 4: The user language. [The early name for Magma was Cayley V4.] In: P. Gianni (ed.), *Proceedings of the 1988 International Symposium on Symbolic and Algebraic Computation, Rome, July 4–8, 1988, Lecture Notes in Computer Science 358*, 456–466. New York: Springer.

[BuC90] Butler, G. and Cannon, J. J. (1990). The design of Cayley, a language for modern algebra. In: A. Miola (ed.), *Design and Implementation of Symbolic Computation Systems, Lecture Notes in Computer Science 429*, 10–19.

[Can81] Cannon, J. J. (1981). The basis of a computer system for modern algebra. *Proceedings of the 1981 ACM Symposium on Symbolic and Algebraic Computation, (SYMSAC '81), Snowbird, Utah, August, 1981*, 1–5. New York: Association for Computing Machinery.

[Can82] Cannon, J. J. (1st ed. 1982). *A Language for Group Theory*. Sydney: Department of Pure Mathematics, University of Sydney.

[Can84] Cannon, J. J. (1984). An introduction to the group theory language Cayley. In: M. D. Atkinson (ed.), *Computational Group Theory*, 145–183. London: Academic Press.

[CaP96] Cannon, J. J. and Playoust, C. (1996). *First Steps in Magma*. Sydney: School of Mathematics and Statistics, University of Sydney.

[CKM96] Collart, S., Kalkbrener, M., Mall, D. (1996). Converting Bases with the Gröbner Walk. Submitted to *J. Symbolic Computation*.

[CLO92] Cox, D., Little, J. and O'Shea, D. (1992). *Ideals, Varieties and Algorithms*. New York: Springer Verlag.

[CoM72] Coxeter, H.S.M. and Moser, W.O.J. (1972). *Generators and Relations for Discrete Groups* New York, Berlin and Heidelberg: Springer.

[Cre92] Cremona, J. E. (1992). *Algorithms for Modular Elliptic Curves*. Cambridge: Cambridge University Press.

[DGK83] Diaconis, P., Graham, R. L., Kantor, W. M. (1983). The mathematics of perfect shuffles, *Adv. Appl. Math.* **4** (1983), 175–196.

[DiM96] Dixon, J. D. and Mortimer, B. (1996). *Permutation Groups*. New York, Berlin and Heidelberg: Springer.

[Hal86] Hall, M. (2nd ed. 1986). *Combinatorial Theory*. New York: Wiley.

[HuP73] Hughes, D. R. and Piper, F. C. (1973). *Projective Planes*. New York: Springer Verlag.

[Joh90] Johnson, D. L. (1990). *Presentations of Groups*. Cambridge: Cambridge University Press.

[LiW92] Lint, J. H. van and Wilson, R. M. (1992). *A Course in Combinatorics*. Cambridge: Cambridge University Press.

[LyS77] Lyndon, R. C. and Schupp, P. E. (1977). *Combinatorial Group Theory*. Berlin-Heidelberg-New York: Springer-Verlag.

[MaS77] MacWilliams, F. J. and Sloane, N. J. A. (1977). *The Theory of Error-Correcting Codes*. Amsterdam: North-Holland.

[MeM87] Medvedoff, S. and Morrison, K. (1987). Groups of Perfect Shuffles. *Mathematics Magazine* **60.1** (Feb 1987), 3–14.

[NST94] Neumann, P. M., Stoy, G. A. and Thompson, E. C. (1994). *Groups and Geometry*. Oxford: Oxford University Press.

[New90] Newman, M. F. (1990). Proving a group infinite. *Arch. Math.*, **54**, 209–211.

[Rio58] Rordan, J. (1958). *An Introduction to Combinatorial Analysis*. New York: John Wiley.

[Rot73] Rotman, Joseph J. (1973). *The Theory of Groups: an Introduction*. Boston: Allyn and Bacon.

Index

!, 53
", 254
( ), *see* parentheses
+, 54, 327, 330
−, 54, 327
/, 24
<, *see* **lt**
=, *see* **eq**
>, 3, *see* **gt**
R-module, 687
[∗ ∗], 98
[ ], *see* brackets
#, 83, 114, 115, 131, 132, 135, 140, 420
∗, 687
≥, *see* **ge**
≤, *see* **le**
< >, *see* angle brackets
≠, *see* **ne**
˜, 116
↑, 24
{! !}, 98, 129
{∗ ∗}, 98
{ }, *see* braces
{@ @}, 98
t-balanced, 753
', 217
’, 254
*, 24, 54, 258, 327, 330
*, 163, 266
+, 254, 266
,, 23
-n, 280
., 67, 421, 422
., 266
.., 105
/, 54, 327
/, 242
:, 108
:=, 26, 29
;, 3
?, 238, 255, 266
??, 239, 255
#, 331
#, 258, 593
$, 74, 163, 266
$$, 185, 188
%m, 272
%o, 272
&, 117
&meet, 391
@, 152
@@, 154
@@, 76
^, 266
/* */, 40
//, 40
% -- , 286
^^, 101
\\, 262
\, 335
_, 27, 32, 299
\", 262
\n, 262
\r, 262
\t, 262
‘, 142
‘ ‘, 143
^, 24, 54, 258, 327, 550, 647, 652, 687, 744, 749, 750, 821
%, 292
", 26, 257
|, 52, 102, 108, 409
|, 266

abbreviations, 279
abelian groups, 543, 627
– finitely-presented, 79, 629
– not in **GrpAb**, 633
abelian socle, 666
AbelianGroup, 544, 548, 630, 633, 634

AbelianQuotient, 601
AbelianQuotientInvariants, 601
Abs, 88, 340, 342, 368, 495
absolute extensions, 464
absolute value, 88
AbsoluteField, 466, 468
AbsoluteMinimalPolynomial, 456
AbsoluteNorm, 456
AbsoluteOrder, 468
AbsolutePrecision, 408
AbsoluteTrace, 456
AbsoluteValue, 368
AbsoluteValues, 481
accessing MAGMA, 3
Action, 610, 663, 664
action of groups, 648
– on codes, 820
– on graphs, 748
ActionImage, 664
ActionKernel, 664
actions, 638
– induced on a block, 657
– induced on a G-set, 664
– induced on an orbit, 654, 690
actual arguments, 168, 171
AddColumn, 521
additive operators, 54, 327
AdditiveGroup, 352
AddRow, 521
adj, 299, 738
affine planes, 7, 777
AffineGammaLinearGroup, 644
AffineGeneralLinearGroup, 644
AffinePlane, 778, 780, 783, 794
AffineSigmaLinearGroup, 644
AffineSpecialLinearGroup, 644
AGammaL, 644
aggregate structures, 97
AGL, 644
AGM, 501
aInvariants, 706
algebraic number fields, 443
algebraic structures, 4
algebras
– finitely-presented, 58
– Laurent series, 407
– matrix, 72, 527
– polynomial, 364
– power series, 407
AlgMat, 527
algorithms
– verbose printing, 271
Alldeg, 738
AllInformationSets, 806
AllIrreduciblePolynomials, 432
AllPassants, 796
AllSecants, 796
AllSqrts, 352
AllSquareRoots, 352
AllTangents, 796, 797
Alphabet, 806
alphabetic characters, case conversion, 264
Alt, 548, 643
alternant code decoding, 831
AlternantCode, 812
alternating groups
– recognizing, 665
AlternatingGroup, 548
AlternatingSum, 499
analysis of strings, 263
and, 38, 299
angle brackets, 48, 52, 72, 98, 132, 135, 409
anti-Hermitian, 535
Append, 120, 121, 123, 131, 140
appending to files, 282
Arccos, 498
Arccosec, 498
Arccot, 498
Arcsec, 498
Arcsin, 498
Arctan, 498
Arctan2, 498
Arg, 496
Argcosech, 498
Argcosh, 498
Argcoth, 498
Argsech, 498
Argsinh, 498
Argtanh, 498
Argument, 496
arguments, 168
– actual, 171
– formal, 171, 176
–– local scope, 181
arithmetic, 24, 54, 327
arithmetic progression, 122
arithmetic progression constructors, 105
ArithmeticGeometricMean, 501
arrays, 125
ASigmaL, 644
ASL, 644
assert, 195, 299
AssertAttribute, 411, 436, 490, 686

assigned, 28, 143, 299
assignment
– generators, 48
assignment statements, 26, 29
– generators, 48
– multiple, 31
– mutation, 31
assignments
– start-up, 279
AssignNames, 73
Attach, 222
AttachSpec, 232
attributes, 411
AugmentCode, 814
automated decoding, 830
automatic coercion, 56, 115
automatic updating of packages, 224
automatically-created magmas, 49
automorphism groups, 727
AutomorphismGroup, 730, 750, 752, 775, 799, 824
AutomorphismGroupStabilizer, 733
AutomorphismSubgroup, 733

