

HANDBOOK OF MAGMA FUNCTIONS

Volume 1

Language, Aggregates and Semigroups

John Cannon Wieb Bosma

Claus Fieker Allan Steel

Editors

Version 2.19

Sydney

December 17, 2012

HANDBOOK OF MAGMA FUNCTIONS

Editors:

John Cannon Wieb Bosma Claus Fieker Allan Steel

Handbook Contributors:

Geoff Bailey, Wieb Bosma, Gavin Brown, Nils Bruin, John Cannon, Jon Carlson, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Tim Dokchitser, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Volker Gebhardt, Sergei Haller, Michael Harrison, Florian Hess, Derek Holt, David Howden, Al Kasprzyk, Markus Kirschmer, David Kohel, Axel Kohnert, Dimitri Leemans, Paulette Lieby, Graham Matthews, Scott Murray, Eamonn O'Brien, Dan Roozemon, Ben Smith, Bernd Souvignier, William Stein, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, Alexa van der Waall, Paul van Wamelen, Helena Verrill, John Voight, Mark Watkins, Greg White

Production Editors:

Wieb Bosma Claus Fieker Allan Steel Nicole Sutherland

HTML Production:

Claus Fieker Allan Steel

PREFACE

The computer algebra system MAGMA is designed to provide a software environment for computing with the structures which arise in areas such as algebra, number theory, algebraic geometry and (algebraic) combinatorics. MAGMA enables users to define and to compute with structures such as groups, rings, fields, modules, algebras, schemes, curves, graphs, designs, codes and many others. The main features of MAGMA include:

- *Algebraic Design Philosophy:* The design principles underpinning both the user language and system architecture are based on ideas from universal algebra and category theory. The language attempts to approximate as closely as possible the usual mathematical modes of thought and notation. In particular, the principal constructs in the user language are set, (algebraic) structure and morphism.
- *Explicit Typing:* The user is required to explicitly define most of the algebraic structures in which calculations are to take place. Each object arising in the computation is then defined in terms of these structures.
- *Integration:* The facilities for each area are designed in a similar manner using generic constructors wherever possible. The uniform design makes it a simple matter to program calculations that span different classes of mathematical structures or which involve the interaction of structures.
- *Relationships:* MAGMA provides a mechanism that manages “relationships” between complex bodies of information. For example, when substructures and quotient structures are created by the system, the natural homomorphisms that arise are always stored. These are then used to support automatic coercion between parent and child structures.
- *Mathematical Databases:* MAGMA has access to a large number of databases containing information that may be used in searches for interesting examples or which form an integral part of certain algorithms. Examples of current databases include factorizations of integers of the form $p^n \pm 1$, p a prime; modular equations; strongly regular graphs; maximal subgroups of simple groups; integral lattices; $K3$ surfaces; best known linear codes and many others.
- *Performance:* The intention is that MAGMA provide the best possible performance both in terms of the algorithms used and their implementation. The design philosophy permits the kernel implementor to choose optimal data structures at the machine level. Most of the major algorithms currently installed in the MAGMA kernel are state-of-the-art and give performance similar to, or better than, specialized programs.

The theoretical basis for the design of MAGMA is founded on the concepts and methodology of modern algebra. The central notion is that of an *algebraic structure*. Every object created during the course of a computation is associated with a unique parent algebraic structure. The *type* of an object is then simply its parent structure.

Algebraic structures are first classified by *variety*: a variety being a class of structures having the same set of defining operators and satisfying a common set of axioms. Thus, the collection of all rings forms a variety. Within a variety, structures are partitioned into *categories*. Informally, a family of algebraic structures forms a category if its members all share a common *representation*. All varieties possess an *abstract* category of structures (the finitely presented structures). However, categories based on a concrete representation are at least as important as the abstract category in most varieties. For example, within the variety of algebras, the family of finitely presented algebras constitutes an abstract category, while the family of matrix algebras constitutes a concrete category.

MAGMA comprises a novel user programming language based on the principles outlined above together with program code and databases designed to support computational research in those areas of mathematics which are algebraic in nature. The major areas represented in MAGMA V2.19 include group theory, ring theory, commutative algebra, arithmetic fields and their completions, module theory and lattice theory, finite dimensional algebras, Lie theory, representation theory, homological algebra, general schemes and curve schemes, modular forms and modular curves, L -functions, finite incidence structures, linear codes and much else.

This set of volumes (known as the Handbook) constitutes the main reference work on MAGMA. It aims to provide a comprehensive description of the MAGMA language and the mathematical facilities of the system. In particular, it documents every function and operator available to the user. Our aim (not yet achieved) is to list not only the functionality of the MAGMA system but also to show how the tools may be used to solve problems in the various areas that fall within the scope of the system. This is attempted through the inclusion of tutorials and sophisticated examples. Finally, starting with the edition corresponding to release V2.8, this work aims to provide some information about the algorithms and techniques employed in performing sophisticated or time-consuming operations. It will take some time before this goal is fully realised.

We give a brief overview of the organization of the Handbook.

- Volume 1 contains a terse summary of the language together with a description of the central datatypes: sets, sequences, tuples, mappings, etc. An index of all intrinsics appears at the end of the volume.
- Volume 2 deals with basic rings and linear algebra. The rings include the integers, the rationals, finite fields, univariate and multivariate polynomial rings as well as real and complex fields. The linear algebra section covers matrices and vector spaces.
- Volume 3 covers global arithmetic fields. The major topics are number fields, their orders and function fields. More specialised topics include quadratic fields, cyclotomic fields and algebraically closed fields.
- Volume 4 is concerned with local arithmetic fields. This covers p -adic rings and their extension and power series rings including Laurent and Puiseux series rings,

- Volume 5 describes the facilities for finite groups and, in particular, discusses permutation groups, matrix groups and finite soluble groups defined by a power-conjugate presentation. A chapter is devoted to databases of groups.
- Volume 6 describes the machinery provided for finitely presented groups. Included are abelian groups, general finitely presented groups, polycyclic groups, braid groups and automatic groups. This volume gives a description of the machinery provided for computing with finitely presented semigroups and monoids.
- Volume 7 is devoted to aspects of Lie theory and module theory. The Lie theory includes root systems, root data, Coxeter groups, reflection groups and Lie groups.
- Volume 8 covers algebras and representation theory. Associative algebras include structure-constant algebras, matrix algebras, basic algebras and quaternion algebras. Following an account of Lie algebras there is a chapter on quantum groups and another on universal enveloping algebras. The representation theory includes group algebras, $K[G]$ -modules, character theory, representations of the symmetric group and representations of Lie groups.
- Volume 9 covers commutative algebra and algebraic geometry. The commutative algebra material includes constructive ideal theory, affine algebras and their modules, invariant rings and differential rings. In algebraic geometry the main topics are schemes, sheaves and toric varieties. Also included are chapters describing specialised machinery for curves and surfaces.
- Volume 10 describes the machinery pertaining to arithmetic geometry. The main topics include the arithmetic properties of low genus curves such as conics, elliptic curves and hyperelliptic curves. The volume concludes with a chapter on L -series.
- Volume 11 is concerned with modular forms.
- Volume 12 covers various aspects of geometry and combinatorial theory. The geometry section includes finite planes, finite incidence geometry and convex polytopes. The combinatorial theory topics comprise enumeration, designs, Hadamard matrices, graphs and networks.
- Volume 13 is primarily concerned with coding theory. Linear codes over both fields and finite rings are considered at length. Further chapters discuss machinery for AG-codes, LDPC codes, additive codes and quantum error-correcting codes. The volume concludes with short chapters on pseudo-random sequences and on linear programming.

Although the Handbook has been compiled with care, it is possible that the semantics of some facilities have not been described adequately. We regret any inconvenience that this may cause, and we would be most grateful for any comments and suggestions for improvement. We would like to thank users for numerous helpful suggestions for improvement and for pointing out misprints in previous versions.

The development of MAGMA has only been possible through the dedication and enthusiasm of a group of very talented mathematicians and computer scientists. Since 1990, the principal members of the MAGMA group have included: Geoff Bailey, Mark Bofinger, Wieb Bosma, Gavin Brown, John Brownie, Herbert Brückner, Nils Bruin, Steve Collins, Scott Contini, Bruce Cox, Brendan Creutz, Steve Donnelly, Willem de Graaf, Andreas-Stephan Elsenhans, Claus Fieker, Damien Fisher, Alexandra Flynn, Volker Gebhardt, Katharina Geißler, Sergei Haller, Michael Harrison, Emanuel Herrmann, Florian Heß, David Howden, Al Kasprzyk, David Kohel, Paulette Lieby, Graham Matthews, Scott Murray, Anne O’Kane, Catherine Playoust, Richard Rannard, Colva Roney-Dougal, Dan Roozemon, Andrew Solomon, Bernd Souvignier, Ben Smith, Allan Steel, Damien Stehlé, Nicole Sutherland, Don Taylor, Bill Unger, John Voight, Alexa van der Waall, Mark Watkins and Greg White.

John Cannon
Sydney, December 2012

ACKNOWLEDGEMENTS

The Magma Development Team

Current Members

Geoff Bailey, BSc (Hons) (Sydney), [1995-]: Main interests include elliptic curves (especially those defined over the rationals), virtual machines and computer language design. Has implemented part of the elliptic curve facilities especially the calculation of Mordell-Weil groups. Other main areas of contribution include combinatorics, local fields and the MAGMA system internals.

John Cannon, Ph.D. (Sydney), [1971-]: Research interests include computational methods in algebra, geometry, number theory and combinatorics; the design of mathematical programming languages and the integration of databases with Computer Algebra systems. Contributions include overall concept and planning, language design, specific design for many categories, numerous algorithms (especially in group theory) and general management.

Brendan Creutz, Ph.D. (Jacobs University Bremen) [2011-]: Primary research interests are in arithmetic geometry. Main contributions focus on descent obstructions to the existence of rational points on curves and torsors under their Jacobians. Currently developing a package for cyclic covers of the projective line.

Steve Donnelly, Ph.D. (Georgia) [2005-]: Research interests are in arithmetic geometry, particularly elliptic curves and modular forms. Major contributions include descent methods for elliptic curves (including over function fields) and Cassels-Tate pairings, classical modular forms of half-integral weight, Hilbert modular forms and fast algorithms for definite quaternion algebras. Currently working on Hilbert modular forms, and elliptic curves over number fields.

Andreas-Stephan Elsenhans, Ph.D. (Göttingen) [2012-]: Main research interests are in arithmetic and algebraic geometry, particularly cubic and K3 surfaces. Main contributions focus on cubic surfaces from the arithmetic and the algebraic point of view. Currently working on computation of invariants.

Michael Harrison, Ph.D. (Cambridge) [2003-]: Research interests are in number theory, arithmetic and algebraic geometry. Implemented the p -adic methods for counting points on hyperelliptic curves and their Jacobians over finite fields: Kedlaya's method and the modular parameter method of Mestre. Currently working on machinery for general surfaces and cohomology for projective varieties.

David Howden, Ph.D. (Warwick) [2012-]: Primary research interests are in computational group theory. Main contributions focus on computing automorphism groups and isomorphism testing for soluble groups.

Allan Steel, Ph.D. (Sydney), [1989-]: Has developed many of the fundamental data structures and algorithms in MAGMA for multiprecision integers, finite fields, matrices and modules, polynomials and Gröbner bases, aggregates, memory management, environmental features, and the package system, and has also worked on the MAGMA language interpreter. In collaboration, he has developed the code for lattice theory (with Bernd Souvignier), invariant theory (with Gregor Kemper) and module theory (with Jon Carlson and Derek Holt).

Nicole Sutherland, BSc (Hons) (Macquarie), [1999-]: Works in the areas of number theory and algebraic geometry. Developed the machinery for Newton polygons and lazy power series and contributed to the code for local fields, number fields, modules over Dedekind domains, function fields, schemes and has worked on aspects of algebras.

Don Taylor, D.Phil. (Oxford), [2010-] Research interests are in reflection groups, finite group theory, and geometry. Implemented algorithms for complex reflection groups and complex root data. Contributed to the packages for Chevalley groups and groups of Lie type. Currently developing algorithms for classical groups of isometries, Clifford algebras and spin groups.

Bill Unger, Ph.D. (Sydney), [1998-]: Main area of interest is computational group theory, with particular emphasis on algorithms for permutation and matrix groups. Implemented many of the current permutation and matrix group algorithms for MAGMA, in particular BSGS verification, solvable radical and chief series algorithms. Recently discovered a new method for computing the character table of a finite group.

Mark Watkins, Ph.D. (Athens, Ga), [2003, 2004-2005, 2008-]: Works in the area of number theory, particularly analytic methods for arithmetic objects. Implemented a range of analytic tools for the study of elliptic curves including analytic rank, modular degree, Heegner points and (general) point searching methods. Also deals with conics, lattices, modular forms, and descent machinery over the rationals.

Former Members

Wieb Bosma, [1989-1996]: Responsible for the initial development of number theory in MAGMA and the coordination of work on commutative rings. Also has continuing involvement with the design of MAGMA.

Gavin Brown, [1998-2001]: Developed code in basic algebraic geometry, applications of Gröbner bases, number field and function field kernel operations; applications of Hilbert series to lists of varieties.

Herbert Brückner, [1998–1999]: Developed code for constructing the ordinary irreducible representations of a finite soluble group and the maximal finite soluble quotient of a finitely presented group.

Nils Bruin, [2002–2003]: Contributions include Selmer groups of elliptic curves and hyperelliptic Jacobians over arbitrary number fields, local solubility testing for arbitrary projective varieties and curves, Chabauty-type computations on Weil-restrictions of elliptic curves and some algorithms for, and partial design of, the differential rings module.

Bruce Cox, [1990–1998]: A member of the team that worked on the design of the MAGMA language. Responsible for implementing much of the first generation MAGMA machinery for permutation and matrix groups.

Claus Fieker, [2000-2011]: Formerly a member of the KANT project. Research interests are in constructive algebraic number theory and, especially, relative extensions and computational class field theory. Main contributions are the development of explicit algorithmic class field theory in the case of both number and function fields and the computation of Galois groups.

Damien Fisher, [2002-2006]: Implemented a package for p -adic rings and their extensions and undertook a number of extensions to the MAGMA language.

Alexandra Flynn, [1995–1998]: Incorporated various Pari modules into MAGMA, and developed much of the machinery for designs and finite planes.

Volker Gebhardt, [1999–2003]: Author of the MAGMA categories for infinite polycyclic groups and for braid groups. Other contributions include machinery for general finitely presented groups.

Katharina Geißler, [1999–2001]: Developed the code for computing Galois groups of number fields and function fields.

Willem de Graaf, [2004-2005]: Contributed functions for computing with finite-dimensional Lie algebras, finitely-presented Lie algebras, universal enveloping algebras and quantum groups.

Sergei Haller, [2004, 2006-2007]: Developed code for many aspects of Lie Theory. Of particular note was his work on the construction of twisted groups of Lie type and the determination of conjugacy classes of elements in the classical groups (jointly with Scott Murray).

Emanuel Herrmann, [1999]: Contributed code for finding S -integral points on genus 1 curves (not elliptic curves).

Florian Heß, [1999–2001]: Developed a substantial part of the algebraic function field module in MAGMA including algorithms for the computation of Riemann-Roch spaces and class groups. His most recent contribution (2005) is a package for computing all isomorphisms between a pair of function fields.

Alexander Kasprzyk, [2009–2010]: Developed the toric geometry and polyhedra packages (along with Gavin Brown and Jaroslaw Buczynski).

David Kohel, [1999–2002]: Contributions include a model for schemes (with G Brown); algorithms for curves of low genus; implementation of elliptic curves, binary quadratic forms, quaternion algebras, Brandt modules, spinor genera and genera of lattices, modular curves, conics (with P Lieby), modules of supersingular points (with W Stein), Witt rings.

Paulette Lieby, [1999–2003]: Contributed to the development of algorithms for algebraic geometry, abelian groups and incidence structures. Developed datastructures for multigraphs and implemented algorithms for planarity, triconnectivity and network flows.

Graham Matthews, [1989–1993]: Involved in the design of the MAGMA semantics, user interface, and internal organisation.

Scott Murray, [2001–2002, 2004–2010]: Implemented algorithms for element operations in split groups of Lie type, representations of split groups of Lie type, split Cartan subalgebras of modular Lie algebras, and Lang’s Theorem in finite reductive groups. More recently implemented solutions to conjugacy problems in the classical groups (with S. Haller and D. Taylor).

Catherine Playoust, [1989–1996]: Wrote extensive documentation and implemented an early help system. Contributed to system-wide consistency of design and functionality. Also pioneered the use of MAGMA for teaching undergraduates.

Richard Rannard, [1997–1998]: Contributed to the code for elliptic curves over finite fields including a first version of the SEA algorithm.

Colva M. Roney-Dougal, [2001–2003]: Completed the classification of primitive permutation groups up to degree 999 (with Bill Unger). Also undertook a constructive classification of the maximal subgroups of the classical simple groups.

Dan Roozmond, [2010–2012]: Research focused on the computational aspects of Lie theory. Ported algorithms for the Weight Multisets from LiE to Magma and developed a number of algorithms for reductive Lie algebras, particularly over fields of small characteristic.

Michael Slattery, [1987–2006]: Contributed a large part of the machinery for finite soluble groups including subgroup lattice and automorphism group.

Ben Smith, [2000–2003]: Contributed to an implementation of the Number Field Sieve and a package for integer linear programming.

Bernd Souvignier, [1996–1997]: Contributed to the development of algorithms and code for lattices, local fields, finite dimensional algebras and permutation groups.

Damien Stehlé, [2006, 2008–2010]: Implemented the proveably correct floating-point LLL algorithm together with a number of fast non-rigorous variants. Also developed a fast method for enumerating short vectors.

John Voight, [2005–2006]: Implemented algorithms for quaternion algebras over number fields, associative orders (with Nicole Sutherland), and Shimura curves.

Alexa van der Waall, [2003]: Implemented the module for differential Galois theory.

Paul B. van Wamelen, [2002–2003]: Implemented analytic Jacobians of hyperelliptic curves in MAGMA.

Greg White, [2000–2006]: Contributions include fast minimum weight determination, linear codes over Z/mZ , additive codes, LDPC codes, quantum error-correcting codes, and a database of best known linear codes (with Cannon and Grassl).

External Contributors

The MAGMA system has benefited enormously from contributions made by many members of the mathematical community. We list below those persons and research groups who have given the project substantial assistance either by allowing us to adapt their software for inclusion within MAGMA or through general advice and criticism. We wish to express our gratitude both to the people listed here and to all those others who participated in some aspect of the MAGMA development.

Algebraic Geometry

A major package for algebraic surfaces providing formal desingularization, the calculation of adjoints, and rational parameterization was developed by **Tobias Beck** (RICAM, Linz). He also implemented a package for computing with algebraic power series. This work was done while he was a student of **Josef Schicho**.

The machinery for working with Hilbert series of polarised varieties and the associated databases of K3 surfaces and Fano 3-folds has been constructed by **Gavin Brown** (Warwick).

Jaroslaw Buczynski (Texas A&M), along with **Gavin Brown** and **Alexander Kasprzyk**, developed the toric geometry and polyhedra packages.

Jana Pilnikova (Univerzita Komenskeho, Bratislava) (while a student of **Josef Schicho** in Linz) contributed code for the parameterization of degree 8 and 9 Del Pezzo surfaces, jointly written with **Willem de Graaf**.

Miles Reid (Warwick) has been heavily involved in the design and development of a database of K3 surfaces within MAGMA.

Josef Schicho (RICAM, Linz) has played a major role in the design and implementation of the algebraic surfaces package. In particular, Josef has also implemented several of the modules for rational surface parameterization.

Andrew Wilson (Edinburgh) has contributed a package to compute the log canonical threshold for singular points on a curve.

Arithmetic Geometry

The method of Chabauty for finding points on elliptic curves was originally implemented by **Nils Bruin** in 2003 while a member of the MAGMA group. In 2009 Nils improved it considerably by combining it with *Mordell-Weil sieving*.

Two-cover-descent has been implemented by **Nils Bruin** (Simon Fraser) for hyperelliptic curves. Given the Jacobian of a genus 2 curve, Nils has also provided code to compute all $(2, 2)$ -isogenous abelian surfaces.

The MAGMA facility for determining the Mordell-Weil group of an elliptic curve over the rational field is based on the `MWRANK` programs of **John Cremona** (Nottingham).

John Cremona (Nottingham) has contributed his code implementing Tate’s algorithm for computing local minimal models for elliptic curves defined over number fields.

The widely-used database of all elliptic curves over \mathbb{Q} having conductor up to 200,000 constructed by **John Cremona** (Nottingham) is also included.

Tim Dokchitser (Durham) wrote code for computing root numbers of elliptic curves over number fields.

Andreas-Stephan Elsenhans (Bayreuth) has provided routines for performing minimisation and reduction for Del Pezzo surfaces of degrees 3 and 4. He has also contributed code for determining isomorphism of cubic surfaces. Finally, Stephan has provided tools that calculate information about the Picard rank of a surface.

A package contributed by **Tom Fisher** (Cambridge) deals with curves of genus 1 given by models of a special kind (genus one normal curves) having degree 2, 3, 4 and 5.

The implementation of three descent on elliptic curves was mainly written by **Tom Fisher** (Cambridge), while certain parts of it (and also an earlier version) are by **Michael Stoll** (Bremen).

The algorithms and implementations of 6– and 12-descent are due to **Tom Fisher** (Cambridge). The new algorithm/implementation of 8-descent is likewise by Tom Fisher; this partly incorporates and partly replaces the earlier one by **Sebastian Stamminger**.

Various point-counting algorithms for hyperelliptic curves have been implemented by **Pier-rick Gaudry** (Ecole Polytechnique, Paris). These include an implementation of the Schoof algorithm for genus 2 curves.

Martine Girard (Sydney) has contributed her fast code for determining the heights of a point on an elliptic curve defined over a number field or a function field.

An implementation of GHS Weil descent for ordinary elliptic curves in characteristic 2 has been provided by **Florian Heß** (TU, Berlin).

A MAGMA package for calculating Igusa and other invariants for genus 2 hyperelliptic curves functions was written by **Everett Howe** (CCR, San Diego) and is based on `gp` routines developed by **Fernando Rodriguez–Villegas** (Texas) as part of the Computational Number Theory project funded by a TARP grant.

Hendrik Hubrechts (Leuven) has contributed his package for fast point-counting on elliptic and hyperelliptic curves over large finite fields, based on the deformation method pioneered by Alan Lauder.

David Kohel (Singapore, Magma) has provided implementations of division polynomials and isogeny structures for elliptic curves.

Allan Lauder (Oxford) has contributed code for computing the characteristic polynomial of a Hecke operator acting on spaces of overconvergent modular forms.

Reynard Lercier (Rennes) provided much advice and assistance to the MAGMA group concerning the implementation of the SEA point counting algorithm for elliptic curves.

Reynard Lercier (Rennes) and **Christophe Ritzenthaler** provided extensions to the machinery for genus 2 curves defined over finite fields. These include the reconstruction of a curve from invariants which applies to every characteristic p (previously $p > 5$), the geometric automorphism group and the calculation of all twists (not just quadratic).

Full and partial descents on cyclic covers of the projective line were implemented by **Michael Mourao** (Warwick).

A package for computing canonical heights on hyperelliptic curves has been contributed by **Steffan Müller** (Bayreuth).

David Roberts (Nottingham) contributed some descent machinery for elliptic curves over function fields.

David Roberts and **John Cremona** (Nottingham) implemented the Cremona-van Hoeij algorithm for parametrization of conics over rational function fields.

Jasper Scholten (Leuven) has developed much of the code for computing with elliptic curves over function fields.

Much of the initial development of the package for computing with hyperelliptic curves is due to **Michael Stoll** (Bayreuth). He also contributed many of the high level routines involving curves over the rationals and their Jacobians, such as Chabauty's method.

A database of 136,924,520 elliptic curves with conductors up to 10^8 has been provided by **William Stein** (Harvard) and **Mark Watkins** (Penn State).

Frederik Vercauteren (Leuven) has produced efficient implementations of the Tate, Eta and Ate pairings in MAGMA.

For elliptic curves defined over finite fields of characteristic 2, Kedlaya's algorithm for point counting has been implemented by **Frederick Vercauteren** (Leuven).

Tom Womack (Nottingham) contributed code for performing four-descent, from which the current implementation was adapted.

Associative Algebras

Fast algorithms for computing the Jacobson radical and unit group of a matrix algebra over a finite field were designed and implemented by **Peter Brooksbank** (Bucknell) and **Eamonn O'Brien** (Auckland).

A package for computing with algebras equipped with an involution (*-algebras) has been contributed by **Peter Brooksbank** (Bucknell) and **James Wilson**.

An algorithm designed and implemented by **Jon Carlson** and **Graham Matthews** (Athens, Ga.) provide an efficient means for constructing presentations for matrix algebras.

Markus Kirschmer (Aachen) has written a number of optimized routines for definite quaternion algebras over number fields. Markus has also contributed a package for quaternion algebras over the function fields $F_q[t]$, for q odd.

Quaternion algebras over the rational field Q were originally implemented by **David Kohel** (Singapore, Magma).

The vector enumeration program of **Steve Linton** (St. Andrews) provides an alternative to the use of Gröbner basis for constructing a matrix representation of a finitely presented associative algebra.

John Voight (Vermont) produced the package for quaternion algebras over number fields.

Coding Theory

The PERM package developed by **Jeff Leon** (UIC) is used to determine automorphism groups of codes, designs and matrices.

The construction of a database of Best Known Linear Codes over $GF(2)$ was a joint project with **Markus Grassl** (Karlsruhe, NUS). Other contributors to this project include: **Andries Brouwer**, **Zhi Chen**, **Stephan Grosse**, **Aaron Gulliver**, **Ray Hill**, **David Jaffe**, **Simon Litsyn**, **James B. Shearer** and **Henk van Tilborg**. Markus Grassl has also made many other contributions to the MAGMA coding theory machinery.

The databases of Best Known Linear Codes over $GF(3)$, $GF(4)$, $GF(5)$, $GF(7)$, $GF(8)$ and $GF(9)$ were constructed by **Markus Grassl** (IAKS, Karlsruhe).

A substantial collection of invariants for constructing and computing properties of Z_4 codes has been contributed by **Jaume Perras**, **Jaume Pujol** and **Merç Villanueva** (Universitat Autònoma de Barcelona).

Combinatorics

Michel Berkelaar (Eindhoven) gave us permission to incorporate his `LP-SOLVE` package for linear programming.

The first stage of the MAGMA database of Hadamard and skew-Hadamard matrices was prepared with the assistance of **Stelios Georgiou** (Athens), **Ilias Kotsireas** (Wilfrid Laurier) and **Christos Koukouvinos** (Athens). In particular, they made available their tables of Hadamard matrices of orders 32, 36, 44, 48 and 52. Further Hadamard matrices were contributed by Dragomir Djokovic.

The MAGMA machinery for symmetric functions is based on the Symmetriza package developed by **Abalbert Kerber** (Bayreuth) and colleagues. The MAGMA version was implemented by **Axel Kohnert** of the Bayreuth group.

The PERM package developed by **Jeff Leon** (UIC) is used to determine automorphism groups of designs and also to determine isomorphism of pairs of designs.

Automorphism groups and isomorphism of Hadamard matrices are determined by converting to a similar problem for graphs and then applying **Brendan McKay's** (ANU) program `NAUTY`. The adaption was undertaken by **Paulette Lieby** and **Geoff Bailey**.

The calculation of the automorphism groups of graphs and the determination of graph isomorphism is performed using **Brendan McKay's** (ANU) program `NAUTY` (version 2.2).

Databases of graphs and machinery for generating such databases have also been made available by Brendan. He has also collaborated in the design of the sparse graph machinery.

The code to perform the regular expression matching in the `regex` intrinsic function comes from the V8 regex package written by **Henry Spencer** (Toronto).

Commutative Algebra

Gregor Kemper (TU München) has contributed most of the major algorithms of the Invariant Theory module of MAGMA, together with many other helpful suggestions in the area of Commutative Algebra.

Alexa van der Waall (Simon Fraser) has implemented the module for differential Galois theory.

Geometry

The MAGMA code for computing with incidence geometries has been developed by **Dimitri Leemans** (Brussels).

Global and Local Arithmetic Fields

Jean-Francois Biasse has implemented a quadratic sieve for computing the class group of a quadratic field. He has also developed a generalisation to compute class groups for number fields of degree greater than 2.

Florian Heß (TU Berlin) has contributed a major package for determining all isomorphisms between a pair of algebraic function fields.

David Kohel (Singapore, Magma) has contributed to the machinery for binary quadratic forms and has implemented rings of Witt vectors.

Jürgen Klüners (Kassel) has made major contributions to the Galois theory machinery for function fields and number fields. In particular, he implemented functions for constructing the subfield lattice and automorphism group of a field and also the subfield lattice of the normal closure of a field. In joint work with Claus Fieker (Magma), Jürgen has recently developed a new method for determining the Galois group of a polynomial of arbitrary high degree.

Jürgen Klüners (Kassel) and **Gunter Malle** (Kassel) made available their extensive tables of polynomials realising all Galois groups over \mathbb{Q} up to degree 15.

Jürgen Klüners (Düsseldorf) and **Sebastian Pauli** (UNC Greensboro) have developed algorithms for computing the Picard group of non-maximal orders and for embedding the unit group of non-maximal orders into the unit group of the field.

Sebastian Pauli (TU Berlin) has implemented his algorithm for factoring polynomials over local fields within Magma. This algorithm may also be used for the factorization of ideals, the computation of completions of global fields, and for splitting extensions of local fields into towers of unramified and totally ramified extensions.

Class fields over local fields and the multiplicative structure of local fields are computed using new algorithms and implementations due to **Sebastian Pauli** (TU Berlin).

The module for Lazy Power Series is based on the ideas of **Josef Schicho** (Linz).

The facilities for general number fields and global function fields in MAGMA are based on the KANT V4 package developed by **Michael Pohst** and collaborators, first at Düsseldorf and then at TU Berlin. This package provides extensive machinery for computing with maximal orders of number fields and their ideals, Galois groups and function fields. Particularly noteworthy are functions for computing the class and unit group, and for solving Diophantine equations.

The fast algorithm of Bosma and Stevenhagen for computing the 2-part of the ideal class group of a quadratic field has been implemented by **Mark Watkins** (Bristol).

Group Theory: Finitely-Presented Groups

The soluble quotient algorithm in MAGMA was designed and implemented by **Herbert Brückner** (Aachen).

A new algorithm for computing all normal subgroups of a finitely presented group up to a specified index has been designed and implemented by **David Firth** and **Derek Holt** (Warwick).

The function for determining whether a given finite permutation group is a homomorphic image of a finitely presented group has been implemented in C by Volker Gebhardt (Magma) from a Magma language prototype developed by **Derek Holt** (Warwick). A variant developed by Derek, allows one to determine whether a small soluble group is a homomorphic image.

A small package for working with subgroups of free groups has been developed by **Derek Holt** (Warwick). He has also provided code for computing the automorphism group of a free group.

Code producing descriptions of the groups of order p^4, p^5, p^6, p^7 for $p > 3$ were contributed by **Boris Gornat**, **Robert McKibbin**, **Mike Newman**, **Eamonn O'Brien**, and **Mike Vaughan-Lee**.

Versions of MAGMA from V2.8 onwards employ the Advanced Coset Enumerator designed by **George Havas** (Queensland) and implemented by **Colin Ramsay** (also of Queensland). George has also contributed to the design of the machinery for finitely presented groups.

Derek Holt (Warwick) developed a modified version of his program, `KBMAG`, for inclusion within MAGMA. The MAGMA facilities for groups and monoids defined by confluent rewrite systems, as well as automatic groups, are supported by this code.

Derek Holt (Warwick) has provided a MAGMA implementation of his algorithm for testing whether two finitely presented groups are isomorphic.

Most of the algorithms for p -groups and many of the algorithms implemented in MAGMA for finite soluble groups are largely due to **Charles Leedham-Green** (QMUL, London).

The NQ program of **Werner Nickel** (Darmstadt) is used to compute nilpotent quotients of finitely presented groups. Version 2.2 of NQ was installed in MAGMA V2.14 by **Bill Unger** (Magma) and **Michael Vaughan-Lee** (Oxford).

The p -quotient program, developed by **Eamonn O'Brien** (Auckland) based on earlier work by **George Havas** and **Mike Newman** (ANU), provides a key facility for studying p -groups in MAGMA. Eamonn's extensions in MAGMA of this package for generating p -groups, computing automorphism groups of p -groups, and deciding isomorphism of p -groups are also included. He has contributed software to count certain classes of p -groups and to construct central extensions of soluble groups.

The package for classifying metacyclic p -groups has been developed by **Eamonn O'Brien** (Auckland) and **Mike Vaughan-Lee** (Oxford).

The low index subgroup function is implemented by code that is based on a Pascal program written by **Charlie Sims** (Rutgers).

Group Theory: Finite Groups

A variation of the Product Replacement Algorithm for generating random elements of a group due to **Henrik Bäårnhielm** and **Charles Leedham-Green** has been coded with their assistance.

Michael Downward and **Eamonn O'Brien** (Auckland) provided functions to access much of the data in the on-line Atlas of Finite Simple Groups for the sporadic groups. A function to select "good" base points for sporadic groups was provided by Eamonn and **Robert Wilson** (London).

The calculation of automorphism groups (for permutation and matrix groups) and determining group isomorphism is performed by code written by **Derek Holt** (Warwick).

Magma includes a database of almost-simple groups defined on standard generators. The database was originally conceived by **Derek Holt** (Warwick) with a major extension by **Volker Gebhardt** (Magma) and sporadic additions by **Bill Unger** (Magma).

The routine for computing the subgroup lattice of a group (as distinct from the list of all conjugacy classes of subgroups) is based closely on code written by **Dimitri Leemans** (Brussels).

A Small Groups database containing all groups having order at most 2000, excluding order 1024 has been made available by **Hans Ulrich Besche** (Aachen), **Bettina Eick** (Braunschweig), and **Eamonn O'Brien** (Auckland). This library incorporates "directly" the libraries of 2-groups of order dividing 256 and the 3-groups of order dividing 729, which were prepared and distributed at various intervals by **Mike Newman** (ANU) and **Eamonn O'Brien** and various assistants, the first release dating from 1987.

The Small Groups database has been augmented in V2.14 by code that will enumerate all groups of any square-free order. This code was developed by **Bettina Eick** (Braunschweig) and **Eamonn O'Brien** (Auckland).

Robert Wilson (QMUL) has made available the data contained in the on-line *ATLAS of Finite Group Representations* for use in a MAGMA database of permutation and matrix representations for finite simple groups. See <http://brauer.maths.qmul.ac.uk/Atlas/>.

Group Theory: Matrix Groups

The Composition Tree (CT) package developed by **Henrik Bäärnhielm** (Auckland), **Derek Holt** (Warwick), **Charles Leedham-Green** (QMUL) and **Eamonn O'Brien** (Auckland), working with numerous collaborators, was first released in V2.17. This package is designed for computing structural information for large matrix groups defined over a finite field.

Constructive recognition of quasi-simple groups belonging to the Suzuki and two Ree families have been implemented by **Hendrik Bäärnhielm** (QMUL). The package includes code for constructing their Sylow p -subgroups and maximal subgroups.

The maximal subgroups of all classical groups having degree not exceeding 12 have been constructed and implemented in MAGMA by **John Bray** (QMUL), **Derek Holt** (Warwick) and **Colva Roney-Dougal** (St Andrews).

Peter Brooksbank (Bucknell) implemented a MAGMA version of his algorithm for performing constructive black-box recognition of low-dimensional symplectic and unitary groups. He also gave the MAGMA group permission to base its implementation of the Kantor-Seress algorithm for black-box recognition of linear groups on his GAP implementation.

Code which computes the normaliser of a linear group defined over a finite field, using a theorem of Aschbacher rather than backtrack search, has been provided by **Hannah Coutts** (St Andrews).

A package, “Infinite”, has been developed by **Alla Detinko** (Galway), **Dane Flannery** (Galway) and **Eamonn O'Brien** (Auckland) for computing with groups defined over number fields, or (rational) function fields in zero or positive characteristic.

Markus Kirschmer (RWTH, Aachen) has provided a package for computing with finite subgroups of $GL(n, \mathbf{Z})$. A MAGMA database of the maximal finite irreducible subgroups of $Sp_{2n}(\mathbf{Q})$ for $1 \leq i \leq 11$ has also been made available by Markus.

Procedures to list irreducible (soluble) subgroups of $GL(2, q)$ and $GL(3, q)$ for arbitrary q have been provided by **Dane Flannery** (Galway) and **Eamonn O'Brien** (Auckland).

A Monte-Carlo algorithm to determine the defining characteristic of a quasisimple group of Lie type has been contributed by **Martin Liebeck** (Imperial) and **Eamonn O'Brien** (Auckland).

A Monte-Carlo algorithm for non-constructive recognition of simple groups has been contributed by **Gunter Malle** (Kaiserslautern) and **Eamonn O'Brien** (Auckland). This procedure includes the algorithm of Babai et al. to name a quasisimple group of Lie type.

MAGMA incorporates a database of the maximal finite rational subgroups of $GL(n, \mathbb{Q})$ up to dimension 31. This database is due to **Gabriele Nebe** (Aachen) and **Wilhelm Plesken** (Aachen). A database of quaternionic matrix groups constructed by Gabriele is also included.

A function that determines whether a matrix group G (defined over a finite field) is the normaliser of an extraspecial group in the case where the degree of G is an odd prime uses the new Monte-Carlo algorithm of **Alice Niemeyer** (Perth) and has been implemented in MAGMA by **Eamonn O'Brien** (Auckland).

The package for recognizing large degree classical groups over finite fields was designed and implemented by **Alice Niemeyer** (Perth) and **Cheryl Praeger** (Perth). It has been extended to include 2-dimensional linear groups by **Eamonn O'Brien** (Auckland).

Eamonn O'Brien (Auckland) has contributed a MAGMA implementation of algorithms for determining the Aschbacher category of a subgroup of $GL(n, q)$.

Eamonn O'Brien (Auckland) has provided implementations of constructive recognition algorithms for the matrix groups $(P)SL(2, q)$ and $(P)SL(3, q)$.

A fast algorithm for determining subgroup conjugacy based on Aschbacher's theorem classifying the maximal subgroups of a linear group has been designed and implemented by **Colva Roney-Dougal** (St Andrews).

A package for constructing the Sylow p -subgroups of the classical groups has been implemented by **Mark Stather** (Warwick).

Generators in the natural representation of a finite group of Lie type were constructed and implemented by **Don Taylor** (Sydney) with some assistance from **Leanne Rylands** (Western Sydney).

Group Theory: Permutation Groups

Derek Holt (Warwick) has implemented the MAGMA version of the Bratus/Pak algorithm for black-box recognition of the symmetric and alternating groups.

Alexander Hulpke (Colorado State) has made available his database of all transitive permutation groups of degree up to 30. This incorporates the earlier database of **Greg Butler** (Concordia) and **John McKay** (Concordia) containing all transitive groups of degree up to 15.

The PERM package developed by **Jeff Leon** (UIC) for efficient backtrack searching in permutation groups is used for most of the permutation group constructions that employ backtrack search.

A table containing all primitive groups having degree less than 2,500 has been provided by **Colva Roney-Dougal** (St Andrews). The groups of degree up to 1,000 were done jointly with Bill Unger (MAGMA).

A table containing all primitive groups having degrees in the range 2,500 to 4,095 has been provided by **Hannah Coutts**, **Martyn Quick** and **Colva Roney-Dougal** (all at St Andrews).

Colva Roney-Dougal (St Andrews) has implemented the Beals et al algorithm for performing black-box recognition on the symmetric and alternating groups.

A MAGMA database has been constructed from the permutation and matrix representations contained in the on-line Atlas of Finite Simple Groups with the assistance of its author **Robert Wilson** (QMUL, London).

Homological Algebra

The packages for chain complexes and basic algebras have been developed by **Jon F. Carlson** (Athens, GA).

Machinery for computing group cohomology and for producing group extensions has been developed by **Derek Holt** (Warwick). There are two parts to this machinery. The first part comprises Derek's older C-language package for permutation groups while the second part comprises a recent MAGMA language package for group cohomology.

Sergei Haller developed MAGMA code for computing the first cohomology group of a finite group with coefficients in a finite (not necessarily abelian) group. This formed the basis of a package for computing Galois cohomology of linear algebra groups.

The code for computing A_∞ -structures in group cohomology was developed by **Mikael Vejdemo Johansson** (Jena).

L -Functions

Tim Dokchitser (Cambridge) has implemented efficient computation of many kinds of L -functions, including those attached to Dirichlet characters, number fields, Artin representations, elliptic curves and hyperelliptic curves. **Vladimir Dokchitser** (Cambridge) has contributed theoretical ideas.

Anton Mellit has contributed code for computing symmetric powers and tensor products of L -functions.

Lattices and Quadratic Forms

The construction of the sublattice of an integral lattice is performed by code developed by **Markus Kirschmer** (Aachen).

A collection of lattices derived from the on-line tables of lattices prepared by **Neil Sloane** (AT&T Research) and **Gabriele Nebe** (Aachen) is included in MAGMA.

The original functions for computing automorphism groups and isometries of integral lattices are based on the AUTO and ISOM programs of **Bernd Souvignier** (Nijmegen). In V2.16 they are replaced by much faster versions developed by **Bill Unger** (Magma).

Coppersmith's method (based on LLL) for finding small roots of univariate polynomials modulo an integer has been implemented by **Damien Stehlé** (ENS Lyon).

Given a quadratic form F in an arbitrary number of variables, **Mark Watkins** (Bristol) has used Denis Simon's ideas as the basis of an algorithm he has implemented in MAGMA for finding a large (totally) isotropic subspace of F .

Lie Theory

The major structural machinery for Lie algebras has been implemented for MAGMA by **Willem de Graaf** (Utrecht) and is based on his ELIAS package written in GAP. He has also implemented a separate package for finitely presented Lie rings.

A database of soluble Lie algebras of dimensions 2, 3 and 4 over all fields has been implemented by **Willem de Graaf** (Trento). Willem has also provided a database of all nilpotent Lie algebras of dimension up to 6 over all base fields (except characteristic 2 when the dimension is 6).

More recent extensions to the Lie algebra package developed by **Willem de Graaf** (Trento) include quantum groups, universal enveloping algebras, the semisimple subalgebras of a simple Lie algebra and nilpotent orbits for simple Lie algebras.

A fast algorithm for multiplying the elements of Coxeter groups based on their automatic structure has been designed and implemented by **Bob Howlett** (Sydney). Bob has also contributed MAGMA code for computing the growth function of a Coxeter group.

Machinery for computing the W -graphs for Lie types A_n , E_6 , E_7 and E_8 has been supplied by **Bob Howlett** (Sydney).

The original version of the code for root systems and permutation Coxeter groups was modelled, in part, on the Chevie package of GAP and implemented by **Don Taylor** (Sydney) with the assistance of **Frank Lübeck** (Aachen).

Functions that construct any finite irreducible unitary reflection group in C^n have been implemented by **Don Taylor** (Sydney). Extension to the infinite case was implemented by **Scott H. Murray** (Sydney).

The current version of Lie groups in MAGMA has been implemented by **Scott H. Murray** (Sydney) and **Sergei Haller** with some assistance from **Don Taylor** (Sydney).

An extensive package for computing the combinatorial properties of highest weight representations of a Lie algebra has been written by **Dan Roozmond** (Eindhoven). This code is based in the LiE package with permission of the authors.