BachBound, 482
back, 247, 255
backslash, 266, 335
backslash character, 262
balanced, 753, 765
Ball, 746
Bang, 150
base and strong generating set, 684
base and strong generating sets (BSGS), 669
base field
– changing, 526
base-b representation of integer, 340
BaseField, 517
BaseModule, 537
BaseRing, 468, 535, 683, 706
Basis, 388, 451–454, 471, 508, 517
basis
– changing, 523
– echelonized, 515
– embedded, reduced, 507
– standard, 508, 509
BasisElement, 388, 517
BasisMatrix, 453, 454, 471, 517
BCH code decoding, 831
BCHCode, 810
Bell numbers, 724
Berlekamp's factorization algorithms, 371
BernoulliApproximation, 501, 719
BernoulliNumber, 501, 719
BernoulliPolynomial, 719
BesselFunction, 501
BestApproximation, 500
BetterPolynomial, 467, 468
bigger than, *see* **gt**
bigger than or equal to, *see* **ge**
bijections, 638
binary quadratic forms, 476
Binomial, 719
bInvariants, 706
BipartiteGraph, 736
Block, 759, 760
block codes, 801
block designs, 753
block intersection number, 763
block systems, 655
BlockDegree, 760, 763
BlockDegrees, 759
BlockGraph, 776
BlockGroup, 732, 775
Blocks, 755, 759, 760
blocks, 753
BlocksAction, 658, 665
BlockSet, 759, 760
BlocksImage, 658
BlockSize, 760, 763
blocksize, 753
BlockSizes, 759
BlocksKernel, 658
BM, 564
books about MAGMA, 237
Bool, 50
Boolean functions, 39
Booleans, 38
Booleans, 50
Bosma, Wieb, 16
braces, 98
brackets, 98, 102, 120, 121, 131, 140, 258, 266, 512
break, 210, 299
Brent, R. P., 485
Brownie, 670
browser
– online help system, 237, 239
browsers for Internet, 237
BSGS, 684
BSGS, 669
Buchberger algorithm, 13
buffer
– output, 273
– previous value, 41

bugs, 195
built-in functions and procedures, 167
by, 105, 299

cached data, 85
call-by-name evaluation, 92, 94
call-by-value evaluation, 38
calling (invoking) functions/procedures, 168
calls, *see* invocations
Cannon, 670
canonical forms of elements, 57
CanonicalGraph, 752
Cantor-Zassenhaus factorization algorithm, 371
capital letters, 264
car, 98, 132
cardinality, 83, 114, 132, 135, 331, 420
– finite, infinite, 133
Carlson, Jon, 9
CarmichaelLambda, 343, 359
carrier sets, 727
Cartesian product, 132
cartesian products, 98
CartesianPower, 132
CartesianProduct, 132, 744
case, 92, 94, 299
case of alphabetic characters, 264
cat, 120–122, 131, 140, 255, 258, 299
Catalan, 497
Catalan numbers, 417
categories, 46
Category, 46, 325, 327
cd, 241, 255
Ceiling, 342, 495
CentralCollineationGroup, 799
Centre, 330
ChangeDirectory, 277
ChangeRing, 526
ChangeUniverse, 104
changing the basis of a vector space, 523
changing the coefficient field of a vector space, 526
Characteristic, 330, 420
CharacteristicPolynomial, 432, 440, 539
characters, 257, *see* strings
CharacterTable, 462
Check, 758, 780
checking for bugs, 195
CheckPolynomial, 811
ChienChoyCode, 812
ChineseRemainderTheorem, 349
cInvariants, 706
Class, 567
Classes, 14, 567
ClassGroup, 474
ClassGroupStructure, 474
classical groups, 680
classical planes, 777, 782
ClassNumber, 474
ClassRepresentative, 569
clear, 299
ClearPrevious, 42
Clique, 747
CliqueNumber, 747
cmpeq, 83, 85, 299
CO_1, CO_2, CO_3, 564
codes, 727, 801
– equivalent, isomorphic, 823
CodeToString, 264
Codomain, 156, 518
Coefficient, 366, 381, 408
coefficient field
– changing, 526
CoefficientField, 510, 517
CoefficientRing, 364, 392, 408, 468, 510, 535, 683, 706
Coefficients, 366, 380, 381, 408, 706
Coercion, 150
coercion, 53, 65, 109, 115, 490, 496
Cokernel, 520
collineation groups, 7
CollineationGroup, 799
CollineationGroupStabilizer, 799
CollineationSubgroup, 799
ColonIdeal, 391
colourings of graphs, 747
combinatorial and incidence structures, 727
combinatorics, 417, 719
comma, 23
commands, 299
comments, 40
common overstructures, 56
commutator, 551
commutators, 549
comp, 319, 321
comparable, 84
comparing magma elements, 57
comparing ring elements, 328
compatible, 115, 118
Complement, 517, 743, 766
ComplementaryErrorFunction, 501
complete, 765, 795

complete coset spaces, 609
complete sequences, 113
CompleteGraph, 736
CompleteKArc, 796
CompleteUnion, 744
CompleteWeightEnumerator, 820
Completion, 321
completion, 250, 256
completion, tab, 289
complex fields, 485
complex numbers, 35
ComplexConjugate, 456, 458, 496
ComplexField, 36, 318, 485, 494
ComplexToPolar, 37, 496
Component, 746
Components, 746
Composition, 415, 478
composition of mappings, 163
CompositionFactors, 564
concatenate, 120, 121, 258
concatenation, 122
concrete groups
– presentations, 600
conditional evaluation, 38
conditional expressions, 87, 92
conditional statements, 87
conditions, turning off checking, 758, 780
Conductor, 448, 450, 452
congruences, 349
Conic, 797
conjugacy, 653
conjugacy classes, 14
ConjugacyClasses, 567
Conjugate, 457, 458, 481
Conjugates, 481, 567
conjugates, 549
conjugation, 550
ConnectionNumber, 760
constants, 497
constants in packages, 233
constituents, 136
constructors, 48
– **AbelianGroup**, 630
– **Group**, 598
– **MatrixAlgebra**, 530
– **MatrixGroup**, 678
– **elt**, 52, 409, 493
– **func**, 171
– **hom**, 157
– **ideal**, 61
– **lideal**, **rideal**, 61
– **map**, 157
– **ncl**, 61
– **proc**, 176
– **quo**, 75, 515, 597, 629
– **random**, 126
– **recformat**, **rec**, 141
– **rep**, 126
– **sub**, 61, 515, 629
– arithmetic progression, 105
– formal sets, 129
– set, sequence, 107, 331
– tuple, 134
ContainsQuadrangle, 789
Content, 368
contiguous substring, 259
continue, 210, 299
ContinuedFraction, 500
Contpp, 368
Contract, 743
Contraction, 766
control characters, 291
control-c, 293
control-d, 255
control-n, 292
control-p, 292
Convergents, 500
Convolution, 415
ConwayPolynomial, 434
Coordinates, 388, 790, 805, 830
Cop, 98
cop, 98
coproducts, 98, 136, 140
Core, 611
Cos, 415, 498
Cosec, 498
Cosech, 498
coset representatives, 611
coset tables, 607, 609
CosetAction, 614
CosetDistanceDistribution, 819
CosetImage, 614
CosetKernel, 614
cosets
– finitely-presented groups, 613
CosetSpace, 610, 614
Cosh, 415, 498
Cot, 498
Coth, 498
counting theory, 417
Covalence, 759, 763
CoveringRadius, 819
Coxeter, 670
CPU time, 33
Cputime, 294

creating objects, 4
creation, 325
cube, group of symmetries of, 646
curly brackets, *see* braces
current directory, 277
current input line, 290
CurveEll, 703
CurveEllPt, 703
customizing MAGMA, 233
cycle form for permutations, 639
CycleStructure, 641
CyclicCode, 810
CyclicGroup, 548
CyclotomicField, 447, 448
CyclotomicOrder, 448