Code has been contributed by **Robert Zeier** (Technical University of Munich) for determining the irreducible simple subalgebras of the Lie algebra $su(k)$.

Linear Algebra and Module Theory

Parts of the ATLAS (Automatically Tuned Linear Algebra Software) of **R. Clint Whaley et al.** are used for some fundamental matrix algorithms over machine-int-sized prime finite fields.

Modular Forms

Jeremy Le Borgne (Rennes) contributed his package for working with mod p Galois representations.

Kevin Buzzard (Imperial College) made available his code for computing modular forms of weight one. The MAGMA implementation was developed using this as a starting point.

Lassina Dembélé (Warwick) wrote part of the code implementing his algorithm for computing Hilbert modular forms.

Enrique González-Jiménez (Madrid) contributed a package to compute curves over \mathbf{Q} , of genus at least 2, which are images of $X_1(N)$ for a given level N .

Matthew Greenberg (Calgary) and **John Voight** (Vermont) developed and implemented an algorithm for computing Hilbert modular forms using Shimura curves.

A new implementation of Brandt modules associated to definite quaternion orders, over \mathbf{Z} and over function fields $\mathbf{F}_q[t]$, has been developed by **Markus Kirschmer** (Aachen) and **Steve Donnelly**.

David Kohel (Singapore, Magma) has provided implementations of division polynomials and isogeny structures for Brandt modules and modular curves. Jointly with **William Stein** (Harvard), he implemented the module of supersingular points.

MAGMA routines for constructing building blocks of modular abelian varieties were contributed by **Jordi Quer** (Cataluna).

A package for computing with modular symbols (known as HECKE) has been developed by **William Stein** (Harvard). William has also provided much of the package for modular forms.

In 2003–2004, **William Stein** (Harvard) developed extensive machinery for computing with modular abelian varieties within MAGMA.

A package for computing with congruence subgroups of the group $\mathrm{PSL}(2, \mathbf{R})$ has been developed by **Helena Verrill** (LSU).

John Voight (Vermont) produced the package for Shimura curves and arithmetic Fuchsian groups.

Dan Yasaki (UNC) provided the package for Bianchi modular forms.

Jared Weinstein (UCLA) wrote the package on admissible representations of $GL_2(\mathbf{Q}_p)$.

Primality and Factorisation

Richard Brent (ANU) has also made available his database of 237,578 factorizations of integers of the form $p^n \pm 1$, together with his intelligent factorization code FACTOR.

One of the main integer factorization tools available in MAGMA is due to **Arjen K. Lenstra** (EPFL) and his collaborators: a multiple polynomial quadratic sieve developed by Arjen from his “factoring by email” MPQS during visits to Sydney in 1995 and 1998.

The primality of integers is proven using the ECPP (Elliptic Curves and Primality Proving) package written by **François Morain** (Ecole Polytechnique and INRIA). The ECPP program in turn uses the BigNum package developed jointly by **INRIA** and **Digital PRL**.

MAGMA uses the **GMP-ECM** implementation of the Elliptic Curve Method (ECM) for integer factorisation. This was developed by **Pierrick Gaudry**, **Jim Fougeron**, **Laurent Fousse**, **Alexander Kruppa**, **Dave Newman**, and **Paul Zimmermann**. See <http://gforge.inria.fr/projects/ecm/>.

Real and Complex Arithmetic

The complex arithmetic in MAGMA uses the **MPC** package which is being developed by **Andreas Enge**, Philippe Théveny and Paul Zimmermann. (For more information see www.multiprecision.org/mpc/).

Xavier Gourdon (INRIA, Paris) made available his C implementation of A. Schönhage’s splitting-circle algorithm for the fast computation of the roots of a polynomial to a specified precision. Xavier also assisted with the adaptation of his code for the MAGMA kernel.

Some portions of the **GNU GMP** multiprecision integer library (<http://gmplib.org>) are used for integer multiplication.

Most real arithmetic in MAGMA is based on the **MPFR** package which is developed by **Paul Zimmermann** (Nancy) and associates. (See www.mpfr.org).

Representation Theory

Derek Holt (Warwick) has made a number of important contributions to the design of the module theory algorithms employed in MAGMA. In 2009–2010 he developed a MAGMA package for computing principal indecomposable modules.

The algorithm of John Dixon for constructing the ordinary irreducible representation of a finite group from its character has been implemented by **Derek Holt** (Warwick).

An algorithm of Sam Conlon for determining the degrees of the ordinary irreducible characters of a soluble group (without determining the full character table) has been implemented by **Derek Holt** (Warwick).

Charles Leedham-Green (QMW, London) was responsible for the original versions of the submodule lattice and endomorphism ring algorithms.

Topology

A basic module for defining and computing with simplicial complexes was developed by **Mikael Johansson** (Jena).

Nathan Dunfield and **William Thurston** (Cornell) have made available their database of the fundamental groups of the 10,986 small-volume closed hyperbolic manifolds in the Hodgson-Weeks census.

Handbook Contributors

Introduction

The Handbook of Magma Functions is the work of many individuals. It was based on a similar Handbook written for Cayley in 1990. Up until 1997 the Handbook was mainly written by Wieb Bosma, John Cannon and Allan Steel but in more recent times, as Magma expanded into new areas of mathematics, additional people became involved. It is not uncommon for some chapters to comprise contributions from 8 to 10 people. Because of the complexity and dynamic nature of chapter authorship, rather than ascribe chapter authors, in the table below we attempt to list those people who have made *significant* contributions to chapters.

We distinguish between:

- **Principal Author**, i.e. one who primarily conceived the core element(s) of a chapter and who was also responsible for the writing of a large part of its current content, and
- **Contributing Author**, i.e. one who has written a significant amount of content but who has not had primary responsibility for chapter design and overall content.

It should be noted that attribution of a person as an author of a chapter carries no implications about the authorship of the associated computer code: for some chapters it will be true that the author(s) listed for a chapter are also the authors of the corresponding code, but in many chapters this is either not the case or only partly true. Some information about code authorship may be found in the sections *Magma Development Team* and *External Contributors*.

The attributions given below reflect the authorship of the material comprising the V2.19 edition. Since many of the authors have since moved on to other careers, we have not been able to check that all of the attributions below are completely correct. We would appreciate hearing of any omissions.

In the chapter listing that follows, for each chapter the start of the list of principal authors (if any) is denoted by • while the start of the list of contributing authors is denoted by ◦.

People who have made minor contributions to one or more chapters are listed in a general acknowledgement following the chapter listing.

The Chapters

- 1 Statements and Expressions • *W. Bosma, A. Steel*
- 2 Functions, Procedures and Packages • *W. Bosma, A. Steel*
- 3 Input and Output • *W. Bosma, A. Steel*
- 4 Environment and Options • *A. Steel* ◦ *W. Bosma*
- 5 Magma Semantics • *G. Matthews*
- 6 The Magma Profiler • *D. Fisher*
- 7 Debugging Magma Code • *D. Fisher*
- 8 Introduction to Aggregates • *W. Bosma*
- 9 Sets • *W. Bosma, J. Cannon* ◦ *A. Steel*
- 10 Sequences • *W. Bosma, J. Cannon*
- 11 Tuples and Cartesian Products • *W. Bosma*
- 12 Lists • *W. Bosma*
- 13 Associative Arrays • *A. Steel*
- 14 Coproducts • *A. Steel*
- 15 Records • *W. Bosma*
- 16 Mappings • *W. Bosma*
- 17 Introduction to Rings • *W. Bosma*
- 18 Ring of Integers • *W. Bosma, A. Steel* ◦ *S. Contini, B. Smith*
- 19 Integer Residue Class Rings • *W. Bosma* ◦ *S. Donnelly, W. Stein*
- 20 Rational Field • *W. Bosma*
- 21 Finite Fields • *W. Bosma, A. Steel*
- 22 Nearfields • *D. Taylor*
- 23 Univariate Polynomial Rings • *A. Steel*
- 24 Multivariate Polynomial Rings • *A. Steel*
- 25 Real and Complex Fields • *W. Bosma*
- 26 Matrices • *A. Steel*
- 27 Sparse Matrices • *A. Steel*
- 28 Vector Spaces • *J. Cannon, A. Steel*
- 29 Polar Spaces • *D. Taylor*
- 30 Lattices • *A. Steel, D. Stehlé*
- 31 Lattices With Group Action • *B. Souvignier* ◦ *M. Kirschmer*
- 32 Quadratic Forms • *S. Donnelly*
- 33 Binary Quadratic Forms • *D. Kohel*
- 34 Number Fields • *C. Fieker* ◦ *W. Bosma, N. Sutherland*
- 35 Quadratic Fields • *W. Bosma*
- 36 Cyclotomic Fields • *W. Bosma, C. Fieker*
- 37 Orders and Algebraic Fields • *C. Fieker* ◦ *W. Bosma, N. Sutherland*
- 38 Galois Theory of Number Fields • *C. Fieker* ◦ *J. Klüners, K. Geißler*

- 39 Class Field Theory • *C. Fieker*
- 40 Algebraically Closed Fields • *A. Steel*
- 41 Rational Function Fields • *A. Steel* ◦ *A. van der Waall*
- 42 Algebraic Function Fields • *F. Heß* ◦ *C. Fieker, N. Sutherland*
- 43 Class Field Theory For Global Function Fields • *C. Fieker*
- 44 Artin Representations • *T. Dokchitser*
- 45 Valuation Rings • *W. Bosma*
- 46 Newton Polygons • *G. Brown, N. Sutherland*
- 47 p -adic Rings and their Extensions • *D. Fisher, B. Souvignier* ◦ *N. Sutherland*
- 48 Galois Rings • *A. Steel*
- 49 Power, Laurent and Puiseux Series • *A. Steel*
- 50 Lazy Power Series Rings • *N. Sutherland*
- 51 General Local Fields • *N. Sutherland*
- 52 Algebraic Power Series Rings • *T. Beck, M. Harrison*
- 53 Introduction to Modules • *J. Cannon*
- 54 Free Modules • *J. Cannon, A. Steel*
- 55 Modules over Dedekind Domains • *C. Fieker, N. Sutherland*
- 56 Chain Complexes • *J. Carlson*
- 57 Groups • *J. Cannon* ◦ *W. Unger*
- 58 Permutation Groups • *J. Cannon* ◦ *B. Cox, W. Unger*
- 59 Matrix Groups over General Rings • *J. Cannon* ◦ *B. Cox, E.A. O'Brien, A. Steel*
- 60 Matrix Groups over Finite Fields • *E.A. O'Brien*
- 61 Matrix Groups over Infinite Fields • *E.A. O'Brien*
- 62 Matrix Groups over \mathbb{Q} and \mathbb{Z} • *M. Kirschmer, B. Souvignier*
- 63 Finite Soluble Groups • *J. Cannon, M. Slattery* ◦ *E.A. O'Brien*
- 64 Black-box Groups • *W. Unger*
- 65 Almost Simple Groups ◦ *H. Bäärnhielm, J. Cannon, D. Holt, M. Stather*
- 66 Databases of Groups • *W. Unger* ◦ *V. Gebhardt*
- 67 Automorphism Groups • *D. Holt* ◦ *W. Unger*
- 68 Cohomology and Extensions • *D. Holt* ◦ *S. Haller*
- 69 Abelian Groups • *J. Cannon* ◦ *P. Lieby*
- 70 Finitely Presented Groups • *J. Cannon* ◦ *V. Gebhardt, E.A. O'Brien, M. Vaughan-Lee*
- 71 Finitely Presented Groups: Advanced • *H. Brückner, V. Gebhardt* ◦ *E.A. O'Brien*
- 72 Polycyclic Groups • *V. Gebhardt*
- 73 Braid Groups • *V. Gebhardt*
- 74 Groups Defined by Rewrite Systems • *D. Holt* ◦ *G. Matthews*
- 75 Automatic Groups • *D. Holt* ◦ *G. Matthews*
- 76 Groups of Straight-line Programs • *J. Cannon*

- 77 Finitely Presented Semigroups • *J. Cannon*
- 78 Monoids Given by Rewrite Systems • *D. Holt* ◦ *G. Matthews*
- 79 Algebras • *J. Cannon, B. Souvignier*
- 80 Structure Constant Algebras • *J. Cannon, B. Souvignier*
- 81 Associative Algebras ◦ *J. Cannon, S. Donnelly, N. Sutherland, B. Souvignier, J. Voight*
- 82 Finitely Presented Algebras • *A. Steel, S. Linton*
- 83 Matrix Algebras • *J. Cannon, A. Steel* ◦ *J. Carlson*
- 84 Group Algebras • *J. Cannon, B. Souvignier*
- 85 Basic Algebras • *J. Carlson* ◦ *M. Vejdemo-Johansson*
- 86 Quaternion Algebras • *D. Kohel, J. Voight* ◦ *S. Donnelly, M. Kirschmer*
- 87 Algebras With Involution • *P. Brooksbank, J. Wilson*
- 88 Clifford Algebras • *D. Taylor*
- 89 Modules over An Algebra • *J. Cannon, A. Steel*
- 90 $K[G]$ -Modules and Group Representations • *J. Cannon, A. Steel*
- 91 Characters of Finite Groups • *W. Bosma, J. Cannon*
- 92 Representations of Symmetric Groups • *A. Kohnert*
- 93 Mod P Galois Representations • *J. Le Borgne*
- 94 Introduction to Lie Theory • *S. Murray* ◦ *D. Taylor*
- 95 Coxeter Systems • *S. Murray* ◦ *D. Taylor*
- 96 Root Systems • *S. Murray* ◦ *S. Haller, D. Taylor*
- 97 Root Data • *S. Haller, S. Murray* ◦ *D. Taylor*
- 98 Coxeter Groups • *S. Murray* ◦ *D. Taylor*
- 99 Reflection Groups • *S. Murray* ◦ *D. Taylor*
- 100 Lie Algebras • *W. de Graaf, D. Roozemonnd* ◦ *S. Haller, S. Murray*
- 101 Kac-moody Lie Algebras • *D. Roozemonnd*
- 102 Quantum Groups • *W. de Graaf*
- 103 Groups of Lie Type • *S. Murray* ◦ *S. Haller, D. Taylor*
- 104 Representations of Lie Groups and Algebras • *D. Roozemonnd* ◦ *S. Murray*
- 105 Gröbner Bases • *A. Steel* ◦ *M. Harrison*
- 106 Polynomial Ring Ideal Operations • *A. Steel* ◦ *M. Harrison*
- 107 Local Polynomial Rings • *A. Steel*
- 108 Affine Algebras • *A. Steel*
- 109 Modules over Multivariate Rings • *A. Steel* ◦ *M. Harrison*
- 110 Invariant Theory • *A. Steel*
- 111 Differential Rings • *A. van der Waall*
- 112 Schemes • *G. Brown* ◦ *J. Cannon, M. Harrison, N. Sutherland*
- 113 Coherent Sheaves • *M. Harrison*
- 114 Algebraic Curves • *G. Brown* ◦ *N. Bruin, J. Cannon, M. Harrison, A. Wilson*
- 115 Resolution Graphs and Splice Diagrams • *G. Brown*

- 116 Algebraic Surfaces • *T. Beck, M. Harrison*
- 117 Hilbert Series of Polarised Varieties • *G. Brown*
- 118 Toric Varieties • *G. Brown, A. Kasprzyk*
- 119 Rational Curves and Conics • *D. Kohel, P. Lieby* ◦ *S. Donnelly, M. Watkins*
- 120 Elliptic Curves • *G. Bailey* ◦ *S. Donnelly, D. Kohel*
- 121 Elliptic Curves over Finite Fields • *M. Harrison* ◦ *P. Lieby*
- 122 Elliptic Curves over \mathbf{Q} and Number Fields ◦ *G. Bailey, N. Bruin, B. Creutz, S. Donnelly, D. Kohel, M. Watkins*
- 123 Elliptic Curves over Function Fields • *J. Scholten* ◦ *S. Donnelly*
- 124 Models of Genus One Curves • *T. Fisher, S. Donnelly*
- 125 Hyperelliptic Curves ◦ *N. Bruin, B. Creutz, S. Donnelly, M. Harrison, D. Kohel, P. van Wamelen*
- 126 Hypergeometric Motives • *M. Watkins*
- 127 L-functions • *T. Dokchitser* ◦ *M. Watkins*
- 128 Modular Curves • *D. Kohel* ◦ *M. Harrison, E. González-Jiménez*
- 129 Small Modular Curves • *M. Harrison*
- 130 Congruence Subgroups of $\mathrm{PSL}_2(\mathbf{R})$ • *H. Verrill*
- 131 Arithmetic Fuchsian Groups and Shimura Curves • *J. Voight*
- 132 Modular Forms • *W. Stein* ◦ *K. Buzzard, S. Donnelly*
- 133 Modular Symbols • *W. Stein* ◦ *K. Buzzard*
- 134 Brandt Modules • *D. Kohel*
- 135 Supersingular Divisors on Modular Curves • *D. Kohel, W. Stein*
- 136 Modular Abelian Varieties • *W. Stein* ◦ *J. Quer*
- 137 Hilbert Modular Forms • *S. Donnelly*
- 138 Modular Forms over Imaginary Quadratic Fields • *D. Yasaki* ◦ *S. Donnelly*
- 139 Admissible Representations of $\mathrm{GL}_2(\mathbf{Q}_p)$ • *J. Weinstein* ◦ *S. Donnelly*
- 140 Simplicial Homology • *M. Vejdemo-Johansson*
- 141 Finite Planes • *J. Cannon*
- 142 Incidence Geometry • *D. Leemans*
- 143 Convex Polytopes and Polyhedra • *G. Brown, A. Kasprzyk*
- 144 Enumerative Combinatorics • *G. Bailey* ◦ *G. White*
- 145 Partitions, Words and Young Tableaux • *G. White*
- 146 Symmetric Functions • *A. Kohnert*
- 147 Incidence Structures and Designs • *J. Cannon*
- 148 Hadamard Matrices • *G. Bailey*
- 149 Graphs • *J. Cannon, P. Lieby* ◦ *G. Bailey*
- 150 Multigraphs • *J. Cannon, P. Lieby*
- 151 Networks • *P. Lieby*
- 152 Linear Codes over Finite Fields • *J. Cannon, A. Steel* ◦ *G. White*

- 153 Algebraic-geometric Codes • *J. Cannon, G. White*
- 154 Low Density Parity Check Codes • *G. White*
- 155 Linear Codes over Finite Rings • *A. Steel* ◦ *G. White, M. Villanueva*
- 156 Additive Codes • *G. White*
- 157 Quantum Codes • *G. White*
- 158 Pseudo-random Bit Sequences • *S. Contini*
- 159 Linear Programming • *B. Smith*

General Acknowledgements

In addition to the contributors listed above, we gratefully acknowledge the contributions to the Handbook made by the following people:

- J. Brownie* (group theory)
- K. Geißler* (Galois groups)
- A. Flynn* (algebras and designs)
- E. Herrmann* (elliptic curves)
- E. Howe* (Igusa invariants)
- B. McKay* (graph theory)
- S. Pauli* (local fields)
- C. Playoust* (data structures, rings)
- C. Roney-Dougal* (groups)
- T. Womack* (elliptic curves)

USING THE HANDBOOK

Most sections within a chapter of this Handbook consist of a brief introduction and explanation of the notation, followed by a list of MAGMA functions, procedures and operators.

Each entry in this list consists of an expression in a box, and an indented explanation of use and effects. The `typewriter` typefont is used for commands that can be used literally; however, one should be aware that most functions operate on variables that must have values assigned to them beforehand, and return values that should be assigned to variables (or the first value should be used in an expression). Thus the entry:

`Xgcd(a, b)`

The extended gcd; returns integers d , l and m such that d is the greatest common divisor of the integers a and b , and $d = l * a + m * b$.

indicates that this function could be called in MAGMA as follows:

```
g, a, b := Xgcd(23, 28);
```

If the function has optional named *parameters*, a line like the following will be found in the description:

Proof	BOOLELT	<i>Default : true</i>
--------------	---------	-----------------------

The first word will be the name of the parameter, the second word will be the type which its value should have, and the rest of the line will indicate the default for the parameter, if there is one. Parameters for a function call are specified by appending a colon to the last argument, followed by a comma-separated list of assignments (using `:=`) for each parameter. For example, the function call `IsPrime(n: Proof := false)` calls the function `IsPrime` with argument n but also with the value for the parameter `Proof` set to `false`.

Whenever the symbol `#` precedes a function name in a box, it indicates that the particular function is not yet available but should be in the future.

An index is provided at the end of each volume which contains all the intrinsics in the Handbook.

Running the Examples

All examples presented in this Handbook are available to MAGMA users. If your MAGMA environment has been set up correctly, you can load the source for an example by using the name of the example as printed in boldface at the top (the name has the form $HmEn$, where m is the Chapter number and n is the Example number). So, to run the first example in the Chapter 28, type:

```
load "H28E1";
```


VOLUME 1: OVERVIEW

I	THE MAGMA LANGUAGE	1
1	STATEMENTS AND EXPRESSIONS	3
2	FUNCTIONS, PROCEDURES AND PACKAGES	33
3	INPUT AND OUTPUT	63
4	ENVIRONMENT AND OPTIONS	93
5	MAGMA SEMANTICS	115
6	THE MAGMA PROFILER	135
7	DEBUGGING MAGMA CODE	145
II	SETS, SEQUENCES, AND MAPPINGS	151
8	INTRODUCTION TO AGGREGATES	153
9	SETS	163
10	SEQUENCES	191
11	TUPLES AND CARTESIAN PRODUCTS	213
12	LISTS	221
13	ASSOCIATIVE ARRAYS	227
14	COPRODUCTS	233
15	RECORDS	239
16	MAPPINGS	245

VOLUME 2: OVERVIEW

III	BASIC RINGS	255
17	INTRODUCTION TO RINGS	257
18	RING OF INTEGERS	277
19	INTEGER RESIDUE CLASS RINGS	329
20	RATIONAL FIELD	349
21	FINITE FIELDS	361
22	NEARFIELDS	389
23	UNIVARIATE POLYNOMIAL RINGS	407
24	MULTIVARIATE POLYNOMIAL RINGS	441
25	REAL AND COMPLEX FIELDS	469
IV	MATRICES AND LINEAR ALGEBRA	515
26	MATRICES	517
27	SPARSE MATRICES	557
28	VECTOR SPACES	583
29	POLAR SPACES	607

VOLUME 3: OVERVIEW

V	LATTICES AND QUADRATIC FORMS	637
30	LATTICES	639
31	LATTICES WITH GROUP ACTION	717
32	QUADRATIC FORMS	743
33	BINARY QUADRATIC FORMS	751
VI	GLOBAL ARITHMETIC FIELDS	765
34	NUMBER FIELDS	767
35	QUADRATIC FIELDS	831
36	CYCLOTOMIC FIELDS	845
37	ORDERS AND ALGEBRAIC FIELDS	853
38	GALOIS THEORY OF NUMBER FIELDS	959
39	CLASS FIELD THEORY	995
40	ALGEBRAICALLY CLOSED FIELDS	1033
41	RATIONAL FUNCTION FIELDS	1055
42	ALGEBRAIC FUNCTION FIELDS	1077
43	CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS	1187
44	ARTIN REPRESENTATIONS	1213

VOLUME 4: OVERVIEW

VII	LOCAL ARITHMETIC FIELDS	1223
45	VALUATION RINGS	1225
46	NEWTON POLYGONS	1231
47	p -ADIC RINGS AND THEIR EXTENSIONS	1259
48	GALOIS RINGS	1309
49	POWER, LAURENT AND PUISEUX SERIES	1317
50	LAZY POWER SERIES RINGS	1345
51	GENERAL LOCAL FIELDS	1361
52	ALGEBRAIC POWER SERIES RINGS	1373
VIII	MODULES	1387
53	INTRODUCTION TO MODULES	1389
54	FREE MODULES	1393
55	MODULES OVER DEDEKIND DOMAINS	1417
56	CHAIN COMPLEXES	1439

VOLUME 5: OVERVIEW

IX	FINITE GROUPS	1455
57	GROUPS	1457
58	PERMUTATION GROUPS	1513
59	MATRIX GROUPS OVER GENERAL RINGS	1635
60	MATRIX GROUPS OVER FINITE FIELDS	1707
61	MATRIX GROUPS OVER INFINITE FIELDS	1757
62	MATRIX GROUPS OVER \mathbb{Q} AND \mathbb{Z}	1777
63	FINITE SOLUBLE GROUPS	1787
64	BLACK-BOX GROUPS	1867
65	ALMOST SIMPLE GROUPS	1873
66	DATABASES OF GROUPS	1933
67	AUTOMORPHISM GROUPS	1991
68	COHOMOLOGY AND EXTENSIONS	2009

VOLUME 6: OVERVIEW

X	FINITELY-PRESENTED GROUPS	2037
69	ABELIAN GROUPS	2039
70	FINITELY PRESENTED GROUPS	2075
71	FINITELY PRESENTED GROUPS: ADVANCED	2201
72	POLYCYCLIC GROUPS	2247
73	BRAID GROUPS	2287
74	GROUPS DEFINED BY REWRITE SYSTEMS	2337
75	AUTOMATIC GROUPS	2355
76	GROUPS OF STRAIGHT-LINE PROGRAMS	2375
77	FINITELY PRESENTED SEMIGROUPS	2385
78	MONOIDS GIVEN BY REWRITE SYSTEMS	2397

VOLUME 7: OVERVIEW

XI	ALGEBRAS	2415
79	ALGEBRAS	2417
80	STRUCTURE CONSTANT ALGEBRAS	2429
81	ASSOCIATIVE ALGEBRAS	2439
82	FINITELY PRESENTED ALGEBRAS	2465
83	MATRIX ALGEBRAS	2503
84	GROUP ALGEBRAS	2543
85	BASIC ALGEBRAS	2557
86	QUATERNION ALGEBRAS	2617
87	ALGEBRAS WITH INVOLUTION	2661
88	CLIFFORD ALGEBRAS	2677
XII	REPRESENTATION THEORY	2681
89	MODULES OVER AN ALGEBRA	2683
90	$K[G]$ -MODULES AND GROUP REPRESENTATIONS	2719
91	CHARACTERS OF FINITE GROUPS	2755
92	REPRESENTATIONS OF SYMMETRIC GROUPS	2777
93	MOD P GALOIS REPRESENTATIONS	2785

VOLUME 8: OVERVIEW

XIII	LIE THEORY	2793
94	INTRODUCTION TO LIE THEORY	2795
95	COXETER SYSTEMS	2801
96	ROOT SYSTEMS	2825
97	ROOT DATA	2847
98	COXETER GROUPS	2899
99	REFLECTION GROUPS	2939
100	LIE ALGEBRAS	2971
101	KAC-MOODY LIE ALGEBRAS	3059
102	QUANTUM GROUPS	3069
103	GROUPS OF LIE TYPE	3095
104	REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS	3135

VOLUME 9: OVERVIEW

XIV	COMMUTATIVE ALGEBRA	3175
105	GRÖBNER BASES	3177
106	POLYNOMIAL RING IDEAL OPERATIONS	3221
107	LOCAL POLYNOMIAL RINGS	3269
108	AFFINE ALGEBRAS	3283
109	MODULES OVER MULTIVARIATE RINGS	3299
110	INVARIANT THEORY	3351
111	DIFFERENTIAL RINGS	3397
XV	ALGEBRAIC GEOMETRY	3465
112	SCHEMES	3467
113	COHERENT SHEAVES	3597
114	ALGEBRAIC CURVES	3629
115	RESOLUTION GRAPHS AND SPLICE DIAGRAMS	3735
116	ALGEBRAIC SURFACES	3751
117	HILBERT SERIES OF POLARISED VARIETIES	3817
118	TORIC VARIETIES	3851

VOLUME 10: OVERVIEW

XVI	ARITHMETIC GEOMETRY	3901
119	RATIONAL CURVES AND CONICS	3903
120	ELLIPTIC CURVES	3927
121	ELLIPTIC CURVES OVER FINITE FIELDS	3969
122	ELLIPTIC CURVES OVER \mathbf{Q} AND NUMBER FIELDS	3993
123	ELLIPTIC CURVES OVER FUNCTION FIELDS	4075
124	MODELS OF GENUS ONE CURVES	4093
125	HYPERELLIPTIC CURVES	4111
126	HYPERGEOMETRIC MOTIVES	4215
127	L-FUNCTIONS	4235

VOLUME 11: OVERVIEW

XVII	MODULAR ARITHMETIC GEOMETRY	4281
128	MODULAR CURVES	4283
129	SMALL MODULAR CURVES	4303
130	CONGRUENCE SUBGROUPS OF $\mathrm{PSL}_2(\mathbf{R})$	4327
131	ARITHMETIC FUCHSIAN GROUPS AND SHIMURA CURVES	4353
132	MODULAR FORMS	4377
133	MODULAR SYMBOLS	4419
134	BRANDT MODULES	4475
135	SUPERSINGULAR DIVISORS ON MODULAR CURVES	4489
136	MODULAR ABELIAN VARIETIES	4505
137	HILBERT MODULAR FORMS	4643
138	MODULAR FORMS OVER IMAGINARY QUADRATIC FIELDS	4661
139	ADMISSIBLE REPRESENTATIONS OF $\mathrm{GL}_2(\mathbf{Q}_p)$	4669

VOLUME 12: OVERVIEW

XVIII	TOPOLOGY	4681
140	SIMPLICIAL HOMOLOGY	4683
XIX	GEOMETRY	4703
141	FINITE PLANES	4705
142	INCIDENCE GEOMETRY	4741
143	CONVEX POLYTOPES AND POLYHEDRA	4763
XX	COMBINATORICS	4795
144	ENUMERATIVE COMBINATORICS	4797
145	PARTITIONS, WORDS AND YOUNG TABLEAUX	4803
146	SYMMETRIC FUNCTIONS	4837
147	INCIDENCE STRUCTURES AND DESIGNS	4863
148	HADAMARD MATRICES	4899
149	GRAPHS	4909
150	MULTIGRAPHS	4991
151	NETWORKS	5039

VOLUME 13: OVERVIEW

XXI	CODING THEORY	5059
152	LINEAR CODES OVER FINITE FIELDS	5061
153	ALGEBRAIC-GEOMETRIC CODES	5137
154	LOW DENSITY PARITY CHECK CODES	5147
155	LINEAR CODES OVER FINITE RINGS	5159
156	ADDITIVE CODES	5199
157	QUANTUM CODES	5225
XXII	CRYPTOGRAPHY	5263
158	PSEUDO-RANDOM BIT SEQUENCES	5265
XXIII	OPTIMIZATION	5273
159	LINEAR PROGRAMMING	5275

VOLUME 1: CONTENTS

I	THE MAGMA LANGUAGE	1
1	STATEMENTS AND EXPRESSIONS	3
1.1	<i>Introduction</i>	5
1.2	<i>Starting, Interrupting and Terminating</i>	5
1.3	<i>Identifiers</i>	5
1.4	<i>Assignment</i>	6
1.4.1	Simple Assignment	6
1.4.2	Indexed Assignment	7
1.4.3	Generator Assignment	8
1.4.4	Mutation Assignment	9
1.4.5	Deletion of Values	10
1.5	<i>Boolean values</i>	10
1.5.1	Creation of Booleans	11
1.5.2	Boolean Operators	11
1.5.3	Equality Operators	11
1.5.4	Iteration	12
1.6	<i>Coercion</i>	13
1.7	<i>The where ... is Construction</i>	14
1.8	<i>Conditional Statements and Expressions</i>	16
1.8.1	The Simple Conditional Statement	16
1.8.2	The Simple Conditional Expression	17
1.8.3	The Case Statement	18
1.8.4	The Case Expression	18
1.9	<i>Error Handling Statements</i>	19
1.9.1	The Error Objects	19
1.9.2	Error Checking and Assertions	19
1.9.3	Catching Errors	20
1.10	<i>Iterative Statements</i>	21
1.10.1	Definite Iteration	21
1.10.2	Indefinite Iteration	22
1.10.3	Early Exit from Iterative Statements	23
1.11	<i>Runtime Evaluation: the eval Expression</i>	24
1.12	<i>Comments and Continuation</i>	26
1.13	<i>Timing</i>	26
1.14	<i>Types, Category Names, and Structures</i>	28
1.15	<i>Random Object Generation</i>	30
1.16	<i>Miscellaneous</i>	32
1.17	<i>Bibliography</i>	32

2	FUNCTIONS, PROCEDURES AND PACKAGES	33
2.1	<i>Introduction</i>	35
2.2	<i>Functions and Procedures</i>	35
2.2.1	Functions	35
2.2.2	Procedures	39
2.2.3	The forward Declaration	41
2.3	<i>Packages</i>	42
2.3.1	Introduction	42
2.3.2	Intrinsics	43
2.3.3	Resolving Calls to Intrinsics	45
2.3.4	Attaching and Detaching Package Files	46
2.3.5	Related Files	47
2.3.6	Importing Constants	47
2.3.7	Argument Checking	48
2.3.8	Package Specification Files	49
2.3.9	User Startup Specification Files	50
2.4	<i>Attributes</i>	51
2.4.1	Predefined System Attributes	51
2.4.2	User-defined Attributes	52
2.4.3	Accessing Attributes	52
2.5	<i>User-defined Verbose Flags</i>	53
2.5.1	Examples	53
2.6	<i>User-Defined Types</i>	56
2.6.1	Declaring User-Defined Types	56
2.6.2	Creating an Object	57
2.6.3	Special Intrinsics Provided by the User	57
2.6.4	Examples	58
3	INPUT AND OUTPUT	63
3.1	<i>Introduction</i>	65
3.2	<i>Character Strings</i>	65
3.2.1	Representation of Strings	65
3.2.2	Creation of Strings	66
3.2.3	Integer-Valued Functions	67
3.2.4	Character Conversion	67
3.2.5	Boolean Functions	68
3.2.6	Parsing Strings	71
3.3	<i>Printing</i>	72
3.3.1	The print -Statement	72
3.3.2	The printf and fprintf Statements	73
3.3.3	Verbose Printing (vprint , vprintf)	75
3.3.4	Automatic Printing	76
3.3.5	Indentation	78
3.3.6	Printing to a File	78
3.3.7	Printing to a String	79
3.3.8	Redirecting Output	80
3.4	<i>External Files</i>	80
3.4.1	Opening Files	80
3.4.2	Operations on File Objects	81
3.4.3	Reading a Complete File	82
3.5	<i>Pipes</i>	83
3.5.1	Pipe Creation	83
3.5.2	Operations on Pipes	84
3.6	<i>Sockets</i>	85
3.6.1	Socket Creation	85

3.6.2	Socket Properties	86
3.6.3	Socket Predicates	86
3.6.4	Socket I/O	87
3.7	<i>Interactive Input</i>	88
3.8	<i>Loading a Program File</i>	89
3.9	<i>Saving and Restoring Workspaces</i>	89
3.10	<i>Logging a Session</i>	90
3.11	<i>Memory Usage</i>	90
3.12	<i>System Calls</i>	90
3.13	<i>Creating Names</i>	91
4	ENVIRONMENT AND OPTIONS	93
4.1	<i>Introduction</i>	95
4.2	<i>Command Line Options</i>	95
4.3	<i>Environment Variables</i>	97
4.4	<i>Set and Get</i>	98
4.5	<i>Verbose Levels</i>	102
4.6	<i>Other Information Procedures</i>	103
4.7	<i>History</i>	104
4.8	<i>The Magma Line Editor</i>	106
4.8.1	Key Bindings (Emacs and VI mode)	106
4.8.2	Key Bindings in Emacs mode only	108
4.8.3	Key Bindings in VI mode only	109
4.9	<i>The Magma Help System</i>	112
4.9.1	Internal Help Browser	113
5	MAGMA SEMANTICS	115
5.1	<i>Introduction</i>	117
5.2	<i>Terminology</i>	117
5.3	<i>Assignment</i>	118
5.4	<i>Uninitialized Identifiers</i>	118
5.5	<i>Evaluation in Magma</i>	119
5.5.1	Call by Value Evaluation	119
5.5.2	Magma's Evaluation Process	120
5.5.3	Function Expressions	121
5.5.4	Function Values Assigned to Identifiers	122
5.5.5	Recursion and Mutual Recursion	122
5.5.6	Function Application	123
5.5.7	The Initial Context	124
5.6	<i>Scope</i>	124
5.6.1	Local Declarations	125
5.6.2	The 'first use' Rule	125
5.6.3	Identifier Classes	126
5.6.4	The Evaluation Process Revisited	126
5.6.5	The 'single use' Rule	127
5.7	<i>Procedure Expressions</i>	127
5.8	<i>Reference Arguments</i>	129
5.9	<i>Dynamic Typing</i>	130
5.10	<i>Traps for Young Players</i>	131
5.10.1	Trap 1	131
5.10.2	Trap 2	131
5.11	<i>Appendix A: Precedence</i>	133
5.12	<i>Appendix B: Reserved Words</i>	134

6	THE MAGMA PROFILER	135
6.1	<i>Introduction</i>	137
6.2	<i>Profiler Basics</i>	137
6.3	<i>Exploring the Call Graph</i>	139
6.3.1	Internal Reports	139
6.3.2	HTML Reports	141
6.4	<i>Recursion and the Profiler</i>	141
7	DEBUGGING MAGMA CODE	145
7.1	<i>Introduction</i>	147
7.2	<i>Using the Debugger</i>	147

II	SETS, SEQUENCES, AND MAPPINGS	151
8	INTRODUCTION TO AGGREGATES	153
8.1	<i>Introduction</i>	155
8.2	<i>Restrictions on Sets and Sequences</i>	155
8.2.1	Universe of a Set or Sequence	156
8.2.2	Modifying the Universe of a Set or Sequence	157
8.2.3	Parents of Sets and Sequences	159
8.3	<i>Nested Aggregates</i>	160
8.3.1	Multi-indexing	160
9	SETS	163
9.1	<i>Introduction</i>	165
9.1.1	Enumerated Sets	165
9.1.2	Formal Sets	165
9.1.3	Indexed Sets	165
9.1.4	Multisets	165
9.1.5	Compatibility	166
9.1.6	Notation	166
9.2	<i>Creating Sets</i>	166
9.2.1	The Formal Set Constructor	166
9.2.2	The Enumerated Set Constructor	167
9.2.3	The Indexed Set Constructor	169
9.2.4	The Multiset Constructor	170
9.2.5	The Arithmetic Progression Constructors	172
9.3	<i>Power Sets</i>	173
9.3.1	The Cartesian Product Constructors	175
9.4	<i>Sets from Structures</i>	175
9.5	<i>Accessing and Modifying Sets</i>	176
9.5.1	Accessing Sets and their Associated Structures	176
9.5.2	Selecting Elements of Sets	177
9.5.3	Modifying Sets	180
9.6	<i>Operations on Sets</i>	183
9.6.1	Boolean Functions and Operators	183
9.6.2	Binary Set Operators	184
9.6.3	Other Set Operations	185
9.7	<i>Quantifiers</i>	186
9.8	<i>Reduction and Iteration over Sets</i>	189
10	SEQUENCES	191
10.1	<i>Introduction</i>	193
10.1.1	Enumerated Sequences	193
10.1.2	Formal Sequences	193
10.1.3	Compatibility	194
10.2	<i>Creating Sequences</i>	194
10.2.1	The Formal Sequence Constructor	194
10.2.2	The Enumerated Sequence Constructor	195
10.2.3	The Arithmetic Progression Constructors	196
10.2.4	Literal Sequences	197
10.3	<i>Power Sequences</i>	197
10.4	<i>Operators on Sequences</i>	198
10.4.1	Access Functions	198
10.4.2	Selection Operators on Enumerated Sequences	199

10.4.3	Modifying Enumerated Sequences	200
10.4.4	Creating New Enumerated Sequences from Existing Ones	205
10.5	<i>Predicates on Sequences</i>	208
10.5.1	Membership Testing	208
10.5.2	Testing Order Relations	209
10.6	<i>Recursion, Reduction, and Iteration</i>	210
10.6.1	Recursion	210
10.6.2	Reduction	211
10.7	<i>Iteration</i>	211
10.8	<i>Bibliography</i>	212
11	TUPLES AND CARTESIAN PRODUCTS	213
11.1	<i>Introduction</i>	215
11.2	<i>Cartesian Product Constructor and Functions</i>	215
11.3	<i>Creating and Modifying Tuples</i>	216
11.4	<i>Tuple Access Functions</i>	218
11.5	<i>Equality</i>	218
11.6	<i>Other operations</i>	219
12	LISTS	221
12.1	<i>Introduction</i>	223
12.2	<i>Construction of Lists</i>	223
12.3	<i>Creation of New Lists</i>	223
12.4	<i>Access Functions</i>	224
12.5	<i>Assignment Operator</i>	225
13	ASSOCIATIVE ARRAYS	227
13.1	<i>Introduction</i>	229
13.2	<i>Operations</i>	229
14	COPRODUCTS	233
14.1	<i>Introduction</i>	235
14.2	<i>Creation Functions</i>	235
14.2.1	Creation of Coproducts	235
14.2.2	Creation of Coproduct Elements	235
14.3	<i>Accessing Functions</i>	236
14.4	<i>Retrieve</i>	236
14.5	<i>Flattening</i>	237
14.6	<i>Universal Map</i>	237
15	RECORDS	239
15.1	<i>Introduction</i>	241
15.2	<i>The Record Format Constructor</i>	241
15.3	<i>Creating a Record</i>	242
15.4	<i>Access and Modification Functions</i>	243

16	MAPPINGS	245
16.1	<i>Introduction</i>	247
16.1.1	The Map Constructors	247
16.1.2	The Graph of a Map	248
16.1.3	Rules for Maps	248
16.1.4	Homomorphisms	248
16.1.5	Checking of Maps	248
16.2	<i>Creation Functions</i>	249
16.2.1	Creation of Maps	249
16.2.2	Creation of Partial Maps	250
16.2.3	Creation of Homomorphisms	250
16.2.4	Coercion Maps	251
16.3	<i>Operations on Mappings</i>	251
16.3.1	Composition	251
16.3.2	(Co)Domain and (Co)Kernel	252
16.3.3	Inverse	252
16.3.4	Function	252
16.4	<i>Images and Preimages</i>	253
16.5	<i>Parents of Maps</i>	254

VOLUME 2: CONTENTS

III	BASIC RINGS	255
17	INTRODUCTION TO RINGS	257
17.1	Overview	259
17.2	The World of Rings	260
17.2.1	New Rings from Existing Ones	260
17.2.2	Attributes	261
17.3	Coercion	261
17.3.1	Automatic Coercion	262
17.3.2	Forced Coercion	264
17.4	Generic Ring Functions	266
17.4.1	Related Structures	266
17.4.2	Numerical Invariants	266
17.4.3	Predicates and Boolean Operations	267
17.5	Generic Element Functions	268
17.5.1	Parent and Category	268
17.5.2	Creation of Elements	269
17.5.3	Arithmetic Operations	269
17.5.4	Equality and Membership	270
17.5.5	Predicates on Ring Elements	271
17.5.6	Comparison of Ring Elements	272
17.6	Ideals and Quotient Rings	273
17.6.1	Defining Ideals and Quotient Rings	273
17.6.2	Arithmetic Operations on Ideals	273
17.6.3	Boolean Operators on Ideals	274
17.7	Other Ring Constructions	274
17.7.1	Residue Class Fields	274
17.7.2	Localization	274
17.7.3	Completion	275
17.7.4	Transcendental Extension	275
18	RING OF INTEGERS	277
18.1	Introduction	281
18.1.1	Representation	281
18.1.2	Coercion	281
18.1.3	Homomorphisms	281
18.2	Creation Functions	282
18.2.1	Creation of Structures	282
18.2.2	Creation of Elements	282
18.2.3	Printing of Elements	283
18.2.4	Element Conversions	284
18.3	Structure Operations	285
18.3.1	Related Structures	285
18.3.2	Numerical Invariants	286
18.3.3	Ring Predicates and Booleans	286
18.4	Element Operations	286
18.4.1	Arithmetic Operations	286
18.4.2	Bit Operations	287