Darstellungsgruppe, 599
data files, 281
databases, 835
databases of definitions, 42, 277
DawsonIntegral, 501
Decode, 830
decoding, 827
Decomposition, 473
DedekindEta, 501
Default, 269, 270, 281
default, 299
default precision, 411
default real field, 486
DefiningPolynomial, 425, 432, 444, 446, 448, 450
definitions in libraries and databases, 277
Degree, 366, 381, 408, 420, 444, 446–448, 450, 470, 517, 535, 641, 648, 663, 683, 738
degree, 443, 638
degree of transitivity, 649
DegreeOnePrimeIdeals, 473
degrees (angles), 498
DegreeSequence, 738
delete, 28, 143, 285, 299
delve, 245, 255
Denominator, 37, 342, 366, 385, 470
derivation, 7
Derivative, 367, 415
derived G-sets, 638, 648, 660
derived subgroups, 610
Desarguesian, 792
Design, 766, 800
designs, 727, 753, 765
Detach, 222
DetachSpec, 232
Determinant, 536
Development, 769
DiagonalJoin, 514
DiagonalMatrix, 528
Diameter, 746, 819
DiameterPath, 746
diff, 118, 130, 299
difference
– of elements, 54, 327
– of sets, 118, 130
difference sets, 769
DifferenceSet, 769
digraphs, 735
DihedralGroup, 548
Dilog, 498
Dimension, 392, 517, 535, 806
directed graphs, 735
directories of files, 277
DirectProduct, 599, 645, 682, 814
DirectSum, 534, 536, 634, 814
Discriminant, 372, 444, 446, 448, 450, 478, 706
disjoint cycle form for permutations, 639
Distance, 478, 746, 805
distance, 818
DistinctDegreeFactorization, 371
div, 24, 54, 299, 332, 339, 367
division, 54, 327, 332
– exact, 24
– in rings (quotient-and-remainder), 24
– integers, 24
– quotient, remainder, *see* **Quotrem**
DivisorSigma, 343, 359
do, 199, 205, 299
documentation of MAGMA, 237
Domain, 156, 518
double cosets, 611
double quotation mark character, 26, 257, 262
Dual, 766, 794, 806
DualWeightDistribution, 819
dummy variables, 107

e, 486
e, 36
EARNS, 666
EchelonForm, 519, 538
echelonized basis, 515
echoing external input, 280
EdgeGroup, 732, 750, 751
Edges, 737
edges

– addition, subtraction, 739
EdgeSet, 737
EdgeUnion, 744
editing input, 290
EDITOR, 293
ei, 501
Eigenspace, 539
Eigenvalues, 539
element-description, 107
elements, 4, 51
– canonical forms, 57
– literal, 52
– representation, 58
– testing membership, 83, 115, 130
ElementToSequence, 435, 529, 628, 673
elif, 90, 299
Eliminate, 594
elliptic curves, 703
else, 87, 92, 299
elt, 52, 134, 135, 336, 409, 493
Eltseq, 435, 444, 446, 448, 452, 453, 511, 529, 593, 594, 628, 673, 790
Emacs editor, 290
Embed, 430, 432
embedded basis, 507
embedding in complex field, 458
embedding of finite fields, 429
empty sets and sequences, 104
empty word, 593
EmptyGraph, 736
encoding, 827
end, 299
endo-trivial modules, 9
EndomorphismAlgebra, 524
EndVertices, 738
entering MAGMA, 3
enumerated sequences, 98
enumerated sets, 98
enumerative combinatorics, 719
environment variables, 278, 279, 296
environment, user, 289
eq, 57, 83, 115, 132, 135, 299, 327, 391, 593, 729
equal to, *see* **eq**
equality
– words in fp-magmas, 56
equation orders, 449
EquationOrder, 450
equivalent codes, 822, 823
Erf, 501
Erfc, 501
error, 299
error-correcting codes, *see* codes
ErrorFunction, 501
errors, 195, 225
errors in sending codewords, 829
escape to shell, 286
η, 501
Euclidean decoding, 831
Euclidean decoding algorithm, 831
EuclideanNorm, 332
EulerGamma, 497
EulerianNumber, 719
EulerPhi, 343, 359, 447
Evaluate, 367, 414
evaluation
– call-by-name, call-by-value, 92, 94
evaluations, 23
examples, 7
– in the textbooks, 44
– online help nodes, 248
Exclude, 116, 120, 121
ExcludedConjugates, 610
exclusive-or, 38
execute, 3
existential quantifiers, 127
exists, 127, 299
ExistsConwayPolynomial, 434
exit, 255, 299
exiting an iteration or loop quickly, 210
exiting MAGMA, 5, 293
Exp, 415, 498
Explode, 91, 120, 121, 135
exponentation, 35
exponential numbers, 724
ExponentialIntegral, 501
exponentiation, 24, 54, 327, 349
– modular, 348
ExponentSum, 594
expressions, 23
– **case**, 94
– conditional, 87, 92
– regular (strings), 265
ExpurgateCode, 814
ext, 319, 465
ExtendBasis, 525, 526
ExtendCode, 814
extended greatest common divisor, 339
ExtendField, 526
ExtensionField, 425
extensions
– of finitely-presented groups, 599
– relative, 455
– relative, absolute, 464
extensions of finite fields, 423

Exterior, 797
ExteriorSquare, 534, 536
external files and processes, 277
external line, 795
external processes, 285
ExternalLines, 796
ExtractGenerators, 622
ExtractGroup, 622
ExtractRep, 119

Facint, 347
FactorBasis, 482
FactoredOrder, 435, 536
Factorial, 719
Factorization, 136, 332, 356, 369, 473
factorization, 16
factorization sequences, 136, 345
FactorizationToInteger, 347
false, 38, 50, 299
families of groups, 564
Fibonacci, 112, 719
Field, 806
field of fractions, 322
FieldOfFractions, 322, 365, 385, 449, 450
fields, 317
– complex, 485
– cyclotomic, 447
– finite (Galois), 51, 351, 419
– number, 16, 443
– quadratic, 446
– rational, 335, 443
– real, 485
– residue, 323
File, 282
file objects, 282
files, 277
– data, 281
– input, 42, 277
– log, output, 281
– start-up, 278
find, 243, 255
finite fields, 51
finite geometry, 7, 777
finite planes, 7, 777
FiniteField, 318, 351, 419
finitely-presented, 78
finitely-presented abelian groups, 629
finitely-presented groups, 591, 595
– cosets, permutation representations, 613
– quotients, 601
– subgroups of finite index, 619
finitely-presented magmas, 58, 59
finitely-presented subgroups
– subgroup presentation, 615
First Steps in Magma, V
first textual use rule, 181
first-class objects, 170, 183
Fix, 641, 648, 664, 823
fixed precision real/complex fields, 485
fixed precision series rings, 408
fixed-point set, 648
flags, verbose printing, 271
Flat, 132
FldCyc, 444
FldNum, 444
FldPr, 50
FldQuad, 444
FldRat, 50, 443
Floor, 342, 495
flushing the output buffer, 273
for, 205, 299, 331
forall, 127, 299
formal arguments, 168, 171, 176
– local scope, 181
formal sets, 98, 129
FormalSet, 83, 129, 648
Format, 143
format string, 272
formats for records, 141
formatted printing, 272
forms
– binary quadratic, 476
forward, 185, 299
fp, *see* finitely-presented
FPGroup, 600
free identifiers, 107
free magmas, 50, 78
free precision series rings, 408
free real/complex fields, 485
free reductions, 58, 594
FreeAbelianGroup, 544, 627
FreeGroup, 50, 544, 592
FreeProduct, 599
freeze, 224, 299
frozen packages, 224
full matrix space, 508
full pathnames, 277
full vector space, 507
func, 171
Function, 159
function, 299
function assignment statements, 187
function expressions
– constructor form, 171

– statement form, 174, 187
function fields, 323, 365, 385
FunctionField, 318, 323, 365
functions, 167
– **return**, 174
– Boolean, 39
– defined recursively, 183, 188
– intrinsic, 25
– principal value, 32
– returning several values, 31, 172
– signatures, 216
– transfer, 146
– user-defined, 170
fundamental units, 479
FundamentalUnit, 476