18.4.3	Equality and Membership	287
18.4.4	Parent and Category	287
18.4.5	Predicates on Ring Elements	288
18.4.6	Comparison of Ring Elements	289
18.4.7	Conjugates, Norm and Trace	289
18.4.8	Other Elementary Functions	290
18.5	<i>Random Numbers</i>	291
18.6	<i>Common Divisors and Common Multiples</i>	292
18.7	<i>Arithmetic Functions</i>	293
18.8	<i>Combinatorial Functions</i>	296
18.9	<i>Primes and Primality Testing</i>	297
18.9.1	Primality	297
18.9.2	Other Functions Relating to Primes	300
18.10	<i>Factorization</i>	301
18.10.1	General Factorization	302
18.10.2	Storing Potential Factors	304
18.10.3	Specific Factorization Algorithms	304
18.10.4	Factorization Related Functions	308
18.11	<i>Factorization Sequences</i>	310
18.11.1	Creation and Conversion	310
18.11.2	Arithmetic	310
18.11.3	Divisors	311
18.11.4	Predicates	311
18.12	<i>Modular Arithmetic</i>	311
18.12.1	Arithmetic Operations	311
18.12.2	The Solution of Modular Equations	312
18.13	<i>Infinities</i>	313
18.13.1	Creation	314
18.13.2	Arithmetic	314
18.13.3	Comparison	314
18.13.4	Miscellaneous	314
18.14	<i>Advanced Factorization Techniques: The Number Field Sieve</i>	315
18.14.1	The MAGMA Number Field Sieve Implementation	315
18.14.2	Naive NFS	316
18.14.3	Factoring with NFS Processes	316
18.14.4	Data files	321
18.14.5	Distributing NFS Factorizations	322
18.14.6	MAGMA and CWI NFS Interoperability	323
18.14.7	Tools for Finding a Suitable Polynomial	324
18.15	<i>Bibliography</i>	326
19	INTEGER RESIDUE CLASS RINGS	329
19.1	<i>Introduction</i>	331
19.2	<i>Ideals of \mathbf{Z}</i>	331
19.3	<i>\mathbf{Z} as a Number Field Order</i>	332
19.4	<i>Residue Class Rings</i>	333
19.4.1	Creation	333
19.4.2	Coercion	334
19.4.3	Elementary Invariants	335
19.4.4	Structure Operations	335
19.4.5	Ring Predicates and Booleans	336
19.4.6	Homomorphisms	336
19.5	<i>Elements of Residue Class Rings</i>	336
19.5.1	Creation	336
19.5.2	Arithmetic Operators	337

19.5.3	Equality and Membership	337
19.5.4	Parent and Category	337
19.5.5	Predicates on Ring Elements	337
19.5.6	Solving Equations over $\mathbf{Z}/m\mathbf{Z}$	337
19.6	<i>Ideal Operations</i>	339
19.7	<i>The Unit Group</i>	340
19.8	<i>Dirichlet Characters</i>	341
19.8.1	Creation	342
19.8.2	Element Creation	342
19.8.3	Properties of Dirichlet Groups	343
19.8.4	Properties of Elements	344
19.8.5	Evaluation	345
19.8.6	Arithmetic	346
19.8.7	Example	346
20	RATIONAL FIELD	349
20.1	<i>Introduction</i>	351
20.1.1	Representation	351
20.1.2	Coercion	351
20.1.3	Homomorphisms	352
20.2	<i>Creation Functions</i>	353
20.2.1	Creation of Structures	353
20.2.2	Creation of Elements	353
20.3	<i>Structure Operations</i>	354
20.3.1	Related Structures	354
20.3.2	Numerical Invariants	356
20.3.3	Ring Predicates and Booleans	356
20.4	<i>Element Operations</i>	357
20.4.1	Parent and Category	357
20.4.2	Arithmetic Operators	357
20.4.3	Numerator and Denominator	357
20.4.4	Equality and Membership	357
20.4.5	Predicates on Ring Elements	358
20.4.6	Comparison	358
20.4.7	Conjugates, Norm and Trace	358
20.4.8	Absolute Value and Sign	359
20.4.9	Rounding and Truncating	359
20.4.10	Rational Reconstruction	360
20.4.11	Valuation	360
20.4.12	Sequence Conversions	360
21	FINITE FIELDS	361
21.1	<i>Introduction</i>	363
21.1.1	Representation of Finite Fields	363
21.1.2	Conway Polynomials	363
21.1.3	Ground Field and Relationships	364
21.2	<i>Creation Functions</i>	364
21.2.1	Creation of Structures	364
21.2.2	Creating Relations	368
21.2.3	Special Options	368
21.2.4	Homomorphisms	370
21.2.5	Creation of Elements	370
21.2.6	Special Elements	371
21.2.7	Sequence Conversions	372
21.3	<i>Structure Operations</i>	372

21.3.1	Related Structures	373
21.3.2	Numerical Invariants	375
21.3.3	Defining Polynomial	375
21.3.4	Ring Predicates and Booleans	375
21.3.5	Roots	376
21.4	<i>Element Operations</i>	377
21.4.1	Arithmetic Operators	377
21.4.2	Equality and Membership	377
21.4.3	Parent and Category	377
21.4.4	Predicates on Ring Elements	378
21.4.5	Minimal and Characteristic Polynomial	378
21.4.6	Norm, Trace and Frobenius	379
21.4.7	Order and Roots	380
21.5	<i>Polynomials for Finite Fields</i>	382
21.6	<i>Discrete Logarithms</i>	383
21.7	<i>Permutation Polynomials</i>	386
21.8	<i>Bibliography</i>	387
22	NEARFIELDS	389
22.1	<i>Introduction</i>	391
22.2	<i>Nearfield Properties</i>	391
22.2.1	Sharply Doubly Transitive Groups	392
22.3	<i>Constructing Nearfields</i>	393
22.3.1	Dickson Nearfields	393
22.3.2	Zassenhaus Nearfields	396
22.4	<i>Operations on Elements</i>	397
22.4.1	Nearfield Arithmetic	397
22.4.2	Equality and Membership	397
22.4.3	Parent and Category	397
22.4.4	Predicates on Nearfield Elements	397
22.5	<i>Operations on Nearfields</i>	399
22.6	<i>The Group of Units</i>	400
22.7	<i>Automorphisms</i>	401
22.8	<i>Nearfield Planes</i>	402
22.8.1	Hughes Planes	403
22.9	<i>Bibliography</i>	404
23	UNIVARIATE POLYNOMIAL RINGS	407
23.1	<i>Introduction</i>	411
23.1.1	Representation	411
23.2	<i>Creation Functions</i>	411
23.2.1	Creation of Structures	411
23.2.2	Print Options	412
23.2.3	Creation of Elements	413
23.3	<i>Structure Operations</i>	415
23.3.1	Related Structures	415
23.3.2	Changing Rings	415
23.3.3	Numerical Invariants	416
23.3.4	Ring Predicates and Booleans	416
23.3.5	Homomorphisms	416
23.4	<i>Element Operations</i>	417
23.4.1	Parent and Category	417
23.4.2	Arithmetic Operators	417
23.4.3	Equality and Membership	417

23.4.4	Predicates on Ring Elements	418
23.4.5	Coefficients and Terms	418
23.4.6	Degree	419
23.4.7	Roots	420
23.4.8	Derivative, Integral	422
23.4.9	Evaluation, Interpolation	422
23.4.10	Quotient and Remainder	422
23.4.11	Modular Arithmetic	424
23.4.12	Other Operations	424
23.5	<i>Common Divisors and Common Multiples</i>	424
23.5.1	Common Divisors and Common Multiples	425
23.5.2	Content and Primitive Part	426
23.6	<i>Polynomials over the Integers</i>	427
23.7	<i>Polynomials over Finite Fields</i>	427
23.8	<i>Factorization</i>	428
23.8.1	Factorization and Irreducibility	428
23.8.2	Resultant and Discriminant	432
23.8.3	Hensel Lifting	433
23.9	<i>Ideals and Quotient Rings</i>	434
23.9.1	Creation of Ideals and Quotients	434
23.9.2	Ideal Arithmetic	434
23.9.3	Other Functions on Ideals	435
23.9.4	Other Functions on Quotients	436
23.10	<i>Special Families of Polynomials</i>	436
23.10.1	Orthogonal Polynomials	436
23.10.2	Permutation Polynomials	437
23.10.3	The Bernoulli Polynomial	438
23.10.4	Swinerton-Dyer Polynomials	438
23.11	<i>Bibliography</i>	438
24	MULTIVARIATE POLYNOMIAL RINGS	441
24.1	<i>Introduction</i>	443
24.1.1	Representation	443
24.2	<i>Polynomial Rings and Polynomials</i>	444
24.2.1	Creation of Polynomial Rings	444
24.2.2	Print Names	446
24.2.3	Graded Polynomial Rings	446
24.2.4	Creation of Polynomials	447
24.3	<i>Structure Operations</i>	447
24.3.1	Related Structures	447
24.3.2	Numerical Invariants	448
24.3.3	Ring Predicates and Booleans	448
24.3.4	Changing Coefficient Ring	448
24.3.5	Homomorphisms	448
24.4	<i>Element Operations</i>	449
24.4.1	Arithmetic Operators	449
24.4.2	Equality and Membership	449
24.4.3	Predicates on Ring Elements	450
24.4.4	Coefficients, Monomials and Terms	450
24.4.5	Degrees	455
24.4.6	Univariate Polynomials	456
24.4.7	Derivative, Integral	457
24.4.8	Evaluation, Interpolation	458
24.4.9	Quotient and Reductum	459
24.4.10	Diagonalizing a Polynomial of Degree 2	460
24.5	<i>Greatest Common Divisors</i>	461

24.5.1	Common Divisors and Common Multiples	461
24.5.2	Content and Primitive Part	462
24.6	<i>Factorization and Irreducibility</i>	463
24.7	<i>Resultants and Discriminants</i>	467
24.8	<i>Polynomials over the Integers</i>	467
24.9	<i>Bibliography</i>	468
25	REAL AND COMPLEX FIELDS	469
25.1	<i>Introduction</i>	473
25.1.1	Overview of Real Numbers in MAGMA	473
25.1.2	Coercion	474
25.1.3	Homomorphisms	475
25.1.4	Special Options	475
25.1.5	Version Functions	476
25.2	<i>Creation Functions</i>	476
25.2.1	Creation of Structures	476
25.2.2	Creation of Elements	478
25.3	<i>Structure Operations</i>	479
25.3.1	Related Structures	479
25.3.2	Numerical Invariants	479
25.3.3	Ring Predicates and Booleans	480
25.3.4	Other Structure Functions	480
25.4	<i>Element Operations</i>	480
25.4.1	Generic Element Functions and Predicates	480
25.4.2	Comparison of and Membership	481
25.4.3	Other Predicates	481
25.4.4	Arithmetic	481
25.4.5	Conversions	481
25.4.6	Rounding	482
25.4.7	Precision	483
25.4.8	Constants	483
25.4.9	Simple Element Functions	484
25.4.10	Roots	485
25.4.11	Continued Fractions	490
25.4.12	Algebraic Dependencies	491
25.5	<i>Transcendental Functions</i>	491
25.5.1	Exponential, Logarithmic and Polylogarithmic Functions	491
25.5.2	Trigonometric Functions	493
25.5.3	Inverse Trigonometric Functions	495
25.5.4	Hyperbolic Functions	497
25.5.5	Inverse Hyperbolic Functions	498
25.6	<i>Elliptic and Modular Functions</i>	499
25.6.1	Eisenstein Series	499
25.6.2	Weierstrass Series	501
25.6.3	The Jacobi θ and Dedekind η -functions	502
25.6.4	The j -invariant and the Discriminant	503
25.6.5	Weber's Functions	504
25.7	<i>Theta Functions</i>	505
25.8	<i>Gamma, Bessel and Associated Functions</i>	506
25.9	<i>The Hypergeometric Function</i>	508
25.10	<i>Other Special Functions</i>	509
25.11	<i>Numerical Functions</i>	511
25.11.1	Summation of Infinite Series	511
25.11.2	Integration	511
25.11.3	Numerical Derivatives	512
25.12	<i>Bibliography</i>	512

IV	MATRICES AND LINEAR ALGEBRA	515
26	MATRICES	517
26.1	Introduction	521
26.2	Creation of Matrices	521
26.2.1	General Matrix Construction	521
26.2.2	Shortcuts	523
26.2.3	Construction of Structured Matrices	525
26.2.4	Construction of Random Matrices	528
26.2.5	Creating Vectors	529
26.3	Elementary Properties	529
26.4	Accessing or Modifying Entries	530
26.4.1	Indexing	530
26.4.2	Extracting and Inserting Blocks	531
26.4.3	Row and Column Operations	534
26.5	Building Block Matrices	537
26.6	Changing Ring	538
26.7	Elementary Arithmetic	539
26.8	Nullspaces and Solutions of Systems	540
26.9	Predicates	543
26.10	Determinant and Other Properties	544
26.11	Minimal and Characteristic Polynomials and Eigenvalues	546
26.12	Canonical Forms	548
26.12.1	Canonical Forms over General Rings	548
26.12.2	Canonical Forms over Fields	548
26.12.3	Canonical Forms over Euclidean Domains	551
26.13	Orders of Invertible Matrices	554
26.14	Miscellaneous Operations on Matrices	555
26.15	Bibliography	555
27	SPARSE MATRICES	557
27.1	Introduction	559
27.2	Creation of Sparse Matrices	559
27.2.1	Construction of Initialized Sparse Matrices	559
27.2.2	Construction of Trivial Sparse Matrices	560
27.2.3	Construction of Structured Matrices	562
27.2.4	Parents of Sparse Matrices	562
27.3	Accessing Sparse Matrices	563
27.3.1	Elementary Properties	563
27.3.2	Weights	564
27.4	Accessing or Modifying Entries	564
27.4.1	Extracting and Inserting Blocks	566
27.4.2	Row and Column Operations	568
27.5	Building Block Matrices	569
27.6	Conversion to and from Dense Matrices	570
27.7	Changing Ring	570
27.8	Predicates	571
27.9	Elementary Arithmetic	572
27.10	Multiplying Vectors or Matrices by Sparse Matrices	573
27.11	Non-trivial Properties	573
27.11.1	Nullspace and Rowspace	573
27.11.2	Rank	574
27.12	Determinant and Other Properties	574

27.12.1	Elementary Divisors (Smith Form)	575
27.12.2	Verbosity	575
27.13	<i>Linear Systems (Structured Gaussian Elimination)</i>	575
27.14	<i>Bibliography</i>	582
28	VECTOR SPACES	583
28.1	<i>Introduction</i>	585
28.1.1	Vector Space Categories	585
28.1.2	The Construction of a Vector Space	585
28.2	<i>Creation of Vector Spaces and Arithmetic with Vectors</i>	586
28.2.1	Construction of a Vector Space	586
28.2.2	Construction of a Vector Space with Inner Product Matrix	587
28.2.3	Construction of a Vector	587
28.2.4	Deconstruction of a Vector	589
28.2.5	Arithmetic with Vectors	589
28.2.6	Indexing Vectors and Matrices	592
28.3	<i>Subspaces, Quotient Spaces and Homomorphisms</i>	594
28.3.1	Construction of Subspaces	594
28.3.2	Construction of Quotient Vector Spaces	596
28.4	<i>Changing the Coefficient Field</i>	598
28.5	<i>Basic Operations</i>	599
28.5.1	Accessing Vector Space Invariants	599
28.5.2	Membership and Equality	600
28.5.3	Operations on Subspaces	601
28.6	<i>Reducing Vectors Relative to a Subspace</i>	601
28.7	<i>Bases</i>	602
28.8	<i>Operations with Linear Transformations</i>	604
29	POLAR SPACES	607
29.1	<i>Introduction</i>	609
29.2	<i>Reflexive Forms</i>	609
29.2.1	Quadratic Forms	610
29.3	<i>Inner Products</i>	611
29.3.1	Orthogonality	613
29.4	<i>Isotropic and Singular Vectors and Subspaces</i>	614
29.5	<i>The Standard Forms</i>	617
29.6	<i>Constructing Polar Spaces</i>	620
29.6.1	Symplectic Spaces	621
29.6.2	Unitary Spaces	621
29.6.3	Quadratic Spaces	622
29.7	<i>Isometries and Similarities</i>	625
29.7.1	Isometries	625
29.7.2	Similarities	628
29.8	<i>Wall Forms</i>	629
29.9	<i>Invariant Forms</i>	630
29.9.1	Semi-invariant Forms	633
29.10	<i>Bibliography</i>	635

VOLUME 3: CONTENTS

V	LATTICES AND QUADRATIC FORMS	637
30	LATTICES	639
30.1	<i>Introduction</i>	643
30.2	<i>Presentation of Lattices</i>	644
30.3	<i>Creation of Lattices</i>	645
30.3.1	Elementary Creation of Lattices	645
30.3.2	Lattices from Linear Codes	649
30.3.3	Lattices from Algebraic Number Fields	650
30.3.4	Special Lattices	652
30.4	<i>Lattice Elements</i>	653
30.4.1	Creation of Lattice Elements	653
30.4.2	Operations on Lattice Elements	653
30.4.3	Predicates and Boolean Operations	655
30.4.4	Access Operations	655
30.5	<i>Properties of Lattices</i>	657
30.5.1	Associated Structures	657
30.5.2	Attributes of Lattices	658
30.5.3	Predicates and Booleans on Lattices	659
30.5.4	Base Ring and Base Change	660
30.6	<i>Construction of New Lattices</i>	660
30.6.1	Sub- and Superlattices and Quotients	660
30.6.2	Standard Constructions of New Lattices	662
30.7	<i>Reduction of Matrices and Lattices</i>	665
30.7.1	LLL Reduction	665
30.7.2	Pair Reduction	675
30.7.3	Seysen Reduction	676
30.7.4	HKZ Reduction	677
30.7.5	Recovering a Short Basis from Short Lattice Vectors	680
30.8	<i>Minima and Element Enumeration</i>	680
30.8.1	Minimum, Density and Kissing Number	681
30.8.2	Shortest and Closest Vectors	683
30.8.3	Short and Close Vectors	685
30.8.4	Short and Close Vector Processes	691
30.8.5	Successive Minima and Theta Series	692
30.8.6	Lattice Enumeration Utilities	693
30.9	<i>Theta Series as Modular Forms</i>	696
30.10	<i>Voronoi Cells, Holes and Covering Radius</i>	697
30.11	<i>Orthogonalization</i>	699
30.12	<i>Testing Matrices for Definiteness</i>	701
30.13	<i>Genera and Spinor Genera</i>	702
30.13.1	Genus Constructions	702
30.13.2	Invariants of Genera and Spinor Genera	702
30.13.3	Invariants of p -adic Genera	704
30.13.4	Neighbour Relations and Graphs	704
30.14	<i>Attributes of Lattices</i>	708
30.15	<i>Database of Lattices</i>	708
30.15.1	Creating the Database	709
30.15.2	Database Information	709

30.15.3	Accessing the Database	710
30.15.4	Hermitian Lattices	712
30.16	<i>Bibliography</i>	714
31	LATTICES WITH GROUP ACTION	717
31.1	<i>Introduction</i>	719
31.2	<i>Automorphism Group and Isometry Testing</i>	719
31.2.1	Automorphism Group and Isometry Testing over $\mathbf{F}_q[t]$	726
31.3	<i>Lattices from Matrix Groups</i>	728
31.3.1	Creation of G -Lattices	728
31.3.2	Operations on G -Lattices	729
31.3.3	Invariant Forms	729
31.3.4	Endomorphisms	730
31.3.5	G -invariant Sublattices	731
31.3.6	Lattice of Sublattices	735
31.4	<i>Bibliography</i>	741
32	QUADRATIC FORMS	743
32.1	<i>Introduction</i>	745
32.2	<i>Constructions and Conversions</i>	745
32.3	<i>Local Invariants</i>	746
32.4	<i>Isotropic Subspaces</i>	747
32.5	<i>Bibliography</i>	750
33	BINARY QUADRATIC FORMS	751
33.1	<i>Introduction</i>	753
33.2	<i>Creation Functions</i>	753
33.2.1	Creation of Structures	753
33.2.2	Creation of Forms	754
33.3	<i>Basic Invariants</i>	754
33.4	<i>Operations on Forms</i>	755
33.4.1	Arithmetic	755
33.4.2	Attribute Access	756
33.4.3	Boolean Operations	756
33.4.4	Related Structures	757
33.5	<i>Class Group</i>	757
33.6	<i>Class Group Coercions</i>	760
33.7	<i>Discrete Logarithms</i>	760
33.8	<i>Elliptic and Modular Invariants</i>	761
33.9	<i>Class Invariants</i>	762
33.10	<i>Matrix Action on Forms</i>	763
33.11	<i>Bibliography</i>	763

VI	GLOBAL ARITHMETIC FIELDS	765
34	NUMBER FIELDS	767
34.1	<i>Introduction</i>	771
34.2	<i>Creation Functions</i>	773
34.2.1	Creation of Number Fields	773
34.2.2	Maximal Orders	779
34.2.3	Creation of Elements	780
34.2.4	Creation of Homomorphisms	781
34.3	<i>Structure Operations</i>	782
34.3.1	General Functions	782
34.3.2	Related Structures	783
34.3.3	Representing Fields as Vector Spaces	786
34.3.4	Invariants	788
34.3.5	Basis Representation	790
34.3.6	Ring Predicates	792
34.3.7	Field Predicates	793
34.4	<i>Element Operations</i>	793
34.4.1	Parent and Category	793
34.4.2	Arithmetic	794
34.4.3	Equality and Membership	794
34.4.4	Predicates on Elements	795
34.4.5	Finding Special Elements	795
34.4.6	Real and Complex Valued Functions	796
34.4.7	Norm, Trace, and Minimal Polynomial	798
34.4.8	Other Functions	800
34.5	<i>Class and Unit Groups</i>	800
34.6	<i>Galois Theory</i>	803
34.7	<i>Solving Norm Equations</i>	804
34.8	<i>Places and Divisors</i>	807
34.8.1	Creation of Structures	807
34.8.2	Operations on Structures	807
34.8.3	Creation of Elements	807
34.8.4	Arithmetic with Places and Divisors	808
34.8.5	Other Functions for Places and Divisors	808
34.9	<i>Characters</i>	811
34.9.1	Creation Functions	811
34.9.2	Functions on Groups and Group Elements	811
34.9.3	Predicates on Group Elements	814
34.9.4	Passing between Dirichlet and Hecke Characters	814
34.9.5	L-functions of Hecke Characters	819
34.9.6	Hecke Größencharacters and their L-functions	819
34.10	<i>Number Field Database</i>	826
34.10.1	Creation	826
34.10.2	Access	827
34.11	<i>Bibliography</i>	829

35	QUADRATIC FIELDS	831
35.1	<i>Introduction</i>	833
35.1.1	Representation	833
35.2	<i>Creation of Structures</i>	834
35.3	<i>Operations on Structures</i>	835
35.3.1	Ideal Class Group	836
35.3.2	Norm Equations	839
35.4	<i>Special Element Operations</i>	840
35.4.1	Greatest Common Divisors	840
35.4.2	Modular Arithmetic	840
35.4.3	Factorization	841
35.4.4	Conjugates	841
35.4.5	Other Element Functions	841
35.5	<i>Special Functions for Ideals</i>	843
35.6	<i>Bibliography</i>	843
36	CYCLOTOMIC FIELDS	845
36.1	<i>Introduction</i>	847
36.2	<i>Creation Functions</i>	847
36.2.1	Creation of Cyclotomic Fields	847
36.2.2	Creation of Elements	848
36.3	<i>Structure Operations</i>	849
36.3.1	Invariants	850
36.4	<i>Element Operations</i>	850
36.4.1	Predicates on Elements	850
36.4.2	Conjugates	850
37	ORDERS AND ALGEBRAIC FIELDS	853
37.1	<i>Introduction</i>	859
37.2	<i>Creation Functions</i>	861
37.2.1	Creation of General Algebraic Fields	861
37.2.2	Creation of Orders and Fields from Orders	865
37.2.3	Maximal Orders	870
37.2.4	Creation of Elements	875
37.2.5	Creation of Homomorphisms	877
37.3	<i>Special Options</i>	879
37.4	<i>Structure Operations</i>	881
37.4.1	General Functions	882
37.4.2	Related Structures	883
37.4.3	Representing Fields as Vector Spaces	889
37.4.4	Invariants	891
37.4.5	Basis Representation	895
37.4.6	Ring Predicates	899
37.4.7	Order Predicates	900
37.4.8	Field Predicates	901
37.4.9	Setting Properties of Orders	902
37.5	<i>Element Operations</i>	903
37.5.1	Parent and Category	903
37.5.2	Arithmetic	903
37.5.3	Equality and Membership	904
37.5.4	Predicates on Elements	904
37.5.5	Finding Special Elements	905
37.5.6	Real and Complex Valued Functions	906
37.5.7	Norm, Trace, and Minimal Polynomial	908

37.5.8	Other Functions	910
37.6	<i>Ideal Class Groups</i>	911
37.6.1	Setting the Class Group Bounds Globally	919
37.7	<i>Unit Groups</i>	920
37.8	<i>Solving Equations</i>	923
37.8.1	Norm Equations	923
37.8.2	Thue Equations	927
37.8.3	Unit Equations	929
37.8.4	Index Form Equations	929
37.9	<i>Ideals and Quotients</i>	930
37.9.1	Creation of Ideals in Orders	931
37.9.2	Invariants	932
37.9.3	Basis Representation	935
37.9.4	Two-Element Presentations	936
37.9.5	Predicates on Ideals	937
37.9.6	Ideal Arithmetic	939
37.9.7	Roots of Ideals	942
37.9.8	Factorization and Primes	942
37.9.9	Other Ideal Operations	944
37.9.10	Quotient Rings	949
37.10	<i>Places and Divisors</i>	952
37.10.1	Creation of Structures	952
37.10.2	Operations on Structures	952
37.10.3	Creation of Elements	953
37.10.4	Arithmetic with Places and Divisors	954
37.10.5	Other Functions for Places and Divisors	954
37.11	<i>Bibliography</i>	956
38	GALOIS THEORY OF NUMBER FIELDS	959
38.1	<i>Automorphism Groups</i>	962
38.2	<i>Galois Groups</i>	969
38.2.1	Straight-line Polynomials	973
38.2.2	Invariants	975
38.2.3	Subfields and Subfield Towers	977
38.2.4	Solvability by Radicals	984
38.2.5	Linear Relations	985
38.2.6	Other	988
38.3	<i>Subfields</i>	988
38.3.1	The Subfield Lattice	989
38.4	<i>Galois Cohomology</i>	992
38.5	<i>Bibliography</i>	993
39	CLASS FIELD THEORY	995
39.1	<i>Introduction</i>	997
39.1.1	Overview	997
39.1.2	MAGMA	998
39.2	<i>Creation</i>	1001
39.2.1	Ray Class Groups	1001
39.2.2	Maps	1006
39.2.3	Abelian Extensions	1007
39.2.4	Binary Operations	1012
39.3	<i>Galois Module Structure</i>	1012
39.3.1	Predicates	1013
39.3.2	Constructions	1013

	39.4	<i>Conversion to Number Fields</i>	1014
	39.5	<i>Invariants</i>	1015
	39.6	<i>Automorphisms</i>	1018
	39.7	<i>Norm Equations</i>	1020
	39.8	<i>Attributes</i>	1023
	39.8.1	Orders	1023
	39.8.2	Abelian Extensions	1026
	39.9	<i>Group Theoretic Functions</i>	1030
	39.9.1	Generic Groups	1030
	39.10	<i>Bibliography</i>	1031
40		ALGEBRAICALLY CLOSED FIELDS	1033
	40.1	<i>Introduction</i>	1035
	40.2	<i>Representation</i>	1035
	40.3	<i>Creation of Structures</i>	1036
	40.4	<i>Creation of Elements</i>	1037
	40.4.1	Coercion	1037
	40.4.2	Roots	1037
	40.4.3	Variables	1038
	40.5	<i>Related Structures</i>	1043
	40.6	<i>Properties</i>	1043
	40.7	<i>Ring Predicates and Properties</i>	1044
	40.8	<i>Element Operations</i>	1044
	40.8.1	Arithmetic Operators	1045
	40.8.2	Equality and Membership	1045
	40.8.3	Parent and Category	1045
	40.8.4	Predicates on Ring Elements	1045
	40.8.5	Minimal Polynomial, Norm and Trace	1046
	40.9	<i>Simplification</i>	1048
	40.10	<i>Absolute Field</i>	1049
	40.11	<i>Bibliography</i>	1053
41		RATIONAL FUNCTION FIELDS	1055
	41.1	<i>Introduction</i>	1057
	41.2	<i>Creation Functions</i>	1057
	41.2.1	Creation of Structures	1057
	41.2.2	Names	1058
	41.2.3	Creation of Elements	1059
	41.3	<i>Structure Operations</i>	1059
	41.3.1	Related Structures	1059
	41.3.2	Invariants	1060
	41.3.3	Ring Predicates and Booleans	1060
	41.3.4	Homomorphisms	1060
	41.4	<i>Element Operations</i>	1061
	41.4.1	Arithmetic	1061
	41.4.2	Equality and Membership	1061
	41.4.3	Numerator, Denominator and Degree	1062
	41.4.4	Predicates on Ring Elements	1062
	41.4.5	Evaluation	1062
	41.4.6	Derivative	1063
	41.4.7	Partial Fraction Decomposition	1063
	41.5	<i>Padé-Hermite Approximants</i>	1066
	41.5.1	Introduction	1066
	41.5.2	Ordering of Sequences	1066

41.5.3	Approximants	1070
41.6	<i>Bibliography</i>	1075
42	ALGEBRAIC FUNCTION FIELDS	1077
42.1	<i>Introduction</i>	1085
42.1.1	Representations of Fields	1085
42.2	<i>Creation of Algebraic Function Fields and their Orders</i>	1086
42.2.1	Creation of Algebraic Function Fields	1086
42.2.2	Creation of Orders of Algebraic Function Fields	1089
42.2.3	Orders and Ideals	1094
42.3	<i>Related Structures</i>	1095
42.3.1	Parent and Category	1095
42.3.2	Other Related Structures	1095
42.4	<i>General Structure Invariants</i>	1099
42.5	<i>Galois Groups</i>	1104
42.6	<i>Subfields</i>	1108
42.7	<i>Automorphism Group</i>	1109
42.7.1	Automorphisms over the Base Field	1110
42.7.2	General Automorphisms	1112
42.7.3	Field Morphisms	1114
42.8	<i>Global Function Fields</i>	1117
42.8.1	Functions relative to the Exact Constant Field	1117
42.8.2	Functions Relative to the Constant Field	1119
42.8.3	Functions related to Class Group	1120
42.9	<i>Structure Predicates</i>	1124
42.10	<i>Homomorphisms</i>	1125
42.11	<i>Elements</i>	1126
42.11.1	Creation of Elements	1127
42.11.2	Parent and Category	1128
42.11.3	Sequence Conversions	1129
42.11.4	Arithmetic Operators	1130
42.11.5	Equality and Membership	1130
42.11.6	Predicates on Elements	1130
42.11.7	Functions related to Norm and Trace	1131
42.11.8	Functions related to Orders and Integrality	1133
42.11.9	Functions related to Places and Divisors	1134
42.11.10	Other Operations on Elements	1137
42.12	<i>Ideals</i>	1140
42.12.1	Creation of Ideals	1140
42.12.2	Parent and Category	1140
42.12.3	Arithmetic Operators	1141
42.12.4	Roots of Ideals	1141
42.12.5	Equality and Membership	1143
42.12.6	Predicates on Ideals	1143
42.12.7	Further Ideal Operations	1145
42.13	<i>Places</i>	1151
42.13.1	Creation of Structures	1151
42.13.2	Creation of Elements	1151
42.13.3	Related Structures	1153
42.13.4	Structure Invariants	1153
42.13.5	Structure Predicates	1154
42.13.6	Element Operations	1154
42.13.7	Completion at Places	1157
42.14	<i>Divisors</i>	1157
42.14.1	Creation of Structures	1157

42.14.2	Creation of Elements	1157
42.14.3	Related Structures	1158
42.14.4	Structure Invariants	1158
42.14.5	Structure Predicates	1158
42.14.6	Element Operations	1158
42.14.7	Functions related to Divisor Class Groups of Global Function Fields	1169
42.15	<i>Differentials</i>	1174
42.15.1	Creation of Structures	1174
42.15.2	Creation of Elements	1174
42.15.3	Related Structures	1175
42.15.4	Subspaces	1175
42.15.5	Structure Predicates	1176
42.15.6	Operations on Elements	1176
42.16	<i>Weil Descent</i>	1180
42.17	<i>Function Field Database</i>	1182
42.17.1	Creation	1183
42.17.2	Access	1183
42.18	<i>Bibliography</i>	1184
43	CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS	1187
43.1	<i>Ray Class Groups</i>	1189
43.2	<i>Creation of Class Fields</i>	1192
43.3	<i>Properties of Class Fields</i>	1194
43.4	<i>The Ring of Witt Vectors of Finite Length</i>	1197
43.5	<i>The Ring of Twisted Polynomials</i>	1199
43.5.1	Creation of Twisted Polynomial Rings	1199
43.5.2	Operations with the Ring of Twisted Polynomials	1200
43.5.3	Creation of Twisted Polynomials	1200
43.5.4	Operations with Twisted Polynomials	1202
43.6	<i>Analytic Theory</i>	1203
43.7	<i>Related Functions</i>	1209
43.8	<i>Enumeration of Places</i>	1211
43.9	<i>Bibliography</i>	1212
44	ARTIN REPRESENTATIONS	1213
44.1	Overview	1215
44.2	Constructing Artin Representations	1215
44.3	Basic Invariants	1217
44.4	Arithmetic	1220
44.5	Implementation Notes	1222
44.6	Bibliography	1222

VOLUME 4: CONTENTS

VII	LOCAL ARITHMETIC FIELDS	1223
45	VALUATION RINGS	1225
45.1	<i>Introduction</i>	1227
45.2	<i>Creation Functions</i>	1227
45.2.1	Creation of Structures	1227
45.2.2	Creation of Elements	1227
45.3	<i>Structure Operations</i>	1228
45.3.1	Related Structures	1228
45.3.2	Numerical Invariants	1228
45.4	<i>Element Operations</i>	1228
45.4.1	Arithmetic Operations	1228
45.4.2	Equality and Membership	1228
45.4.3	Parent and Category	1228
45.4.4	Predicates on Ring Elements	1229
45.4.5	Other Element Functions	1229
46	NEWTON POLYGONS	1231
46.1	<i>Introduction</i>	1233
46.2	<i>Newton Polygons</i>	1235
46.2.1	Creation of Newton Polygons	1235
46.2.2	Vertices and Faces of Polygons	1237
46.2.3	Tests for Points and Faces	1241
46.3	<i>Polynomials Associated with Newton Polygons</i>	1242
46.4	<i>Finding Valuations of Roots of Polynomials from Newton Polygons</i>	1243
46.5	<i>Using Newton Polygons to Find Roots of Polynomials over Series Rings</i>	1243
46.5.1	Operations not associated with Duval's Algorithm	1244
46.5.2	Operations associated with Duval's algorithm	1249
46.5.3	Roots of Polynomials	1256
46.6	<i>Bibliography</i>	1258
47	<i>p</i> -ADIC RINGS AND THEIR EXTENSIONS	1259
47.1	<i>Introduction</i>	1263
47.2	<i>Background</i>	1263
47.3	<i>Overview of the p-adics in MAGMA</i>	1264
47.3.1	<i>p</i> -adic Rings	1264
47.3.2	<i>p</i> -adic Fields	1264
47.3.3	Free Precision Rings and Fields	1265
47.3.4	Precision of Extensions	1265
47.4	<i>Creation of Local Rings and Fields</i>	1265
47.4.1	Creation Functions for the p -adics	1265
47.4.2	Creation Functions for Unramified Extensions	1267
47.4.3	Creation Functions for Totally Ramified Extensions	1269
47.4.4	Creation Functions for Unbounded Precision Extensions	1270
47.4.5	Miscellaneous Creation Functions	1271
47.4.6	Other Elementary Constructions	1272

47.4.7	Attributes of Local Rings and Fields	1272
47.5	<i>Elementary Invariants</i>	1272
47.6	<i>Operations on Structures</i>	1276
47.6.1	Ramification Predicates	1278
47.7	<i>Element Constructions and Conversions</i>	1279
47.7.1	Constructions	1279
47.7.2	Element Decomposers	1282
47.8	<i>Operations on Elements</i>	1283
47.8.1	Arithmetic	1283
47.8.2	Equality and Membership	1284
47.8.3	Properties	1286
47.8.4	Precision and Valuation	1286
47.8.5	Logarithms and Exponentials	1288
47.8.6	Norm and Trace Functions	1289
47.8.7	Teichmüller Lifts	1291
47.9	<i>Linear Algebra</i>	1291
47.10	<i>Roots of Elements</i>	1291
47.11	<i>Polynomials</i>	1292
47.11.1	Operations for Polynomials	1292
47.11.2	Roots of Polynomials	1294
47.11.3	Factorization	1298
47.12	<i>Automorphisms of Local Rings and Fields</i>	1302
47.13	<i>Completions</i>	1304
47.14	<i>Class Field Theory</i>	1305
47.14.1	Unit Group	1305
47.14.2	Norm Group	1306
47.14.3	Class Fields	1307
47.15	<i>Extensions</i>	1307
47.16	<i>Bibliography</i>	1308
48	GALOIS RINGS	1309
48.1	<i>Introduction</i>	1311
48.2	<i>Creation Functions</i>	1311
48.2.1	Creation of Structures	1311
48.2.2	Names	1312
48.2.3	Creation of Elements	1313
48.2.4	Sequence Conversions	1313
48.3	<i>Structure Operations</i>	1314
48.3.1	Related Structures	1314
48.3.2	Numerical Invariants	1315
48.3.3	Ring Predicates and Booleans	1315
48.4	<i>Element Operations</i>	1315
48.4.1	Arithmetic Operators	1315
48.4.2	Euclidean Operations	1316
48.4.3	Equality and Membership	1316
48.4.4	Parent and Category	1316
48.4.5	Predicates on Ring Elements	1316

49	POWER, LAURENT AND PUISEUX SERIES	1317
49.1	<i>Introduction</i>	1319
49.1.1	Kinds of Series	1319
49.1.2	Puiseux Series	1319
49.1.3	Representation of Series	1320
49.1.4	Precision	1320
49.1.5	Free and Fixed Precision	1320
49.1.6	Equality	1321
49.1.7	Polynomials over Series Rings	1321
49.2	<i>Creation Functions</i>	1321
49.2.1	Creation of Structures	1321
49.2.2	Special Options	1323
49.2.3	Creation of Elements	1324
49.3	<i>Structure Operations</i>	1325
49.3.1	Related Structures	1325
49.3.2	Invariants	1326
49.3.3	Ring Predicates and Booleans	1326
49.4	<i>Basic Element Operations</i>	1326
49.4.1	Parent and Category	1326
49.4.2	Arithmetic Operators	1326
49.4.3	Equality and Membership	1327
49.4.4	Predicates on Ring Elements	1327
49.4.5	Precision	1327
49.4.6	Coefficients and Degree	1328
49.4.7	Evaluation and Derivative	1329
49.4.8	Square Root	1330
49.4.9	Composition and Reversion	1330
49.5	<i>Transcendental Functions</i>	1332
49.5.1	Exponential and Logarithmic Functions	1332
49.5.2	Trigonometric Functions and their Inverses	1334
49.5.3	Hyperbolic Functions and their Inverses	1334
49.6	<i>The Hypergeometric Series</i>	1335
49.7	<i>Polynomials over Series Rings</i>	1335
49.8	<i>Extensions of Series Rings</i>	1338
49.8.1	Constructions of Extensions	1338
49.8.2	Operations on Extensions	1339
49.8.3	Elements of Extensions	1342
49.8.4	Optimized Representation	1343
49.9	<i>Bibliography</i>	1344
50	LAZY POWER SERIES RINGS	1345
50.1	<i>Introduction</i>	1347
50.2	<i>Creation of Lazy Series Rings</i>	1348
50.3	<i>Functions on Lazy Series Rings</i>	1348
50.4	<i>Elements</i>	1349
50.4.1	Creation of Finite Lazy Series	1349
50.4.2	Arithmetic with Lazy Series	1352
50.4.3	Finding Coefficients of Lazy Series	1353
50.4.4	Predicates on Lazy Series	1356
50.4.5	Other Functions on Lazy Series	1357

51	GENERAL LOCAL FIELDS	1361
51.1	<i>Introduction</i>	1363
51.2	<i>Constructions</i>	1363
51.3	<i>Operations with Fields</i>	1364
51.3.1	Predicates on Fields	1367
51.4	<i>Maximal Order</i>	1367
51.5	<i>Homomorphisms from Fields</i>	1368
51.6	<i>Automorphisms and Galois Theory</i>	1368
51.7	<i>Local Field Elements</i>	1369
51.7.1	Arithmetic	1369
51.7.2	Predicates on Elements	1369
51.7.3	Other Operations on Elements	1370
51.8	<i>Polynomials over General Local Fields</i>	1371
52	ALGEBRAIC POWER SERIES RINGS	1373
52.1	<i>Introduction</i>	1375
52.2	<i>Basics</i>	1375
52.2.1	Data Structures	1375
52.2.2	Verbose Output	1376
52.3	<i>Constructors</i>	1376
52.3.1	Rational Puiseux Expansions	1377
52.4	<i>Accessors and Expansion</i>	1381
52.5	<i>Arithmetic</i>	1382
52.6	<i>Predicates</i>	1383
52.7	<i>Modifiers</i>	1384
52.8	<i>Bibliography</i>	1385

VIII	MODULES	1387
53	INTRODUCTION TO MODULES	1389
53.1	Overview	1391
53.2	General Modules	1391
53.3	The Presentation of Submodules	1392
54	FREE MODULES	1393
54.1	Introduction	1395
54.1.1	Free Modules	1395
54.1.2	Module Categories	1395
54.1.3	Presentation of Submodules	1396
54.1.4	Notation	1396
54.2	Definition of a Module	1396
54.2.1	Construction of Modules of n -tuples	1396
54.2.2	Construction of Modules of $m \times n$ Matrices	1397
54.2.3	Construction of a Module with Specified Basis	1397
54.3	Accessing Module Information	1397
54.4	Standard Constructions	1398
54.4.1	Changing the Coefficient Ring	1398
54.4.2	Direct Sums	1398
54.5	Elements	1399
54.6	Construction of Elements	1399
54.6.1	Deconstruction of Elements	1400
54.6.2	Operations on Module Elements	1400
54.6.3	Properties of Vectors	1402
54.6.4	Inner Products	1402
54.7	Bases	1403
54.8	Submodules	1403
54.8.1	Construction of Submodules	1403
54.8.2	Operations on Submodules	1404
54.8.3	Membership and Equality	1404
54.8.4	Operations on Submodules	1405
54.9	Quotient Modules	1405
54.9.1	Construction of Quotient Modules	1405
54.10	Homomorphisms	1406
54.10.1	$\text{Hom}_R(M, N)$ for R -modules	1406
54.10.2	$\text{Hom}_R(M, N)$ for Matrix Modules	1407
54.10.3	Modules $\text{Hom}_R(M, N)$ with Given Basis	1409
54.10.4	The Endomorphsim Ring	1409
54.10.5	The Reduced Form of a Matrix Module	1410
54.10.6	Construction of a Matrix	1413
54.10.7	Element Operations	1414
55	MODULES OVER DEDEKIND DOMAINS	1417
55.1	Introduction	1419
55.2	Creation of Modules	1420
55.3	Elementary Functions	1424
55.4	Predicates on Modules	1426
55.5	Arithmetic with Modules	1427
55.6	Basis of a Module	1428
55.7	Other Functions on Modules	1429