G-invariant, 638, 654, 655
G-set, 730
G-sets, 638
– derived, 660
– general, 660
– natural, 639
– standard, 638
Galois correspondences, 16
Galois groups, 457
GaloisField, 51, 419
GaloisGroup, 17
Γ, 497, 501
Gamma, 501
γ, 501
GammaD, 501
GCD, 196, 332, 339, 347, 367
Gcd, 339
ge, 57, 299, 331, 593
general element in aggregate constructor, 107
general G-sets, 660
general incidence structures, 727
'general' number fields, 443
GeneralizedSrivastavaCode, 812
GeneralLinearGroup, 546, 672, 680
GeneralUnitaryGroup, 681
generating functions, 417, 726
GeneratingWords, 601
Generator, 350, 352, 421, 422
generator assignments, 48
GeneratorMatrix, 806
GeneratorNumber, 594
GeneratorPolynomial, 811
Generators, 66, 471, 508, 517, 601, 648, 712
generators, 58, 515
– naming, 72
Generic, 63, 391, 517, 531, 535, 648, 679, 683
generic magmas, 63
Geodesic, 746
geometry
– finite, 7, 777
GetAssertions, 196
Getc, 282
GetCurrentDirectory, 277
GetDefaultRealField, 487, 491
GetEchoInput, 280
GetHistorySize, 291
GetMemoryLimit, 296
GetPath, 278
Getpid, 287
GetPreviousSize, 41
Gets, 282
GetVerbose, 271
GF, 419
Girth, 746
GirthCycle, 746
GL, 81, 546, 672, 680
GolayCode, 809
Goppa code decoding, 831
GoppaCode, 812
goto, 241, 255
grammar, 303
graphs, 727, 735
– group action, 748
– standard, 736
– standard constructions, 743
– tests, 745
– unions and products, 744
greater than, *see* **gt**
greater than or equal to, *see* **ge**
greatest common divisor, 339, *see* **GCD**
GreatestCommonDivisor, 339
Groebner, 388
GroebnerBasis, 388
GroundField, 424, 464, 468
Group, 544, 598, 663
group actions, 638
groups, 543
– abelian, 79, 627
– abelian (not in **GrpAb**), 633
– action, 648
– action on codes, 820
– action on graphs, 748
– categories, 547
– class, 473
– classical, 680
– commutator, 551

– conjugation, 550
– families, 564
– finitely-presented, 58, 79, 591, 595
– finitely-presented abelian, 629
– free, 592
– Galois, 457
– ideal class, 473
– identity element, 551
– linear, 680
– matrix, 13, 671
– orthogonal, 680
– permutation, 637
– polycyclic, 70, 79
– presentations for concrete groups, 600
– projective, 680
– sporadic, 564
– standard, 548
– symmetric, 639
– symplectic, 680
– unit, 473
– unitary, 680
GrpAb, 543, 627
GrpFP, 543
GrpMat, 543, 672
GrpPC, 543
GrpPerm, 543
GRSCode, 809
GSet, 639, 648, 661, 662
gt, 57, 299, 327, 331, 593
GU, 681

HA, 564
HadamardColumnDesign, 767
HadamardRowDesign, 767
Hall ovals, 7
Hamming distance, 801
Hamming weight, 818
HammingCode, 809
Handbook of Magma Functions, V, 44, 237, 248
HarmonicNumber, 719
HasAttribute, 411, 436, 490
hash table, 99
HasOutputFile, 281
HE, 564
Height, 714
HeightPairing, 714
help, 255
help on MAGMA, 237
HenselLift, 372
Hermitian, 535
highest common factor, *see* **GCD**
Hilbert-driven Buchberger algorithm, 13
history, 291
Hom, 509, 517
hom, 157, 299
homogeneous, 98
homomorphism modules, 509, 517
homomorphisms, 149
– preimages, 154
HorizontalJoin, 514
HS, 564
hyperbolic functions, 415, 498
HypergeometricU, 501
hypertext help system, 237

Id, 54, 551, 627, 673
ideal, 61, 319, 350, 388, 470, 531
ideal class group, 474
ideals, 61
Idempotent, 811
identifiers, 26
– deleting, 28
– free, 107
– interchanging, 30
– local and non-local, 181, 207
– loop, 183, 207
– start-up, 279
– value class, 183, 208
Identity, 54, 551, 673
identity elements, 54, 327
if, 87, 89, 299
Ilog2, 340
Im, 494, 496
Image, 156, 519, 537, 647, 664, 821
images of mappings, 152
Imaginary, 494, 496
import, 233, 299
imprimitive groups, 655
in, 83, 115, 130, 135, 205, 259, 299, 331, 391, 738, 760, 789
IncBlk, 761
IncBlkSet, 760
incidence structures, 727, 753
IncidenceGraph, 776, 800
IncidenceMatrix, 729, 757, 759, 784
IncidenceStructure, 766, 776
IncidentEdges, 738
Include, 116, 120, 121
inclusion monomorphisms, 65
inclusive-or, 38
incomplete coset spaces, 609
incomplete sequences, 113
IncPt, 761

IncPtSet, 760
IndependenceNumber, 747
Independent, 747
IndependentUnits, 479
Index, 114, 258, 470, 607
indexed sets, 98
IndexedSetToSequence, 146
IndexedSetToSet, 146
indexing vectors and matrices, 512
induced actions, 654, 657, 664, 690
infinite series, 499, 725
infinite sets, 129, 130
InfiniteSum, 499
InformationSet, 806
InformationSpace, 806
InnerProduct, 512, 805
input, 3, 274
– echoed from files, 280
– long input lines, 262
input files, 42, 277
input, editing, 290
Insert, 120, 121, 124
InsertBlock, 514
InsertVertex, 743
integer arithmetic, 24, 36
integer part of n^{th} root, 340
integer part of square root, 340
integer-string, 263
IntegerRing, 50, 318, 322, 336, 365, 385, 450
Integers, 336
integers, 35
integers in base-b representation, 340
IntegerToString, 263
Integral, 195, 415
IntegralBasis, 450, 451
IntegralModel, 709, 710
interacting with MAGMA, 3
interface, user, 237
Interior, 797
Internet browsers, 237
interrupting a computation, 293
intersection, 83, 118, 130, 330
IntersectionArray, 750
IntersectionNumber, 763
intrinsic, 299
intrinsics, 25, 167, 215
– help nodes, 248
– signatures, 216
An Introduction to Magma, V, 44
Intseq, 340
InvariantFactors, 539
InvariantRing, 697
Invariants, 631, 634
invariants, 13
InvariantsOfDegree, 15, 697
inverse elements, 54, 327
inverse trigonometric and hyperbolic functions, 415, 498
invocations of functions/procedures, 168, 176
Iroot, 340
IrreduciblePolynomial, 425, 432
is, 33, 299
IsAbelian, 559
IsAlternating, 665
IsAltSym, 665
IsArc, 796
IsAssociate, 332
IsBalanced, 765
IsBipartite, 39, 745
IsBlock, 657, 760
IsBlockTransitive, 775
IsCentral, 559
IsCentralCollineation, 799
IsCollinear, 789
IsCommutative, 329
IsComplete, 113, 610, 745, 765, 796
IsConcurrent, 789
IsConjugate, 559, 612, 653, 664
IsConnected, 746
IsConsistent, 523
IsConway, 434
IsCyclic, 559, 807
IsDefined, 113
IsDesarguesian, 792
IsDesign, 765
IsDiagonal, 536
IsDifferenceSet, 769
IsDistanceRegular, 750
IsDistanceTransitive, 750
IsDivisible, 332, 339
IsDivisionRing, 329
IsDomain, 329
IsEdgeTransitive, 750, 751
IsElementaryAbelian, 559
IsEllipticCurve, 705
IsEmpty, 104, 131, 140, 624, 745
IsEof, 282
IsEquationOrder, 450
IsEquivalent, 823
Isetseq, 146
Isetset, 146
IsEuclideanDomain, 329, 332
IsEuclideanRing, 329, 332
IsEulerian, 746