55.8	<i>Homomorphisms between Modules</i>	1431
55.9	<i>Elements of Modules</i>	1434
55.9.1	Creation of Elements	1434
55.9.2	Arithmetic with Elements	1435
55.9.3	Other Functions on Elements	1435
55.10	<i>Pseudo Matrices</i>	1436
55.10.1	Construction of a Pseudo Matrix	1436
55.10.2	Elementary Functions	1436
55.10.3	Basis of a Pseudo Matrix	1437
55.10.4	Predicates	1437
55.10.5	Operations with Pseudo Matrices	1437
56	CHAIN COMPLEXES	1439
56.1	<i>Complexes of Modules</i>	1441
56.1.1	Creation	1441
56.1.2	Subcomplexes and Quotient Complexes	1442
56.1.3	Access Functions	1442
56.1.4	Elementary Operations	1443
56.1.5	Extensions	1444
56.1.6	Predicates	1445
56.2	<i>Chain Maps</i>	1447
56.2.1	Creation	1448
56.2.2	Access Functions	1448
56.2.3	Elementary Operations	1449
56.2.4	Predicates	1449
56.2.5	Maps on Homology	1452

VOLUME 5: CONTENTS

IX	FINITE GROUPS	1455
57	GROUPS	1457
57.1	<i>Introduction</i>	1461
57.1.1	The Categories of Finite Groups	1461
57.2	<i>Construction of Elements</i>	1462
57.2.1	Construction of an Element	1462
57.2.2	Coercion	1462
57.2.3	Homomorphisms	1462
57.2.4	Arithmetic with Elements	1464
57.3	<i>Construction of a General Group</i>	1466
57.3.1	The General Group Constructors	1466
57.3.2	Construction of Subgroups	1470
57.3.3	Construction of Quotient Groups	1471
57.4	<i>Standard Groups and Extensions</i>	1473
57.4.1	Construction of a Standard Group	1473
57.4.2	Construction of Extensions	1475
57.5	<i>Transfer Functions Between Group Categories</i>	1476
57.6	<i>Basic Operations</i>	1479
57.6.1	Accessing Group Information	1480
57.7	<i>Operations on the Set of Elements</i>	1481
57.7.1	Order and Index Functions	1481
57.7.2	Membership and Equality	1482
57.7.3	Set Operations	1483
57.7.4	Random Elements	1484
57.7.5	Action on a Coset Space	1487
57.8	<i>Standard Subgroup Constructions</i>	1488
57.8.1	Abstract Group Predicates	1489
57.9	<i>Characteristic Subgroups and Normal Structure</i>	1491
57.9.1	Characteristic Subgroups and Subgroup Series	1491
57.9.2	The Abstract Structure of a Group	1493
57.10	<i>Conjugacy Classes of Elements</i>	1494
57.11	<i>Conjugacy Classes of Subgroups</i>	1498
57.11.1	Conjugacy Classes of Subgroups	1498
57.11.2	The Poset of Subgroup Classes	1502
57.12	<i>Cohomology</i>	1507
57.13	<i>Characters and Representations</i>	1508
57.13.1	Character Theory	1508
57.13.2	Representation Theory	1509
57.14	<i>Databases of Groups</i>	1511
57.15	<i>Bibliography</i>	1511

58	PERMUTATION GROUPS	1513
58.1	<i>Introduction</i>	1519
58.1.1	Terminology	1519
58.1.2	The Category of Permutation Groups	1519
58.1.3	The Construction of a Permutation Group	1519
58.2	<i>Creation of a Permutation Group</i>	1520
58.2.1	Construction of the Symmetric Group	1520
58.2.2	Construction of a Permutation	1521
58.2.3	Construction of a General Permutation Group	1523
58.3	<i>Elementary Properties of a Group</i>	1524
58.3.1	Accessing Group Information	1524
58.3.2	Group Order	1526
58.3.3	Abstract Properties of a Group	1526
58.4	<i>Homomorphisms</i>	1527
58.5	<i>Building Permutation Groups</i>	1530
58.5.1	Some Standard Permutation Groups	1530
58.5.2	Direct Products and Wreath Products	1532
58.6	<i>Permutations</i>	1534
58.6.1	Coercion	1534
58.6.2	Arithmetic with Permutations	1534
58.6.3	Properties of Permutations	1535
58.6.4	Predicates for Permutations	1536
58.6.5	Set Operations	1537
58.7	<i>Conjugacy</i>	1539
58.8	<i>Subgroups</i>	1546
58.8.1	Construction of a Subgroup	1546
58.8.2	Membership and Equality	1548
58.8.3	Elementary Properties of a Subgroup	1549
58.8.4	Standard Subgroups	1550
58.8.5	Maximal Subgroups	1553
58.8.6	Conjugacy Classes of Subgroups	1555
58.8.7	Classes of Subgroups Satisfying a Condition	1560
58.9	<i>Quotient Groups</i>	1561
58.9.1	Construction of Quotient Groups	1561
58.9.2	Abelian, Nilpotent and Soluble Quotients	1562
58.10	<i>Permutation Group Actions</i>	1564
58.10.1	<i>G</i> -Sets	1564
58.10.2	Creating a <i>G</i> -Set	1564
58.10.3	Images, Orbits and Stabilizers	1567
58.10.4	Action on a <i>G</i> -Space	1572
58.10.5	Action on Orbits	1573
58.10.6	Action on a <i>G</i> -invariant Partition	1575
58.10.7	Action on a Coset Space	1580
58.10.8	Reduced Permutation Actions	1581
58.10.9	The Jellyfish Algorithm	1581
58.11	<i>Normal and Subnormal Subgroups</i>	1583
58.11.1	Characteristic Subgroups and Normal Series	1583
58.11.2	Maximal and Minimal Normal Subgroups	1586
58.11.3	Lattice of Normal Subgroups	1586
58.11.4	Composition and Chief Series	1587
58.11.5	The Socle	1590
58.11.6	The Soluble Radical and its Quotient	1593
58.11.7	Complements and Supplements	1595
58.11.8	Abelian Normal Subgroups	1597
58.12	<i>Cosets and Transversals</i>	1598
58.12.1	Cosets	1598

58.12.2	Transversals	1600
58.13	<i>Presentations</i>	1600
58.13.1	Generators and Relations	1601
58.13.2	Permutations as Words	1601
58.14	<i>Automorphism Groups</i>	1602
58.15	<i>Cohomology</i>	1604
58.16	<i>Representation Theory</i>	1606
58.17	<i>Identification</i>	1608
58.17.1	Identification as an Abstract Group	1608
58.17.2	Identification as a Permutation Group	1608
58.18	<i>Base and Strong Generating Set</i>	1613
58.18.1	Construction of a Base and Strong Generating Set	1613
58.18.2	Defining Values for Attributes	1616
58.18.3	Accessing the Base and Strong Generating Set	1617
58.18.4	Working with a Base and Strong Generating Set	1618
58.18.5	Modifying a Base and Strong Generating Set	1620
58.19	<i>Permutation Representations of Linear Groups</i>	1620
58.20	<i>Permutation Group Databases</i>	1626
58.21	<i>Ordered Partition Stacks</i>	1627
58.21.1	Construction of Ordered Partition Stacks	1627
58.21.2	Properties of Ordered Partition Stacks	1627
58.21.3	Operations on Ordered Partition Stacks	1628
58.22	<i>Bibliography</i>	1630
59	MATRIX GROUPS OVER GENERAL RINGS	1635
59.1	<i>Introduction</i>	1639
59.1.1	Introduction to Matrix Groups	1639
59.1.2	The Support	1640
59.1.3	The Category of Matrix Groups	1640
59.1.4	The Construction of a Matrix Group	1640
59.2	<i>Creation of a Matrix Group</i>	1640
59.2.1	Construction of the General Linear Group	1640
59.2.2	Construction of a Matrix Group Element	1641
59.2.3	Construction of a General Matrix Group	1643
59.2.4	Changing Rings	1644
59.2.5	Coercion between Matrix Structures	1645
59.2.6	Accessing Associated Structures	1645
59.3	<i>Homomorphisms</i>	1646
59.3.1	Construction of Extensions	1648
59.4	<i>Operations on Matrices</i>	1649
59.4.1	Arithmetic with Matrices	1650
59.4.2	Predicates for Matrices	1652
59.4.3	Matrix Invariants	1652
59.5	<i>Global Properties</i>	1655
59.5.1	Group Order	1656
59.5.2	Membership and Equality	1657
59.5.3	Set Operations	1658
59.6	<i>Abstract Group Predicates</i>	1660
59.7	<i>Conjugacy</i>	1662
59.8	<i>Subgroups</i>	1666
59.8.1	Construction of Subgroups	1666
59.8.2	Elementary Properties of Subgroups	1667
59.8.3	Standard Subgroups	1667
59.8.4	Low Index Subgroups	1669
59.8.5	Conjugacy Classes of Subgroups	1670

59.9	<i>Quotient Groups</i>	1672
59.9.1	Construction of Quotient Groups	1673
59.9.2	Abelian, Nilpotent and Soluble Quotients	1674
59.10	<i>Matrix Group Actions</i>	1675
59.10.1	Orbits and Stabilizers	1676
59.10.2	Orbit and Stabilizer Functions for Large Groups	1678
59.10.3	Action on Orbits	1684
59.10.4	Action on a Coset Space	1686
59.10.5	Action on the Natural G -Module	1687
59.11	<i>Normal and Subnormal Subgroups</i>	1688
59.11.1	Characteristic Subgroups and Subgroup Series	1688
59.11.2	The Soluble Radical and its Quotient	1690
59.11.3	Composition and Chief Factors	1691
59.12	<i>Coset Tables and Transversals</i>	1693
59.13	<i>Presentations</i>	1693
59.13.1	Presentations	1693
59.13.2	Matrices as Words	1694
59.14	<i>Automorphism Groups</i>	1694
59.15	<i>Representation Theory</i>	1697
59.16	<i>Base and Strong Generating Set</i>	1700
59.16.1	Introduction	1700
59.16.2	Controlling Selection of a Base	1700
59.16.3	Construction of a Base and Strong Generating Set	1701
59.16.4	Defining Values for Attributes	1703
59.16.5	Accessing the Base and Strong Generating Set	1703
59.17	<i>Soluble Matrix Groups</i>	1704
59.17.1	Conversion to a PC-Group	1704
59.17.2	Soluble Group Functions	1704
59.17.3	p -group Functions	1705
59.17.4	Abelian Group Functions	1705
59.18	<i>Bibliography</i>	1705
60	MATRIX GROUPS OVER FINITE FIELDS	1707
60.1	<i>Introduction</i>	1709
60.2	<i>Finding Elements with Prescribed Properties</i>	1709
60.3	<i>Monte Carlo Algorithms for Subgroups</i>	1710
60.4	<i>Aschbacher Reduction</i>	1713
60.4.1	Introduction	1713
60.4.2	Primitivity	1714
60.4.3	Semilinearity	1716
60.4.4	Tensor Products	1718
60.4.5	Tensor-induced Groups	1720
60.4.6	Normalisers of Extraspecial r -groups and Symplectic 2-groups	1722
60.4.7	Writing Representations over Subfields	1724
60.4.8	Decompositions with Respect to a Normal Subgroup	1727
60.5	<i>Constructive Recognition for Simple Groups</i>	1731
60.6	<i>Composition Trees for Matrix Groups</i>	1735
60.7	<i>The LMG functions</i>	1744
60.8	<i>Unipotent Matrix Groups</i>	1752
60.9	<i>Bibliography</i>	1754

61	MATRIX GROUPS OVER INFINITE FIELDS	1757
61.1	Overview	1759
61.2	Construction of Congruence Homomorphisms	1760
61.3	Testing Finiteness	1761
61.4	Deciding Virtual Properties of Linear Groups	1763
61.5	Other Properties of Linear Groups	1766
61.6	Other Functions for Nilpotent Matrix Groups	1768
61.7	Examples	1768
61.8	Bibliography	1775
62	MATRIX GROUPS OVER \mathbb{Q} AND \mathbb{Z}	1777
62.1	Overview	1779
62.2	Invariant Forms	1779
62.3	Endomorphisms	1780
62.4	New Groups From Others	1781
62.5	Perfect Forms and Normalizers	1781
62.6	Conjugacy	1782
62.7	Conjugacy Tests for Matrices	1783
62.8	Examples	1783
62.9	Bibliography	1785
63	FINITE SOLUBLE GROUPS	1787
63.1	Introduction	1791
63.1.1	Power-Conjugate Presentations	1791
63.2	Creation of a Group	1792
63.2.1	Construction Functions	1792
63.2.2	Definition by Presentation	1793
63.2.3	Possibly Inconsistent Presentations	1796
63.3	Basic Group Properties	1797
63.3.1	Infrastructure	1797
63.3.2	Numerical Invariants	1798
63.3.3	Predicates	1798
63.4	Homomorphisms	1799
63.5	New Groups from Existing	1802
63.6	Elements	1806
63.6.1	Definition of Elements	1806
63.6.2	Arithmetic Operations on Elements	1808
63.6.3	Properties of Elements	1809
63.6.4	Predicates for Elements	1809
63.6.5	Set Operations	1810
63.7	Conjugacy	1813
63.8	Subgroups	1815
63.8.1	Definition of Subgroups by Generators	1815
63.8.2	Membership and Coercion	1816
63.8.3	Inclusion and Equality	1818
63.8.4	Standard Subgroup Constructions	1819
63.8.5	Properties of Subgroups	1820
63.8.6	Predicates for Subgroups	1821
63.8.7	Hall π -Subgroups and Sylow Systems	1823
63.8.8	Conjugacy Classes of Subgroups	1824
63.9	Quotient Groups	1828
63.9.1	Construction of Quotient Groups	1828
63.9.2	Abelian and p -Quotients	1829

63.10	<i>Normal Subgroups and Subgroup Series</i>	1830
63.10.1	Characteristic Subgroups	1830
63.10.2	Subgroup Series	1831
63.10.3	Series for p -groups	1833
63.10.4	Normal Subgroups and Complements	1833
63.11	<i>Cosets</i>	1835
63.11.1	Coset Tables and Transversals	1835
63.11.2	Action on a Coset Space	1835
63.12	<i>Automorphism Group</i>	1836
63.12.1	General Soluble Group	1836
63.12.2	p -group	1840
63.12.3	Isomorphism and Standard Presentations	1842
63.13	<i>Generating p-groups</i>	1845
63.14	<i>Representation Theory</i>	1849
63.15	<i>Central Extensions</i>	1852
63.16	<i>Transfer Between Group Categories</i>	1855
63.16.1	Transfer to GrpPC	1855
63.16.2	Transfer from GrpPC	1856
63.17	<i>More About Presentations</i>	1858
63.17.1	Conditioned Presentations	1858
63.17.2	Special Presentations	1859
63.17.3	CompactPresentation	1862
63.18	<i>Optimizing Magma Code</i>	1863
63.18.1	PowerGroup	1863
63.19	<i>Bibliography</i>	1864
64	BLACK-BOX GROUPS	1867
64.1	<i>Introduction</i>	1869
64.2	<i>Construction of an SLP-Group and its Elements</i>	1869
64.2.1	Structure Constructors	1869
64.2.2	Construction of an Element	1869
64.3	<i>Arithmetic with Elements</i>	1869
64.3.1	Accessing the Defining Generators	1870
64.4	<i>Operations on Elements</i>	1870
64.4.1	Equality and Comparison	1870
64.4.2	Attributes of Elements	1870
64.5	<i>Set-Theoretic Operations</i>	1871
64.5.1	Membership and Equality	1871
64.5.2	Set Operations	1872
64.5.3	Coercions Between Related Groups	1872
65	ALMOST SIMPLE GROUPS	1873
65.1	<i>Introduction</i>	1877
65.1.1	Overview	1877
65.2	<i>Creating Finite Groups of Lie Type</i>	1878
65.2.1	Generic Creation Function	1878
65.2.2	The Orders of the Chevalley Groups	1879
65.2.3	Classical Groups	1880
65.2.4	Exceptional Groups	1887
65.3	<i>Group Recognition</i>	1889
65.3.1	Constructive Recognition of Alternating Groups	1890
65.3.2	Determining the Type of a Finite Group of Lie Type	1893
65.3.3	Classical Forms	1896
65.3.4	Recognizing Classical Groups in their Natural Representation	1900

65.3.5	Constructive Recognition of Linear Groups	1902
65.3.6	Constructive Recognition of Symplectic Groups	1906
65.3.7	Constructive Recognition of Unitary Groups	1906
65.3.8	Constructive Recognition of $SL(d, q)$ in Low Degree	1907
65.3.9	Constructive Recognition of Suzuki Groups	1908
65.3.10	Constructive Recognition of Small Ree Groups	1914
65.3.11	Constructive Recognition of Large Ree Groups	1917
65.4	<i>Properties of Finite Groups Of Lie Type</i>	1919
65.4.1	Maximal Subgroups of the Classical Groups	1919
65.4.2	Maximal Subgroups of the Exceptional Groups	1920
65.4.3	Sylow Subgroups of the Classical Groups	1921
65.4.4	Sylow Subgroups of Exceptional Groups	1922
65.4.5	Conjugacy of Subgroups of the Classical Groups	1925
65.4.6	Conjugacy of Elements of the Exceptional Groups	1926
65.4.7	Irreducible Subgroups of the General Linear Group	1926
65.5	<i>Atlas Data for the Sporadic Groups</i>	1927
65.6	<i>Bibliography</i>	1930
66	DATABASES OF GROUPS	1933
66.1	<i>Introduction</i>	1937
66.2	<i>Database of Small Groups</i>	1938
66.2.1	Basic Small Group Functions	1939
66.2.2	Processes	1943
66.2.3	Small Group Identification	1945
66.2.4	Accessing Internal Data	1946
66.3	<i>The p-groups of Order Dividing p^7</i>	1948
66.4	<i>Metacyclic p-groups</i>	1949
66.5	<i>Database of Perfect Groups</i>	1951
66.5.1	Specifying an Entry of the Database	1952
66.5.2	Creating the Database	1952
66.5.3	Accessing the Database	1952
66.5.4	Finding Legal Keys	1954
66.6	<i>Database of Almost-Simple Groups</i>	1956
66.6.1	The Record Fields	1956
66.6.2	Creating the Database	1957
66.6.3	Accessing the Database	1958
66.7	<i>Database of Transitive Groups</i>	1960
66.7.1	Accessing the Databases	1960
66.7.2	Processes	1963
66.7.3	Transitive Group Identification	1964
66.8	<i>Database of Primitive Groups</i>	1965
66.8.1	Accessing the Databases	1965
66.8.2	Processes	1967
66.8.3	Primitive Group Identification	1969
66.9	<i>Database of Rational Maximal Finite Matrix Groups</i>	1969
66.10	<i>Database of Integral Maximal Finite Matrix Groups</i>	1971
66.11	<i>Database of Finite Quaternionic Matrix Groups</i>	1973
66.12	<i>Database of Finite Symplectic Matrix Groups</i>	1974
66.13	<i>Database of Irreducible Matrix Groups</i>	1976
66.13.1	Accessing the Database	1976
66.14	<i>Database of Quasisimple Matrix Groups</i>	1977
66.15	<i>Database of Soluble Irreducible Groups</i>	1978
66.15.1	Basic Functions	1978
66.15.2	Searching with Predicates	1980
66.15.3	Associated Functions	1981

66.15.4	Processes	1981
66.16	<i>Database of ATLAS Groups</i>	1983
66.16.1	Accessing the Database	1984
66.16.2	Accessing the ATLAS Groups	1984
66.16.3	Representations of the ATLAS Groups	1985
66.17	<i>Fundamental Groups of 3-Manifolds</i>	1986
66.17.1	Basic Functions	1986
66.17.2	Accessing the Data	1987
66.18	<i>Bibliography</i>	1988
67	AUTOMORPHISM GROUPS	1991
67.1	<i>Introduction</i>	1993
67.2	<i>Creation of Automorphism Groups</i>	1994
67.3	<i>Access Functions</i>	1996
67.4	<i>Order Functions</i>	1997
67.5	<i>Representations of an Automorphism Group</i>	1999
67.6	<i>Automorphisms</i>	2001
67.7	<i>Stored Attributes of an Automorphism Group</i>	2004
67.8	<i>Holomorphs</i>	2007
67.9	<i>Bibliography</i>	2008
68	COHOMOLOGY AND EXTENSIONS	2009
68.1	<i>Introduction</i>	2011
68.2	<i>Creation of a Cohomology Module</i>	2012
68.3	<i>Accessing Properties of the Cohomology Module</i>	2013
68.4	<i>Calculating Cohomology</i>	2014
68.5	<i>Cocycles</i>	2016
68.6	<i>The Restriction to a Subgroup</i>	2019
68.7	<i>Other Operations on Cohomology Modules</i>	2020
68.8	<i>Constructing Extensions</i>	2021
68.9	<i>Constructing Distinct Extensions</i>	2024
68.10	<i>Finite Group Cohomology</i>	2028
68.10.1	Creation of Gamma-groups	2029
68.10.2	Accessing Information	2030
68.10.3	One Cocycles	2031
68.10.4	Group Cohomology	2032
68.11	<i>Bibliography</i>	2035

VOLUME 6: CONTENTS

X	FINITELY-PRESENTED GROUPS	2037
69	ABELIAN GROUPS	2039
69.1	<i>Introduction</i>	2041
69.2	<i>Construction of a Finitely Presented Abelian Group and its Elements</i>	2041
69.2.1	The Free Abelian Group	2041
69.2.2	Relations	2042
69.2.3	Specification of a Presentation	2043
69.2.4	Accessing the Defining Generators and Relations	2044
69.3	<i>Construction of a Generic Abelian Group</i>	2045
69.3.1	Specification of a Generic Abelian Group	2045
69.3.2	Accessing Generators	2048
69.3.3	Computing Abelian Group Structure	2048
69.4	<i>Elements</i>	2050
69.4.1	Construction of Elements	2050
69.4.2	Representation of an Element	2051
69.4.3	Arithmetic with Elements	2052
69.5	<i>Construction of Subgroups and Quotient Groups</i>	2053
69.5.1	Construction of Subgroups	2053
69.5.2	Construction of Quotient Groups	2055
69.6	<i>Standard Constructions and Conversions</i>	2055
69.7	<i>Operations on Elements</i>	2057
69.7.1	Order of an Element	2057
69.7.2	Discrete Logarithm	2058
69.7.3	Equality and Comparison	2059
69.8	<i>Invariants of an Abelian Group</i>	2060
69.9	<i>Canonical Decomposition</i>	2060
69.10	<i>Set-Theoretic Operations</i>	2061
69.10.1	Functions Relating to Group Order	2061
69.10.2	Membership and Equality	2061
69.10.3	Set Operations	2062
69.11	<i>Coset Spaces</i>	2063
69.11.1	Coercions Between Groups and Subgroups	2063
69.12	<i>Subgroup Constructions</i>	2064
69.13	<i>Subgroup Chains</i>	2065
69.14	<i>General Group Properties</i>	2065
69.14.1	Properties of Subgroups	2066
69.14.2	Enumeration of Subgroups	2066
69.15	<i>Representation Theory</i>	2068
69.16	<i>The Hom Functor</i>	2068
69.17	<i>Automorphism Groups</i>	2070
69.18	<i>Cohomology</i>	2070
69.19	<i>Homomorphisms</i>	2070
69.20	<i>Bibliography</i>	2073

70	FINITELY PRESENTED GROUPS	2075
70.1	<i>Introduction</i>	2079
70.1.1	Overview of Facilities	2079
70.1.2	The Construction of Finitely Presented Groups	2079
70.2	<i>Free Groups and Words</i>	2080
70.2.1	Construction of a Free Group	2080
70.2.2	Construction of Words	2081
70.2.3	Access Functions for Words	2081
70.2.4	Arithmetic Operators for Words	2083
70.2.5	Comparison of Words	2084
70.2.6	Relations	2085
70.3	<i>Construction of an FP-Group</i>	2087
70.3.1	The Quotient Group Constructor	2087
70.3.2	The FP-Group Constructor	2089
70.3.3	Construction from a Finite Permutation or Matrix Group	2090
70.3.4	Construction of the Standard Presentation for a Coxeter Group	2092
70.3.5	Conversion from a Special Form of FP-Group	2093
70.3.6	Construction of a Standard Group	2094
70.3.7	Construction of Extensions	2096
70.3.8	Accessing the Defining Generators and Relations	2098
70.4	<i>Homomorphisms</i>	2098
70.4.1	General Remarks	2098
70.4.2	Construction of Homomorphisms	2099
70.4.3	Accessing Homomorphisms	2099
70.4.4	Computing Homomorphisms to Finite Groups	2102
70.4.5	The L_2 -Quotient Algorithm	2110
70.4.6	Infinite L_2 quotients	2117
70.4.7	Searching for Isomorphisms	2121
70.5	<i>Abelian, Nilpotent and Soluble Quotient</i>	2123
70.5.1	Abelian Quotient	2123
70.5.2	p -Quotient	2126
70.5.3	The Construction of a p -Quotient	2127
70.5.4	Nilpotent Quotient	2129
70.5.5	Soluble Quotient	2135
70.6	<i>Subgroups</i>	2138
70.6.1	Specification of a Subgroup	2138
70.6.2	Index of a Subgroup: The Todd-Coxeter Algorithm	2140
70.6.3	Implicit Invocation of the Todd-Coxeter Algorithm	2145
70.6.4	Constructing a Presentation for a Subgroup	2146
70.7	<i>Subgroups of Finite Index</i>	2150
70.7.1	Low Index Subgroups	2150
70.7.2	Subgroup Constructions	2159
70.7.3	Properties of Subgroups	2164
70.8	<i>Coset Spaces and Tables</i>	2168
70.8.1	Coset Tables	2168
70.8.2	Coset Spaces: Construction	2170
70.8.3	Coset Spaces: Elementary Operations	2171
70.8.4	Accessing Information	2172
70.8.5	Double Coset Spaces: Construction	2176
70.8.6	Coset Spaces: Selection of Cosets	2177
70.8.7	Coset Spaces: Induced Homomorphism	2178
70.9	<i>Simplification</i>	2181
70.9.1	Reducing Generating Sets	2181
70.9.2	Tietze Transformations	2181
70.10	<i>Representation Theory</i>	2192
70.11	<i>Small Group Identification</i>	2196

70.11.1	Concrete Representations of Small Groups	2198
70.12	<i>Bibliography</i>	2198
71	FINITELY PRESENTED GROUPS: ADVANCED	2201
71.1	<i>Introduction</i>	2203
71.2	<i>Low Level Operations on Presentations and Words</i>	2203
71.2.1	Modifying Presentations	2204
71.2.2	Low Level Operations on Words	2206
71.3	<i>Interactive Coset Enumeration</i>	2208
71.3.1	Introduction	2208
71.3.2	Constructing and Modifying a Coset Enumeration Process	2209
71.3.3	Starting and Restarting an Enumeration	2214
71.3.4	Accessing Information	2216
71.3.5	Induced Permutation Representations	2225
71.3.6	Coset Spaces and Transversals	2226
71.4	<i>p-Quotients (Process Version)</i>	2229
71.4.1	The p -Quotient Process	2229
71.4.2	Using p -Quotient Interactively	2230
71.5	<i>Soluble Quotients</i>	2239
71.5.1	Introduction	2239
71.5.2	Construction	2239
71.5.3	Calculating the Relevant Primes	2241
71.5.4	The Functions	2241
71.6	<i>Bibliography</i>	2245
72	POLYCYCLIC GROUPS	2247
72.1	<i>Introduction</i>	2249
72.2	<i>Polycyclic Groups and Polycyclic Presentations</i>	2249
72.2.1	Introduction	2249
72.2.2	Specification of Elements	2250
72.2.3	Access Functions for Elements	2250
72.2.4	Arithmetic Operations on Elements	2251
72.2.5	Operators for Elements	2252
72.2.6	Comparison Operators for Elements	2252
72.2.7	Specification of a Polycyclic Presentation	2253
72.2.8	Properties of a Polycyclic Presentation	2257
72.3	<i>Subgroups, Quotient Groups, Homomorphisms and Extensions</i>	2257
72.3.1	Construction of Subgroups	2257
72.3.2	Coercions Between Groups and Subgroups	2258
72.3.3	Construction of Quotient Groups	2259
72.3.4	Homomorphisms	2259
72.3.5	Construction of Extensions	2260
72.3.6	Construction of Standard Groups	2260
72.4	<i>Conversion between Categories</i>	2263
72.5	<i>Access Functions for Groups</i>	2264
72.6	<i>Set-Theoretic Operations in a Group</i>	2265
72.6.1	Functions Relating to Group Order	2265
72.6.2	Membership and Equality	2265
72.6.3	Set Operations	2266
72.7	<i>Coset Spaces</i>	2267
72.8	<i>The Subgroup Structure</i>	2270
72.8.1	General Subgroup Constructions	2270
72.8.2	Subgroup Constructions Requiring a Nilpotent Covering Group	2270
72.9	<i>General Group Properties</i>	2271

72.9.1	General Properties of Subgroups	2272
72.9.2	Properties of Subgroups Requiring a Nilpotent Covering Group	2272
72.10	<i>Normal Structure and Characteristic Subgroups</i>	2274
72.10.1	Characteristic Subgroups and Subgroup Series	2274
72.10.2	The Abelian Quotient Structure of a Group	2278
72.11	<i>Conjugacy</i>	2278
72.12	<i>Representation Theory</i>	2279
72.13	<i>Power Groups</i>	2285
72.14	<i>Bibliography</i>	2286
73	BRAID GROUPS	2287
73.1	<i>Introduction</i>	2289
73.1.1	Lattice Structure and Simple Elements	2290
73.1.2	Representing Elements of a Braid Group	2291
73.1.3	Normal Form for Elements of a Braid Group	2292
73.1.4	Mixed Canonical Form and Lattice Operations	2293
73.1.5	Conjugacy Testing and Conjugacy Search	2294
73.2	<i>Constructing and Accessing Braid Groups</i>	2296
73.3	<i>Creating Elements of a Braid Group</i>	2297
73.4	<i>Working with Elements of a Braid Group</i>	2303
73.4.1	Accessing Information	2303
73.4.2	Computing Normal Forms of Elements	2306
73.4.3	Arithmetic Operators and Functions for Elements	2309
73.4.4	Boolean Predicates for Elements	2313
73.4.5	Lattice Operations	2317
73.4.6	Invariants of Conjugacy Classes	2321
73.5	<i>Homomorphisms</i>	2330
73.5.1	General Remarks	2330
73.5.2	Constructing Homomorphisms	2330
73.5.3	Accessing Homomorphisms	2331
73.5.4	Representations of Braid Groups	2334
73.6	<i>Bibliography</i>	2336
74	GROUPS DEFINED BY REWRITE SYSTEMS	2337
74.1	<i>Introduction</i>	2339
74.1.1	Terminology	2339
74.1.2	The Category of Rewrite Groups	2339
74.1.3	The Construction of a Rewrite Group	2339
74.2	<i>Constructing Confluent Presentations</i>	2340
74.2.1	The Knuth-Bendix Procedure	2340
74.2.2	Defining Orderings	2341
74.2.3	Setting Limits	2343
74.2.4	Accessing Group Information	2345
74.3	<i>Properties of a Rewrite Group</i>	2347
74.4	<i>Arithmetic with Words</i>	2348
74.4.1	Construction of a Word	2348
74.4.2	Element Operations	2349
74.5	<i>Operations on the Set of Group Elements</i>	2351
74.6	<i>Homomorphisms</i>	2353
74.6.1	General Remarks	2353
74.6.2	Construction of Homomorphisms	2353
74.7	<i>Conversion to a Finitely Presented Group</i>	2354
74.8	<i>Bibliography</i>	2354

75	AUTOMATIC GROUPS	2355
75.1	<i>Introduction</i>	2357
75.1.1	Terminology	2357
75.1.2	The Category of Automatic Groups	2357
75.1.3	The Construction of an Automatic Group	2357
75.2	<i>Creation of Automatic Groups</i>	2358
75.2.1	Construction of an Automatic Group	2358
75.2.2	Modifying Limits	2359
75.2.3	Accessing Group Information	2363
75.3	<i>Properties of an Automatic Group</i>	2364
75.4	<i>Arithmetic with Words</i>	2366
75.4.1	Construction of a Word	2366
75.4.2	Operations on Elements	2367
75.5	<i>Homomorphisms</i>	2369
75.5.1	General Remarks	2369
75.5.2	Construction of Homomorphisms	2370
75.6	<i>Set Operations</i>	2370
75.7	<i>The Growth Function</i>	2372
75.8	<i>Bibliography</i>	2373
76	GROUPS OF STRAIGHT-LINE PROGRAMS	2375
76.1	<i>Introduction</i>	2377
76.2	<i>Construction of an SLP-Group and its Elements</i>	2377
76.2.1	Structure Constructors	2377
76.2.2	Construction of an Element	2378
76.3	<i>Arithmetic with Elements</i>	2378
76.3.1	Accessing the Defining Generators and Relations	2378
76.4	<i>Addition of Extra Generators</i>	2379
76.5	<i>Creating Homomorphisms</i>	2379
76.6	<i>Operations on Elements</i>	2381
76.6.1	Equality and Comparison	2381
76.7	<i>Set-Theoretic Operations</i>	2381
76.7.1	Membership and Equality	2381
76.7.2	Set Operations	2382
76.7.3	Coercions Between Related Groups	2383
76.8	<i>Bibliography</i>	2383
77	FINITELY PRESENTED SEMIGROUPS	2385
77.1	<i>Introduction</i>	2387
77.2	<i>The Construction of Free Semigroups and their Elements</i>	2387
77.2.1	Structure Constructors	2387
77.2.2	Element Constructors	2388
77.3	<i>Elementary Operators for Words</i>	2388
77.3.1	Multiplication and Exponentiation	2388
77.3.2	The Length of a Word	2388
77.3.3	Equality and Comparison	2389
77.4	<i>Specification of a Presentation</i>	2390
77.4.1	Relations	2390
77.4.2	Presentations	2390
77.4.3	Accessing the Defining Generators and Relations	2391
77.5	<i>Subsemigroups, Ideals and Quotients</i>	2392
77.5.1	Subsemigroups and Ideals	2392
77.5.2	Quotients	2393

77.6	<i>Extensions</i>	2393
77.7	<i>Elementary Tietze Transformations</i>	2393
77.8	<i>String Operations on Words</i>	2395
78	MONOIDS GIVEN BY REWRITE SYSTEMS	2397
78.1	<i>Introduction</i>	2399
78.1.1	Terminology	2399
78.1.2	The Category of Rewrite Monoids	2399
78.1.3	The Construction of a Rewrite Monoid	2399
78.2	<i>Construction of a Rewrite Monoid</i>	2400
78.3	<i>Basic Operations</i>	2405
78.3.1	Accessing Monoid Information	2405
78.3.2	Properties of a Rewrite Monoid	2406
78.3.3	Construction of a Word	2408
78.3.4	Arithmetic with Words	2408
78.4	<i>Homomorphisms</i>	2410
78.4.1	General Remarks	2410
78.4.2	Construction of Homomorphisms	2410
78.5	<i>Set Operations</i>	2410
78.6	<i>Conversion to a Finitely Presented Monoid</i>	2412
78.7	<i>Bibliography</i>	2413

VOLUME 7: CONTENTS

XI	ALGEBRAS	2415
79	ALGEBRAS	2417
79.1	<i>Introduction</i>	2419
79.1.1	The Categories of Algebras	2419
79.2	<i>Construction of General Algebras and their Elements</i>	2419
79.2.1	Construction of a General Algebra	2420
79.2.2	Construction of an Element of a General Algebra	2421
79.3	<i>Construction of Subalgebras, Ideals and Quotient Algebras</i>	2421
79.3.1	Subalgebras and Ideals	2421
79.3.2	Quotient Algebras	2422
79.4	<i>Operations on Algebras and Subalgebras</i>	2422
79.4.1	Invariants of an Algebra	2422
79.4.2	Changing Rings	2423
79.4.3	Bases	2423
79.4.4	Decomposition of an Algebra	2424
79.4.5	Operations on Subalgebras	2426
79.5	<i>Operations on Elements of an Algebra</i>	2427
79.5.1	Operations on Elements	2427
79.5.2	Comparisons and Membership	2428
79.5.3	Predicates on Elements	2428
80	STRUCTURE CONSTANT ALGEBRAS	2429
80.1	<i>Introduction</i>	2431
80.2	<i>Construction of Structure Constant Algebras and Elements</i>	2431
80.2.1	Construction of a Structure Constant Algebra	2431
80.2.2	Construction of Elements of a Structure Constant Algebra	2432
80.3	<i>Operations on Structure Constant Algebras and Elements</i>	2433
80.3.1	Operations on Structure Constant Algebras	2433
80.3.2	Indexing Elements	2434
80.3.3	The Module Structure of a Structure Constant Algebra	2435
80.3.4	Homomorphisms	2435
81	ASSOCIATIVE ALGEBRAS	2439
81.1	<i>Introduction</i>	2441
81.2	<i>Construction of Associative Algebras</i>	2441
81.2.1	Construction of an Associative Structure Constant Algebra	2441
81.2.2	Associative Structure Constant Algebras from other Algebras	2442
81.3	<i>Operations on Algebras and their Elements</i>	2443
81.3.1	Operations on Algebras	2443
81.3.2	Operations on Elements	2445
81.3.3	Representations	2446
81.3.4	Decomposition of an Algebra	2446
81.4	<i>Orders</i>	2448
81.4.1	Creation of Orders	2449
81.4.2	Attributes	2452
81.4.3	Bases of Orders	2453

81.4.4	Predicates	2454
81.4.5	Operations with Orders	2455
81.5	<i>Elements of Orders</i>	2456
81.5.1	Creation of Elements	2456
81.5.2	Arithmetic of Elements	2456
81.5.3	Predicates on Elements	2457
81.5.4	Other Operations with Elements	2457
81.6	<i>Ideals of Orders</i>	2458
81.6.1	Creation of Ideals	2458
81.6.2	Attributes of Ideals	2459
81.6.3	Arithmetic for Ideals	2460
81.6.4	Predicates on Ideals	2460
81.6.5	Other Operations on Ideals	2461
81.7	<i>Quaternionic Orders</i>	2463
81.8	<i>Bibliography</i>	2464
82	FINITELY PRESENTED ALGEBRAS	2465
82.1	<i>Introduction</i>	2467
82.2	<i>Representation and Monomial Orders</i>	2467
82.3	<i>Exterior Algebras</i>	2468
82.4	<i>Creation of Free Algebras and Elements</i>	2468
82.4.1	Creation of Free Algebras	2468
82.4.2	Print Names	2468
82.4.3	Creation of Polynomials	2469
82.5	<i>Structure Operations</i>	2469
82.5.1	Related Structures	2469
82.5.2	Numerical Invariants	2469
82.5.3	Homomorphisms	2470
82.6	<i>Element Operations</i>	2471
82.6.1	Arithmetic Operators	2471
82.6.2	Equality and Membership	2471
82.6.3	Predicates on Algebra Elements	2471
82.6.4	Coefficients, Monomials, Terms and Degree	2472
82.6.5	Evaluation	2474
82.7	<i>Ideals and Gröbner Bases</i>	2475
82.7.1	Creation of Ideals	2475
82.7.2	Gröbner Bases	2476
82.7.3	Verbosity	2477
82.7.4	Related Functions	2478
82.8	<i>Basic Operations on Ideals</i>	2480
82.8.1	Construction of New Ideals	2481
82.8.2	Ideal Predicates	2481
82.8.3	Operations on Elements of Ideals	2482
82.9	<i>Changing Coefficient Ring</i>	2483
82.10	<i>Finitely Presented Algebras</i>	2483
82.11	<i>Creation of FP-Algebras</i>	2483
82.12	<i>Operations on FP-Algebras</i>	2485
82.13	<i>Finite Dimensional FP-Algebras</i>	2486
82.14	<i>Vector Enumeration</i>	2490
82.14.1	Finitely Presented Modules	2490
82.14.2	<i>S</i> -algebras	2490
82.14.3	Finitely Presented Algebras	2491
82.14.4	Vector Enumeration	2491
82.14.5	The Isomorphism	2492
82.14.6	Sketch of the Algorithm	2493

82.14.7	Weights	2493
82.14.8	Setup Functions	2494
82.14.9	The Quotient Module Function	2494
82.14.10	Structuring Presentations	2494
82.14.11	Options and Controls	2495
82.14.12	Weights	2495
82.14.13	Limits	2496
82.14.14	Logging	2497
82.14.15	Miscellaneous	2498
82.15	<i>Bibliography</i>	2501
83	MATRIX ALGEBRAS	2503
83.1	<i>Introduction</i>	2507
83.2	<i>Construction of Matrix Algebras and their Elements</i>	2507
83.2.1	Construction of the Complete Matrix Algebra	2507
83.2.2	Construction of a Matrix	2507
83.2.3	Constructing a General Matrix Algebra	2509
83.2.4	The Invariants of a Matrix Algebra	2510
83.3	<i>Construction of Subalgebras, Ideals and Quotient Rings</i>	2511
83.4	<i>The Construction of Extensions and their Elements</i>	2513
83.4.1	The Construction of Direct Sums and Tensor Products	2513
83.4.2	Construction of Direct Sums and Tensor Products of Elements	2515
83.5	<i>Operations on Matrix Algebras</i>	2516
83.6	<i>Changing Rings</i>	2516
83.7	<i>Elementary Operations on Elements</i>	2516
83.7.1	Arithmetic	2516
83.7.2	Predicates	2517
83.8	<i>Elements of M_n as Homomorphisms</i>	2521
83.9	<i>Elementary Operations on Subalgebras and Ideals</i>	2522
83.9.1	Bases	2522
83.9.2	Intersection of Subalgebras	2522
83.9.3	Membership and Equality	2522
83.10	<i>Accessing and Modifying a Matrix</i>	2523
83.10.1	Indexing	2523
83.10.2	Extracting and Inserting Blocks	2524
83.10.3	Joining Matrices	2524
83.10.4	Row and Column Operations	2525
83.11	<i>Canonical Forms</i>	2525
83.11.1	Canonical Forms for Matrices over Euclidean Domains	2525
83.11.2	Canonical Forms for Matrices over a Field	2527
83.12	<i>Diagonalising Commutative Algebras over a Field</i>	2530
83.13	<i>Solutions of Systems of Linear Equations</i>	2532
83.14	<i>Presentations for Matrix Algebras</i>	2533
83.14.1	Quotients and Idempotents	2533
83.14.2	Generators and Presentations	2536
83.14.3	Solving the Word Problem	2540
83.15	<i>Bibliography</i>	2542