IsEven, 339, 347, 641
IsExtraSpecial, 559
IsFaithful, 664
IsField, 329
IsFinite, 329, 330
IsForest, 745
IsFrobenius, 650
IsHadamard, 767
IsHadamardEquivalent, 767
IsId, 54, 57, 627
IsIdempotent, 328
IsIdentity, 54, 57
IsIndependent, 525
IsInRadical, 391
IsIntegral, 336, 342, 450, 455, 473, 496
IsIntegralDomain, 329
IsInvariant, 697
IsIrreducible, 328, 371, 383, 432
IsIsomorphic, 468, 710, 729, 752, 817, 823
IsLinearGroup, 695
IsLinearSpace, 765
IsLineRegular, 765
IsLineTransitive, 799
IsMaximal, 401, 450, 559, 612
IsMDS, 807
IsMinusOne, 328, 478, 496
IsNearLinearSpace, 765
IsNilpotent, 328, 559
IsNormal, 438, 559, 612
IsNull, 104
IsOdd, 339, 347, 641
isomorphic, 729
isomorphic codes, 822, 823
isomorphisms, 467, 727
IsOne, 54, 57, 328, 347, 478, 496
IsOrbit, 652
IsOrdered, 329
IsOrthogonalGroup, 695
IsOrthogonalType, 695
IsParallel, 789
IsParallelClass, 760
IsPath, 745
IsPerfect, 559
IsPID, 329
IsPIR, 329
IsPointRegular, 765
IsPointTransitive, 775, 799
IsPolygon, 745
IsPower, 457
IsPowerTimesUnit, 457
IsPrimary, 401
IsPrime, 328, 347, 354, 473
IsPrimePower, 347
IsPrimitive, 349, 352, 422, 432, 457, 657, 664, 750
IsPrincipal, 473
IsPrincipalIdealDomain, 329
IsPrincipalIdealRing, 329
IsProper, 391
Isqrt, 340
IsRadical, 401
IsReal, 496
IsReduced, 478
IsRegular, 328, 650, 664, 738
IsRelative, 454
IsResolvable, 765
IsScalar, 536
IsSelfDual, 765, 792, 807
IsSelfNormalizing, 559
IsSelfOrthogonal, 807
IsSemiregular, 650, 664
IsSemisimple, 532
IsSharplyTransitive, 650, 664
IsSimilar, 536
IsSimple, 532, 559, 765
IsSoluble, 559
IsSolvable, 559
IsSpecial, 559
IsSpecialLinearGroup, 695
IsSpecialOrthogonalGroup, 695
IsSpecialUnitaryGroup, 695
IsSquare, 352, 435
IsSquareFree, 347
IsSteiner, 765
IsSubnormal, 559
IsSubsequence, 122
IsSymmetric, 536, 665, 750, 765
IsSymplecticGroup, 695
IsSymplecticType, 695
IsTransitive, 650, 664, 750, 799
IsTree, 745
IsTrivial, 765
IsUFD, 329
IsUniform, 765
IsUniqueFactorizationDomain, 329
IsUnit, 328, 347, 455, 529
IsUnital, 797
IsUnitary, 329
IsUnitaryGroup, 695
IsUnitaryType, 695
IsVertexTransitive, 750
IsWeaklySelfDual, 807
IsWeaklySelfOrthogonal, 807
IsZero, 54, 57, 328, 391, 473, 496, 511
IsZeroDivisor, 328

iterable, 98
iterable homogeneous aggregates, 100, 114
iterable magmas, 107, 206
iteration, 133, 331
iterative statements, 197
– nested, 212
– quick exit, 210

J_1, J_2, J_3, J_4, 564
JacobiSymbol, 343, 359
JacobiTheta, 501
JacobiThetaNull, 501
JacobsonRadical, 532
JBessel, 501
jInvariant, 501, 706
join, 118, 130, 299, 761
JordanForm, 539

K-modules, 507
k-ply transitive, 649
kArc, 796
KBessel, 501
KBessel2, 501
KCubeGraph, 736
Kernel, 156, 520, 537
kinds of help nodes, 248
KMatrixSpace, 55, 508
Knot, 797
KSpace, 507

Labelling, 663
labelling information, 26
λ, 343, 359
language, 303
Laplace, 415
large-prime polynomial factorization, 371
LargestClique, 747
LargestIndependent, 747
Laurent series rings, 321
LaurentSeriesAlgebra, 408
LaurentSeriesRing, 318, 408, 412
LCM, 332, 339, 347, 367
Lcm, 339
le, 57, 299, 331, 593
LeadingCoefficient, 366, 380, 381
LeadingMonomial, 366, 380
LeadingTerm, 366, 380, 381
least common multiple, *see* **LCM**
leaving MAGMA, 5, 293
left ideal, 531
left-normed commutators, 549
LegendreSymbol, 343, 359
Length, 481, 806
length
– of a sequence, 115
– of a string, 258
– of a word, 593
LengthenCode, 814
Leon, 670
less than, *see* **lt**
less than or equal to, *see* **le**
levels of printing, 269, 270, 281
lexicographic ordering
– strings, 260
– words in free group, 594
LexProduct, 744
Li, 498
li, 501
libraries of definitions, 42, 277
lideal, 61, 531
Lift, 708
limits on memory, 295
Line, 760
line continuation, 262
line continuation character, 335
line editor, 290
line regular, 765
linear algebra, 537
linear codes, 801
– encoding and decoding, 827
– group action, 820
– noisy communications channel, 804
– operations, 814
– weight distribution, 818
linear congruences, 349
linear groups, 680
linear spaces, 727, 753, 765
linear transformations, 509, 517
LinearCode, 81, 776, 800, 802, 803
LinearSpace, 766
LineGraph, 743, 800
LineGroup, 732, 799
LineOrbits, 689
Lines, 784, 785
lines, 753
lines of input, editing, 290
LineSet, 784, 785
List, 98, 130
ListCategories, 47
lists, 98, 130, 131
ListSignatures, 218
literal, 23
literal elements, 52
load, 42, 299
loc, 319, 322

local, 299
local identifiers, 181, 207
local scope, 107
LocalHeight, 714
locality, principle of, 64
Localization, 322
Log, 415, 434, 435, 498
log file, 281
LogDerivative, 501
LogGamma, 501
LogIntegral, 501
Logs, 481
long input lines, 262
loop identifiers, 183, 207
loops, 197
– nested, 212
– quick exit, 210
lower case letters, 264
lowest common multiple, 339, *see* **LCM**
LowestCommonMultiple, 339
LowIndexProcess, 622
LowIndexSubgroups, 619
ls, 240, 255
lt, 57, 299, 331, 496, 593
LY, 564

$M(22), M(23), M(24)$, 564
$M, M_{11}, M_{12}, M_{22}, M_{23}, M_{24}$, 564
MacWilliamsTransform, 819, 820
MagForm, 477
MagFormElt, 477
Magma print-level, 269, 270, 281
MAGMA, V, 3
– documentation, 237
– entering, 3
– interrupting, 293
– multiple processes, 287
– online help, 237
– quitting, 5, 293
– user environment, 289
Magma HTML Help Document, 237
`MAGMA_MEMORY_LIMIT`, 296
`MAGMA_PATH`, 277, 278
`MAGMA_STARTUP_FILE`, 279
`MAGMA_USER_SPEC`, 232
magmas, 4, 45
– automatically-created, 49
– finitely-presented, 78
– free, 50, 78
– generators of, 66
– generic, 63
– ideals, 61
– iterable, 107, 206
– normal closures, 61
– operations, 83
– quotients, 75
– submagmas, 61
– warning about, 85
MantissaExponent, 487
manuals on MAGMA, 237
map, 157
mapping form for permutations, 639
mappings, 149, 170
– defined recursively, 163
– images, 152
– preimages, 154
mark, 252, 255
marks, 252, 255
Match, 594
matrices
– constructing, 511
matrix algebras, 527
matrix groups, 13, 543, 671
– natural R-module, 687
matrix modules, 537
matrix rings, 527
matrix spaces, 55, 507, 530, 537, 674
MatrixAlgebra, 81, 392, 439, 440, 527, 530
MatrixGroup, 81, 678
MatrixRing, 72, 81, 527
MatrixUnit, 528
Max, 331, 340
Max, *see* **Maximum**
Maxdeg, 738
Maximal, 269, 270, 281
maximal orders, 449
MaximalIdeals, 532
MaximalLeftIdeals, 532
MaximalOrder, 450
MaximalOvergroup, 611
MaximalPartition, 657
MaximalRightIdeals, 532
Maximum, 116, 331
maximum-distance separable codes, 807
MaxNorm, 368
MCL, 564
meet, 83, 118, 130, 299, 330, 391, 517, 761, 789
memory limits, 295
MergeFields, 467, 468
MergeUnits, 479
Miller-Rabin probabilistic primality test, 354
Min, 331, 340
Min, *see* **Minimum**