84	GROUP ALGEBRAS	2543
84.1	<i>Introduction</i>	2545
84.2	<i>Construction of Group Algebras and their Elements</i>	2545
84.2.1	Construction of a Group Algebra	2545
84.2.2	Construction of a Group Algebra Element	2547
84.3	<i>Construction of Subalgebras, Ideals and Quotient Algebras</i>	2548
84.4	<i>Operations on Group Algebras and their Subalgebras</i>	2550
84.4.1	Operations on Group Algebras	2550
84.4.2	Operations on Subalgebras of Group Algebras	2551
84.5	<i>Operations on Elements</i>	2553
85	BASIC ALGEBRAS	2557
85.1	<i>Introduction</i>	2561
85.2	<i>Basic Algebras</i>	2561
85.2.1	Creation	2561
85.2.2	Special Basic Algebras	2562
85.2.3	Access Functions	2568
85.2.4	Elementary Operations	2569
85.2.5	Boolean Functions	2573
85.3	<i>Homomorphisms</i>	2573
85.4	<i>Subalgebras and Quotient Algebras</i>	2574
85.4.1	Subalgebras and their Constructions	2574
85.4.2	Ideals and their Construction	2575
85.4.3	Quotient Algebras	2576
85.5	<i>Minimal Forms and Gradings</i>	2577
85.6	<i>Automorphisms and Isomorphisms</i>	2579
85.7	<i>Modules over Basic Algebras</i>	2581
85.7.1	Indecomposable Projective Modules	2581
85.7.2	Creation	2582
85.7.3	Access Functions	2583
85.7.4	Predicates	2585
85.7.5	Elementary Operations	2586
85.8	<i>Homomorphisms of Modules</i>	2588
85.8.1	Creation	2588
85.8.2	Access Functions	2589
85.8.3	Projective Covers and Resolutions	2590
85.9	<i>Duals and Injectives</i>	2594
85.9.1	Injective Modules	2595
85.10	<i>Cohomology</i>	2598
85.10.1	Ext-Algebras	2603
85.11	<i>Group Algebras of p-groups</i>	2605
85.11.1	Access Functions	2606
85.11.2	Projective Resolutions	2606
85.11.3	Cohomology Generators	2607
85.11.4	Cohomology Rings	2608
85.11.5	Restrictions and Inflation	2608
85.12	<i>A-infinity Algebra Structures on Group Cohomology</i>	2612
85.12.1	Homological Algebra Toolkit	2614
85.13	<i>Bibliography</i>	2616

86	QUATERNION ALGEBRAS	2617
86.1	<i>Introduction</i>	2619
86.2	<i>Creation of Quaternion Algebras</i>	2620
86.3	<i>Creation of Quaternion Orders</i>	2624
86.3.1	Creation of Orders from Elements	2625
86.3.2	Creation of Maximal Orders	2626
86.3.3	Creation of Orders with given Discriminant	2628
86.3.4	Creation of Orders with given Discriminant over the Integers	2629
86.4	<i>Elements of Quaternion Algebras</i>	2630
86.4.1	Creation of Elements	2630
86.4.2	Arithmetic of Elements	2630
86.5	<i>Attributes of Quaternion Algebras</i>	2632
86.6	<i>Hilbert Symbols and Embeddings</i>	2634
86.7	<i>Predicates on Algebras</i>	2637
86.8	<i>Recognition Functions</i>	2638
86.9	<i>Attributes of Orders</i>	2640
86.10	<i>Predicates of Orders</i>	2641
86.11	<i>Operations with Orders</i>	2642
86.12	<i>Ideal Theory of Orders</i>	2643
86.12.1	Creation and Access Functions	2643
86.12.2	Enumeration of Ideal Classes	2646
86.12.3	Operations on Ideals	2649
86.13	<i>Norm Spaces and Basis Reduction</i>	2650
86.14	<i>Isomorphisms</i>	2652
86.14.1	Isomorphisms of Algebras	2652
86.14.2	Isomorphisms of Orders	2653
86.14.3	Isomorphisms of Ideals	2653
86.14.4	Examples	2655
86.15	<i>Units and Unit Groups</i>	2657
86.16	<i>Bibliography</i>	2659
87	ALGEBRAS WITH INVOLUTION	2661
87.1	<i>Introduction</i>	2663
87.2	<i>Algebras with Involution</i>	2663
87.2.1	Reflexive Forms	2664
87.2.2	Systems of Reflexive Forms	2664
87.2.3	Basic Attributes of *-Algebras	2665
87.2.4	Adjoint Algebras	2666
87.2.5	Group Algebras	2667
87.2.6	Simple *-Algebras	2668
87.3	<i>Decompositions of *-Algebras</i>	2669
87.4	<i>Recognition of *-Algebras</i>	2670
87.4.1	Recognition of Simple *-Algebras	2670
87.4.2	Recognition of Arbitrary *-Algebras	2671
87.5	<i>Intersections of Classical Groups</i>	2673
87.6	<i>Bibliography</i>	2675
88	CLIFFORD ALGEBRAS	2677
88.1	<i>Introduction</i>	2679
88.2	<i>Quadratic Spaces</i>	2679
88.3	<i>Bibliography</i>	2680

XII	REPRESENTATION THEORY	2681
89	MODULES OVER AN ALGEBRA	2683
89.1	<i>Introduction</i>	2685
89.2	<i>Modules over a Matrix Algebra</i>	2686
89.2.1	Construction of an A -Module	2686
89.2.2	Accessing Module Information	2687
89.2.3	Standard Constructions	2689
89.2.4	Element Construction and Operations	2690
89.2.5	Submodules	2692
89.2.6	Quotient Modules	2695
89.2.7	Structure of a Module	2696
89.2.8	Decomposability and Complements	2702
89.2.9	Lattice of Submodules	2704
89.2.10	Homomorphisms	2708
89.3	<i>Modules over a General Algebra</i>	2714
89.3.1	Introduction	2714
89.3.2	Construction of Algebra Modules	2714
89.3.3	The Action of an Algebra Element	2715
89.3.4	Related Structures of an Algebra Module	2715
89.3.5	Properties of an Algebra Module	2716
89.3.6	Creation of Algebra Modules from other Algebra Modules	2716
90	$K[G]$ -MODULES AND GROUP REPRESENTATIONS	2719
90.1	<i>Introduction</i>	2721
90.2	<i>Construction of $K[G]$-Modules</i>	2721
90.2.1	General $K[G]$ -Modules	2721
90.2.2	Natural $K[G]$ -Modules	2723
90.2.3	Action on an Elementary Abelian Section	2724
90.2.4	Permutation Modules	2725
90.2.5	Action on a Polynomial Ring	2727
90.3	<i>The Representation Afforded by a $K[G]$-module</i>	2728
90.4	<i>Standard Constructions</i>	2730
90.4.1	Changing the Coefficient Ring	2730
90.4.2	Writing a Module over a Smaller Field	2731
90.4.3	Direct Sum	2735
90.4.4	Tensor Products of $K[G]$ -Modules	2735
90.4.5	Induction and Restriction	2736
90.4.6	The Fixed-point Space of a Module	2737
90.4.7	Changing Basis	2737
90.5	<i>The Construction of all Irreducible Modules</i>	2738
90.5.1	Generic Functions for Finding Irreducible Modules	2738
90.5.2	The Burnside Algorithm	2741
90.5.3	The Schur Algorithm for Soluble Groups	2742
90.5.4	The Rational Algorithm	2745
90.6	<i>Extensions of Modules</i>	2748
90.7	<i>The Construction of Projective Indecomposable Modules</i>	2749

91	CHARACTERS OF FINITE GROUPS	2755
91.1	<i>Creation Functions</i>	2757
91.1.1	Structure Creation	2757
91.1.2	Element Creation	2757
91.1.3	The Table of Irreducible Characters	2758
91.2	<i>Character Ring Operations</i>	2762
91.2.1	Related Structures	2762
91.3	<i>Element Operations</i>	2763
91.3.1	Arithmetic	2763
91.3.2	Predicates and Booleans	2763
91.3.3	Accessing Class Functions	2764
91.3.4	Conjugation of Class Functions	2765
91.3.5	Functions Returning a Scalar	2765
91.3.6	The Schur Index	2766
91.3.7	Attribute	2769
91.3.8	Induction, Restriction and Lifting	2769
91.3.9	Symmetrization	2770
91.3.10	Permutation Character	2771
91.3.11	Composition and Decomposition	2771
91.3.12	Finding Irreducibles	2771
91.3.13	Brauer Characters	2774
91.4	<i>Bibliography</i>	2776
92	REPRESENTATIONS OF SYMMETRIC GROUPS	2777
92.1	<i>Introduction</i>	2779
92.2	<i>Representations of the Symmetric Group</i>	2779
92.2.1	Integral Representations	2779
92.2.2	The Seminormal and Orthogonal Representations	2780
92.3	<i>Characters of the Symmetric Group</i>	2781
92.3.1	Single Values	2781
92.3.2	Irreducible Characters	2781
92.3.3	Character Table	2781
92.4	<i>Representations of the Alternating Group</i>	2781
92.5	<i>Characters of the Alternating Group</i>	2782
92.5.1	Single Values	2782
92.5.2	Irreducible Characters	2782
92.5.3	Character Table	2782
92.6	<i>Bibliography</i>	2783
93	MOD P GALOIS REPRESENTATIONS	2785
93.1	<i>Introduction</i>	2787
93.1.1	Motivation	2787
93.1.2	Definitions	2787
93.1.3	Classification of φ -modules	2788
93.1.4	Connection with Galois Representations	2788
93.2	<i>φ-modules and Galois Representations in Magma</i>	2788
93.2.1	φ -modules	2789
93.2.2	Semisimple Galois Representations	2790
93.3	<i>Examples</i>	2791

VOLUME 8: CONTENTS

XIII	LIE THEORY	2793
94	INTRODUCTION TO LIE THEORY	2795
94.1	<i>Descriptions of Coxeter Groups</i>	2797
94.2	<i>Root Systems and Root Data</i>	2798
94.3	<i>Coxeter and Reflection Groups</i>	2798
94.4	<i>Lie Algebras and Groups of Lie Type</i>	2799
94.5	<i>Highest Weight Representations</i>	2799
94.6	<i>Universal Enveloping Algebras and Quantum Groups</i>	2799
94.7	<i>Bibliography</i>	2800
95	COXETER SYSTEMS	2801
95.1	<i>Introduction</i>	2803
95.2	<i>Coxeter Matrices</i>	2803
95.3	<i>Coxeter Graphs</i>	2805
95.4	<i>Cartan Matrices</i>	2807
95.5	<i>Dynkin Digraphs</i>	2810
95.6	<i>Finite and Affine Coxeter Groups</i>	2812
95.7	<i>Hyperbolic Groups</i>	2820
95.8	<i>Related Structures</i>	2821
95.9	<i>Bibliography</i>	2823
96	ROOT SYSTEMS	2825
96.1	<i>Introduction</i>	2827
96.1.1	<i>Reflections</i>	2827
96.1.2	<i>Definition of a Root System</i>	2827
96.1.3	<i>Simple and Positive Roots</i>	2828
96.1.4	<i>The Coxeter Group</i>	2828
96.1.5	<i>Nonreduced Root Systems</i>	2829
96.2	<i>Constructing Root Systems</i>	2829
96.3	<i>Operators on Root Systems</i>	2833
96.4	<i>Properties of Root Systems</i>	2835
96.5	<i>Roots and Coroots</i>	2836
96.5.1	<i>Accessing Roots and Coroots</i>	2836
96.5.2	<i>Reflections</i>	2839
96.5.3	<i>Operations and Properties for Roots and Coroot Indices</i>	2841
96.6	<i>Building Root Systems</i>	2844
96.7	<i>Related Structures</i>	2846
96.8	<i>Bibliography</i>	2846

97	ROOT DATA	2847
97.1	<i>Introduction</i>	2851
97.1.1	Reflections	2851
97.1.2	Definition of a Split Root Datum	2852
97.1.3	Simple and Positive Roots	2852
97.1.4	The Coxeter Group	2852
97.1.5	Nonreduced Root Data	2853
97.1.6	Isogeny of Split Reduced Root Data	2853
97.1.7	Extended Root Data	2854
97.2	<i>Constructing Root Data</i>	2854
97.2.1	Constructing Sparse Root Data	2860
97.3	<i>Operations on Root Data</i>	2862
97.4	<i>Properties of Root Data</i>	2869
97.5	<i>Roots, Coroots and Weights</i>	2872
97.5.1	Accessing Roots and Coroots	2872
97.5.2	Reflections	2879
97.5.3	Operations and Properties for Root and Coroot Indices	2881
97.5.4	Weights	2884
97.6	<i>Building Root Data</i>	2886
97.7	<i>Morphisms of Root Data</i>	2892
97.8	<i>Constants Associated with Root Data</i>	2894
97.9	<i>Related Structures</i>	2897
97.10	<i>Bibliography</i>	2898
98	COXETER GROUPS	2899
98.1	<i>Introduction</i>	2901
98.1.1	The Normal Form for Words	2902
98.2	<i>Constructing Coxeter Groups</i>	2902
98.3	<i>Converting Between Types of Coxeter Group</i>	2905
98.4	<i>Operations on Coxeter Groups</i>	2908
98.5	<i>Properties of Coxeter Groups</i>	2913
98.6	<i>Operations on Elements</i>	2914
98.7	<i>Roots, Coroots and Reflections</i>	2916
98.7.1	Accessing Roots and Coroots	2916
98.7.2	Operations and Properties for Root and Coroot Indices	2919
98.7.3	Weights	2922
98.8	<i>Reflections</i>	2923
98.9	<i>Reflection Subgroups</i>	2925
98.10	<i>Root Actions</i>	2928
98.11	<i>Standard Action</i>	2930
98.12	<i>Braid Groups</i>	2930
98.13	<i>W-graphs</i>	2931
98.14	<i>Related Structures</i>	2936
98.15	<i>Bibliography</i>	2937

99	REFLECTION GROUPS	2939
99.1	<i>Introduction</i>	2941
99.2	<i>Construction of Pseudo-reflections</i>	2941
99.2.1	Pseudo-reflections Preserving Reflexive Forms	2944
99.3	<i>Construction of Reflection Groups</i>	2946
99.4	<i>Construction of Real Reflection Groups</i>	2946
99.5	<i>Construction of Finite Complex Reflection Groups</i>	2949
99.6	<i>Operations on Reflection Groups</i>	2957
99.7	<i>Properties of Reflection Groups</i>	2961
99.8	<i>Roots, Coroots and Reflections</i>	2963
99.8.1	Accessing Roots and Coroots	2963
99.8.2	Reflections	2966
99.8.3	Weights	2967
99.9	<i>Related Structures</i>	2969
99.10	<i>Bibliography</i>	2969
100	LIE ALGEBRAS	2971
100.1	<i>Introduction</i>	2975
100.1.1	Guide for the Reader	2975
100.2	<i>Constructors for Lie Algebras</i>	2976
100.3	<i>Finitely Presented Lie Algebras</i>	2979
100.3.1	Construction of the Free Lie Algebra	2980
100.3.2	Properties of the Free Lie Algebra	2980
100.3.3	Operations on Elements of the Free Lie Algebra	2981
100.3.4	Construction of a Finitely-Presented Lie Algebra	2982
100.3.5	Homomorphisms of the Free Lie Algebra	2986
100.4	<i>Lie Algebras Generated by Extremal Elements</i>	2987
100.4.1	Constructing Lie Algebras Generated by Extremal Elements	2988
100.4.2	Properties of Lie Algebras Generated by Extremal Elements	2989
100.4.3	Instances of Lie Algebras Generated by Extremal Elements	2993
100.4.4	Studying the Parameter Space	2995
100.5	<i>Families of Lie Algebras</i>	2998
100.5.1	Almost Reductive Lie Algebras	2998
100.5.2	Cartan-Type Lie Algebras	3001
100.5.3	Melikian Lie Algebras	3006
100.6	<i>Construction of Elements</i>	3007
100.6.1	Construction of Elements of Structure Constant Algebras	3008
100.6.2	Construction of Matrix Elements	3008
100.7	<i>Construction of Subalgebras, Ideals and Quotients</i>	3009
100.8	<i>Operations on Lie Algebras</i>	3011
100.8.1	Basic Invariants	3014
100.8.2	Changing Base Rings	3015
100.8.3	Bases	3015
100.8.4	Operations for Semisimple and Reductive Lie Algebras	3016
100.9	<i>Operations on Subalgebras and Ideals</i>	3023
100.9.1	Standard Ideals and Subalgebras	3024
100.9.2	Cartan and Toral Subalgebras	3025
100.9.3	Standard Series	3027
100.9.4	The Lie Algebra of Derivations	3029
100.10	<i>Properties of Lie Algebras and Ideals</i>	3030
100.11	<i>Operations on Elements</i>	3032
100.11.1	Indexing	3033
100.12	<i>The Natural Module</i>	3034
100.13	<i>Operations for Matrix Lie Algebras</i>	3035

100.14	<i>Homomorphisms</i>	3035
100.15	<i>Automorphisms of Classical-type Reductive Algebras</i>	3036
100.16	<i>Restrictable Lie Algebras</i>	3037
100.17	<i>Universal Enveloping Algebras</i>	3039
100.17.1	Background	3039
100.17.2	Construction of Universal Enveloping Algebras	3040
100.17.3	Related Structures	3041
100.17.4	Elements of Universal Enveloping Algebras	3041
100.18	<i>Solvable and Nilpotent Lie Algebras Classification</i>	3044
100.18.1	The List of Solvable Lie Algebras	3044
100.18.2	Comments on the Classification over Finite Fields	3045
100.18.3	The List of Nilpotent Lie Algebras	3046
100.18.4	Intrinsics for Working with the Classifications	3047
100.19	<i>Semisimple Subalgebras of Simple Lie Algebras</i>	3051
100.20	<i>Nilpotent Orbits in Simple Lie Algebras</i>	3053
100.21	<i>Bibliography</i>	3057
101	KAC-MOODY LIE ALGEBRAS	3059
101.1	<i>Introduction</i>	3061
101.2	<i>Generalized Cartan Matrices</i>	3062
101.3	<i>Affine Kac-Moody Lie Algebras</i>	3063
101.3.1	Constructing Affine Kac-Moody Lie Algebras	3063
101.3.2	Properties of Affine Kac-Moody Lie Algebras	3064
101.3.3	Constructing Elements of Affine Kac-Moody Lie Algebras	3065
101.3.4	Properties of Elements of Affine Kac-Moody Lie Algebras	3066
101.4	<i>Bibliography</i>	3067
102	QUANTUM GROUPS	3069
102.1	<i>Introduction</i>	3071
102.2	<i>Background</i>	3071
102.2.1	Gaussian Binomials	3071
102.2.2	Quantized Enveloping Algebras	3072
102.2.3	Representations of $U_q(L)$	3073
102.2.4	PBW-type Bases	3073
102.2.5	The \mathbf{Z} -form of $U_q(L)$	3074
102.2.6	The Canonical Basis	3075
102.2.7	The Path Model	3076
102.3	<i>Gauss Numbers</i>	3077
102.4	<i>Construction</i>	3078
102.5	<i>Related Structures</i>	3079
102.6	<i>Operations on Elements</i>	3080
102.7	<i>Representations</i>	3082
102.8	<i>Hopf Algebra Structure</i>	3085
102.9	<i>Automorphisms</i>	3086
102.10	<i>Kashiwara Operators</i>	3088
102.11	<i>The Path Model</i>	3088
102.12	<i>Elements of the Canonical Basis</i>	3091
102.13	<i>Homomorphisms to the Universal Enveloping Algebra</i>	3093
102.14	<i>Bibliography</i>	3094

103	GROUPS OF LIE TYPE	3095
103.1	<i>Introduction</i>	3099
103.1.1	The Steinberg Presentation	3099
103.1.2	Bruhat Normalisation	3099
103.1.3	Twisted Groups of Lie type	3100
103.2	<i>Constructing Groups of Lie Type</i>	3100
103.2.1	Split Groups	3100
103.2.2	Galois Cohomology	3103
103.2.3	Twisted Groups	3107
103.3	<i>Operations on Groups of Lie Type</i>	3108
103.4	<i>Properties of Groups of Lie Type</i>	3112
103.5	<i>Constructing Elements</i>	3113
103.6	<i>Operations on Elements</i>	3115
103.6.1	Basic Operations	3115
103.6.2	Decompositions	3117
103.6.3	Conjugacy and Cohomology	3117
103.7	<i>Properties of Elements</i>	3118
103.8	<i>Roots, Coroots and Weights</i>	3118
103.8.1	Accessing Roots and Coroots	3119
103.8.2	Reflections	3121
103.8.3	Operations and Properties for Root and Coroot Indices	3122
103.8.4	Weights	3123
103.9	<i>Building Groups of Lie Type</i>	3123
103.10	<i>Automorphisms</i>	3125
103.10.1	Basic Functionality	3125
103.10.2	Constructing Special Automorphisms	3126
103.10.3	Operations and Properties of Automorphisms	3127
103.11	<i>Algebraic Homomorphisms</i>	3128
103.12	<i>Twisted Tori</i>	3128
103.13	<i>Sylow Subgroups</i>	3130
103.14	<i>Representations</i>	3131
103.15	<i>Bibliography</i>	3133
104	REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS . .	3135
104.1	<i>Introduction</i>	3137
104.1.1	Highest Weight Modules	3137
104.1.2	Toral Elements	3138
104.1.3	Other Highest Weight Representations	3138
104.2	<i>Constructing Weight Multisets</i>	3139
104.3	<i>Constructing Representations</i>	3140
104.3.1	Lie Algebras	3140
104.3.2	Groups of Lie Type	3144
104.4	<i>Operations on Weight Multisets</i>	3146
104.4.1	Basic Operations	3146
104.4.2	Conversion Functions	3149
104.4.3	Calculating with Representations	3150
104.5	<i>Operations on Representations</i>	3160
104.5.1	Lie Algebras	3160
104.5.2	Groups of Lie Type	3164
104.6	<i>Other Functions for Representation Decompositions</i>	3165
104.6.1	Operations Related to the Symmetric Group	3169
104.6.2	FusionRules	3170
104.7	<i>Subgroups of Small Rank</i>	3171
104.8	<i>Subalgebras of $su(d)$</i>	3172
104.9	<i>Bibliography</i>	3174

VOLUME 9: CONTENTS

XIV	COMMUTATIVE ALGEBRA	3175
105	GRÖBNER BASES	3177
105.1	<i>Introduction</i>	3179
105.2	<i>Representation and Monomial Orders</i>	3179
105.2.1	Lexicographical: <code>lex</code>	3180
105.2.2	Graded Lexicographical: <code>glex</code>	3180
105.2.3	Graded Reverse Lexicographical: <code>grevlex</code>	3180
105.2.4	Graded Reverse Lexicographical (Weighted): <code>grevlexw</code>	3181
105.2.5	Elimination (k): <code>elim</code>	3181
105.2.6	Elimination List: <code>elim</code>	3181
105.2.7	Inverse Block: <code>invblock</code>	3182
105.2.8	Univariate: <code>univ</code>	3182
105.2.9	Weight: <code>weight</code>	3182
105.3	<i>Polynomial Rings and Ideals</i>	3183
105.3.1	Creation of Polynomial Rings and Accessing their Monomial Orders	3183
105.3.2	Creation of Graded Polynomial Rings	3185
105.3.3	Element Operations Using the Grading	3186
105.3.4	Creation of Ideals and Accessing their Bases	3189
105.4	<i>Gröbner Bases</i>	3190
105.4.1	Gröbner Bases over Fields	3190
105.4.2	Gröbner Bases over Euclidean Rings	3190
105.4.3	Construction of Gröbner Bases	3192
105.4.4	Related Functions	3197
105.4.5	Gröbner Bases of Boolean Polynomial Rings	3199
105.4.6	Verbosity	3200
105.4.7	Degree- <i>d</i> Gröbner Bases	3212
105.5	<i>Changing Coefficient Ring</i>	3214
105.6	<i>Changing Monomial Order</i>	3214
105.7	<i>Hilbert-driven Gröbner Basis Construction</i>	3216
105.8	<i>SAT solver</i>	3218
105.9	<i>Bibliography</i>	3219
106	POLYNOMIAL RING IDEAL OPERATIONS	3221
106.1	<i>Introduction</i>	3223
106.2	<i>Creation of Polynomial Rings and their Ideals</i>	3224
106.3	<i>First Operations on Ideals</i>	3224
106.3.1	Simple Ideal Constructions	3224
106.3.2	Basic Commutative Algebra Operations	3224
106.3.3	Ideal Predicates	3227
106.3.4	Element Operations with Ideals	3229
106.4	<i>Computation of Varieties</i>	3231
106.5	<i>Multiplicities</i>	3233
106.6	<i>Elimination</i>	3234
106.6.1	Construction of Elimination Ideals	3234
106.6.2	Univariate Elimination Ideal Generators	3236
106.6.3	Relation Ideals	3239

106.7	<i>Variable Extension of Ideals</i>	3240
106.8	<i>Homogenization of Ideals</i>	3241
106.9	<i>Extension and Contraction of Ideals</i>	3241
106.10	<i>Dimension of Ideals</i>	3242
106.11	<i>Radical and Decomposition of Ideals</i>	3243
106.11.1	Radical	3243
106.11.2	Primary Decomposition	3244
106.11.3	Triangular Decomposition	3250
106.11.4	Equidimensional Decomposition	3252
106.12	<i>Normalisation and Noether Normalisation</i>	3253
106.12.1	Noether Normalisation	3253
106.12.2	Normalisation	3254
106.13	<i>Hilbert Series and Hilbert Polynomial</i>	3257
106.14	<i>Syzygies</i>	3260
106.15	<i>Maps between Rings</i>	3261
106.16	<i>Symmetric Polynomials</i>	3262
106.17	<i>Functions for Polynomial Algebra and Module Generators</i>	3263
106.18	<i>Bibliography</i>	3266
107	LOCAL POLYNOMIAL RINGS	3269
107.1	<i>Introduction</i>	3271
107.2	<i>Elements and Local Monomial Orders</i>	3271
107.2.1	Local Lexicographical: <code>llex</code>	3272
107.2.2	Local Graded Lexicographical: <code>lplex</code>	3272
107.2.3	Local Graded Reverse Lexicographical: <code>lprevlex</code>	3272
107.3	<i>Local Polynomial Rings and Ideals</i>	3273
107.3.1	Creation of Local Polynomial Rings and Accessing their Monomial Orders	3273
107.3.2	Creation of Ideals and Accessing their Bases	3274
107.4	<i>Standard Bases</i>	3275
107.4.1	Construction of Standard Bases	3276
107.5	<i>Operations on Ideals</i>	3278
107.5.1	Basic Operations	3278
107.5.2	Ideal Predicates	3279
107.5.3	Operations on Elements of Ideals	3281
107.6	<i>Changing Coefficient Ring</i>	3281
107.7	<i>Changing Monomial Order</i>	3282
107.8	<i>Dimension of Ideals</i>	3282
107.9	<i>Bibliography</i>	3282
108	AFFINE ALGEBRAS	3283
108.1	<i>Introduction</i>	3285
108.2	<i>Creation of Affine Algebras</i>	3285
108.3	<i>Operations on Affine Algebras</i>	3287
108.4	<i>Maps between Affine Algebras</i>	3290
108.5	<i>Finite Dimensional Affine Algebras</i>	3290
108.6	<i>Affine Algebras which are Fields</i>	3292
108.7	<i>Rings and Fields of Fractions of Affine Algebras</i>	3294

109	MODULES OVER MULTIVARIATE RINGS	3299
109.1	Introduction	3301
109.2	Module Basics: Embedded and Reduced Modules	3301
109.3	Monomial Orders	3303
109.3.1	Term Over Position: TOP	3304
109.3.2	Term Over Position (Weighted): TOPW	3304
109.3.3	Position Over Term: POT	3304
109.3.4	Position Over Term (Permutation): POTPERM	3305
109.3.5	Block TOP-TOP: TOPTOP	3305
109.3.6	Block TOP-POT: TOPPOT	3305
109.4	Basic Creation and Access	3305
109.4.1	Creation of Ambient Embedded Modules	3305
109.4.2	Creation of Reduced Modules	3306
109.4.3	Localization	3306
109.4.4	Basic Invariants	3307
109.4.5	Creation of Module Elements	3308
109.4.6	Element Operations	3309
109.5	The Homomorphism Type	3313
109.6	Submodules and Quotient Modules	3316
109.6.1	Creation	3316
109.6.2	Module Bases	3317
109.7	Basic Module Constructions	3320
109.8	Predicates	3321
109.9	Module Operations	3322
109.10	Changing Ring	3324
109.11	Hilbert Series	3324
109.12	Free Resolutions	3326
109.12.1	Constructing Free Resolutions	3326
109.12.2	Betti Numbers and Related Invariants	3330
109.13	The Hom Module and Ext	3340
109.14	Tensor Products and Tor	3343
109.15	Cohomology Of Coherent Sheaves	3345
109.16	Bibliography	3349
110	INVARIANT THEORY	3351
110.1	Introduction	3353
110.2	Invariant Rings of Finite Groups	3354
110.2.1	Creation	3354
110.2.2	Access	3354
110.3	Group Actions on Polynomials	3355
110.4	Permutation Group Actions on Polynomials	3355
110.5	Matrix Group Actions on Polynomials	3356
110.6	Algebraic Group Actions on Polynomials	3357
110.7	Verbosity	3357
110.8	Construction of Invariants of Specified Degree	3357
110.9	Construction of G-modules	3361
110.10	Molien Series	3362
110.11	Primary Invariants	3363
110.12	Secondary Invariants	3364
110.13	Fundamental Invariants	3366
110.14	The Module of an Invariant Ring	3371
110.15	The Algebra of an Invariant Ring and Algebraic Relations	3372
110.16	Properties of Invariant Rings	3376

110.17	<i>Steenrod Operations</i>	3377
110.18	<i>Minimalization and Homogeneous Module Testing</i>	3378
110.19	<i>Attributes of Invariant Rings and Fields</i>	3381
110.20	<i>Invariant Rings of Linear Algebraic Groups</i>	3383
110.20.1	Creation	3384
110.20.2	Access	3384
110.20.3	Functions	3384
110.21	<i>Invariant Fields</i>	3390
110.21.1	Creation	3390
110.21.2	Access	3391
110.21.3	Functions for Invariant Fields	3391
110.22	<i>Invariants of the Symmetric Group</i>	3394
110.23	<i>Bibliography</i>	3396
111	DIFFERENTIAL RINGS	3397
111.1	<i>Introduction</i>	3401
111.2	<i>Differential Rings and Fields</i>	3402
111.2.1	Creation	3402
111.2.2	Creation of Differential Ring Elements	3404
111.3	<i>Structure Operations on Differential Rings</i>	3405
111.3.1	Category and Parent	3405
111.3.2	Related Structures	3405
111.3.3	Derivation and Differential	3407
111.3.4	Numerical Invariants	3407
111.3.5	Predicates and Booleans	3408
111.3.6	Precision	3409
111.4	<i>Element Operations on Differential Ring Elements</i>	3411
111.4.1	Category and Parent	3411
111.4.2	Arithmetic	3411
111.4.3	Predicates and Booleans	3412
111.4.4	Coefficients and Terms	3413
111.4.5	Conjugates, Norm and Trace	3414
111.4.6	Derivatives and Differentials	3415
111.5	<i>Changing Related Structures</i>	3415
111.6	<i>Ring and Field Extensions</i>	3419
111.7	<i>Ideals and Quotient Rings</i>	3424
111.7.1	Defining Ideals and Quotient Rings	3424
111.7.2	Boolean Operations on Ideals	3425
111.8	<i>Wronskian Matrix</i>	3425
111.9	<i>Differential Operator Rings</i>	3426
111.9.1	Creation	3426
111.9.2	Creation of Differential Operators	3427
111.10	<i>Structure Operations on Differential Operator Rings</i>	3428
111.10.1	Category and Parent	3428
111.10.2	Related Structures	3428
111.10.3	Derivation and Differential	3428
111.10.4	Predicates and Booleans	3429
111.10.5	Precision	3430
111.11	<i>Element Operations on Differential Operators</i>	3431
111.11.1	Category and Parent	3431
111.11.2	Arithmetic	3431
111.11.3	Predicates and Booleans	3432
111.11.4	Coefficients and Terms	3432
111.11.5	Order and Degree	3433
111.11.6	Related Differential Operators	3434

111.11.7	Application of Operators	3435
111.12	<i>Related Maps</i>	3436
111.13	<i>Changing Related Structures</i>	3437
111.14	<i>Euclidean Algorithms, GCDs and LCMs</i>	3441
111.14.1	Euclidean Right and Left Division	3441
111.14.2	Greatest Common Right and Left Divisors	3442
111.14.3	Least Common Left Multiples	3443
111.15	<i>Related Matrices</i>	3444
111.16	<i>Singular Places and Indicial Polynomials</i>	3445
111.16.1	Singular Places	3445
111.16.2	Indicial Polynomials	3447
111.17	<i>Rational Solutions</i>	3448
111.18	<i>Newton Polygons</i>	3449
111.19	<i>Symmetric Powers</i>	3451
111.20	<i>Differential Operators of Algebraic Functions</i>	3452
111.21	<i>Factorisation of Operators over Differential Laurent Series Rings</i>	3452
111.21.1	Slope Valuation of an Operator	3453
111.21.2	Coprime Index 1 and LCLM Factorisation	3454
111.21.3	Right Hand Factors of Operators	3459
111.22	<i>Bibliography</i>	3464

XV	ALGEBRAIC GEOMETRY	3465
112	SCHEMES	3467
112.1	<i>Introduction and First Examples</i>	3473
112.1.1	Ambient Spaces	3474
112.1.2	Schemes	3475
112.1.3	Rational Points	3476
112.1.4	Projective Closure	3478
112.1.5	Maps	3479
112.1.6	Linear Systems	3481
112.1.7	Aside: Types of Schemes	3482
112.2	<i>Ambients</i>	3483
112.2.1	Affine and Projective Spaces	3483
112.2.2	Scrolls and Products	3485
112.2.3	Functions and Homogeneity on Ambient Spaces	3487
112.2.4	Prelude to Points	3488
112.3	<i>Constructing Schemes</i>	3490
112.4	<i>Different Types of Scheme</i>	3495
112.5	<i>Basic Attributes of Schemes</i>	3496
112.5.1	Functions of the Ambient Space	3496
112.5.2	Functions of the Equations	3497
112.6	<i>Function Fields and their Elements</i>	3499
112.7	<i>Rational Points and Point Sets</i>	3502
112.8	<i>Zero-dimensional Schemes</i>	3506
112.9	<i>Local Geometry of Schemes</i>	3508
112.9.1	Point Conditions	3509
112.9.2	Point Computations	3509
112.9.3	Analytically Hypersurface Singularities	3509
112.10	<i>Global Geometry of Schemes</i>	3512
112.11	<i>Base Change for Schemes</i>	3515
112.12	<i>Affine Patches and Projective Closure</i>	3517
112.13	<i>Arithmetic Properties of Schemes and Points</i>	3520
112.13.1	Height	3520
112.13.2	Restriction of Scalars	3520
112.13.3	Local Solubility	3521
112.13.4	Searching for Points	3524
112.14	<i>Maps between Schemes</i>	3525
112.14.1	Creation of Maps	3526
112.14.2	Basic Attributes	3536
112.14.3	Maps and Points	3538
112.14.4	Maps and Schemes	3540
112.14.5	Maps and Closure	3543
112.14.6	Automorphisms	3545
112.14.7	Scheme Graph Maps	3555
112.15	<i>Tangent and Secant Varieties and Isomorphic Projections</i>	3559
112.15.1	Tangent Varieties	3559
112.15.2	Secant Varieties	3560
112.15.3	Isomorphic Projection to Subspaces	3561
112.16	<i>Linear Systems</i>	3563
112.16.1	Creation of Linear Systems	3564
112.16.2	Basic Algebra of Linear Systems	3570
112.16.3	Linear Systems and Maps	3575
112.17	<i>Divisors</i>	3575
112.17.1	Divisor Groups	3576
112.17.2	Creation Of Divisors	3576

112.17.3	Ideals and Factorisations	3578
112.17.4	Basic Divisor Predicates	3579
112.17.5	Arithmetic of Divisors	3580
112.17.6	Further Divisor Properties	3580
112.17.7	Riemann-Roch Spaces	3582
112.18	<i>Isolated Points on Schemes</i>	3583
112.19	<i>Advanced Examples</i>	3591
112.19.1	A Pair of Twisted Cubics	3591
112.19.2	Curves in Space	3594
112.20	<i>Bibliography</i>	3595
113	COHERENT SHEAVES	3597
113.1	<i>Introduction</i>	3599
113.2	<i>Creation Functions</i>	3600
113.3	<i>Accessor Functions</i>	3603
113.4	<i>Basic Constructions</i>	3605
113.5	<i>Sheaf Homomorphisms</i>	3607
113.6	<i>Divisor Maps and Riemann-Roch Spaces</i>	3608
113.7	<i>Predicates</i>	3612
113.8	<i>Miscellaneous</i>	3615
113.9	<i>Examples</i>	3616
113.10	<i>Bibliography</i>	3627
114	ALGEBRAIC CURVES	3629
114.1	<i>First Examples</i>	3635
114.1.1	Ambients	3635
114.1.2	Curves	3636
114.1.3	Projective Closure	3637
114.1.4	Points	3638
114.1.5	Choosing Coordinates	3639
114.1.6	Function Fields and Divisors	3640
114.2	<i>Ambient Spaces</i>	3643
114.3	<i>Algebraic Curves</i>	3645
114.3.1	Creation	3645
114.3.2	Base Change	3647
114.3.3	Basic Attributes	3649
114.3.4	Basic Invariants	3651
114.3.5	Random Curves	3651
114.3.6	Ordinary Plane Curves	3653
114.4	<i>Local Geometry</i>	3657
114.4.1	Creation of Points on Curves	3657
114.4.2	Operations at a Point	3658
114.4.3	Singularity Analysis	3659
114.4.4	Resolution of Singularities	3660
114.4.5	Log Canonical Thresholds	3662
114.4.6	Local Intersection Theory	3665
114.5	<i>Global Geometry</i>	3667
114.5.1	Genus and Singularities	3667
114.5.2	Projective Closure and Affine Patches	3668
114.5.3	Special Forms of Curves	3670
114.6	<i>Maps and Curves</i>	3672
114.6.1	Elementary Maps	3672
114.6.2	Maps Induced by Morphisms	3674
114.7	<i>Automorphism Groups of Curves</i>	3675

114.7.1	Group Creation Functions	3676
114.7.2	Automorphisms	3677
114.7.3	Automorphism Group Operations	3678
114.7.4	Pullbacks and Pushforwards	3679
114.7.5	Quotients of Curves	3683
114.8	<i>Function Fields</i>	3687
114.8.1	Function Fields	3687
114.8.2	Representations of the Function Field	3692
114.8.3	Differentials	3693
114.9	<i>Divisors</i>	3697
114.9.1	Places	3698
114.9.2	Divisor Group	3702
114.9.3	Creation of Divisors	3703
114.9.4	Arithmetic of Divisors	3706
114.9.5	Other Operations on Divisors	3708
114.10	<i>Linear Equivalence of Divisors</i>	3709
114.10.1	Linear Equivalence and Class Group	3709
114.10.2	Riemann–Roch Spaces	3711
114.10.3	Index Calculus	3714
114.11	<i>Advanced Examples</i>	3717
114.11.1	Trigonal Curves	3717
114.11.2	Algebraic Geometric Codes	3719
114.12	<i>Curves over Global Fields</i>	3721
114.12.1	Finding Rational Points	3721
114.12.2	Regular Models of Arithmetic Surfaces	3722
114.12.3	Minimization and Reduction	3723
114.13	<i>Minimal Degree Functions and Plane Models</i>	3725
114.13.1	General Functions and Clifford Index One	3725
114.13.2	Small Genus Functions	3727
114.13.3	Small Genus Plane Models	3731
114.14	<i>Bibliography</i>	3734
115	RESOLUTION GRAPHS AND SPLICE DIAGRAMS	3735
115.1	<i>Introduction</i>	3737
115.2	<i>Resolution Graphs</i>	3737
115.2.1	Graphs, Vertices and Printing	3738
115.2.2	Creation from Curve Singularities	3740
115.2.3	Creation from Pencils	3742
115.2.4	Creation by Hand	3743
115.2.5	Modifying Resolution Graphs	3744
115.2.6	Numerical Data Associated to a Graph	3745
115.3	<i>Splice Diagrams</i>	3746
115.3.1	Creation of Splice Diagrams	3746
115.3.2	Numerical Functions of Splice Diagrams	3748
115.4	<i>Translation Between Graphs</i>	3749
115.4.1	Splice Diagrams from Resolution Graphs	3749
115.5	<i>Bibliography</i>	3750

116	ALGEBRAIC SURFACES	3751
116.1	<i>Introduction</i>	3753
116.2	<i>General Surfaces</i>	3754
116.2.1	Introduction	3754
116.2.2	Creation Functions	3754
116.2.3	Invariants	3755
116.2.4	Singularity Properties	3758
116.2.5	Kodaira-Enriques Classification	3760
116.2.6	Minimal Models	3761
116.2.7	Special Surfaces in Projective 4-space	3771
116.3	<i>Surfaces in \mathbf{P}^3</i>	3773
116.3.1	Introduction	3773
116.3.2	Embedded Formal Desingularization of Curves	3773
116.3.3	Formal Desingularization of Surfaces	3777
116.3.4	Adjoint Systems and Birational Invariants	3781
116.3.5	Classification and Parameterization of Rational Surfaces	3783
116.3.6	Reduction to Special Models	3784
116.3.7	Parameterization of Rational Surfaces	3788
116.3.8	Parameterization of Special Surfaces	3792
116.4	<i>Del Pezzo Surfaces</i>	3795
116.4.1	Introduction	3795
116.4.2	Creation of General Del Pezzos	3795
116.4.3	Parameterization of Del Pezzo Surfaces	3796
116.4.4	Minimization and Reduction of Surfaces	3805
116.4.5	Cubic Surfaces over Finite Fields	3807
116.4.6	Construction of Cubic Surfaces	3809
116.4.7	Invariant Theory of Cubic Surfaces	3809
116.4.8	The Pentahedron of a Cubic Surface	3813
116.5	<i>Bibliography</i>	3814
117	HILBERT SERIES OF POLARISED VARIETIES	3817
117.1	<i>Introduction</i>	3819
117.1.1	Key Warning and Disclaimer	3819
117.1.2	Overview of the Chapter	3821
117.2	<i>Hilbert Series and Graded Rings</i>	3822
117.2.1	Hilbert Series and Hilbert Polynomials	3822
117.2.2	Interpreting the Hilbert Numerator	3824
117.3	<i>Baskets of Singularities</i>	3827
117.3.1	Point Singularities	3828
117.3.2	Curve Singularities	3830
117.3.3	Baskets of Singularities	3832
117.3.4	Curves and Dissident Points	3834
117.4	<i>Generic Polarised Varieties</i>	3834
117.4.1	Accessing the Data	3835
117.4.2	Generic Creation, Checking, Changing	3836
117.5	<i>Subcanonical Curves</i>	3837
117.5.1	Creation of Subcanonical Curves	3837
117.5.2	Catalogue of Subcanonical Curves	3838
117.6	<i>K3 Surfaces</i>	3838
117.6.1	Creating and Comparing K3 Surfaces	3838
117.6.2	Accessing the Key Data	3839
117.6.3	Modifying K3 Surfaces	3839
117.7	<i>The K3 Database</i>	3840
117.7.1	Searching the K3 Database	3840
117.7.2	Working with the K3 Database	3843