Mindeg, 738
Minimal, 269, 270, 281
MinimalField, 460
MinimalIdeals, 532
MinimalInteger, 470
MinimalLeftIdeals, 532
MinimalModel, 710
MinimalNormalSubgroups, 667
MinimalOvergroup, 611
MinimalPartition, 657
MinimalPartitions, 657
MinimalPolynomial, 432, 440, 455, 461, 539
MinimalRightIdeals, 532
Minimize, 460
Minimum, 116, 184, 331
MinimumDistance, 806, 819
MinimumWeight, 806, 819
MinimumWords, 819
MinkowskiBound, 482
miscellaneous features, 289
mod, 24, 54, 299, 332, 339, 367
Modexp, 348, 349
Modinv, 349
ModMatFld, 507
ModMatRng, 507, 510
Modsqrt, 349
ModTupFld, 507
ModTupRng, 507, 510
modular exponentiation, 348
modules, 9, 507, 537
– matrix, 537
Modulus, 351, 352, 496
MoebiusMu, 343, 347, 359
Molien series, 13
MolienSeries, 697
monoids, 79
monoids, finitely-presented, 58
monomial actions on a code, 820
MonomialCoefficient, 366, 380
MonomialGroup, 824
Monomials, 366, 380
MonomialsOfDegree, 697
monomorphisms, 65
MonStg, 50, 258
MonStgElt, 258
MordellWeilGroup, 712
MordellWeilRank, 712
MordellWeilRankBounds, 712
more, 255
Morphism, 516, 534
MP, 485
μ, 343, 359
multi-dimensional sequences, 125
multi-index, 120, 125, 131
multi-sorted algebras, 727
multi-valued expressions, 31
Multinomial, 719
MultipartiteGraph, 736
multiple assignment statements, 31
multiple extensions, 464
multiple MAGMA processes, 287
multiplication of permutations, 640
MultiplicationTable, 454
multiplicative operators, 54, 327
MultiplicativeGroup, 352, 353
MultiplicatorRing, 473
Multiplicity, 101
MultiplyColumn, 521
MultiplyRow, 521
Multisets, 721
multisets, 98
MultisetToSet, 146
mutation, 120
mutation assignment, 31
mutual recursion, 186

NaiveHeight, 714
Names, 143
natural G-sets, 639
natural G-sets and actions, 638
ncl, 61
Ncolumns, 536
ne, 57, 83, 115, 299, 327, 593, 729
near-linear spaces, 727, 753, 765
NearLinearSpace, 766
negative, 54
Neighbours, 738
nested sequences, 125
nested statements, 212
newline character, 262
newlines, 265
NextPrime, 360
NextSubgroup, 623
Ngens, 66, 517, 601, 648
Nickel, Werner, 605
Niemeyer, Alice, 605
nilpotent quotients, 605
nodes, online help, 239
noisy communications channel, 804
non-local identifiers, 181
Norm, 435, 455, 470, 495, 496
normal closures, 61
NormalClosure, 611
NormalElement, 438
NormalForm, 391

Normalize, 512
Normalizer, 611
NormEquation, 349, 455, 456
not, 38, 299
not equal to, *see* **ne**
notadj, 299, 738
notin, 83, 115, 130, 259, 299, 738, 760, 789
notsubset, 83, 115, 299, 760, 789
Nrows, 536
null sets and sequences, 104
NullSpace, 520, 538
number fields, 16, 443
number of elements, *see* cardinality
number theory, 335
– factorization, 353, 356
– primality, 353
NumberField, 318, 444, 445
NumberOfBlocks, 759
NumberOfColumns, 536
NumberOfComponents, 132
NumberOfDivisors, 347
NumberOfGenerators, 66
NumberOfLines, 784
NumberOfPartitions, 722
NumberOfPermutations, 719
NumberOfPoints, 759, 784
NumberOfRows, 536
NumberOfWords, 819
numbers, 35
– complex, 485, 494
– integers, 24, 335
– rationals, 335
– real, 485
numbers, 252, 255
Numerator, 37, 342, 366, 385

O'Nan, 665
objects, 4
ON, 564
on-the-fly invocations, 170
One, 54, 325, 327, 496
online examples, *see* examples
online help, 237
Open, 282
operating system access, 285
operators, 217
– arithmetic, 24
– Boolean, 38
– order of evaluation, 25
– precedence, 301
– relational, 38, 115, 258
or, 38, 299
Orbit, 652, 664, 688
OrbitAction, 654, 665, 690
OrbitActionBounded, 690
OrbitImage, 654, 690
OrbitImageBounded, 690
OrbitKernel, 654, 690
OrbitKernelBounded, 690
Orbits, 652, 664, 689
OrbitsPartition, 750
Order, 83, 349, 352, 435, 440, 536, 551, 624, 712, 738, 763, 792
order of operations, 25
ordering, 102
– strings, 260
– words in free group, 593
orders, 444, 449
orthogonal groups, 680
output, 3, 23, 25
– scrolling, 294
– stored as strings, 270
output file, 281
ovals, 7
OverDimension, 517
overview kind of help node, 248
overwriting files, 282

p-adic fields, 321
p, 489, 492
package files, 221
package spec files, 231
packages, 215
pAdicField, 318, 321
pAdicRing, 318
ParallelClass, 789
ParallelClasses, 789
Parameters, 763
parameters, 191
Parent, 46, 102, 135, 325, 327
parent node in online help, 242
ParentGraph, 738
parentheses, 25, 266
ParentPlane, 784
PARI, 485
ParityCheckMatrix, 806
parser, 278
partial maps, 150
partial quotients, 500
Partitions, 722
partitions, 655
PascalTriangle, 763
passant, 795
PathGraph, 736
pathname of help node, 242

pathnames
– full, relative, 277
pattern-matching, 265
PCGroup, 544
Pencil, 789
permutation actions on a code, 820
permutation groups, 637
– action, 648
– imprimitive, primitive, 655
– projective, 680
– transitive, 649
permutation representations, 638
permutation representations of fp-groups, 613
PermutationGroup, 81, 642, 824
Permutations, 721
pFundamentalUnits, 479
PGammaL, 644, 680
PGammaU, 644
PGL, 644, 680
PGU, 644
ϕ, 343, 359
Pi, 495, 497
π, 497
Pipe, 286
pipe, 286
PlaneAff, 777
PlaneLn, 785
PlaneLnSet, 777, 785
PlaneProj, 777
PlanePt, 785
PlanePtSet, 777, 785
planes, 727
– classical, 782
– finite (affine, projective), 7, 777
PlotkinSum, 814
pmap, 150
Point, 759
point regular, 765
PointDegree, 760
PointDegrees, 759
PointGraph, 776, 800
PointGroup, 775, 799
Points, 755, 759, 760, 784, 785
points, 638, 753
PointSet, 759, 760, 784, 785
PolarToComplex, 495, 496
polycyclic groups, 543
PolygonGraph, 736
Polylog, 498
PolylogD, 498
PolylogDold, 498
PolylogP, 498
polynomial rings, 16, 321
PolynomialAlgebra, 364
PolynomialRing, 318, 364
polynomials
– **polynomials**
–– **better**, 467
– Conway, 433
– for finite fields, 432
POmega, 644
POmegaMinus, 644
POmegaPlus, 644
POpen, 282, 287
Position, 114, 120, 258
PositiveSum, 499, 725
Power, 478
power printing, 436
power series, 13
power, raising to a, 24, 54, 327, 349
PowerSeriesAlgebra, 408
PowerSeriesRing, 318, 408, 412
PowerSet, 721
pPrimaryComponent, 633, 634
pPrimaryInvariants, 633, 634
pQuotient, 603
pRadical, 473
pRank, 792
precedence of operators, 25, 301
Precision, 412, 413, 486, 487
precision, 407, 411
– arbitrary, 36
– changing, 493
preimages, 76, 154
presentation, 78
presentation for subgroup of fp-group, 615
presentations for concrete groups, 600
previous sessions, returning, 285
previously printed values, recalling, 40
PreviousPrime, 360
primality, 353
primary invariants, 13
PrimaryDecomposition, 401
PrimaryInvariantFactors, 539
PrimaryInvariants, 633, 634, 697
PrimaryRationalForm, 539
Prime, 401
prime field, 419
prime numbers, generating, 360
PrimeField, 330, 424
PrimeForm, 478
PrimeRing, 330
primitive, 422
primitive elements, 443