117.8	<i>Fano 3-folds</i>	3844
117.8.1	Creation: $f = 1, 2$ or ≥ 3	3845
117.8.2	A Preliminary Fano Database	3846
117.9	<i>Calabi–Yau 3-folds</i>	3846
117.10	<i>Building Databases</i>	3847
117.10.1	The K3 Database	3847
117.10.2	Making New Databases	3848
117.11	<i>Bibliography</i>	3849
118	TORIC VARIETIES	3851
118.1	<i>Introduction and First Examples</i>	3855
118.1.1	The Projective Plane as a Toric Variety	3855
118.1.2	Resolution of a Nonprojective Toric Variety	3857
118.1.3	The Cox Ring of a Toric Variety	3858
118.2	<i>Fans in Toric Lattices</i>	3861
118.2.1	Construction of Fans	3861
118.2.2	Components of Fans	3864
118.2.3	Properties of Fans	3866
118.2.4	Maps of Fans	3867
118.3	<i>Geometrical Properties of Cones and Polyhedra</i>	3868
118.4	<i>Toric Varieties</i>	3870
118.4.1	Constructors for Toric Varieties	3871
118.4.2	Toric Varieties and Their Fans	3872
118.4.3	Properties of Toric Varieties	3873
118.4.4	Affine Patches on Toric Varieties	3874
118.5	<i>Cox Rings</i>	3874
118.5.1	The Cox Ring of a Toric Variety	3874
118.5.2	Cox Rings in Their Own Right	3876
118.5.3	Recovering a Toric Variety From a Cox Ring	3877
118.6	<i>Invariant Divisors and Riemann–Roch Spaces</i>	3879
118.6.1	Divisor Group	3880
118.6.2	Constructing Invariant Divisors	3880
118.6.3	Properties of Divisors	3882
118.6.4	Linear Equivalence of Divisors	3885
118.6.5	Riemann–Roch Spaces of Invariant Divisors	3885
118.7	<i>Maps of Toric Varieties</i>	3888
118.7.1	Maps from Lattice Maps	3888
118.7.2	Properties of Toric Maps	3889
118.8	<i>The Geometry of Toric Varieties</i>	3890
118.8.1	Resolution of Singularities and Linear Systems	3890
118.8.2	Mori Theory of Toric Varieties	3890
118.8.3	Decomposition of Toric Morphisms	3895
118.9	<i>Schemes in Toric Varieties</i>	3897
118.9.1	Construction of Subschemes	3898
118.10	<i>Bibliography</i>	3900

VOLUME 10: CONTENTS

XVI	ARITHMETIC GEOMETRY	3901
119	RATIONAL CURVES AND CONICS	3903
119.1	<i>Introduction</i>	3905
119.2	<i>Rational Curves and Conics</i>	3906
119.2.1	Rational Curve and Conic Creation	3906
119.2.2	Access Functions	3907
119.2.3	Rational Curve and Conic Examples	3908
119.3	<i>Conics</i>	3911
119.3.1	Elementary Invariants	3911
119.3.2	Alternative Defining Polynomials	3911
119.3.3	Alternative Models	3912
119.3.4	Other Functions on Conics	3912
119.4	<i>Local-Global Correspondence</i>	3913
119.4.1	Local Conditions for Conics	3913
119.4.2	Norm Residue Symbol	3913
119.5	<i>Rational Points on Conics</i>	3915
119.5.1	Finding Points	3915
119.5.2	Point Reduction	3917
119.6	<i>Isomorphisms</i>	3919
119.6.1	Isomorphisms with Standard Models	3919
119.7	<i>Automorphisms</i>	3923
119.7.1	Automorphisms of Rational Curves	3923
119.7.2	Automorphisms of Conics	3924
119.8	<i>Bibliography</i>	3926
120	ELLIPTIC CURVES	3927
120.1	<i>Introduction</i>	3931
120.2	<i>Creation Functions</i>	3932
120.2.1	Creation of an Elliptic Curve	3932
120.2.2	Creation Predicates	3935
120.2.3	Changing the Base Ring	3936
120.2.4	Alternative Models	3937
120.2.5	Predicates on Curve Models	3938
120.2.6	Twists of Elliptic Curves	3939
120.3	<i>Operations on Curves</i>	3942
120.3.1	Elementary Invariants	3942
120.3.2	Associated Structures	3945
120.3.3	Predicates on Elliptic Curves	3945
120.4	<i>Polynomials</i>	3946
120.5	<i>Subgroup Schemes</i>	3947
120.5.1	Creation of Subgroup Schemes	3947
120.5.2	Associated Structures	3948
120.5.3	Predicates on Subgroup Schemes	3948
120.5.4	Points of Subgroup Schemes	3948
120.6	<i>The Formal Group</i>	3949
120.7	<i>Operations on Point Sets</i>	3950
120.7.1	Creation of Point Sets	3950

120.7.2	Associated Structures	3951
120.7.3	Predicates on Point Sets	3951
120.8	<i>Morphisms</i>	3952
120.8.1	Creation Functions	3952
120.8.2	Predicates on Isogenies	3957
120.8.3	Structure Operations	3957
120.8.4	Endomorphisms	3958
120.8.5	Automorphisms	3959
120.9	<i>Operations on Points</i>	3959
120.9.1	Creation of Points	3959
120.9.2	Creation Predicates	3960
120.9.3	Access Operations	3961
120.9.4	Associated Structures	3961
120.9.5	Arithmetic	3961
120.9.6	Division Points	3962
120.9.7	Point Order	3965
120.9.8	Predicates on Points	3965
120.9.9	Weil Pairing	3967
120.10	<i>Bibliography</i>	3968
121	ELLIPTIC CURVES OVER FINITE FIELDS	3969
121.1	<i>Supersingular Curves</i>	3971
121.2	<i>The Order of the Group of Points</i>	3972
121.2.1	Point Counting	3972
121.2.2	Zeta Functions	3978
121.2.3	Cryptographic Elliptic Curve Domains	3979
121.3	<i>Enumeration of Points</i>	3980
121.4	<i>Abelian Group Structure</i>	3981
121.5	<i>Pairings on Elliptic Curves</i>	3982
121.5.1	Weil Pairing	3982
121.5.2	Tate Pairing	3982
121.5.3	Eta Pairing	3983
121.5.4	Ate Pairing	3984
121.6	<i>Weil Descent in Characteristic Two</i>	3988
121.7	<i>Discrete Logarithms</i>	3990
121.8	<i>Bibliography</i>	3991
122	ELLIPTIC CURVES OVER \mathbf{Q} AND NUMBER FIELDS	3993
122.1	<i>Introduction</i>	3997
122.2	<i>Curves over the Rationals</i>	3997
122.2.1	Local Invariants	3997
122.2.2	Kodaira Symbols	3999
122.2.3	Complex Multiplication	4000
122.2.4	Isogenous Curves	4000
122.2.5	Mordell–Weil Group	4001
122.2.6	Heights and Height Pairing	4007
122.2.7	Two-Descent and Two-Coverings	4013
122.2.8	The Cassels-Tate Pairing	4016
122.2.9	Four-Descent	4018
122.2.10	Eight-Descent	4022
122.2.11	Three-Descent	4023
122.2.12	Nine-Descent	4030
122.2.13	p -Isogeny Descent	4031
122.2.14	Heegner Points	4035

122.2.15	Analytic Information	4042
122.2.16	Integral and S -integral Points	4047
122.2.17	Elliptic Curve Database	4050
122.3	<i>Curves over Number Fields</i>	4054
122.3.1	Local Invariants	4054
122.3.2	Complex Multiplication	4055
122.3.3	Mordell–Weil Groups	4055
122.3.4	Heights	4056
122.3.5	Two Descent	4057
122.3.6	Selmer Groups	4057
122.3.7	The Cassels–Tate Pairing	4063
122.3.8	Elliptic Curve Chabauty	4063
122.3.9	Auxiliary Functions for Etale Algebras	4067
122.3.10	Analytic Information	4068
122.3.11	Elliptic Curves of Given Conductor	4069
122.4	<i>Curves over p-adic Fields</i>	4070
122.4.1	Local Invariants	4070
122.5	<i>Bibliography</i>	4071
123	ELLIPTIC CURVES OVER FUNCTION FIELDS	4075
123.1	<i>An Overview of Relevant Theory</i>	4077
123.2	<i>Local Computations</i>	4079
123.3	<i>Elliptic Curves of Given Conductor</i>	4080
123.4	<i>Heights</i>	4081
123.5	<i>The Torsion Subgroup</i>	4082
123.6	<i>The Mordell–Weil Group</i>	4082
123.7	<i>Two Descent</i>	4084
123.8	<i>The L-function and Counting Points</i>	4085
123.9	<i>Action of Frobenius</i>	4088
123.10	<i>Extended Examples</i>	4088
123.11	<i>Bibliography</i>	4091
124	MODELS OF GENUS ONE CURVES	4093
124.1	<i>Introduction</i>	4095
124.2	<i>Related Functionality</i>	4096
124.3	<i>Creation of Genus One Models</i>	4096
124.4	<i>Predicates on Genus One Models</i>	4099
124.5	<i>Access Functions</i>	4099
124.6	<i>Minimisation and Reduction</i>	4100
124.7	<i>Genus One Models as Coverings</i>	4102
124.8	<i>Families of Elliptic Curves with Prescribed n-Torsion</i>	4104
124.9	<i>Transformations between Genus One Models</i>	4104
124.10	<i>Invariants for Genus One Models</i>	4105
124.11	<i>Covariants and Contravariants for Genus One Models</i>	4106
124.12	<i>Examples</i>	4107
124.13	<i>Bibliography</i>	4109

125	HYPERELLIPTIC CURVES	4111
125.1	<i>Introduction</i>	4115
125.2	<i>Creation Functions</i>	4115
125.2.1	Creation of a Hyperelliptic Curve	4115
125.2.2	Creation Predicates	4116
125.2.3	Changing the Base Ring	4117
125.2.4	Models	4118
125.2.5	Predicates on Models	4120
125.2.6	Twisting Hyperelliptic Curves	4121
125.2.7	Type Change Predicates	4123
125.3	<i>Operations on Curves</i>	4123
125.3.1	Elementary Invariants	4124
125.3.2	Igusa Invariants	4124
125.3.3	Shioda Invariants	4128
125.3.4	Base Ring	4130
125.4	<i>Creation from Invariants</i>	4130
125.5	<i>Function Field</i>	4133
125.5.1	Function Field and Polynomial Ring	4133
125.6	<i>Points</i>	4133
125.6.1	Creation of Points	4133
125.6.2	Random Points	4135
125.6.3	Predicates on Points	4135
125.6.4	Access Operations	4135
125.6.5	Arithmetic of Points	4135
125.6.6	Enumeration and Counting Points	4136
125.6.7	Frobenius	4137
125.7	<i>Isomorphisms and Transformations</i>	4138
125.7.1	Creation of Isomorphisms	4138
125.7.2	Arithmetic with Isomorphisms	4139
125.7.3	Invariants of Isomorphisms	4140
125.7.4	Automorphism Group and Isomorphism Testing	4140
125.8	<i>Jacobians</i>	4145
125.8.1	Creation of a Jacobian	4145
125.8.2	Access Operations	4145
125.8.3	Base Ring	4145
125.8.4	Changing the Base Ring	4146
125.9	<i>Richelot Isogenies</i>	4146
125.10	<i>Points on the Jacobian</i>	4149
125.10.1	Creation of Points	4150
125.10.2	Random Points	4153
125.10.3	Booleans and Predicates for Points	4153
125.10.4	Access Operations	4154
125.10.5	Arithmetic of Points	4154
125.10.6	Order of Points on the Jacobian	4155
125.10.7	Frobenius	4155
125.10.8	Weil Pairing	4156
125.11	<i>Rational Points and Group Structure over Finite Fields</i>	4157
125.11.1	Enumeration of Points	4157
125.11.2	Counting Points on the Jacobian	4157
125.11.3	Deformation Point Counting	4162
125.11.4	Abelian Group Structure	4163
125.12	<i>Jacobians over Number Fields or \mathbf{Q}</i>	4164
125.12.1	Searching For Points	4164
125.12.2	Torsion	4164
125.12.3	Heights and Regulator	4166
125.12.4	The 2-Selmer Group	4171

125.13	<i>Two-Selmer Set of a Curve</i>	4179
125.14	<i>Chabauty's Method</i>	4182
125.15	<i>Cyclic Covers of \mathbf{P}^1</i>	4187
125.15.1	Points	4187
125.15.2	Descent	4188
125.15.3	Descent on the Jacobian	4189
125.15.4	Partial Descent	4192
125.16	<i>Kummer Surfaces</i>	4195
125.16.1	Creation of a Kummer Surface	4195
125.16.2	Structure Operations	4195
125.16.3	Base Ring	4195
125.16.4	Changing the Base Ring	4196
125.17	<i>Points on the Kummer Surface</i>	4196
125.17.1	Creation of Points	4196
125.17.2	Access Operations	4197
125.17.3	Predicates on Points	4197
125.17.4	Arithmetic of Points	4197
125.17.5	Rational Points on the Kummer Surface	4198
125.17.6	Pullback to the Jacobian	4198
125.18	<i>Analytic Jacobians of Hyperelliptic Curves</i>	4199
125.18.1	Creation and Access Functions	4200
125.18.2	Maps between Jacobians	4201
125.18.3	From Period Matrix to Curve	4208
125.18.4	Voronoi Cells	4210
125.19	<i>Bibliography</i>	4211
126	HYPERGEOMETRIC MOTIVES	4215
126.1	<i>Introduction</i>	4217
126.2	<i>Functionality</i>	4219
126.2.1	Creation Functions	4219
126.2.2	Access Functions	4220
126.2.3	Functionality with L -series and Euler Factors	4221
126.2.4	Associated Schemes and Curves	4224
126.2.5	Utility Functions	4224
126.3	<i>Examples</i>	4225
126.4	<i>Bibliography</i>	4233
127	L-FUNCTIONS	4235
127.1	<i>Overview</i>	4237
127.2	<i>Built-in L-series</i>	4238
127.3	<i>Computing L-values</i>	4249
127.4	<i>Arithmetic with L-series</i>	4251
127.5	<i>General L-series</i>	4252
127.5.1	Terminology	4253
127.5.2	Constructing a General L -Series	4254
127.5.3	Setting the Coefficients	4258
127.5.4	Specifying the Coefficients Later	4258
127.5.5	Generating the Coefficients from Local Factors	4260
127.6	<i>Accessing the Invariants</i>	4260
127.7	<i>Precision</i>	4263
127.7.1	L -series with Unusual Coefficient Growth	4264
127.7.2	Computing $L(s)$ when $\text{Im}(s)$ is Large (<code>ImS</code> Parameter)	4264
127.7.3	Implementation of L -series Computations (<code>Asymptotics</code> Parameter)	4264
127.8	<i>Verbose Printing</i>	4265

127.9	<i>Advanced Examples</i>	4265
127.9.1	Handmade L -series of an Elliptic Curve	4265
127.9.2	Self-made Dedekind Zeta Function	4266
127.9.3	L -series of a Genus 2 Hyperelliptic Curve	4266
127.9.4	Experimental Mathematics for Small Conductor	4268
127.9.5	Tensor Product of L -series Coming from l -adic Representations	4269
127.9.6	Non-abelian Twist of an Elliptic Curve	4270
127.9.7	Other Tensor Products	4271
127.9.8	Symmetric Powers	4273
127.10	<i>Weil Polynomials</i>	4276
127.11	<i>Bibliography</i>	4279

VOLUME 11: CONTENTS

XVII	MODULAR ARITHMETIC GEOMETRY	4281
128	MODULAR CURVES	4283
128.1	Introduction	4285
128.2	Creation Functions	4285
128.2.1	Creation of a Modular Curve	4285
128.2.2	Creation of Points	4285
128.3	Invariants	4286
128.4	Modular Polynomial Databases	4287
128.5	Parametrized Structures	4289
128.6	Associated Structures	4292
128.7	Automorphisms	4293
128.8	Class Polynomials	4293
128.9	Modular Curves and Quotients (Canonical Embeddings)	4294
128.10	Modular Curves of Given Level and Genus	4296
128.11	Bibliography	4301
129	SMALL MODULAR CURVES	4303
129.1	Introduction	4305
129.2	Small Modular Curve Models	4305
129.3	Projection Maps	4307
129.4	Automorphisms	4309
129.5	Cusps and Rational Points	4313
129.6	Standard Functions and Forms	4315
129.7	Parametrized Structures	4317
129.8	Modular Generators and q -Expansions	4319
129.9	Extended Example	4324
129.10	Bibliography	4326
130	CONGRUENCE SUBGROUPS OF $\mathrm{PSL}_2(\mathbf{R})$	4327
130.1	Introduction	4329
130.2	Congruence Subgroups	4330
130.2.1	Creation of Subgroups of $\mathrm{PSL}_2(\mathbf{R})$	4331
130.2.2	Relations	4332
130.2.3	Basic Attributes	4332
130.3	Structure of Congruence Subgroups	4333
130.3.1	Cusps and Elliptic Points of Congruence Subgroups	4334
130.4	Elements of $\mathrm{PSL}_2(\mathbf{R})$	4336
130.4.1	Creation	4336
130.4.2	Membership and Equality Testing	4336
130.4.3	Basic Functions	4336
130.5	The Upper Half Plane	4337
130.5.1	Creation	4337
130.5.2	Basic Attributes	4338
130.6	Action of $\mathrm{PSL}_2(\mathbf{R})$ on the Upper Half Plane	4339

130.6.1	Arithmetic	4340
130.6.2	Distances, Angles and Geodesics	4340
130.7	<i>Farey Symbols and Fundamental Domains</i>	4341
130.8	<i>Points and Geodesics</i>	4343
130.9	<i>Graphical Output</i>	4343
130.10	<i>Bibliography</i>	4351
131	ARITHMETIC FUCHSIAN GROUPS AND SHIMURA CURVES	4353
131.1	<i>Arithmetic Fuchsian Groups</i>	4355
131.1.1	Creation	4355
131.1.2	Quaternionic Functions	4357
131.1.3	Basic Invariants	4360
131.1.4	Group Structure	4361
131.2	<i>Unit Disc</i>	4363
131.2.1	Creation	4363
131.2.2	Basic Operations	4364
131.2.3	Access Operations	4364
131.2.4	Distance and Angles	4366
131.2.5	Structural Operations	4367
131.3	<i>Fundamental Domains</i>	4369
131.4	<i>Triangle Groups</i>	4371
131.4.1	Creation of Triangle Groups	4372
131.4.2	Fundamental Domain	4372
131.4.3	CM Points	4372
131.5	<i>Bibliography</i>	4375
132	MODULAR FORMS	4377
132.1	<i>Introduction</i>	4379
132.1.1	Modular Forms	4379
132.1.2	About the Package	4380
132.1.3	Categories	4381
132.1.4	Verbose Output	4381
132.1.5	An Illustrative Overview	4382
132.2	<i>Creation Functions</i>	4385
132.2.1	Ambient Spaces	4385
132.2.2	Base Extension	4388
132.2.3	Elements	4389
132.3	<i>Bases</i>	4390
132.4	<i>q-Expansions</i>	4392
132.5	<i>Arithmetic</i>	4394
132.6	<i>Predicates</i>	4395
132.7	<i>Properties</i>	4397
132.8	<i>Subspaces</i>	4399
132.9	<i>Operators</i>	4401
132.10	<i>Eisenstein Series</i>	4403
132.11	<i>Weight Half Forms</i>	4405
132.12	<i>Weight One Forms</i>	4405
132.13	<i>Newforms</i>	4405
132.13.1	Labels	4408
132.14	<i>Reductions and Embeddings</i>	4410
132.15	<i>Congruences</i>	4411
132.16	<i>Overconvergent Modular Forms</i>	4413
132.17	<i>Algebraic Relations</i>	4414

132.18	<i>Elliptic Curves</i>	4416
132.19	<i>Modular Symbols</i>	4417
132.20	<i>Bibliography</i>	4418
133	MODULAR SYMBOLS	4419
133.1	<i>Introduction</i>	4421
133.1.1	Modular Symbols	4421
133.2	<i>Basics</i>	4422
133.2.1	Verbose Output	4422
133.2.2	Categories	4422
133.3	<i>Creation Functions</i>	4423
133.3.1	Ambient Spaces	4423
133.3.2	Labels	4427
133.3.3	Creation of Elements	4428
133.4	<i>Bases</i>	4431
133.5	<i>Associated Vector Space</i>	4434
133.6	<i>Degeneracy Maps</i>	4435
133.7	<i>Decomposition</i>	4437
133.8	<i>Subspaces</i>	4441
133.9	<i>Twists</i>	4443
133.10	<i>Operators</i>	4444
133.11	<i>The Hecke Algebra</i>	4449
133.12	<i>The Intersection Pairing</i>	4450
133.13	<i>q-Expansions</i>	4451
133.14	<i>Special Values of L-functions</i>	4454
133.14.1	Winding Elements	4456
133.15	<i>The Associated Complex Torus</i>	4457
133.15.1	The Period Map	4462
133.15.2	Projection Mappings	4462
133.16	<i>Modular Abelian Varieties</i>	4464
133.16.1	Modular Degree and Torsion	4464
133.16.2	Tamagawa Numbers and Orders of Component Groups	4466
133.17	<i>Elliptic Curves</i>	4469
133.18	<i>Dimension Formulas</i>	4471
133.19	<i>Bibliography</i>	4472
134	BRANDT MODULES	4475
134.1	<i>Introduction</i>	4477
134.2	<i>Brandt Module Creation</i>	4477
134.2.1	Creation of Elements	4479
134.2.2	Operations on Elements	4479
134.2.3	Categories and Parent	4480
134.2.4	Elementary Invariants	4480
134.2.5	Associated Structures	4481
134.2.6	Verbose Output	4482
134.3	<i>Subspaces and Decomposition</i>	4483
134.3.1	Boolean Tests on Subspaces	4484
134.4	<i>Hecke Operators</i>	4485
134.5	<i>q-Expansions</i>	4486
134.6	<i>Dimensions of Spaces</i>	4486
134.7	<i>Brandt Modules Over $F_q[t]$</i>	4487
134.8	<i>Bibliography</i>	4487

135	SUPERSINGULAR DIVISORS ON MODULAR CURVES . . .	4489
135.1	<i>Introduction</i>	4491
135.1.1	Categories	4492
135.1.2	Verbose Output	4492
135.2	<i>Creation Functions</i>	4492
135.2.1	Ambient Spaces	4492
135.2.2	Elements	4493
135.2.3	Subspaces	4494
135.3	<i>Basis</i>	4495
135.4	<i>Properties</i>	4496
135.5	<i>Associated Spaces</i>	4497
135.6	<i>Predicates</i>	4498
135.7	<i>Arithmetic</i>	4499
135.8	<i>Operators</i>	4501
135.9	<i>The Monodromy Pairing</i>	4502
135.10	<i>Bibliography</i>	4503
136	MODULAR ABELIAN VARIETIES	4505
136.1	<i>Introduction</i>	4511
136.1.1	Categories	4512
136.1.2	Verbose Output	4512
136.2	<i>Creation and Basic Functions</i>	4513
136.2.1	Creating the Modular Jacobian $J_0(N)$	4513
136.2.2	Creating the Modular Jacobians $J_1(N)$ and $J_H(N)$	4514
136.2.3	Abelian Varieties Attached to Modular Forms	4516
136.2.4	Abelian Varieties Attached to Modular Symbols	4518
136.2.5	Creation of Abelian Subvarieties	4519
136.2.6	Creation Using a Label	4520
136.2.7	Invariants	4521
136.2.8	Conductor	4524
136.2.9	Number of Points	4524
136.2.10	Inner Twists and Complex Multiplication	4525
136.2.11	Predicates	4528
136.2.12	Equality and Inclusion Testing	4533
136.2.13	Modular Embedding and Parameterization	4534
136.2.14	Coercion	4535
136.2.15	Modular Symbols to Homology	4538
136.2.16	Embeddings	4539
136.2.17	Base Change	4541
136.2.18	Additional Examples	4542
136.3	<i>Homology</i>	4545
136.3.1	Creation	4545
136.3.2	Invariants	4546
136.3.3	Functors to Categories of Lattices and Vector Spaces	4546
136.3.4	Modular Structure	4548
136.3.5	Additional Examples	4549
136.4	<i>Homomorphisms</i>	4550
136.4.1	Creation	4551
136.4.2	Restriction, Evaluation, and Other Manipulations	4552
136.4.3	Kernels	4556
136.4.4	Images	4557
136.4.5	Cokernels	4559
136.4.6	Matrix Structure	4560
136.4.7	Arithmetic	4562
136.4.8	Polynomials	4565

136.4.9	Invariants	4566
136.4.10	Predicates	4567
136.5	<i>Endomorphism Algebras and Hom Spaces</i>	4570
136.5.1	Creation	4570
136.5.2	Subgroups and Subrings	4571
136.5.3	Pullback and Pushforward of Hom Spaces	4574
136.5.4	Arithmetic	4574
136.5.5	Quotients	4575
136.5.6	Invariants	4576
136.5.7	Structural Invariants	4578
136.5.8	Matrix and Module Structure	4579
136.5.9	Predicates	4581
136.5.10	Elements	4583
136.6	<i>Arithmetic of Abelian Varieties</i>	4584
136.6.1	Direct Sum	4584
136.6.2	Sum in an Ambient Variety	4586
136.6.3	Intersections	4587
136.6.4	Quotients	4589
136.7	<i>Decomposing and Factoring Abelian Varieties</i>	4590
136.7.1	Decomposition	4590
136.7.2	Factorization	4591
136.7.3	Decomposition with respect to an Endomorphism or a Commutative Ring	4592
136.7.4	Additional Examples	4592
136.8	<i>Building blocks</i>	4594
136.8.1	Background and Notation	4594
136.9	<i>Orthogonal Complements</i>	4598
136.9.1	Complements	4598
136.9.2	Dual Abelian Variety	4599
136.9.3	Intersection Pairing	4601
136.9.4	Projections	4602
136.9.5	Left and Right Inverses	4603
136.9.6	Congruence Computations	4605
136.10	<i>New and Old Subvarieties and Natural Maps</i>	4606
136.10.1	Natural Maps	4606
136.10.2	New Subvarieties and Quotients	4608
136.10.3	Old Subvarieties and Quotients	4609
136.11	<i>Elements of Modular Abelian Varieties</i>	4610
136.11.1	Arithmetic	4611
136.11.2	Invariants	4612
136.11.3	Predicates	4613
136.11.4	Homomorphisms	4615
136.11.5	Representation of Torsion Points	4616
136.12	<i>Subgroups of Modular Abelian Varieties</i>	4617
136.12.1	Creation	4617
136.12.2	Elements	4619
136.12.3	Arithmetic	4620
136.12.4	Underlying Abelian Group and Lattice	4622
136.12.5	Invariants	4623
136.12.6	Predicates and Comparisons	4624
136.13	<i>Rational Torsion Subgroups</i>	4626
136.13.1	Cuspidal Subgroup	4626
136.13.2	Upper and Lower Bounds	4628
136.13.3	Torsion Subgroup	4629
136.14	<i>Hecke and Atkin-Lehner Operators</i>	4629
136.14.1	Creation	4629
136.14.2	Invariants	4631
136.15	<i>L-series</i>	4632

136.15.1	Creation	4632
136.15.2	Invariants	4633
136.15.3	Characteristic Polynomials of Frobenius Elements	4634
136.15.4	Values at Integers in the Critical Strip	4635
136.15.5	Leading Coefficient	4637
136.16	<i>Complex Period Lattice</i>	4638
136.16.1	Period Map	4638
136.16.2	Period Lattice	4638
136.17	<i>Tamagawa Numbers and Component Groups of Neron Models</i>	4638
136.17.1	Component Groups	4638
136.17.2	Tamagawa Numbers	4639
136.18	<i>Elliptic Curves</i>	4640
136.18.1	Creation	4640
136.18.2	Invariants	4641
136.19	<i>Bibliography</i>	4642
137	HILBERT MODULAR FORMS	4643
137.1	<i>Introduction</i>	4645
137.1.1	Definitions and Background	4645
137.1.2	Algorithms and the Jacquet-Langlands Correspondence	4646
137.1.3	Algorithm I (Using Definite Quaternion Orders)	4647
137.1.4	Algorithm II (Using Indefinite Quaternion Orders)	4647
137.1.5	Categories	4647
137.1.6	Verbose Output	4647
137.2	<i>Creation of Full Cuspidal Spaces</i>	4647
137.3	<i>Basic Properties</i>	4649
137.4	<i>Elements</i>	4651
137.5	<i>Operators</i>	4651
137.6	<i>Creation of Subspaces</i>	4653
137.7	<i>Eigenspace Decomposition and Eigenforms</i>	4656
137.8	<i>Further Examples</i>	4658
137.9	<i>Bibliography</i>	4660
138	MODULAR FORMS OVER IMAGINARY QUADRATIC FIELDS	4661
138.1	<i>Introduction</i>	4663
138.1.1	Algorithms	4663
138.1.2	Categories	4664
138.1.3	Verbose Output	4664
138.2	<i>Creation</i>	4665
138.3	<i>Attributes</i>	4665
138.4	<i>Hecke Operators</i>	4666
138.5	<i>Newforms</i>	4667
138.6	<i>Bibliography</i>	4668
139	ADMISSIBLE REPRESENTATIONS OF $GL_2(\mathbf{Q}_p)$	4669
139.1	<i>Introduction</i>	4671
139.1.1	Motivation	4671
139.1.2	Definitions	4671
139.1.3	The Principal Series	4672
139.1.4	Supercuspidal Representations	4672
139.1.5	The Local Langlands Correspondence	4673
139.1.6	Connection with Modular Forms	4673
139.1.7	Category	4673

139.1.8	Verbose Output	4673
139.2	<i>Creation of Admissible Representations</i>	4674
139.3	<i>Attributes of Admissible Representations</i>	4674
139.4	<i>Structure of Admissible Representations</i>	4675
139.5	<i>Local Galois Representations</i>	4676
139.6	<i>Examples</i>	4676
139.7	<i>Bibliography</i>	4680

VOLUME 12: CONTENTS

XVIII	TOPOLOGY	4681
140	SIMPLICIAL HOMOLOGY	4683
140.1	<i>Introduction</i>	4685
140.2	<i>Simplicial Complexes</i>	4685
140.2.1	Standard Topological Objects	4696
140.3	<i>Homology Computation</i>	4697
140.4	<i>Bibliography</i>	4701

XIX	GEOMETRY	4703
141	FINITE PLANES	4705
141.1	<i>Introduction</i>	4707
141.1.1	Planes in Magma	4707
141.2	<i>Construction of a Plane</i>	4707
141.3	<i>The Point-Set and Line-Set of a Plane</i>	4710
141.3.1	Introduction	4710
141.3.2	Creating Point-Sets and Line-Sets	4710
141.3.3	Using the Point-Set and Line-Set to Create Points and Lines	4710
141.3.4	Retrieving the Plane from Points, Lines, Point-Sets and Line-Sets	4714
141.4	<i>The Set of Points and Set of Lines</i>	4714
141.5	<i>The Defining Points of a Plane</i>	4715
141.6	<i>Subplanes</i>	4716
141.7	<i>Structures Associated with a Plane</i>	4717
141.8	<i>Numerical Invariants of a Plane</i>	4718
141.9	<i>Properties of Planes</i>	4719
141.10	<i>Identity and Isomorphism</i>	4719
141.11	<i>The Connection between Projective and Affine Planes</i>	4720
141.12	<i>Operations on Points and Lines</i>	4721
141.12.1	Elementary Operations	4721
141.12.2	Deconstruction Functions	4722
141.12.3	Other Point and Line Functions	4725
141.13	<i>Arcs</i>	4726
141.14	<i>Unitals</i>	4729
141.15	<i>The Collineation Group of a Plane</i>	4730
141.15.1	The Collineation Group Function	4731
141.15.2	General Action of Collineations	4732
141.15.3	Central Collineations	4736
141.15.4	Transitivity Properties	4737
141.16	<i>Translation Planes</i>	4738
141.17	<i>Planes and Designs</i>	4738
141.18	<i>Planes, Graphs and Codes</i>	4739
142	INCIDENCE GEOMETRY	4741
142.1	<i>Introduction</i>	4743
142.2	<i>Construction of Incidence and Coset Geometries</i>	4744
142.2.1	Construction of an Incidence Geometry	4744
142.2.2	Construction of a Coset Geometry	4748
142.3	<i>Elementary Invariants</i>	4751
142.4	<i>Conversion Functions</i>	4753
142.5	<i>Residues</i>	4754
142.6	<i>Truncations</i>	4755
142.7	<i>Shadows</i>	4755
142.8	<i>Shadow Spaces</i>	4755
142.9	<i>Automorphism Group and Correlation Group</i>	4756
142.10	<i>Properties of Incidence Geometries and Coset Geometries</i>	4756
142.11	<i>Intersection Properties of Coset Geometries</i>	4757
142.12	<i>Primitivity Properties on Coset Geometries</i>	4758
142.13	<i>Diagram of an Incidence Geometry</i>	4759
142.14	<i>Bibliography</i>	4762

143	CONVEX POLYTOPES AND POLYHEDRA	4763
143.1	<i>Introduction and First Examples</i>	4765
143.2	<i>Polytopes, Cones and Polyhedra</i>	4770
143.2.1	Polytopes	4770
143.2.2	Cones	4771
143.2.3	Polyhedra	4772
143.2.4	Arithmetic Operations on Polyhedra	4774
143.3	<i>Basic Combinatorics of Polytopes and Polyhedra</i>	4775
143.3.1	Vertices and Inequalities	4775
143.3.2	Facets and Faces	4777
143.4	<i>The Combinatorics of Polytopes</i>	4778
143.4.1	Points in Polytopes	4778
143.4.2	Ehrhart Theory of Polytopes	4779
143.4.3	Automorphisms of a Polytope	4779
143.4.4	Operations on Polytopes	4780
143.5	<i>Cones and Polyhedra</i>	4780
143.5.1	Generators of Cones	4780
143.5.2	Properties of Polyhedra	4781
143.5.3	Attributes of Polyhedra	4785
143.5.4	Combinatorics of Polyhedral Complexes	4785
143.6	<i>Toric Lattices</i>	4785
143.6.1	Toric Lattices	4786
143.6.2	Points of Toric Lattices	4787
143.6.3	Operations on Toric Lattices	4790
143.6.4	Maps of Toric Lattices	4792
143.7	<i>Bibliography</i>	4793

XX	COMBINATORICS	4795
144	ENUMERATIVE COMBINATORICS	4797
144.1	<i>Introduction</i>	4799
144.2	<i>Combinatorial Functions</i>	4799
144.3	<i>Subsets of a Finite Set</i>	4801
145	PARTITIONS, WORDS AND YOUNG TABLEAUX	4803
145.1	<i>Introduction</i>	4805
145.2	<i>Partitions</i>	4805
145.3	<i>Words</i>	4808
145.3.1	Ordered Monoids	4808
145.3.2	Plactic Monoids	4811
145.4	<i>Tableaux</i>	4814
145.4.1	Tableau Monoids	4814
145.4.2	Creation of Tableaux	4816
145.4.3	Enumeration of Tableaux	4819
145.4.4	Random Tableaux	4821
145.4.5	Basic Access Functions	4822
145.4.6	Properties	4825
145.4.7	Operations	4827
145.4.8	The Robinson-Schensted-Knuth Correspondence	4830
145.4.9	Counting Tableaux	4834
145.5	<i>Bibliography</i>	4836
146	SYMMETRIC FUNCTIONS	4837
146.1	<i>Introduction</i>	4839
146.2	<i>Creation</i>	4841
146.2.1	Creation of Symmetric Function Algebras	4841
146.2.2	Creation of Symmetric Functions	4843
146.3	<i>Structure Operations</i>	4846
146.3.1	Related Structures	4846
146.3.2	Ring Predicates and Booleans	4847
146.3.3	Predicates on Basis Types	4847
146.4	<i>Element Operations</i>	4847
146.4.1	Parent and Category	4847
146.4.2	Print Styles	4848
146.4.3	Additive Arithmetic Operators	4848
146.4.4	Multiplication	4849
146.4.5	Plethysm	4850
146.4.6	Boolean Operators	4850
146.4.7	Accessing Elements	4851
146.4.8	Multivariate Polynomials	4852
146.4.9	Frobenius Homomorphism	4853
146.4.10	Inner Product	4854
146.4.11	Combinatorial Objects	4854
146.4.12	Symmetric Group Character	4854
146.4.13	Restrictions	4855
146.5	<i>Transition Matrices</i>	4856
146.5.1	Transition Matrices from Schur Basis	4856
146.5.2	Transition Matrices from Monomial Basis	4858
146.5.3	Transition Matrices from Homogeneous Basis	4859
146.5.4	Transition Matrices from Power Sum Basis	4860

146.5.5	Transition Matrices from Elementary Basis	4861
146.6	<i>Bibliography</i>	4862
147	INCIDENCE STRUCTURES AND DESIGNS	4863
147.1	<i>Introduction</i>	4865
147.2	<i>Construction of Incidence Structures and Designs</i>	4866
147.3	<i>The Point-Set and Block-Set of an Incidence Structure</i>	4870
147.3.1	Introduction	4870
147.3.2	Creating Point-Sets and Block-Sets	4871
147.3.3	Creating Points and Blocks	4871
147.4	<i>General Design Constructions</i>	4873
147.4.1	The Construction of Related Structures	4873
147.4.2	The Witt Designs	4876
147.4.3	Difference Sets and their Development	4876
147.5	<i>Elementary Invariants of an Incidence Structure</i>	4878
147.6	<i>Elementary Invariants of a Design</i>	4879
147.7	<i>Operations on Points and Blocks</i>	4881
147.8	<i>Elementary Properties of Incidence Structures and Designs</i>	4883
147.9	<i>Resolutions, Parallelisms and Parallel Classes</i>	4885
147.10	<i>Conversion Functions</i>	4888
147.11	<i>Identity and Isomorphism</i>	4889
147.12	<i>The Automorphism Group of an Incidence Structure</i>	4890
147.12.1	Construction of Automorphism Groups	4890
147.12.2	Action of Automorphisms	4893
147.13	<i>Incidence Structures, Graphs and Codes</i>	4895
147.14	<i>Automorphisms of Matrices</i>	4896
147.15	<i>Bibliography</i>	4897
148	HADAMARD MATRICES	4899
148.1	<i>Introduction</i>	4901
148.2	<i>Equivalence Testing</i>	4901
148.3	<i>Associated 3-Designs</i>	4903
148.4	<i>Automorphism Group</i>	4904
148.5	<i>Databases</i>	4904
148.5.1	Updating the Databases	4905
149	GRAPHS	4909
149.1	<i>Introduction</i>	4913
149.2	<i>Construction of Graphs and Digraphs</i>	4914
149.2.1	Bounds on the Graph Order	4914
149.2.2	Construction of a General Graph	4915
149.2.3	Construction of a General Digraph	4918
149.2.4	Operations on the Support	4920
149.2.5	Construction of a Standard Graph	4921
149.2.6	Construction of a Standard Digraph	4923
149.3	<i>Graphs with a Sparse Representation</i>	4924
149.4	<i>The Vertex-Set and Edge-Set of a Graph</i>	4926
149.4.1	Introduction	4926
149.4.2	Creating Edges and Vertices	4926
149.4.3	Operations on Vertex-Sets and Edge-Sets	4928
149.4.4	Operations on Edges and Vertices	4929
149.5	<i>Labelled, Capacitated and Weighted Graphs</i>	4930

149.6	<i>Standard Constructions for Graphs</i>	4930
149.6.1	Subgraphs and Quotient Graphs	4930
149.6.2	Incremental Construction of Graphs	4932
149.6.3	Constructing Complements, Line Graphs; Contraction, Switching	4935
149.7	<i>Unions and Products of Graphs</i>	4937
149.8	<i>Converting between Graphs and Digraphs</i>	4939
149.9	<i>Construction from Groups, Codes and Designs</i>	4939
149.9.1	Graphs Constructed from Groups	4939
149.9.2	Graphs Constructed from Designs	4940
149.9.3	Miscellaneous Graph Constructions	4941
149.10	<i>Elementary Invariants of a Graph</i>	4942
149.11	<i>Elementary Graph Predicates</i>	4943
149.12	<i>Adjacency and Degree</i>	4945
149.12.1	Adjacency and Degree Functions for a Graph	4945
149.12.2	Adjacency and Degree Functions for a Digraph	4946
149.13	<i>Connectedness</i>	4948
149.13.1	Connectedness in a Graph	4948
149.13.2	Connectedness in a Digraph	4949
149.13.3	Graph Triconnectivity	4949
149.13.4	Maximum Matching in Bipartite Graphs	4951
149.13.5	General Vertex and Edge Connectivity in Graphs and Digraphs	4952
149.14	<i>Distances, Paths and Circuits in a Graph</i>	4955
149.14.1	Distances, Paths and Circuits in a Possibly Weighted Graph	4955
149.14.2	Distances, Paths and Circuits in a Non-Weighted Graph	4955
149.15	<i>Maximum Flow, Minimum Cut, and Shortest Paths</i>	4956
149.16	<i>Matrices and Vector Spaces Associated with a Graph or Digraph</i>	4957
149.17	<i>Spanning Trees of a Graph or Digraph</i>	4957
149.18	<i>Directed Trees</i>	4958
149.19	<i>Colourings</i>	4959
149.20	<i>Cliques, Independent Sets</i>	4960
149.21	<i>Planar Graphs</i>	4965
149.22	<i>Automorphism Group of a Graph or Digraph</i>	4968
149.22.1	The Automorphism Group Function	4968
149.22.2	nauty Invariants	4969
149.22.3	Graph Colouring and Automorphism Group	4971
149.22.4	Variants of Automorphism Group	4972
149.22.5	Action of Automorphisms	4976
149.23	<i>Symmetry and Regularity Properties of Graphs</i>	4979
149.24	<i>Graph Databases and Graph Generation</i>	4981
149.24.1	Strongly Regular Graphs	4981
149.24.2	Small Graphs	4983
149.24.3	Generating Graphs	4984
149.24.4	A General Facility	4987
149.25	<i>Bibliography</i>	4989
150	MULTIGRAPHS	4991
150.1	<i>Introduction</i>	4995
150.2	<i>Construction of Multigraphs</i>	4996
150.2.1	Construction of a General Multigraph	4996
150.2.2	Construction of a General Multidigraph	4997
150.2.3	Printing of a Multi(di)graph	4998
150.2.4	Operations on the Support	4999
150.3	<i>The Vertex-Set and Edge-Set of Multigraphs</i>	5000
150.4	<i>Vertex and Edge Decorations</i>	5003
150.4.1	Vertex Decorations: Labels	5003

150.4.2	Edge Decorations	5004
150.4.3	Unlabelled, or Uncapacitated, or Unweighted Graphs	5007
150.5	<i>Standard Construction for Multigraphs</i>	5010
150.5.1	Subgraphs	5010
150.5.2	Incremental Construction of Multigraphs	5012
150.5.3	Vertex Insertion, Contraction	5016
150.5.4	Unions of Multigraphs	5017
150.6	<i>Conversion Functions</i>	5018
150.6.1	Orientated Graphs	5019
150.6.2	Converse	5019
150.6.3	Converting between Simple Graphs and Multigraphs	5019
150.7	<i>Elementary Invariants and Predicates for Multigraphs</i>	5020
150.8	<i>Adjacency and Degree</i>	5022
150.8.1	Adjacency and Degree Functions for Multigraphs	5023
150.8.2	Adjacency and Degree Functions for Multidigraphs	5024
150.9	<i>Connectedness</i>	5025
150.9.1	Connectedness in a Multigraph	5026
150.9.2	Connectedness in a Multidigraph	5026
150.9.3	Triconnectivity for Multigraphs	5027
150.9.4	Maximum Matching in Bipartite Multigraphs	5027
150.9.5	General Vertex and Edge Connectivity in Multigraphs and Multidigraphs	5027
150.10	<i>Spanning Trees</i>	5029
150.11	<i>Planar Graphs</i>	5030
150.12	<i>Distances, Shortest Paths and Minimum Weight Trees</i>	5034
150.13	<i>Bibliography</i>	5038
151	NETWORKS	5039
151.1	<i>Introduction</i>	5041
151.2	<i>Construction of Networks</i>	5041
151.2.1	Magma Output: Printing of a Network	5043
151.3	<i>Standard Construction for Networks</i>	5045
151.3.1	Subgraphs	5045
151.3.2	Incremental Construction: Adding Edges	5049
151.3.3	Union of Networks	5050
151.4	<i>Maximum Flow and Minimum Cut</i>	5051
151.5	<i>Bibliography</i>	5057

VOLUME 13: CONTENTS

XXI	CODING THEORY	5059
152	LINEAR CODES OVER FINITE FIELDS	5061
152.1	<i>Introduction</i>	5065
152.2	<i>Construction of Codes</i>	5066
152.2.1	Construction of General Linear Codes	5066
152.2.2	Some Trivial Linear Codes	5068
152.2.3	Some Basic Families of Codes	5069
152.3	<i>Invariants of a Code</i>	5071
152.3.1	Basic Numerical Invariants	5071
152.3.2	The Ambient Space and Alphabet	5072
152.3.3	The Code Space	5072
152.3.4	The Dual Space	5073
152.3.5	The Information Space and Information Sets	5074
152.3.6	The Syndrome Space	5075
152.3.7	The Generator Polynomial	5075
152.4	<i>Operations on Codewords</i>	5076
152.4.1	Construction of a Codeword	5076
152.4.2	Arithmetic Operations on Codewords	5077
152.4.3	Distance and Weight	5077
152.4.4	Vector Space and Related Operations	5078
152.4.5	Predicates for Codewords	5079
152.4.6	Accessing Components of a Codeword	5079
152.5	<i>Coset Leaders</i>	5080
152.6	<i>Subcodes</i>	5081
152.6.1	The Subcode Constructor	5081
152.6.2	Sum, Intersection and Dual	5083
152.6.3	Membership and Equality	5084
152.7	<i>Properties of Codes</i>	5085
152.8	<i>The Weight Distribution</i>	5087
152.8.1	The Minimum Weight	5087
152.8.2	The Weight Distribution	5092
152.8.3	The Weight Enumerator	5093
152.8.4	The MacWilliams Transform	5094
152.8.5	Words	5095
152.8.6	Covering Radius and Diameter	5097
152.9	<i>Families of Linear Codes</i>	5098
152.9.1	Cyclic and Quasicyclic Codes	5098
152.9.2	BCH Codes and their Generalizations	5100
152.9.3	Quadratic Residue Codes and their Generalizations	5103
152.9.4	Reed–Solomon and Justesen Codes	5105
152.9.5	Maximum Distance Separable Codes	5106
152.10	<i>New Codes from Existing</i>	5106
152.10.1	Standard Constructions	5106
152.10.2	Changing the Alphabet of a Code	5109
152.10.3	Combining Codes	5110
152.11	<i>Coding Theory and Cryptography</i>	5114
152.11.1	Standard Attacks	5115
152.11.2	Generalized Attacks	5116

152.12	<i>Bounds</i>	5117
152.12.1	Best Known Bounds for Linear Codes	5117
152.12.2	Bounds on the Cardinality of a Largest Code	5118
152.12.3	Bounds on the Minimum Distance	5120
152.12.4	Asymptotic Bounds on the Information Rate	5120
152.12.5	Other Bounds	5120
152.13	<i>Best Known Linear Codes</i>	5121
152.14	<i>Decoding</i>	5127
152.15	<i>Transforms</i>	5128
152.15.1	Mattson–Solomon Transforms	5128
152.15.2	Krawchouk Polynomials	5129
152.16	<i>Automorphism Groups</i>	5129
152.16.1	Introduction	5129
152.16.2	Group Actions	5130
152.16.3	Automorphism Group	5131
152.16.4	Equivalence and Isomorphism of Codes	5134
152.17	<i>Bibliography</i>	5134
153	ALGEBRAIC-GEOMETRIC CODES	5137
153.1	<i>Introduction</i>	5139
153.2	<i>Creation of an Algebraic Geometric Code</i>	5140
153.3	<i>Properties of AG–Codes</i>	5142
153.4	<i>Access Functions</i>	5143
153.5	<i>Decoding AG Codes</i>	5143
153.6	<i>Toric Codes</i>	5144
153.7	<i>Bibliography</i>	5145
154	LOW DENSITY PARITY CHECK CODES	5147
154.1	<i>Introduction</i>	5149
154.1.1	Constructing LDPC Codes	5149
154.1.2	Access Functions	5150
154.1.3	LDPC Decoding and Simulation	5152
154.1.4	Density Evolution	5154
155	LINEAR CODES OVER FINITE RINGS	5159
155.1	<i>Introduction</i>	5161
155.2	<i>Construction of Codes</i>	5161
155.2.1	Construction of General Linear Codes	5161
155.2.2	Construction of Simple Linear Codes	5164
155.2.3	Construction of General Cyclic Codes	5165
155.3	<i>Invariants of Codes</i>	5167
155.4	<i>Codes over \mathbf{Z}_4</i>	5168
155.4.1	The Gray Map	5168
155.4.2	Families of Codes over \mathbf{Z}_4	5170
155.4.3	Derived Binary Codes	5176
155.4.4	The Standard Form	5177
155.4.5	Constructing New Codes from Old	5178
155.4.6	Invariants of Codes over \mathbf{Z}_4	5181
155.4.7	Other \mathbf{Z}_4 functions	5182
155.5	<i>Construction of Subcodes of Linear Codes</i>	5182
155.5.1	The Subcode Constructor	5182
155.6	<i>Weight Distributions</i>	5183
155.6.1	Hamming Weight	5183

155.6.2	Lee Weight	5184
155.6.3	Euclidean Weight	5186
155.7	<i>Weight Enumerators</i>	5187
155.8	<i>Constructing New Codes from Old</i>	5190
155.8.1	Sum, Intersection and Dual	5190
155.8.2	Standard Constructions	5191
155.9	<i>Operations on Codewords</i>	5194
155.9.1	Construction of a Codeword	5194
155.9.2	Operations on Codewords and Vectors	5195
155.9.3	Accessing Components of a Codeword	5197
155.10	<i>Boolean Predicates</i>	5197
155.11	<i>Bibliography</i>	5198
156	ADDITIVE CODES	5199
156.1	<i>Introduction</i>	5201
156.2	<i>Construction of Additive Codes</i>	5202
156.2.1	Construction of General Additive Codes	5202
156.2.2	Some Trivial Additive Codes	5204
156.3	<i>Invariants of an Additive Code</i>	5205
156.3.1	The Ambient Space and Alphabet	5205
156.3.2	Basic Numerical Invariants	5206
156.3.3	The Code Space	5207
156.3.4	The Dual Space	5207
156.4	<i>Operations on Codewords</i>	5208
156.4.1	Construction of a Codeword	5208
156.4.2	Arithmetic Operations on Codewords	5208
156.4.3	Distance and Weight	5209
156.4.4	Vector Space and Related Operations	5209
156.4.5	Predicates for Codewords	5210
156.4.6	Accessing Components of a Codeword	5210
156.5	<i>Subcodes</i>	5210
156.5.1	The Subcode Constructor	5210
156.5.2	Sum, Intersection and Dual	5212
156.5.3	Membership and Equality	5213
156.6	<i>Properties of Codes</i>	5213
156.7	<i>The Weight Distribution</i>	5214
156.7.1	The Minimum Weight	5214
156.7.2	The Weight Distribution	5217
156.7.3	The Weight Enumerator	5217
156.7.4	The MacWilliams Transform	5218
156.7.5	Words	5218
156.8	<i>Families of Linear Codes</i>	5219
156.8.1	Cyclic Codes	5219
156.8.2	Quasicyclic Codes	5220
156.9	<i>New Codes from Old</i>	5221
156.9.1	Standard Constructions	5221
156.9.2	Combining Codes	5222
156.10	<i>Automorphism Group</i>	5223

157	QUANTUM CODES	5225
157.1	<i>Introduction</i>	5227
157.2	<i>Constructing Quantum Codes</i>	5229
157.2.1	Construction of General Quantum Codes	5229
157.2.2	Construction of Special Quantum Codes	5234
157.2.3	CSS Codes	5234
157.2.4	Cyclic Quantum Codes	5235
157.2.5	Quasi-Cyclic Quantum Codes	5238
157.3	<i>Access Functions</i>	5239
157.3.1	Quantum Error Group	5240
157.4	<i>Inner Products and Duals</i>	5242
157.5	<i>Weight Distribution and Minimum Weight</i>	5244
157.6	<i>New Codes From Old</i>	5247
157.7	<i>Best Known Quantum Codes</i>	5248
157.8	<i>Best Known Bounds</i>	5251
157.9	<i>Automorphism Group</i>	5252
157.10	<i>Hilbert Spaces</i>	5254
157.10.1	Creation of Quantum States	5255
157.10.2	Manipulation of Quantum States	5257
157.10.3	Inner Product and Probabilities of Quantum States	5258
157.10.4	Unitary Transformations on Quantum States	5261
157.11	<i>Bibliography</i>	5262

XXII CRYPTOGRAPHY 5263

158	PSEUDO-RANDOM BIT SEQUENCES	5265
158.1	<i>Introduction</i>	5267
158.2	<i>Linear Feedback Shift Registers</i>	5267
158.3	<i>Number Theoretic Bit Generators</i>	5268
158.4	<i>Correlation Functions</i>	5270
158.5	<i>Decimation</i>	5271

XXIII	OPTIMIZATION	5273
159	LINEAR PROGRAMMING	5275
159.1	Introduction	5277
159.2	Explicit LP Solving Functions	5278
159.3	Creation of LP objects	5280
159.4	Operations on LP objects	5280
159.5	Bibliography	5283

PART I

THE MAGMA LANGUAGE

1	STATEMENTS AND EXPRESSIONS	3
2	FUNCTIONS, PROCEDURES AND PACKAGES	33
3	INPUT AND OUTPUT	63
4	ENVIRONMENT AND OPTIONS	93
5	MAGMA SEMANTICS	115
6	THE MAGMA PROFILER	135
7	DEBUGGING MAGMA CODE	145

1 STATEMENTS AND EXPRESSIONS

1.1 Introduction	5	<code>IsCoercible(S, x)</code>	13
1.2 Starting, Interrupting and Terminating	5	1.7 The where ... is Construction .	14
<code><Ctrl>-C</code>	5	<code>e₁ where id is e₂</code>	14
<code>quit;</code>	5	<code>e₁ where id := e₂</code>	14
<code><Ctrl>-D</code>	5	1.8 Conditional Statements and Expressions	16
<code><Ctrl>-\</code>	5	<i>1.8.1 The Simple Conditional Statement .</i>	<i>16</i>
1.3 Identifiers	5	<i>1.8.2 The Simple Conditional Expression .</i>	<i>17</i>
1.4 Assignment	6	<code>bool select e₁ else e₂</code>	17
<i>1.4.1 Simple Assignment</i>	<i>6</i>	<i>1.8.3 The Case Statement</i>	<i>18</i>
<code>x := e;</code>	6	<i>1.8.4 The Case Expression</i>	<i>18</i>
<code>x₁, x₂, ..., x_n := e;</code>	6	1.9 Error Handling Statements . . .	19
<code>_ := e;</code>	6	<i>1.9.1 The Error Objects</i>	<i>19</i>
<code>assigned</code>	6	<code>Error(x)</code>	19
<i>1.4.2 Indexed Assignment</i>	<i>7</i>	<code>e'Position</code>	19
<code>x[e₁][e₂]...[e_n] := e;</code>	7	<code>e'Traceback</code>	19
<code>x[e₁, e₂, ..., e_n] := e;</code>	7	<code>e'Object</code>	19
<i>1.4.3 Generator Assignment</i>	<i>8</i>	<code>e'Type</code>	19
<code>E<x₁, x₂, ..., x_n> := e;</code>	8	<i>1.9.2 Error Checking and Assertions . . .</i>	<i>19</i>
<code>E<[x]> := e;</code>	8	<code>error e, ..., e;</code>	19
<code>AssignNames(∼S, [s₁, ..., s_n])</code>	9	<code>error if bool, e, ..., e;</code>	19
<i>1.4.4 Mutation Assignment</i>	<i>9</i>	<code>assert bool;</code>	20
<code>x o:= e;</code>	9	<code>assert2 bool;</code>	20
<i>1.4.5 Deletion of Values</i>	<i>10</i>	<code>assert3 bool;</code>	20
<code>delete</code>	10	<i>1.9.3 Catching Errors</i>	<i>20</i>
1.5 Boolean values	10	1.10 Iterative Statements	21
<i>1.5.1 Creation of Booleans</i>	<i>11</i>	<i>1.10.1 Definite Iteration</i>	<i>21</i>
<code>Booleans()</code>	11	<i>1.10.2 Indefinite Iteration</i>	<i>22</i>
<code>#</code>	11	<i>1.10.3 Early Exit from Iterative Statements .</i>	<i>23</i>
<code>true</code>	11	<code>continue;</code>	23
<code>false</code>	11	<code>continue id;</code>	23
<code>Random(B)</code>	11	<code>break;</code>	23
<i>1.5.2 Boolean Operators</i>	<i>11</i>	<code>break id;</code>	23
<code>and</code>	11	1.11 Runtime Evaluation: the eval Expression	24
<code>or</code>	11	<code>eval expression</code>	24
<code>xor</code>	11	1.12 Comments and Continuation . .	26
<code>not</code>	11	<code>//</code>	26
<i>1.5.3 Equality Operators</i>	<i>11</i>	<code>/* */</code>	26
<code>eq</code>	11	<code>\</code>	26
<code>ne</code>	12	1.13 Timing	26
<code>cmpeq</code>	12	<code>Cputime()</code>	26
<code>cmpne</code>	12	<code>Cputime(t)</code>	26
<i>1.5.4 Iteration</i>	<i>12</i>		
1.6 Coercion	13		
<code>!</code>	13		

Realtime()	26	CoveringStructure(S, T)	29
Realtime(t)	27	ExistsCoveringStructure(S, T)	29
ClockCycles()	27	1.15 Random Object Generation . . .	30
time statement;	27	SetSeed(s, c)	30
vtime flag: statement;	27	SetSeed(s)	30
vtime flag, n: statement;	27	GetSeed()	31
1.14 Types, Category Names, and		Random(S)	31
Structures	28	Random(a, b)	31
Type(x)	28	Random(b)	31
Category(x)	28	1.16 Miscellaneous	32
ExtendedType(x)	28	IsIntrinsic(S)	32
ExtendedCategory(x)	28	1.17 Bibliography	32
ISA(T, U)	28		
MakeType(S)	29		
ElementType(S)	29		

Chapter 1

STATEMENTS AND EXPRESSIONS

1.1 Introduction

This chapter contains a very terse overview of the basic elements of the MAGMA language.

1.2 Starting, Interrupting and Terminating

If MAGMA has been installed correctly, it may be activated by typing ‘magma’.

`<Ctrl>-C`

Interrupt MAGMA while it is performing some task (that is, while the user does not have a ‘prompt’) to obtain a new prompt. MAGMA will try to interrupt at a convenient point (this may take some time). If `<Ctrl>-C` is typed twice within half a second, MAGMA will exit completely immediately.

`quit;`

`<Ctrl>-D`

Terminate the current MAGMA-session.

`<Ctrl>-\`

Immediately quit MAGMA (send the signal SIGQUIT to the MAGMA process on Unix machines). This is occasionally useful when `<Ctrl>-C` does not seem to work.

1.3 Identifiers

Identifiers (names for user variables, functions etc.) must begin with a letter, and this letter may be followed by any combination of letters or digits, provided that the name is not a *reserved word* (see the chapter on reserved words a complete list). In this definition the underscore `_` is treated as a letter; but note that a single underscore is a reserved word. Identifier names are case-sensitive; that is, they are distinguished from one another by lower and upper case.

Intrinsic MAGMA functions usually have names beginning with capital letters (current exceptions are `pCore`, `pQuotient` and the like, where the `p` indicates a prime). Note that these identifiers are *not* reserved words; that is, one may use names of intrinsic functions for variables.

1.4 Assignment

In this section the basic forms of assignment of values to identifiers are described.

1.4.1 Simple Assignment

$x := \text{expression};$

Given an identifier x and an expression expression , assign the value of expression to x .

Example H1E1

```
> x := 13;
> y := x^2-2;
> x, y;
13 167
```

Intrinsic function names are identifiers just like the x and y above. Therefore it is possible to reassign them to your own variable.

```
> f := PreviousPrime;
> f(y);
163
```

In fact, the same can also be done with the infix operators, except that it is necessary to enclose their names in quotes. Thus it is possible to define your own function **Plus** to be the function taking the arguments of the intrinsic $+$ operator.

```
> Plus := '+';
> Plus(1/2, 2);
5/2
```

Note that redefining the infix operator will *not* change the corresponding mutation assignment operator (in this case $+=$).

$x_1, x_2, \dots, x_n := \text{expression};$

Assignment of $n \geq 1$ values, returned by the expression on the right hand side. Here the x_i are identifiers, and the right hand side expression must return $m \geq n$ values; the first n of these will be assigned to x_1, x_2, \dots, x_n respectively.

$_ := \text{expression};$

Ignore the value(s) returned by the expression on the right hand side.

assigned x

An expression which yields the value **true** if the ‘local’ identifier x has a value currently assigned to it and **false** otherwise. Note that the **assigned**-expression will return **false** for intrinsic function names, since they are not ‘local’ variables (the identifiers can be assigned to something else, hiding the intrinsic function).

Example H1E2

The extended greatest common divisor function `Xgcd` returns 3 values: the gcd d of the arguments m and n , as well as multipliers x and y such that $d = xm + yn$. If one is only interested in the gcd of the integers $m = 12$ and $n = 15$, say, one could use:

```
> d := Xgcd(12, 15);
```

To obtain the multipliers as well, type

```
> d, x, y := Xgcd(12, 15);
```

while the following offers ways to retrieve two of the three return values.

```
> d, x := Xgcd(12, 15);
```

```
> d, _, y := Xgcd(12, 15);
```

```
> _, x, y := Xgcd(12, 15);
```

1.4.2 Indexed Assignment

<code>x[expression₁][expression₂]...[expression_n] := expression;</code>
--

<code>x[expression₁, expression₂, ..., expression_n] := expression;</code>
--

If the argument on the left hand side allows *indexing* at least n levels deep, and if this indexing can be used to modify the argument, this offers two equivalent ways of accessing and modifying the entry indicated by the expressions `expri`. The most important case is that of (nested) sequences.

Example H1E3

Left hand side indexing can be used (as is explained in more detail in the chapter on sequences) to modify existing entries.

```
> s := [ [1], [1, 2], [1, 2, 3] ];
```

```
> s;
```

```
[
  [ 1 ],
  [ 1, 2 ],
  [ 1, 2, 3 ]
]
```

```
> s[2, 2] := -1;
```

```
> s;
```

```
[
  [ 1 ],
  [ 1, -1 ],
  [ 1, 2, 3 ]
]
```

1.4.3 Generator Assignment

Because of the importance of naming the generators in the case of finitely presented magmas, special forms of assignment allow names to be assigned at the time the magma itself is assigned.

$$E\langle x_1, x_2, \dots, x_n \rangle := \text{expression};$$

If the right hand side expression returns a structure that allows *naming* of ‘generators’, such as finitely generated groups or algebras, polynomial rings, this assigns the first n names to the variables x_1, x_2, \dots, x_n . Naming of generators usually has two aspects; firstly, the *strings* x_1, x_2, \dots, x_n are used for printing of the generators, and secondly, to the *identifiers* x_1, x_2, \dots, x_n are assigned the values of the generators. Thus, except for this side effect regarding printing, the above assignment is equivalent to the $n + 1$ assignments:

$$\begin{aligned} E &:= \text{expression}; \\ x_1 &:= E.1; \quad x_2 := E.2; \quad \dots \quad x_n := E.n; \end{aligned}$$

$$E\langle [x] \rangle := \text{expression};$$

If the right hand side expression returns a structure S that allows *naming* of ‘generators’, this assigns the names of S to be those formed by appending the numbers 1, 2, etc. in order enclosed in square brackets to x (considered as a string) and assigns x to the sequence of the names of S .

Example H1E4

We demonstrate the sequence method of generator naming.

```
> P<[X]> := PolynomialRing(RationalField(), 5);
> P;
Polynomial ring of rank 5 over Rational Field
Lexicographical Order
Variables: X[1], X[2], X[3], X[4], X[5]
> X;
[
  X[1],
  X[2],
  X[3],
  X[4],
  X[5]
]
> &+X;
X[1] + X[2] + X[3] + X[4] + X[5]
> (&+X)^2;
X[1]^2 + 2*X[1]*X[2] + 2*X[1]*X[3] + 2*X[1]*X[4] +
  2*X[1]*X[5] + X[2]^2 + 2*X[2]*X[3] + 2*X[2]*X[4] +
  2*X[2]*X[5] + X[3]^2 + 2*X[3]*X[4] + 2*X[3]*X[5] +
  X[4]^2 + 2*X[4]*X[5] + X[5]^2
```

`AssignNames($\sim S$, [s_1 , ... s_n])`

If S is a structure that allows *naming* of ‘generators’ (see the Index for a complete list), this procedure assigns the names specified by the strings to these generators. The number of generators has to match the length of the sequence. This will result in the creation of a new structure.

Example H1E5

```
> G<a, b> := Group<a, b | a^2 = b^3 = a*b*b^2>;
> w := a * b;
> w;
a * b
> AssignNames( $\sim G$ , ["c", "d"]);
> G;
Finitely presented group G on 2 generators
Relations
  c^2 = d^-1 * c * d^3
  d^3 = d^-1 * c * d^3
> w;
a * b
> Parent(w);
Finitely presented group on 2 generators
Relations
  a^2 = b^-1 * a * b^3
  b^3 = b^-1 * a * b^3
> G eq Parent(w);
true
```

1.4.4 Mutation Assignment

`x o:= expression;`

This is the *mutation assignment*: the expression is evaluated and the operator o is applied on the result and the current value of x , and assigned to x again. Thus the result is equivalent to (but an optimized version of): `x := x o expression;`. The operator may be any of the operations `join`, `meet`, `diff`, `sdiff`, `cat`, `*`, `+`, `-`, `/`, `^`, `div`, `mod`, `and`, `or`, `xor` provided that the operation is legal on its arguments of course.

Example H1E6

The following simple program to produce a set consisting of the first 10 powers of 2 involves the use of two different mutation assignments.

```
> x := 1;
> S := { };
> for i := 1 to 10 do
>   S join:= { x };
>   x *:= 2;
> end for;
> S;
{ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 }
```

1.4.5 Deletion of Values

delete x

(Statement.) Delete the current value of the identifier x . The memory occupied is freed, unless other variables still refer to it. If x is the name of an intrinsic MAGMA function that has been reassigned to, the identifier will after deletion again refer to that intrinsic function. Intrinsic functions cannot be deleted.

1.5 Boolean values

This section deals with logical values (“Booleans”).

Booleans are primarily of importance as (return) values for (intrinsic) predicates. It is important to know that the truth-value of the operators **and** and **or** is always evaluated *left to right*, that is, the left-most clause is evaluated first, and if that determines the value of the operator evaluation is aborted; if not, the next clause is evaluated, etc. So, for example, if x is a boolean, it is safe (albeit silly) to type:

```
> if x eq true or x eq false or x/0 eq 1 then
>   "fine";
> else
>   "error";
> end if;
```

even though $x/0$ would cause an error (“Bad arguments”, not “Division by zero”!) upon evaluation, because the truth value will have been determined before the evaluation of $x/0$ takes place.

1.5.1 Creation of Booleans

`Booleans()`

The Boolean structure.

`#B`

Cardinality of Boolean structure (2).

`true`

`false`

The Boolean elements.

`Random(B)`

Return a random Boolean.

1.5.2 Boolean Operators

`x and y`

Returns `true` if both x and y are `true`, `false` otherwise. If x is `false`, the expression for y is not evaluated.

`x or y`

Returns `true` if x or y is `true` (or both are `true`), `false` otherwise. If x is `true`, the expression for y is not evaluated.

`x xor y`

Returns `true` if either x or y is `true` (but not both), `false` otherwise.

`not x`

Negate the truth value of x .

1.5.3 Equality Operators

MAGMA provides two equality operators: `eq` for strong (comparable) equality testing, and `cmpeq` for weak equality testing. The operators depend on the concept of *comparability*. Objects x and y in MAGMA are said to be *comparable* if both of the following points hold:

- (a) x and y are both elements of a structure S or there is a structure S such x and y will be coerced into S by automatic coercion;
- (b) There is an equality test for elements of S defined within MAGMA.

The possible automatic coercions are listed in the descriptions of the various MAGMA modules. For instance, the table in the introductory chapter on rings shows that integers can be coerced automatically into the rational field so an integer and a rational are comparable.

`x eq y`

If x and y are comparable, return `true` if x equals y (which will always work by the second rule above). If x and y are not comparable, an error results.

`x ne y`

If x and y are comparable, return **true** if x does not equal y . If x and y are not comparable, an error results.

`x cmpeq y`

If x and y are comparable, return whether x equals y . Otherwise, return **false**. Thus this operator always returns a value and an error never results. It is useful when comparing two objects of completely different types where it is desired that no error can happen. However, it is strongly recommended that **eq** is usually used to allow MAGMA to pick up common unintentional type errors.

`x cmpne y`

If x and y are comparable, return whether x does not equal y . Otherwise, return **true**. Thus this operator always returns a value and an error never results. It is useful when comparing two objects of completely different types where it is desired that no error can happen. However, it is strongly recommended that **ne** is usually used to allow MAGMA to pick up common unintentional type errors.

Example H1E7

We illustrate the different semantics of **eq** and **cmpeq**.

```
> 1 eq 2/2;
true
> 1 cmpeq 2/2;
true
> 1 eq "x";
Runtime error in 'eq': Bad argument types
> 1 cmpeq "x";
false
> [1] eq ["x"];
Runtime error in 'eq': Incompatible sequences
> [1] cmpeq ["x"];
false
```

1.5.4 Iteration

A Boolean structure B may be used for enumeration: **for** x **in** B **do**, and x **in** B **in** set and sequence constructors.

Example H1E8

The following program checks that the functions `ne` and `xor` coincide.

```
> P := Booleans();
> for x, y in P do
>   (x ne y) eq (x xor y);
> end for;
true
true
true
true
```

Similarly, we can test whether for any pair of Booleans x, y it is true that

$$x = y \iff (x \wedge y) \vee (\neg x \wedge \neg y).$$

```
> equal := true;
> for x, y in P do
>   if (x eq y) and not ((x and y) or (not x and not y)) then
>     equal := false;
>   end if;
> end for;
> equal;
true
```

1.6 Coercion

Coercion is a fundamental concept in MAGMA. Given a structures A and B , there is often a natural mathematical mapping from A to B (e.g., embedding, projection), which allows one to transfer elements of A to corresponding elements of B . This is known as coercion. Natural and obvious coercions are supported in MAGMA as much as possible; see the relevant chapters for the coercions possible between various structures.

$S \mid x$

Given a structure S and an object x , attempt to coerce x into S and return the result if successful. If the attempt fails, an error ensues.

`IsCoercible(S, x)`

Given a structure S and an object x , attempt to coerce x into S ; if successful, return `true` and the result of the coercion, otherwise return `false`.

1.7 The where ... is Construction

By the use of the **where ... is** construction, one can within an expression temporarily assign an identifier to a sub-expression. This allows for compact code and efficient re-use of common sub-expressions.

$expression_1$ where $identifier$ is $expression_2$

$expression_1$ where $identifier$:= $expression_2$

This construction is an expression that temporarily assigns the identifier to the second expression and then yields the value of the first expression. The identifier may be referred to in the first expression and it will equal the value of the second expression. The token **:=** can be used as a synonym for **is**. The scope of the identifier is the **where ... is** construction alone except for when the construction is part of an expression list — see below.

The **where** operator is left-associative. This means that there can be multiple uses of **where ... is** constructions and each expression can refer to variables bound in the enclosing constructions.

Another important feature is found in a set or sequence constructor. If there are **where ... is** constructions in the predicate, then any variables bound in them may be referred to in the expression at the beginning of the constructor. If the whole predicate is placed in parentheses, then any variables bound in the predicate do not extend to the expression at the beginning of the constructor.

The **where** operator also extends left in expression lists. That is, if there is an expression E in a expression list which is a **where** construction (or chain of where constructions), the identifiers bound in that where construction (or chain) will be defined in all expressions in the list which are to the left of E . Expression lists commonly arise as argument lists to functions or procedures, return arguments, print statements (with or without the word ‘print’) etc. A where construction also overrides (hides) any where construction to the right of it in the same list. Using parentheses around a where expression ensures that the identifiers bound within it are not seen outside it.

Example H1E9

The following examples illustrate simple uses of **where ... is**.

```
> x := 1;
> x where x is 10;
10
> x;
1
> Order(G) + Degree(G) where G is Sym(3);
9
```

Since **where** is left-associative we may have multiple uses of it. The use of parentheses, of course, can override the usual associativity.

```
> x := 1;
```

```

> y := 2;
> x + y where x is 5 where y is 6;
11
> (x + y where x is 5) where y is 6; // the same
11
> x + y where x is (5 where y is 6);
7
> x + y where x is y where y is 6;
12
> (x + y where x is y) where y is 6; // the same
12
> x + y where x is (y where y is 6);
8

```

We now illustrate how the left expression in a set or sequence constructor can reference the identifiers of **where** constructions in the predicate.

```

> { a: i in [1 .. 10] | IsPrime(a) where a is 3*i + 1 };
{ 7, 13, 19, 31 }
> [<x, y>: i in [1 .. 10] | IsPrime(x) and IsPrime(y)
>   where x is y + 2 where y is 2 * i + 1];
[ <5, 3>, <7, 5>, <13, 11>, <19, 17> ]

```

We next demonstrate the semantics of **where** constructions inside expression lists.

```

> // A simple use:
> [a, a where a is 1];
[ 1, 1 ]
> // An error: where does not extend right
> print [a where a is 1, a];
User error: Identifier 'a' has not been declared
> // Use of parentheses:
> [a, (a where a is 1)] where a is 2;
[ 2, 1 ]
> // Another use of parentheses:
> print [a, (a where a is 1)];
User error: Identifier 'a' has not been declared
> // Use of a chain of where expressions:
> [<a, b>, <b, a> where a is 1 where b is 2];
[ <1, 2>, <2, 1> ]
> // One where overriding another to the right of it:
> [a, a where a is 2, a where a is 3];
[ 2, 2, 3 ]

```

1.8 Conditional Statements and Expressions

The conditional statement has the usual form `if ... then ... else ... end if;`. It has several variants. Within the statement, a special prompt will appear, indicating that the statement has yet to be closed. Conditional statements may be nested.

The conditional expression, `select ... else`, is used for in-line conditionals.

1.8.1 The Simple Conditional Statement

```
if Boolean expression then
    statements1
else
    statements2
end if;
```

```
if Boolean expression then
    statements
end if;
```

The standard conditional statement: the value of the Boolean expression is evaluated. If the result is `true`, the first block of statements is executed, if the result is `false` the second block of statements is executed. If no action is desired in the latter case, the construction may be abbreviated to the second form above.

```
if Boolean expression1 then
    statements1
elif Boolean expression2 then
    statements2
else
    statements3
end if;
```

Since nested conditions occur frequently, `elif` provides a convenient abbreviation for `else if`, which also restricts the ‘level’:

```
if Boolean expression then
    statements1
elif Boolean expression2 then
    statements2
else
    statements3
end if;
```

is equivalent to

```
if Boolean expression1 then
    statements1
else
    if Boolean expression2 then
```

```

        statements2
    else
        statements3
    end if;
end if;

```

Example H1E10

```

> m := Random(2, 10000);
> if IsPrime(m) then
>   m, "is prime";
> else
>   Factorization(m);
> end if;
[ <23, 1>, <37, 1> ]

```

1.8.2 The Simple Conditional Expression

Boolean expression **select** *expression₁* **else** *expression₂*

This is an expression, of which the value is that of *expression₁* or *expression₂*, depending on whether *Boolean expression* is true or false.

Example H1E11

Using the **select ... else** construction, we wish to assign the sign of *y* to the variable *s*.

```

> y := 11;
> s := (y gt 0) select 1 else -1;
> s;
1

```

This is not quite right (when *y* = 0), but fortunately we can nest **select ... else** constructions:

```

> y := -3;
> s := (y gt 0) select 1 else (y eq 0 select 0 else -1);
> s;
-1
> y := 0;
> s := (y gt 0) select 1 else (y eq 0 select 0 else -1);
> s;
0

```

The **select ... else** construction is particularly important in building sets and sequences, because it enables in-line **if** constructions. Here is a sequence containing the first 100 entries of the Fibonacci sequence:

```

> f := [ i gt 2 select Self(i-1)+Self(i-2) else 1 : i in [1..100] ];

```

1.8.3 The Case Statement

```

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

```

The expression following **case** is evaluated. The statements following the first expression whose value equals this value are executed, and then the **case** statement has finished. If none of the values of the expressions equal the value of the **case** expression, then the statements following **else** are executed. If no action is desired in the latter case, the construction may be abbreviated to the second form above.

Example H1E12

```

> x := 73;
> case Sign(x):
>   when 1:
>     x, "is positive";
>   when 0:
>     x, "is zero";
>   when -1:
>     x, "is negative";
> end case;
73 is positive

```

1.8.4 The Case Expression

```

case< expression |
  expressionleft,1 : expressionright,1 ,
  ⋮
  expressionleft,n : expressionright,n ,
  default : expressiondef >

```

This is the expression form of **case**. The *expression* is evaluated to the value v . Then each of the left-hand expressions $expression_{left,i}$ is evaluated until one is found whose value equals v ; if this happens the value of the corresponding right-hand expression $expression_{right,i}$ is returned. If no left-hand expression with value v is found the value of the default expression $expression_{def}$ is returned.

The default case cannot be omitted, and must come last.

1.9 Error Handling Statements

MAGMA has facilities for both reporting and handling errors. Errors can arise in a variety of circumstances within MAGMA's internal code (due to, for instance, incorrect usage of a function, or the unexpected failure of an algorithm). MAGMA allows the user to raise errors in their own code, as well as catch many kinds of errors.

1.9.1 The Error Objects

All errors in MAGMA are of type **Err**. Error objects not only include a description of the error, but also information relating to the location at which the error was raised, and whether the error was a user error, or a system error.

Error(*x*)

Constructs an error object with user information given by *x*, which can be of any type. The object *x* is stored in the **Object** attribute of the constructed error object, and the **Type** attribute of the object is set to "ErrUser". The remaining attributes are uninitialized until the error is raised by an **error** statement; at that point they are initialized with the appropriate positional information.

e.Position

Stores the position at which the error object *e* was raised. If the error object has not yet been raised, the attribute is undefined.

e.Traceback

Stores the stack traceback giving the position at which the error object *e* was raised. If the error object has not yet been raised, the attribute is undefined.

e.Object

Stores the user defined error information for the error. If the error is a system error, then this will be a string giving a textual description of the error.

e.Type

Stores the type of the error. Currently, there are only two types of errors in Magma: "Err" denotes a system error, and "ErrUser" denotes an error raised by the user.

1.9.2 Error Checking and Assertions

error *expression*, ..., *expression*;

Raises an error, with the error information being the printed value of the expressions. This statement is useful, for example, when an illegal value of an argument is passed to a function.

error if *Boolean expression*, *expression*, ..., *expression*;

If the given boolean expression evaluates to **true**, then raises an error, with the error information being the printed value of the expressions. This statement is designed for checking that certain conditions must be met, etc.

<code>assert Boolean expression;</code>

<code>assert2 Boolean expression;</code>
--

<code>assert3 Boolean expression;</code>
--

These assertion statements are useful to check that certain conditions are satisfied. There is an underlying **Assertions** flag, which is set to 1 by default.

For each statement, if the **Assertions** flag is less than the level specified by the statement (respectively 1, 2, 3 for the above statements), then nothing is done. Otherwise, the given boolean expression is evaluated and if the result is **false**, an error is raised, with the error information being an appropriate message.

It is recommended that when developing package code, **assert** is used for important tests (always to be tested in any mode), while **assert2** is used for more expensive tests, only to be checked in the debug mode, while **assert3** is used for extremely stringent tests which are very expensive.

Thus the **Assertions** flag can be set to 0 for no checking at all, 1 for normal checks, 2 for debug checks and 3 for extremely stringent checking.

1.9.3 Catching Errors

<pre>try statements₁ catch e statements₂ end try;</pre>

The **try/catch** statement lets users handle raised errors. The semantics of a **try/catch** statement are as follows: the block of statements *statements₁* is executed. If no error is raised during its execution, then the block of statements *statements₂* is not executed; if an error is raised at any point in *statements₁*, execution *immediately* transfers to *statements₂* (the remainder of *statements₁* is not executed). When transfer is controlled to the **catch** block, the variable named *e* is initialized to the error that was raised by *statements₁*; this variable remains in scope until the end of the **catch** block, and can be both read from and written to. The catch block can, if necessary, reraise *e*, or any other error object, using an **error** statement.

Example H1E13

The following example demonstrates the use of error objects, and **try/catch** statements.

```
> procedure always_fails(x)
>   error Error(x);
> end procedure;
>
> try
>   always_fails(1);
```



```

> always_fails(2); // we never get here
> catch e
>   print "In catch handler";
>   error "Error calling procedure with parameter: ", e'Object;
> end try;
In catch handler
Error calling procedure with parameter:  1

```

1.10 Iterative Statements

Three types of iterative statement are provided in MAGMA: the **for**-statement providing definite iteration and the **while**- and **repeat**-statements providing indefinite iteration.

Iteration may be performed over an arithmetic progression of integers or over any finite enumerated structure. Iterative statements may be nested. If nested iterations occur over the same enumerated structure, abbreviations such as **for** *x*, *y* **in** *X* **do** may be used; the leftmost identifier will correspond to the outermost loop, etc. (For nested iteration in sequence constructors, see Chapter 10.)

Early termination of the body of loop may be specified through use of the ‘jump’ commands **break** and **continue**.

1.10.1 Definite Iteration

```

for i := expression1 to expression2 by expression3 do
  statements
end for;

```

The expressions in this **for** loop must return integer values, say *b*, *e* and *s* (for ‘begin’, ‘end’ and ‘step’) respectively. The loop is ignored if either $s > 0$ and $b > e$, or $s < 0$ and $b < e$. If $s = 0$ an error occurs. In the remaining cases, the value $b + k \cdot s$ will be assigned to *i*, and the statements executed, for $k = 0, 1, 2, \dots$ in succession, as long as $b + k \cdot s \leq e$ (for $e > 0$) or $b + k \cdot s \geq e$ (for $e < 0$).

If the required step size is 1, the above may be abbreviated to:

```

for i := expression1 to expression2 do
  statements
end for;

```

```

for x in S do
  statements
end for;

```

Each of the elements of the finite enumerated structure *S* will be assigned to *x* in succession, and each time the statements will be executed. It is possible to nest several of these **for** loops compactly as follows.

```

for x11, ..., x1n1 in S1, ..., xm1, ..., xmnm in Sm do
  statements
end for;

```

1.10.2 Indefinite Iteration

```
while Boolean expression do
    statements
end while;
```

Check whether or not the Boolean expression has the value **true**; if it has, execute the statements. Repeat this until the expression assumes the value **false**, in which case statements following the **end while**; will be executed.

Example H1E14

The following short program implements a run of the famous $3x + 1$ problem on a random integer between 1 and 100.

```
> x := Random(1, 100);
> while x gt 1 do
> x;
>   if IsEven(x) then
>     x div:= 2;
>   else
>     x := 3*x+1;
>   end if;
> end while;
13
40
20
10
5
16
8
4
2
```

```
repeat
    statements
until Boolean expression;
```

Execute the statements, then check whether or not the Boolean expression has the value **true**. Repeat this until the expression assumes the value **false**, in which case the loop is exited, and statements following it will be executed.

Example H1E15

This example is similar to the previous one, except that it only prints x and the number of steps taken before x becomes 1. We use a **repeat** loop, and show that the use of a **break** statement sometimes makes it unnecessary that the Boolean expression following the **until** ever evaluates to **true**. Similarly, a **while true** statement may be used if the user makes sure the loop will be exited using **break**.

```
> x := Random(1, 1000);
> x;
172
> i := 0;
> repeat
>   while IsEven(x) do
>     i += 1;
>     x div:= 2;
>   end while;
>   if x eq 1 then
>     break;
>   end if;
>   x := 3*x+1;
>   i += 1;
> until false;
> i;
31
```

1.10.3 Early Exit from Iterative Statements

continue;

The **continue** statement can be used to jump to the end of the innermost enclosing loop: the termination condition for the loop is checked immediately.

continue identifier;

As in the case of **break**, this allows jumps out of nested **for** loops: the termination condition of the loop with loop variable *identifier* is checked immediately after **continue identifier** is encountered.

break;

A **break** inside a loop causes immediate exit from the innermost enclosing loop.

break identifier;

In nested **for** loops, this allows breaking out of several loops at once: this will cause an immediate exit from the loop with loop variable *identifier*.

Example H1E16

```

> p := 10037;
> for x in [1 .. 100] do
>   for y in [1 .. 100] do
>     if x^2 + y^2 eq p then
>       x, y;
>       break x;
>     end if;
>   end for;
> end for;
46 89

```

Note that `break` instead of `break x` would have broken only out of the inner loop; the output in that case would have been:

```

46 89
89 46

```

1.11 Runtime Evaluation: the eval Expression

Sometimes it is convenient to be able to evaluate expressions that are dynamically constructed at runtime. For instance, consider the problem of implementing a database of mathematical objects in MAGMA. Suppose that these mathematical objects are very large, but can be constructed in only a few lines of MAGMA code (a good example of this would be MAGMA's database of best known linear codes). It would be very inefficient to store these objects in a file for later retrieval; a better solution would be to instead store a string giving the code necessary to construct each object. MAGMA's `eval` feature can then be used to dynamically parse and execute this code on demand.

`eval` *expression*

The `eval` expression works as follows: first, it evaluates the given *expression*, which must evaluate to a string. This string is then treated as a piece of MAGMA code which yields a result (that is, the code must be an expression, not a statement), and this result becomes the result of the `eval` expression.

The string that is evaluated can be of two forms: it can be a MAGMA expression, e.g., “1+2”, “Random(x)”, or it can be a sequence of MAGMA statements. In the first case, the string does not have to be terminated with a semicolon, and the result of the expression given in the string will be the result of the `eval` expression. In the second case, the last statement given in the string should be a `return` statement; it is easiest to think of this case as defining the body of a function.

The string that is used in the `eval` expression can refer to any variable that is in scope during the evaluation of the `eval` expression. However, it is not possible for the expression to *modify* any of these variables.

Example H1E17

In this example we demonstrate the basic usage of the `eval` keyword.

```
> x := eval "1+1"; // OK
> x;
2
> eval "1+1;"; // not OK
2
>> eval "1+1;"; // not OK
^
Runtime error: eval must return a value
> eval "return 1+1;"; // OK
2
> eval "x + 1"; // OK
3
> eval "x := x + 1; return x";
>> eval "x := x + 1; return x";
^
In eval expression, line 1, column 1:
>> x := x + 1; return x;
^
    Located in:
    >> eval "x := x + 1; return x";
    ^
```

User error: Imported environment value 'x' cannot be used as a local

Example H1E18

In this example we demonstrate how `eval` can be used to construct MAGMA objects specified with code only available at runtime.

```
> M := Random(MatrixRing(GF(2), 5));
> M;
[1 1 1 1 1]
[0 0 1 0 1]
[0 0 1 0 1]
[1 0 1 1 1]
[1 1 0 1 1]
> Write("/tmp/test", M, "Magma");
> s := Read("/tmp/test");
> s;
MatrixAlgebra(GF(2), 5) ! [ GF(2) | 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1,
1, 0, 1, 1, 1, 1, 1, 0, 1, 1 ]
> M2 := eval s;
> assert M eq M2;
```

1.12 Comments and Continuation

//

One-line comment: any text following the double slash on the same line will be ignored by MAGMA.