primitive groups, 655, 665
PrimitiveElement, 351, 352, 423, 434, 444
PrimitivePart, 368
PrimitivePolynomial, 432
PrimitiveRoot, 349, 351
principal argument of complex number, 496
principal value of a function, 32
principle of locality, 64
print, 23, 269, 299
printed values, recalling, 40
printf, 272, 299
PrintFileMagma, 282
printing
– formatted, 272
printing an object to a file, 281
printing an object to a string, 270
printing finite field elements, 436
printing text, 26
printname, 48
proc, 176
procedure, 299
procedure assignment statements, 187
procedure expressions
– constructor form, 176
– statement form, 177, 187
procedures, 167
– **return**, 180
– defined recursively, 183, 188
– signatures, 216
– user-defined, 176
– variable reference arguments, 116
process, 622
process space, 28
processes
– MAGMA, 287
– external, 285
products, 24, 54, 327
– Cartesian, 132
– of finitely-presented groups, 599
– of permutation groups, 645
progressions, arithmetic, 105
projective groups, 680
projective planes, 7, 777
ProjectiveEmbedding, 794
ProjectiveGammaLinearGroup, 644
ProjectiveGammaUnitaryGroup, 644
ProjectiveGeneralLinearGroup, 644
ProjectiveGeneralUnitaryGroup, 644
ProjectiveOmegaGroup, 644
ProjectiveOmegaMinusGroup, 644
ProjectiveOmegaPlusGroup, 644
ProjectivePlane, 778, 780, 782
ProjectiveSigmaLinearGroup, 644
ProjectiveSigmaSymplecticGroup, 644
ProjectiveSigmaUnitaryGroup, 644
ProjectiveSpecialLinearGroup, 644
ProjectiveSpecialUnitaryGroup, 644
ProjectiveSuzukiGroup, 644
ProjectiveSymplecticGroup, 644
prompt symbols, 3, 88, 199
Prune, 120, 121, 124, 131, 140
Ψ, 501
Psi, 501
PSigmaL, 644, 680
PSigmaSp, 644
PSigmaU, 644
PSL, 644, 680
PSp, 644
PSU, 644
PSz, 644
PunctureCode, 814
Put, 282
Puts, 282
pwd, 241, 255

QRCode, 810
quadratic forms, 476
QuadraticField, 446
QuadraticForm, 797
quantifiers, existential and universal, 127
quit
– saving workspace, 285
quit, 5, 255, 299
quo, 75, 319, 323, 351, 515, 597, 629
quotient, 54, 327, 332, 339
quotient groups, 601
quotient magmas, 75
quotient, remainder, *see* **Quotrem**
quotients
– quotient rings as residue class fields, 323
Quotrem, 54, 332, 339, 367

R-(tuple) spaces, 510
R-matrix spaces, 507, 510

R-modules, 507
R-spaces, 507
radians, 498
Radical, 401
RadicalDecomposition, 401
radix representation of integers, 340
raising to a power, 24, 54, 327, 349
RamificationIndex, 470
Random, 83, 126, 132, 325, 327, 340, 511, 594, 635, 759, 760, 784, 789
random, 126, 299
RandomBits, 340
RandomGraph, 736
RandomProcess, 635
RandomSchreier, 670, 685
RandomTree, 736
Rank, 364, 392, 519, 536, 712
RankBounds, 712
rational approximations, 500
rational arithmetic, 36
rational field, 443
rational function fields, 323, 365, 385
rational functions, 13
RationalField, 50, 318, 336
RationalForm, 539
RationalReconstruction, 341, 342
Rationals, 336
rationals, 35
Re, 494, 496
Read, 281
read, 274, 299
readi, 274, 299
reading data files, 281
Real, 494, 496
real fields, 485
real numbers, 35
RealField, 50, 318, 485, 491
Rec, 98
rec, 98, 141
recalling previously printed values, 40
recformat, 98, 141
RecFrmt, 98
recognizing alternating and symmetric groups, 665
record formats, 98
records, 98, 141
rectangular matrices, 530, 674
recursion, 112, 163, 183, 188, 725
– mutual, 186
reduced basis, 507
reduced form for rationals, 336
ReducedDiscriminant, 444, 450
Reduction, 478
reduction, 117, 659
ReductionStep, 478
Reductum, 366
Reed-Solomon code decoding, 831
ReedMullerCode, 809
ReedSolomonCode, 809
reference arguments, 169
reference counts, 180
reference to help node, 244
reflection groups, 13
Regexp, 265
`regexp`, 267
regular
– point, line, 765
regular expressions, 265
Regulator, 474, 481, 714
relation normal subgroup, 595
relational operators, 38, 57, 115, 258
RelationMatrix, 482
Relations, 601
relations for group presentation, 597
relative extensions, 455, 464
relative pathnames, 277
RelativePrecision, 408
remainder, 54, 332, 339
Remove, 120, 121, 124
Rep, 83, 119, 126, 132, 325, 327, 759, 760, 784, 789
rep, 126
repeat, 202, 299
repeated elements, aggregates with, 99
ReplicationNumber, 763
representation of elements, 58
RepresentationMatrix, 454
representations
– permutation, 638
representations of fp-groups, 613
Representative, 325, 327
representative elements, 83, 119, 126
require, 299
requirege, 299
requirerange, 299
reserved words, 27, 299
Residual, 766
ResidueClassField, 323
ResidueClassRing, 351
resolution, 765
resolvable, 765
restore, 285, 299
RestrictedPartitions, 722
RestrictField, 526
Restriction, 766
Resultant, 372

return
– functions, 174
– procedures, 180
return, 299
return character, 262
return values of functions, 169, 172
returning to a previous session, 285
Reverse, 120, 121, 124
Reversion, 415
Rewind, 282
Rewrite, 616
ReynoldsOperator, 15
rideal, 61, 531
Riemann zeta function, 501
right ideal, 531
ring of integers, 322, 365, 385, 449
rings, 317
– integer, 36, 335
– Laurent series, 407
– matrix, 72, 527
– polynomial, 16, 364
– power series, 407
RMatrixSpace, 510
RngInt, 50
RngIntRes, 351
RngMPol, 377
RngUPol, 364
Root, 435, 457, 495
root
– integer part, 340
root node in online help, 242
RootOfUnity, 428, 448, 458
Roots, 374, 375
RootsNonExact, 376
Rotate, 120, 121, 124, 176
RotateWord, 594
Round, 37, 342, 495
round brackets, *see* parentheses
row and column operations, 521
row vectors, 507
row-major order, 509, 527, 673
RowSpace, 519, 538
RSpace, 510, 687
RU, 564
running programs, 42, 277

save, 285, 299
scalar matrices, 528
scalar multiplication, 512
ScalarMatrix, 528
Schreier, 669
Schreier algorithms for BSGS, 685
SchreierGenerators, 611
SchreierSystem, 611
scientific notation, 36, 486
scope
– local, 107
scope of identifiers, 181, 207
Scott, 665
scrolling output, 294
sdiff, 118, 130, 299
search, 245, 255
Sec, 498
secant, 795
Sech, 498
SecondaryInvariants, 697
section kind of help node, 248
Seek, 282
select, 92, 299
Self, 112, 725
self-dual, 765, 792
semicolon, 3
semigroups, 79
semigroups, finitely-presented, 58
Seqelt, 435
SeqEnum, 98
Seqint, 340
Seqlist, 146
Seqset, 146
sequences
– defined recursively, 112, 725
– empty, null, 104
– factorization, 136, 345
– incomplete (undefined terms), 113
– multi-dimensional, 125
SequenceToElement, 435
SequenceToList, 146
SequenceToMultiset, 146
SequenceToSet, 146
series
– Molien, 13
– power, 13
series, summation of infinite, 499, 725
Set, 83, 132, 331, 648, 663, 760, 789
SetAssertions, 196
SetCart, 98, 132
SetDefaultRealField, 487, 491
SetEchoInput, 280
SetEnum, 98
SetFormal, 98
SetHistorySize, 291
SetIndx, 98
SetLogFile, 281
SetMemoryLimit, 296
SetMulti, 98
SetOrderMaximal, 481

SetOrderTorsionUnits, 481
SetOrderUnitsAreFundamental, 481
SetOutputFile, 281
SetPath, 278
SetPreviousSize, 41
SetPrompt, 89
SetRows, 294
sets
– empty, null, 104
– formal, 129
– infinite, 129, 130
Setseq, 146
SetToIndexedSet, 146
SetToMultiset, 146
SetToSequence, 146
SetVerbose, 271
SetViMode, 291
sharing constants in packages, 233
shell-escape, 286
ShortenCode, 814
show, 240, 255
ShowIdentifiers, 28
ShowPrevious, 42
ShowValues, 28
σ, 343, 359
Sign, 340, 342, 368, 495
Signature, 444–446, 448, 450
signatures, 215, 216
significant decimal digits, 487
simple, 753, 765
simplified/unsimplified forms of elements, 57
Simplify, 468, 618, 766
Sims, 669, 670
SimsSchreier, 669
simu ltaneous linear equations, 386
simultaneous linear congruences, 349
simultaneous linear equations, 522
Sin, 37, 415, 498
Sincos, 415, 498
SingerDifferenceSet, 769
single online help request, 238, 255
single use rule, 183
Sinh, 415, 498
Size, 738
size, *see* cardinality
SL, 680
Slope, 790
small-prime polynomial factorization, 371
Smith Normal Form, 33
SmithForm, 33
Socle, 667
socle
– abelian, 666
SocleFactor, 667
SocleFactors, 667
SocleImage, 667
SocleKernel, 667
SocleSeries, 667
soft memory limits, 295
soluble quotients, 605
SolubleSchreier, 670
Solution, 349, 352, 522
Solving Problems with Magma, 237
Sort, 120, 121, 124
sorts, 727
Sp, 681
spaces
– linear, near-linear, 753
– matrix, 507, 530, 537, 674
– tuple, 507
– vector, 507, 537, 674
spec files, 222, 231
special functions, 500
special prompt symbols, 88
SpecialLinearGroup, 680
SpecialUnitaryGroup, 681
Spencer, Henry, 267
Sphere, 746
Split, 264, 265
splitting fields, 16
SplittingField, 426, 428
SPolynomial, 391
sporadic groups, 564
Sprint, 270
Sqrt, 37, 352, 415, 435, 457, 495, 496
square brackets, *see* brackets
square root, 435, 495, 496
– integer part, 340
SquareFreeFactorization, 371, 383
SquareRoot, 352, 415
SrivastavaCode, 812
Stabilizer, 652, 664, 689, 750
standard basis, 508, 509
standard G-sets, 638
standard notation, 36, 486
standard start-up files, 279
StandardForm, 806, 824
start-up files and identifiers, 278
statement form of function, 174, 187
statement form of procedure, 177, 187
statements, 3
– **case**, 92
– **if**, 87, 89

– **print**, 269
– assignment, 26, 29
– conditional, 87
– function, procedure assignment, 187
– iterative, 197
– mutation assignment, 31
– nested, 212
Steiner, 754, 765
StirlingFirst, 719
StirlingSecond, 719
storing information, 26
string, 222
– conversion/transfer functions, 263
string structure, 258
Strings, 50, 258
strings, 26, 257
– analysis, 263
– operations, 258
– printed form of objects, 270
– special characters, 262
StringToCode, 264
StringToInteger, 263
StringToIntegerSequence, 263
strong pseudoprime primality test, 354
structural computation, 65
structures, 4
structures, incidence, 727, 753
students
– customizing MAGMA, 233
SU, 681
sub, 61, 319, 350, 428, 466, 515, 629, 642
Subfields, 467, 468
subfields, 428, 464
SubfieldSubcode, 814
subgroups, 16
submagmas, 61
Submatrix, 514
Subsequences, 721
subset, 83, 115, 299, 391, 760, 761, 789
Subsets, 119, 721
Substitute, 594
Substring, 258
substrings, 258, 264
Subword, 594
Sum, 766
sum, 54, 327
summation of infinite series, 499, 725
SumNorm, 368
SumOfDivisors, 347
Support, 513, 641, 648, 663, 759, 760, 762, 784, 786, 789, 805
support, 648
SUZ, 564
SuzukiGroup, 681
swap space, 295
SwapColumns, 521
SwapRows, 521
Switch, 743
Sym, 81, 548, 639, 643
SymKod, 703
symmetric, 754, 765
symmetric difference, 118, 130
symmetric groups, 639
– recognizing, 665
SymmetricGroup, 548
SymmetricSquare, 534, 536
symmetries of cube, 646
symplectic groups, 680
SymplecticGroup, 681
Syndrome, 805
syndrome decoding, 829, 831
SyndromeSpace, 806
synonyms, 279
System, 285
system intrinsics, 167, 215
systems of imprimitivity, 655
Sz, 681

tab character, 262
tab completion, 250, 256, 289
Tan, 415, 498
Tangent, 796
tangent, 795
Tanh, 415, 498
Tell, 282
Tempname, 287
temporary name, 287
tensor products, 9
TensorProduct, 534, 536, 645, 682, 744
Term, 381
term algebra, 50
Terms, 380, 381
text strings, *see* strings
TH, 564
then, 87, 299
θ, 501
throwaway identifier, 27, 32, 299
Tietze transformations, 616
tilde expansion, 277
time, 33, 294, 299
timing MAGMA, 294
to, 299
Todd-Coxeter, 670
ToddCoxeter, 607

ToddCoxeterSchreier, 670, 685
top level, 43, 278
TorsionFreeRank, 631, 634
TorsionFreeSubgroup, 632, 634
TorsionInvariants, 631, 634
TorsionSubgroup, 474, 479, 632, 634, 712
TotalDegree, 366, 380
Trace, 435, 455, 470, 536
TraceMatrix, 454
TrailingCoefficient, 366, 380, 381
TrailingTerm, 366, 380, 381
transcendental functions, 415, 497
transfer functions, 146
TransformationMatrix, 453, 454, 471
transformations, linear, 509, 517
transitive, 649
transitive constituents, 654, 657
Transitivity, 650, 664
transmission errors in codewords, 829
Transpose, 513, 520, 536
Transversal, 517, 611, 651, 652
trigonometric functions, 415, 498
trivial, 765
trivial blocks, 655
true, 38, 50, 299
Truncate, 37, 342, 408, 495
Tup, 98
tuple constructors, 134
tuple spaces, 507
tuples, 98, 132
TupleToList, 146
Tuplist, 146
two-sided ideal, 531
TwoElement, 471
TwoElementNormal, 471
Type, 46, 325, 327
type-checking, 170

undef, 113
Undefine, 113
undefined terms of enumerated sequences, 113
underscore, 27, 32, 299
undirected graphs, *see* graphs
unfrozen packages, 224
Ungetc, 282
uniform, 753, 765
Union, 744, 766
union, 118, 130
unital, 796
UnitalFeet, 797
unitary groups, 680
UnitGroup, 330, 352, 353, 474, 479
UnitRank, 474
unity, roots of, 458
universal quantifiers, 127
Universe, 102
universe, 98, 102, 109
unmark, 252, 255
UnsetLogFile, 281
UnsetOutputFile, 281
until, 202, 299
up, 242, 255
updating packages, 224
upper case letters, 264
user environment, 289
user errors, 225
user interface, 237
user intrinsics, 167, 215, 218
user-defined functions and procedures, 167
user-input, 274
using MAGMA, 3

Valuation, 408, 415, 473
ValuationRing, 322
value arguments, 169
value class of identifiers, 183, 208
values, 23
variable reference arguments, 116
variables, *see* identifiers
varieties, 47
vector spaces, 507, 537, 674
vectors
– constructing, 510
VectorSpace, 81, 392, 437, 438, 448, 449, 507, 806
VectorSpaceWithBasis, 523
verbose printing, 271
Verify, 670, 686
VertexSet, 737
VerticalJoin, 514
Vertices, 737
vertices
– subtraction, 739
VertSet, 737
VI editor, 290

walk, 247, 255
WeberF, 501
WeberF2, 501
Weight, 805
WeightDistribution, 819
WeightEnumerator, 820

when, 299
where, 33, 109, 241, 255, 299
while, 199, 299
whitespace, 265
WittDesign, 767
word equality, 56
word problem, 58
Words, 819
words, 59, 592, 593
workspace, 28, 285
WreathProduct, 645, 682
Write, 282
writing to data files, 281

XGCD, 332, 339, 368
xor, 38, 299

Zassenhaus' factorization algorithm, 371
Zech logarithms, 435
Zero, 54, 325, 327, 496, 528
Zeros, 481
ζ, 501
ZetaFunction, 501