/* */

Multi-line comment: any text between `/*` and `*/` is ignored by MAGMA.

\

Line continuation character: this symbol and the `<return>` immediately following is ignored by MAGMA. Evaluation will continue on the next line without interruption. This is useful for long input lines.

Example H1E19

```
> // The following produces an error:
> x := 12
> 34;
User error: bad syntax
> /* but this is correct
>    and reads two lines: */
> x := 12\
> 34;
> x;
1234
```

1.13 Timing

Cputime()

Return the CPU time (as a real number of default precision) used since the beginning of the MAGMA session. Note that for the MSDOS version, this is the real time used since the beginning of the session (necessarily, since process CPU time is not available).

Cputime(t)

Return the CPU time (as a real number of default precision) used since time t . Time starts at 0.0 at the beginning of a MAGMA session.

Realtime()

Return the absolute real time (as a real number of default precision), which is the number of seconds since 00:00:00 GMT, January 1, 1970. For the MSDOS version, this is the real time used since the beginning of the session.

Realtime(*t*)

Return the real time (as a real number of default precision) elapsed since time *t*.

ClockCycles()

Return the number of clock cycles of the CPU since Magma's startup. Note that this matches the real time (i.e., not process user/system time). If the operation is not supported on the current processor, zero is returned.

time *statement*;

Execute the statement and print the time taken when the statement is completed.

vtime *flag*: *statement*;

vtime *flag*, *n*: *statement*:

If the verbose flag *flag* (see the function **SetVerbose**) has a level greater than or equal to *n*, execute the statement and print the time taken when the statement is completed. If the flag has level 0 (i.e., is not turned on), still execute the statement, but do not print the timing. In the first form of this statement, where a specific level is not given, *n* is taken to be 1. This statement is useful in MAGMA code found in packages where one wants to print the timing of some sub-algorithm if and only if an appropriate verbose flag is turned on.

Example H1E20

The **time** command can be used to time a single statement.

```
> n := 2^109-1;
> time Factorization(n);
[<745988807, 1>, <870035986098720987332873, 1>]
Time: 0.149
```

Alternatively, we can extract the current time *t* and use **Cputime**. This method can be used to time the execution of several statements.

```
> m := 2^111-1;
> n := 2^113-1;
> t := Cputime();
> Factorization(m);
[<7, 1>, <223, 1>, <321679, 1>, <26295457, 1>, <319020217, 1>, <616318177, 1>]
> Factorization(n);
[<3391, 1>, <23279, 1>, <65993, 1>, <1868569, 1>, <1066818132868207, 1>]
> Cputime(t);
0.121
```

We illustrate a simple use of **vtime** with **vprint** within a function.

```
> function MyFunc(G)
>   vprint User1: "Computing order...";
>   vtime User1: o := #G;
```

```

> return o;
> end function;
> SetVerbose("User1", 0);
> MyFunc(Sym(4));
24
> SetVerbose("User1", 1);
> MyFunc(Sym(4));
Computing order...
Time: 0.000
24

```

1.14 Types, Category Names, and Structures

The following functions deal with *types* or *category names* and general structures. MAGMA has two levels of granularity when referring to types. In most cases, the coarser grained types (of type `Cat`) are used. Examples of these kinds of types are “polynomial rings” (`RngUPol`) and “finite fields” (`FldFin`). However, sometimes more specific typing information is sometimes useful. For instance, the algorithm used to factorize polynomials differs significantly, depending on the coefficient ring. Hence, we might wish to implement a specialized factorization algorithm polynomials over some particular ring type. Due to this need, MAGMA also supports *extended types*.

An extended type (of type `ECat`) can be thought of as a type taking a parameter. Using extended types, we can talk about “polynomial rings over the integers” (`RngUPol[RngInt]`), or “maps from the integers to the rationals” (`Map[RngInt, FldRat]`). Extended types can interact with normal types in all ways, and thus generally only need to be used when the extra level of information is required.

Type(x)

Category(x)

Given any object x , return the type (or category name) of x .

ExtendedType(x)

ExtendedCategory(x)

Given any object x , return the extended type (or category name) of x .

ISA(T, U)

Given types (or extended types) T and U , return whether T ISA U , i.e., whether objects of type T inherit properties of type U . For example, `ISA(RngInt, Rng)` is true, because the ring of integers \mathbf{Z} is a ring.

MakeType(S)

Given a string S specifying a type return the actual type corresponding to S . This is useful when some intrinsic name hides the symbol which normally refers to the actual type.

ElementType(S)

Given any structure S , return the type of the elements of S . For example, the element type of the ring of integers \mathbf{Z} is `RngIntElt` since that is the type of the integers which lie in \mathbf{Z} .

CoveringStructure(S, T)

Given structures S and T , return a covering structure C for S and T , so that S and T both embed into C . An error results if no such covering structure exists.

ExistsCoveringStructure(S, T)

Given structures S and T , return whether a covering structure C for S and T exists, and if so, return such a C , so that S and T both embed into C .

Example H1E21

We demonstrate the type and structure functions.

```
> Type(3);
RngIntElt
> t := MakeType("RngIntElt");
> t;
RngIntElt
> Type(3) eq t;
true
> Z := IntegerRing();
> Type(Z);
RngInt
> ElementType(Z);
RngIntElt
> ISA(RngIntElt, RngElt);
true
> ISA(RngIntElt, GrpElt);
false
> ISA(FldRat, Fld);
true
```

The following give examples of when covering structures exist or do not exist.

```
> Q := RationalField();
> CoveringStructure(Z, Q);
Rational Field
> ExistsCoveringStructure(Z, DihedralGroup(3));
false
```

```

> ExistsCoveringStructure(Z, CyclotomicField(5));
true Cyclotomic Field of order 5 and degree 4
> ExistsCoveringStructure(CyclotomicField(3), CyclotomicField(5));
true Cyclotomic Field of order 15 and degree 8
> ExistsCoveringStructure(GF(2), GF(3));
false
> ExistsCoveringStructure(GF(2^6), GF(2, 15));
true Finite field of size 2^30

```

Our last example demonstrates the use of extended types:

```

> R<x> := PolynomialRing(Integers());
> ExtendedType(R);
RngUPol[RngInt]
> ISA(RngUPol[RngInt], RngUPol);
true
> f := x + 1;
> ExtendedType(f);
RngUPolElt[RngInt]
> ISA(RngUPolElt[RngInt], RngUPolElt);
true

```

1.15 Random Object Generation

Pseudo-random quantities are used in several MAGMA algorithms, and may also be generated explicitly by some intrinsics. Throughout the Handbook, the word ‘random’ is used for ‘pseudo-random’.

Since V2.7 (June 2000), MAGMA contains an implementation of the *Monster* random number generator of G. Marsaglia [Mar00]. The period of this generator is $2^{29430} - 2^{27382}$ (approximately 10^{8859}), and passes all of the stringent tests in Marsaglia’s *Diehard* test suite [Mar95]. Since V2.13 (July 2006), this generator is combined with the MD5 hash function to produce a higher-quality result.

Because the generator uses an internal array of machine integers, one ‘seed’ variable does not express the whole state, so the method for setting or getting the generator state is by way of a pair of values: (1) the seed for initializing the array, and (2) the number of steps performed since the initialization.

SetSeed(s, c)

SetSeed(s)

(Procedure.) Reset the random number generator to have initial seed s ($0 \leq s < 2^{32}$), and advance to step c ($0 \leq c < 2^{64}$). If c is not given, it is taken to be 0. Passing $-Sn$ to MAGMA at startup is equivalent to typing `SetSeed(n)`; after startup.

GetSeed()

Return the initial seed s used to initialize the random-number generator and also the current step c . This is the complement to the **SetSeed** function.

Random(S)

Given a finite set or structure S , return a random element of S .

Random(a , b)

Return a random integer lying in the interval $[a, b]$, where $a \leq b$.

Random(b)

Return a random integer lying in the interval $[0, b]$, where b is a non-negative integer. Because of the good properties of the underlying Monster generator, calling **Random(1)** is a good safe way to produce a sequence of random bits.

Example H1E22

We demonstrate how one can return to a previous random state by the use of **GetSeed** and **SetSeed**. We begin with initial seed 1 at step 0 and create a multi-set of 100,000 random integers in the range $[1..4]$.

```
> SetSeed(1);
> GetSeed();
1 0
> time S := { * Random(1, 4): i in [1..100000] * };
Time: 0.490
> S;
{ * 1^^24911, 2^^24893, 3^^25139, 4^^25057 * }
```

We note the current state by **GetSeed**, and then print 10 random integers in the range $[1..100]$.

```
> GetSeed();
1 100000
> [Random(1, 100): i in [1 .. 10]];
[ 85, 41, 43, 69, 66, 61, 63, 31, 84, 11 ]
> GetSeed();
1 100014
```

We now restart with a different initial seed 23 (again at step 0), and do the same as before, noting the different random integers produced.

```
> SetSeed(23);
> GetSeed();
23 0
> time S := { * Random(1, 4): i in [1..100000] * };
Time: 0.500
> S;
{ * 1^^24962, 2^^24923, 3^^24948, 4^^25167 * }
> GetSeed();
```

```

23 100000
> [Random(1, 100): i in [1 .. 10]];
[ 3, 93, 11, 62, 6, 73, 46, 52, 100, 30 ]
> GetSeed();
23 100013

```

Finally, we restore the random generator state to what it was after the creation of the multi-set for the first seed. We then print the 10 random integers in the range [1..100], and note that they are the same as before.

```

> SetSeed(1, 100000);
> [Random(1, 100): i in [1 .. 10]];
[ 85, 41, 43, 69, 66, 61, 63, 31, 84, 11 ]
> GetSeed();
1 100014

```

1.16 Miscellaneous

IsIntrinsic(S)

Given a string S , return `true` if and only an intrinsic with the name S exists in the current version of MAGMA. If the result is `true`, return also the actual intrinsic.

Example H1E23

We demonstrate the function `IsIntrinsic`.

```

> IsIntrinsic("ABCD");
false
> l, a := IsIntrinsic("Abs");
> l;
true
> a(-3);
3

```

1.17 Bibliography

[Mar95] G. Marsaglia. DIEHARD: a battery of tests of randomness.

URL:<http://stat.fsu.edu/pub/diehard/>, 1995.

[Mar00] G. Marsaglia. The Monster, a random number generator with period 10^{2857} times as long as the previously touted longest-period one. Preprint, 2000.

2 FUNCTIONS, PROCEDURES AND PACKAGES

2.1 Introduction	35	2.3.9 User Startup Specification Files	50
2.2 Functions and Procedures	35	2.4 Attributes	51
2.2.1 Functions	35	2.4.1 Predefined System Attributes	51
f := func< x ₁ , ..., x _n : - e>;	36	2.4.2 User-defined Attributes	52
f := func< x ₁ , ..., x _n , ...: - e>;	36	AddAttribute(C, F)	52
2.2.2 Procedures	39	declare attributes C: F ₁ , ..., F _n ;	52
p := proc< x ₁ , ..., x _n : - e>;	40	2.4.3 Accessing Attributes	52
p := proc< x ₁ , ..., x _n , ...: - e>;	40	S'fieldname	52
2.2.3 The forward Declaration	41	S'N	52
forward	41	assigned	52
2.3 Packages	42	assigned	52
2.3.1 Introduction	42	S'fieldname := e;	53
2.3.2 Intrinsic	43	S'N := e;	53
intrinsic	43	delete S'fieldname;	53
2.3.3 Resolving Calls to Intrinsic	45	delete S'N;	53
2.3.4 Attaching and Detaching Package Files	46	GetAttributes(C)	53
Attach(F)	47	ListAttributes(C)	53
Detach(F)	47	2.5 User-defined Verbose Flags	53
freeze;	47	declare verbose F, m;	53
2.3.5 Related Files	47	2.5.1 Examples	53
2.3.6 Importing Constants	47	2.6 User-Defined Types	56
import "filename": ident_list;	47	2.6.1 Declaring User-Defined Types	56
2.3.7 Argument Checking	48	declare type T;	56
require condition: print_args;	48	declare type T: P ₁ , ..., P _n ;	56
require range v, L, U;	48	declare type T[E];	56
require v, L;	48	declare type T[E]: P ₁ , ..., P _n ;	56
2.3.8 Package Specification Files	49	2.6.2 Creating an Object	57
AttachSpec(S)	49	New(T)	57
DetachSpec(S)	49	2.6.3 Special Intrinsic Provided by the User	57
		2.6.4 Examples	58

Chapter 2

FUNCTIONS, PROCEDURES AND PACKAGES

2.1 Introduction

Functions are one of the most fundamental elements of the MAGMA language. The first section describes the various ways in which a standard function may be defined while the second section describes the definition of a procedure (i.e. a function which doesn't return a value). The second half of the chapter is concerned with user-defined *intrinsic* functions and procedures.

2.2 Functions and Procedures

There are two slightly different syntactic forms provided for the definition of a user function (as opposed to an intrinsic function). For the case of a function whose definition can be expressed as a single expression, an abbreviated form is provided. The syntax for the definition of user procedures is similar. Names for functions and procedures are ordinary identifiers and so obey the rules as given in Chapter 1 for other variables.

2.2.1 Functions

```
f := function(x1, ..., xn: parameters)
    statements
end function;
```

```
function f(x1, ..., xn: parameters)
    statements
end function;
```

This creates a function taking $n \geq 0$ arguments, and assigns it to f . The statements may comprise any number of valid MAGMA statements, but at least one of them must be of the form **return expression**;. The value of that expression (possibly dependent on the values of the arguments x_1, \dots, x_n) will be the return value for the function; failure to return a value will lead to a run-time error when the function is invoked. (In fact, a return statement is also required for every additional 'branch' of the function that has been created using an **if ... then ... else ...** construction.)

The function may return multiple values. Usually one uses the form **return expression, ..., expression**;. If one wishes to make the last return value(s) undefined (so that the number of return values for the function is the same in all 'branches' of

the function) the underscore symbol ($_$) may be used. (The undefined symbol may only be used for final values of the list.) This construct allows behaviour similar to the intrinsic function `IsSquare`, say, which returns `true` and the square root of its argument if that exists, and `false` and the undefined value otherwise. See also the example below.

If there are parameters given, they must consist of a comma-separated list of clauses each of the form `identifier := value`. The identifier gives the name of the parameter, which can then be treated as a normal value argument within the statements. The value gives a default value for the parameter, and may depend on any of the arguments or preceding parameters; if, when the function is called, the parameter is not assigned a value, this default value will be assigned to the parameter. Thus parameters are always initialized. If no parameters are desired, the colon following the last argument, together with *parameters*, may be omitted.

The only difference between the two forms of function declaration lies in recursion. Functions may invoke themselves recursively since their name is part of the syntax; if the first of the above declarations is used, the identifier f cannot be used inside the definition of f (and `$$` will have to be used to refer to f itself instead), while the second form makes it possible to refer to f within its definition.

An invocation of the user function f takes the form `f(m1, ..., mn)`, where m_1, \dots, m_n are the actual arguments.

```
f := function(x1, ..., xn, ...: parameters)
  statements
end function;
```

```
function f(x1, ..., xn, ...: parameters)
  statements
end function;
```

This creates a *variadic* function, which can take n or more arguments. The semantics are identical to the standard function definition described above, with the exception of function invocation. An invocation of a variadic function f takes the form `f(y1, ..., ym)`, where y_1, \dots, y_m are the arguments to the function, and $m \geq n$. These arguments get bound to the parameters as follows: for $i < n$, the argument y_i is bound to the parameter x_i . For $i \geq n$, the arguments y_i are bound to the last parameter x_n as a list `[*yn, ..., ym]`.

```
f := func< x1, ..., xn: parameters | expression>;
```

This is a short form of the function constructor designed for the situation in which the value of the function can be defined by a single expression. A function f is created which returns the value of the expression (possibly involving the function arguments x_1, \dots, x_n). Optional parameters are permitted as in the standard function constructor.

```
f := func< x1, ..., xn, ...: parameters | expression>;
```

This is a short form of the function constructor for *variadic functions*, otherwise identical to the short form describe above.

Example H2E1

This example illustrates recursive functions.

```
> fibonacci := function(n)
>   if n le 2 then
>     return 1;
>   else
>     return $(n-1) + $(n-2);
>   end if;
> end function;
>
> fibonacci(10)+fibonacci(12);
199

> function Lucas(n)
>   if n eq 1 then
>     return 1;
>   elif n eq 2 then
>     return 3;
>   else
>     return Lucas(n-1)+Lucas(n-2);
>   end if;
> end function;
>
> Lucas(11);
199

> fibo := func< n | n le 2 select 1 else $(n-1) + $(n-2) >;
> fibo(10)+fibo(12);
199
```

Example H2E2

This example illustrates the use of parameters.

```
> f := function(x, y: Proof := true, A1 := "Simple")
>   return <x, y, Proof, A1>;
> end function;
>
> f(1, 2);
<1, 2, true, Simple>
> f(1, 2: Proof := false);
<1, 2, false, Simple>
> f(1, 2: A1 := "abc", Proof := false);
<1, 2, false, abc>
```

Example H2E3

This example illustrates the returning of undefined values.

```
> f := function(x)
>   if IsOdd(x) then
>     return true, x;
>   else
>     return false, _;
>   end if;
> end function;
>
> f(1);
true 1
> f(2);
false
> a, b := f(1);
> a;
true
> b;
1
> a, b := f(2);
> a;
false
> // The following produces an error:
> b;
>> b;
^
```

User error: Identifier 'b' has not been assigned

Example H2E4

This example illustrates the use of variadic functions.

```
> f := function(x, y, ...)
>   print "x: ", x;
>   print "y: ", y;
>   return [x + z : z in y];
> end function;
>
> f(1, 2);
x: 1
y: [* 2*]
[ 3 ]
> f(1, 2, 3);
x: 1
y: [* 2, 3*]
[ 3, 4 ]
> f(1, 2, 3, 4);
```

```
x: 1
y: [* 2, 3, 4*]
   [ 3, 4, 5 ]
```

2.2.2 Procedures

```
p := procedure(x1, ..., xn: parameters)
    statements
end procedure;
```

```
procedure p(x1, ..., xn: parameters)
    statements
end procedure;
```

The procedure, taking $n \geq 0$ arguments and defined by the statements is created and assigned to p . Each of the arguments may be either a variable (y_i) or a referenced variable ($\sim y_i$). Inside the procedure only referenced variables (and local variables) may be (re-)assigned to. The procedure p is invoked by typing $p(x_1, \dots, x_n)$, where the same succession of variables and referenced variables is used (see the example below). Procedures cannot return values.

If there are parameters given, they must consist of a comma-separated list of clauses each of the form **identifier** := **value**. The identifier gives the name of the parameter, which can then be treated as a normal value argument within the statements. The value gives a default value for the parameter, and may depend on any of the arguments or preceding parameters; if, when the function is called, the parameter is not assigned a value, this default value will be assigned to the parameter. Thus parameters are always initialized. If no parameters are desired, the colon following the last argument, together with *parameters*, may be omitted.

As in the case of **function**, the only difference between the two declarations lies in the fact that the second version allows recursive calls to the procedure within itself using the identifier (p in this case).

```
p := procedure(x1, ..., xn, ...: parameters)
    statements
end procedure;
```

```
procedure p(x1, ..., xn, ...: parameters)
    statements
end procedure;
```

Creates and assigns a new *variadic* procedure to p . The use of a variadic procedure is identical to that of a variadic function, described previously.

$$p := \text{proc} \langle x_1, \dots, x_n: \text{parameters} \mid \text{expression} \rangle;$$

This is a short form of the procedure constructor designed for the situation in which the action of the procedure may be accomplished by a single statement. A procedure p is defined which calls the procedure given by the expression. This expression must be a simple procedure call (possibly involving the procedure arguments x_1, \dots, x_n). Optional parameters are permitted as in the main procedure constructor.

$$p := \text{proc} \langle x_1, \dots, x_n, \dots: \text{parameters} \mid \text{expression} \rangle;$$

This is a short form of the procedure constructor for variadic procedures.

Example H2E5

By way of simple example, the following (rather silly) procedure assigns a Boolean to the variable `holds`, according to whether or not the first three arguments x, y, z satisfy $x^2 + y^2 = z^2$. Note that the fourth argument is referenced, and hence can be assigned to; the first three arguments cannot be changed inside the procedure.

```
> procedure CheckPythagoras(x, y, z, ~h)
>   if x^2+y^2 eq z^2 then
>     h := true;
>   else
>     h := false;
>   end if;
> end procedure;
```

We use this to find some Pythagorean triples (in a particularly inefficient way):

```
> for x, y, z in { 1..15 } do
>   CheckPythagoras(x, y, z, ~h);
>   if h then
>     "Yes, Pythagorean triple!", x, y, z;
>   end if;
> end for;
Yes, Pythagorean triple! 3 4 5
Yes, Pythagorean triple! 4 3 5
Yes, Pythagorean triple! 5 12 13
Yes, Pythagorean triple! 6 8 10
Yes, Pythagorean triple! 8 6 10
Yes, Pythagorean triple! 9 12 15
Yes, Pythagorean triple! 12 5 13
Yes, Pythagorean triple! 12 9 15
```

2.2.3 The forward Declaration

forward f;

The forward declaration of a function or procedure f ; although the assignment of a value to f is deferred, f may be called from within another function or procedure already.

The **forward** statement must occur on the ‘main’ level, that is, outside other functions or procedures. (See also Chapter 5.)

Example H2E6

We give an example of mutual recursion using the **forward** declaration. In this example we define a primality testing function which uses the factorization of $n - 1$, where n is the number to be tested. To obtain the complete factorization we need to test whether or not factors found are prime. Thus the prime divisor function and the primality tester call each other.

First we define a simple function that proves primality of n by finding an integer of multiplicative order $n - 1$ modulo n .

```
> function strongTest(primdiv, n)
>   return 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 }
>   };
> end function;
```

Next we define a rather crude **isPrime** function: for odd $n > 3$ it first checks for a few (3) random values of a that $a^{n-1} \equiv 1 \pmod n$, and if so, it applies the above primality prover. For that we need the not yet defined function for finding the prime divisors of an integer.

```
> forward primeDivisors;
> function isPrime(n)
>   if n in { 2, 3 } or
>     IsOdd(n) and
>     forall{ a : a in { Random(2, n-2): i in [1..3] } |
>       Modexp(a, n-1, n) eq 1 } and
>       strongTest( primeDivisors(n-1), n )
>   then
>     return true;
>   else
>     return false;
>   end if;
> end function;
```

Finally, we define a function that finds the prime divisors. Note that it calls the **isPrime** function. Note also that this function is recursive, and that it calls a function upon its definition, in the form `func< ..> (..)`.

```
> primeDivisors := function(n)
>   if isPrime(n) then
>     return { n };
```

```
> else
>     return func< d | primeDivisors(d) join primeDivisors(n div d) >
>         ( rep{ d : d in [2..Isqrt(n)] | n mod d eq 0 } );
> end if;
> end function;
> isPrime(1087);
true;
```

2.3 Packages

2.3.1 Introduction

For brevity, in this section we shall use the term *function* to include both functions and procedures.

The term *intrinsic function* or *intrinsic* refers to a function whose signature is stored in the system table of signatures. In terms of their origin, there are two kinds of intrinsics, *system intrinsics* (or *standard functions*) and *user intrinsics*, but they are indistinguishable in their use. A *system intrinsic* is an intrinsic that is part of the definition of the MAGMA system, whereas a user intrinsic is an informal addition to MAGMA, created by a user of the system. While most of the standard functions in MAGMA are implemented in C, a growing number are implemented in the MAGMA language. User intrinsics are defined in the MAGMA language using a *package* mechanism (the same syntax, in fact, as that used by developers to write standard functions in the MAGMA language).

This section explains the construction of user intrinsics by means of packages. From now on, *intrinsic* will be used as an abbreviation for *user intrinsic*.

It is useful to summarize the properties possessed by an intrinsic function that are not possessed by an ordinary user-defined function. Firstly, the signature of every intrinsic function is stored in the system's table of signatures. In particular, such functions will appear when signatures are listed and printing the function's name will produce a summary of the behaviour of the function. Secondly, intrinsic functions are compiled into the MAGMA internal pseudo-code. Thus, once an intrinsic function has been debugged, it does not have to be compiled every time it is needed. If the definition of the function involves a large body of code, this can save a significant amount of time when the function definition has to be loaded.

An intrinsic function is defined in a special type of file known as a *package*. In general terms a package is a MAGMA source file that defines constants, one or more intrinsic functions, and optionally, some ordinary functions. The definition of an intrinsic function may involve MAGMA standard functions, functions imported from other packages and functions whose definition is part of the package. It should be noted that constants and functions (other than intrinsic functions) defined in a package will not be visible outside the package, unless they are explicitly imported.

The syntax for the definition of an intrinsic function is similar to that of an ordinary function except that the function header must define the function's signature together with

text summarizing the semantics of the function. As noted above, an intrinsic function definition must reside in a package file. It is necessary for MAGMA to know the location of all necessary package files. A package may be attached or detached through use of the **Attach** or **Detach** procedures. More generally, a family of packages residing in a directory tree may be specified through provision of a **spec** file which specifies the locations of a collection of packages relative to the position of the spec file. Automatic attaching of the packages in a spec file may be set by means of an environment variable (**MAGMA.SYSTEM.SPEC** for the MAGMA system packages and **MAGMA.USER.SPEC** for a users personal packages).

So that the user does not have to worry about explicitly compiling packages, MAGMA has an auto-compile facility that will automatically recompile and reload any package that has been modified since the last compilation. It does this by comparing the time stamp on the source file (as specified in an **Attach** procedure call or spec file) with the time stamp on the compiled code. To avoid the possible inefficiency caused by MAGMA checking whether the file is up to date every time an intrinsic function is referenced, the user can indicate that the package is stable by including the **freeze;** directive at the top of the package containing the function definition.

A constant value or function defined in the body of a package may be accessed in a context outside of its package through use of the **import** statement. The arguments for an intrinsic function may be checked through use of the **require** statement and its variants. These statements have the effect of generating an error message at the level of the caller rather than in the called intrinsic function.

See also the section on user-defined attributes for the **declare attributes** directive to declare user-defined attributes used by the package and related packages.

2.3.2 Intrinsics

Besides the definition of *constants* at the top, a package file just consists of *intrinsics*. There is only one way a intrinsic can be referred to (whether from within or without the package). When a package is *attached*, its intrinsics are incorporated into MAGMA. Thus intrinsics are ‘global’ — they affect the global MAGMA state and there is only one set of MAGMA intrinsics at any time. There are no ‘local’ intrinsics.

A package may contain undefined references to identifiers. These are presumed to be intrinsics from other packages which will be attached subsequent to the loading of this package.

```
intrinsic name(arg-list [, ...]) [ -> ret-list ]
{comment-text}
    statements
end intrinsic;
```

The syntax of a intrinsic declaration is as above, where *name* is the name of the intrinsic (any identifier; use single quotes for non-alphanumeric names like '+'); *arg-list* is the argument list (optionally including parameters preceded by a colon); optionally there is an arrow and return type list *ret-list*; the comment text is any text within the braces (use `\}` to get a right brace within the text, and use `"` to repeat the comment from the immediately preceding intrinsic); and *statements* is a list of

statements making up the body. *arg-list* is a list of comma-separated arguments of the form

```

name :: type
~name :: type
~name

```

where *name* is the name of the argument (any identifier), and *type* designates the type, which can be either a simple category name, an extended type, or one of the following:

.	Any type
[]	Sequence type
{ }	Set type
{ [] }	Set or Sequence type
{ @ @ }	Iset type
{ * * }	Multiset type
< >	Tuple type

or a *composite type*:

[type]	Sequences over <i>type</i>
{type}	Sets over <i>type</i>
{ [type] }	Sets or sequences over <i>type</i>
{ @type@ }	Indexed sets over <i>type</i>
{ *type* }	Multisets over <i>type</i>

where *type* is either a simple or extended type. The reference form *type* *~name* requires that the input argument must be initialized to an object of that type. The reference form *~name* is a plain reference argument — it need not be initialized. Parameters may also be specified—these are just as in functions and procedures (preceded by a colon). If *arg-list* is followed by “...” then the intrinsic is **variadic**, with semantics similar to that of a variadic function, described previously.

ret-list is a list of comma-separated simple types. If there is an arrow and the return list, the intrinsic is assumed to be functional; otherwise it is assumed to be procedural.

The body of *statements* should return the correct number and types of arguments if the intrinsic is functional, while the body should return nothing if the intrinsic is procedural.

Example H2E7

A functional intrinsic for greatest common divisors taking two integers and returning another:

```

intrinsic myGCD(x::RngIntElt, y::RngIntElt) -> RngIntElt
{ Return the GCD of x and y }
return ...;

```



```
end intrinsic;
```

A procedural intrinsic for Append taking a reference to a sequence Q and any object then modifying Q :

```
intrinsic Append(~ Q::SeqEnum, . x)
{ Append x to Q }
...;
end intrinsic;
```

A functional intrinsic taking a sequence of sets as arguments 2 and 3:

```
intrinsic IsConjugate(G::GrpPerm, R::[ { } ], S::[ { } ]) -> BoolElt
{ True iff partitions R and S of the support of G are conjugate in G }
  return ...;
end intrinsic;
```

2.3.3 Resolving Calls to Intrinsics

It is often the case that many intrinsics share the same name. For instance, the intrinsic **Factorization** has many implementations for various object types. We will call such intrinsics *overloaded intrinsics*, or refer to each of the participating intrinsics as an *overload*. When the user calls such an overloaded intrinsic, MAGMA must choose the “best possible” overload.

MAGMA’s overload resolution process is quite simple. Suppose the user is calling an intrinsic of arity r , with a list of parameters $\langle p_1, \dots, p_r \rangle$. Let the tuple of the types of these parameters be $\langle t_1, \dots, t_r \rangle$, and let S be the set of all relevant overloads (that is, overloads with the appropriate name and of arity r). We will represent overloads as r -tuples of types.

To pick the “best possible” overload, for each parameter $p \in \{p_1, \dots, p_r\}$, MAGMA finds the set $S_i \subseteq S$ of participating intrinsics which are the best matches for that parameter. More specifically, an intrinsic $s = \langle u_1, \dots, u_r \rangle$ is included in S_i if and only if t_i is a u_i , and no participating intrinsic $s' = \langle v_1, \dots, v_r \rangle$ exists such that t_i is a v_i and v_i is a u_i . Once the sets S_i are computed, MAGMA finds their intersection. If this intersection is empty, then there is no match. If this intersection has cardinality greater than one, then the match is ambiguous. Otherwise, MAGMA calls the overload thus obtained.

An example at this point will make the above process clearer:

Example H2E8

We demonstrate MAGMA’s lookup mechanism with the following example. Suppose we have the following overloaded intrinsics:

```
intrinsic overloaded(x::RngUPolElt, y::RngUPolElt) -> RngIntElt
{ Overload 1 }
  return 1;
end intrinsic;

intrinsic overloaded(x::RngUPolElt[RngInt], y::RngUPolElt) -> RngIntElt
```

```

{ Overload 2 }
  return 2;
end intrinsic;

intrinsic overloaded(x::RngUPolElt, y::RngUPolElt[RngInt]) -> RngIntElt
{ Overload 3 }
  return 3;
end intrinsic;

intrinsic overloaded(x::RngUPolElt[RngInt], y::RngUPolElt[RngInt]) -> RngIntElt
{ Overload 4 }
  return 4;
end intrinsic;

```

The following MAGMA session illustrates how the lookup mechanism operates for the intrinsic `overloaded`:

```

> R1<x> := PolynomialRing(Integers());
> R2<y> := PolynomialRing(Rationals());
> f1 := x + 1;
> f2 := y + 1;
> overloaded(f2, f2);
1
> overloaded(f1, f2);
2
> overloaded(f2, f1);
3
> overloaded(f1, f1);
4

```

2.3.4 Attaching and Detaching Package Files

The procedures `Attach` and `Detach` are provided to attach or detach package files. Once a file is attached, all intrinsics within it are included in MAGMA. If the file is modified, it is automatically recompiled just after the user hits return and just before the next statement is executed. So there is no need to re-attach the file (or ‘re-load’ it). If the recompilation of a package file fails (syntax errors, etc.), all of the intrinsics of the package file are removed from the MAGMA session and none of the intrinsics of the package file are included again until the package file is successfully recompiled. When errors occur during compilation of a package, the appropriate messages are printed with the string ‘[PC]’ at the beginning of the line, indicating that the errors are detected by the MAGMA package compiler.

If a package file contains the single directive `freeze`; at the top then the package file becomes **frozen** — it will not be automatically recompiled after each statement is entered into MAGMA. A frozen package is recompiled if need be, however, when it is attached (thus allowing fixes to be updated) — the main point of freezing a package which is ‘stable’ is to stop MAGMA looking at it between every statement entered into MAGMA interactively.

When a package file is complete and tested, it is usually installed in a spec file so it is automatically attached when the spec file is attached. Thus **Attach** and **Detach** are generally only used when one is developing a single package file containing new intrinsics.

Attach(F)

Procedure to attach the package file *F*.

Detach(F)

Procedure to detach the package file *F*.

freeze;

Freeze the package file in which this appears at the top.

2.3.5 Related Files

There are two files related to any package source file **file.m**:

file.sig	sig file containing signature information;
file.lck	lock file.

The lock file exists while a package file is being compiled. If someone else tries to compile the file, it will just sit there till the lock file disappears. In various circumstances (system down, MAGMA crash) **.lck** files may be left around; this will mean that the next time MAGMA attempts to compile the associated source file it will just sit there indefinitely waiting for the **.lck** file to disappear. In this case the user should search for **.lck** files that should be removed.

2.3.6 Importing Constants

import "filename": ident_list;

This is the general form of the import statement, where **"filename"** is a string and **ident_list** is a list of identifiers.

The import statement is a normal statement and can in fact be used anywhere in MAGMA, but it is recommended that it only be used to import common constants and functions/procedures shared between a collection of package files. It has the following semantics: for each identifier *I* in the list **ident_list**, that identifier is declared just like a normal identifier within MAGMA. Within the package file referenced by **filename**, there should be an assignment of the same identifier *I* to some object *O*. When the identifier *I* is then used as an expression after the import statement, the value yielded is the object *O*.

The file that is named in the import statement must already have been attached by the time the identifiers are needed. The best way to achieve this in practice is to place this file in the spec file, along with the package files, so that all the files can be attached together.

Thus the only way objects (whether they be normal objects, procedures or functions) assigned within packages can be referenced from outside the package is by an explicit import with the 'import' statement.

Example H2E9

Suppose we have a spec file that lists several package files. Included in the spec file is the file `defs.m` containing:

```
MY_LIMIT := 10000;
function fred(x)
  return 1/x;
end function;
```

Then other package files (in the same directory) listed in the spec file which wish to use these definitions would have the line

```
import "defs.m": MY_LIMIT, fred;
```

at the top. These could then be used inside any intrinsics of such package files. (If the package files are not in the same directory, the pathname of `defs.m` will have to be given appropriately in the import statement.)

2.3.7 Argument Checking

Using ‘require’ etc. one can do argument checking easily within intrinsics. If a necessary condition on the argument fails to hold, then the relevant error message is printed and the error pointer refers to the caller of the intrinsic. This feature allows user-defined intrinsics to treat errors in actual arguments in exactly the same way as they are treated by the MAGMA standard functions.

```
require condition: print_args;
```

The expression *condition* may be any yielding a Boolean value. If the value is false, then *print_args* is printed and execution aborts with the error pointer pointing to the caller. The print arguments *print_args* can consist of any expressions (depending on arguments or variables already defined in the intrinsic).

```
requirerange v, L, U;
```

The argument variable *v* must be the name of one of the argument variables (including parameters) and must be of integer type. *L* and *U* may be any expressions each yielding an integer value. If *v* is not in the range $[L, \dots, U]$, then an appropriate error message is printed and execution aborts with the error pointer pointing to the caller.

```
requirege v, L;
```

The argument variable *v* must be the name of one of the argument variables (including parameters) and must be of integer type. *L* must yield an integer value. If *v* is not greater than or equal to *L*, then an appropriate error message is printed and execution aborts with the error pointer pointing to the caller.

Example H2E10

A trivial version of `Binomial(n, k)` which checks that $n \geq 0$ and $0 \leq k \leq n$.

```
intrinsic Binomial(n::RngIntElt, k::RngIntElt) -> RngIntElt
{ Return n choose k }
  requirege n, 0;
  requirerange k, 0, n;
  return Factorial(n) div Factorial(n - k) div Factorial(k);
end intrinsic;
```

A simple function to find a random p -element of a group G .

```
intrinsic pElement(G::Grp, p::RngIntElt) -> GrpElt
{ Return p-element of group G }
  require IsPrime(p): "Argument 2 is not prime";
  x := random{x: x in G | Order(x) mod p eq 0};
  return x^(Order(x) div p);
end intrinsic;
```

2.3.8 Package Specification Files

A *spec file* (short for ‘specification file’) lists a complete tree of MAGMA package files. This makes it easy to collect many package files together and attach them simultaneously.

The specification file consists of a list of tokens which are just space-separated words. The tokens describe a list of package files and directories containing other packages. The list is described as follows. The files that are to be attached in the directory indicated by S are listed enclosed in `{` and `}` characters. A directory may be listed there as well, if it is followed by a list of files from that directory (enclosed in braces again); arbitrary nesting is allowed this way. A filename of the form `+spec` is interpreted as another specification file whose contents will be recursively attached when `AttachSpec` (below) is called. The files are taken relative to the directory that contains the specification file. See also the example below.

AttachSpec(S)

If S is a string indicating the name of a spec file, this command attaches all the files listed in S . The format of the spec file is given above.

DetachSpec(S)

If S is a string indicating the name of a spec file, this command detaches all the files listed in S . The format of the spec file is given above.

Example H2E11

Suppose we have a spec file `/home/user/spec` consisting of the following lines:

```
{
  Group
  {
    chiefseries.m
    socle.m
  }
  Ring
  {
    funcs.m
    Field
    {
      galois.m
    }
  }
}
```

Then there should be the files

```
/home/user/spec/Group/chiefseries.m
/home/user/spec/Group/socle.m
/home/user/spec/Ring/funcs.m
/home/user/spec/Ring/Field/galois.m
```

and if one typed within MAGMA

```
AttachSpec("/home/user/spec");
```

then each of the above files would be attached. If instead of the filename `galois.m` we have `+galspec`, then the file `/home/user/spec/Ring/Field/galspec` would be a specification file itself whose contents would be recursively attached.

2.3.9 User Startup Specification Files

The user may specify a list of spec files to be attached automatically when MAGMA starts up. This is done by setting the environment variable `MAGMA_USER_SPEC` to a colon separated list of spec files.

Example H2E12

One could have

```
setenv MAGMA_USER_SPEC "$HOME/Magma/spec:/home/friend/Magma/spec"
```

in one's `.cshrc`. Then when MAGMA starts up, it will attach all packages listed in the spec files `$HOME/Magma/spec` and `/home/friend/Magma/spec`.

2.4 Attributes

This section is placed beside the section on packages because the use of attributes is most common within packages.

For any structure within MAGMA, it is possible to have *attributes* associated with it. These are simply values stored within the structure and are referred to by named fields in exactly the same manner as MAGMA records.

There are two kinds of structure attributes: predefined system attributes and user-defined attributes. Both kinds are discussed in the following subsections. A description of how attributes are accessed and assigned then follows.

2.4.1 Predefined System Attributes

The valid fields of predefined system attributes are automatically defined at the startup of Magma. These fields now replace the old method of using the procedure `AssertAttribute` and the function `HasAttribute` (which will still work for some time to preserve backwards compatibility). For each name which is a valid first argument for `AssertAttribute` and `HasAttribute`, that name is a valid attribute field for structures of the appropriate category. Thus the backquote method for accessing attributes described in detail below should now be used instead of the old method. For such attributes, the code:

```
> S'Name := x;
```

is completely equivalent to the code:

```
> AssertAttribute(S, "Name", x);
```

(note that the function `AssertAttribute` takes a string for its second argument so the name must be enclosed in double quotes). Similarly, the code:

```
> if assigned S'Name then
>     x := S'Name;
>     // do something with x...
> end if;
```

is completely equivalent to the code:

```
> l, x := HasAttribute(S, "Name");
> if l then
>     // do something with x...
> end if;
```

(note again that the function `HasAttribute` takes a string for its second argument so the name must be enclosed in double quotes).

Note also that if a system attribute is not set, referring to it in an expression (using the backquote operator) will *not* trigger the calculation of it (while the corresponding intrinsic function will if it exists); rather an error will ensue. Use the `assigned` operator to test whether an attribute is actually set.

2.4.2 User-defined Attributes

For any category C , the user can stipulate valid attribute fields for structures of C . After this is done, any structure of category C may have attributes assigned to it and accessed from it.

There are two ways of adding new valid attributes to a category C : by the procedure `AddAttribute` or by the `declare attributes` package declaration. The former should be used outside of packages (e.g. in interactive usage), while the latter must be used within packages to declare attribute fields used by the package and related packages.

AddAttribute(C , F)

(Procedure.) Given a category C , and a string F , append the field name F to the list of valid attribute field names for structures belonging to category C . This procedure should not be used within packages but during interactive use. Previous fields for C are still valid – this just adds another valid one.

declare attributes C : F_1, \dots, F_n ;

Given a category C , and a comma-separated list of identifiers F_1, \dots, F_n append the field names specified by the identifiers to the list of valid attribute field names for structures belonging to category C . This declaration directive must be used within (and only within) packages to declare attribute fields used by the package and packages related to it which use the same fields. It is *not* a statement but a directive which is stored with the other information of the package when it is compiled and subsequently attached – *not* when any code is actually executed.

2.4.3 Accessing Attributes

Attributes of structures are accessed in the same way that records are: using the backquote (‘) operator. The double backquote operator (‘‘) can also be used if the field name is a string.

S ‘*fieldname*

S ‘‘ N

Given a structure S and a field name, return the current value for the given field in S . If the value is not assigned, an error results. The field name must be valid for the category of S . In the S ‘‘ N form, N is a string giving the field name.

assigned S ‘*fieldname*

assigned S ‘‘ N

Given a structure S and a field name, return whether the given field in S currently has a value. The field name must be valid for the category of S . In the S ‘‘ N form, N is a string giving the field name.

`S'fieldname := expression;`

`S' 'N := expression;`

Given a structure S and a field name, assign the given field of S to be the value of the expression (any old value is first discarded). The field name must be valid for the category of S . In the $S' 'N$ form, N is a string giving the field name.

`delete S'fieldname;`

`delete S' 'N;`

Given a structure S and a field name, delete the given field of S . The field then becomes unassigned in S . The field name must be valid for the category of S and the field must be currently assigned in S . This statement is not allowed for predefined system attributes. In the $S' 'N$ form, N is a string giving the field name.

`GetAttributes(C)`

Given a category C , return the valid attribute field names for structures belonging to category C as a sorted sequence of strings.

`ListAttributes(C)`

(Procedure.) Given a category C , list the valid attribute field names for structures belonging to category C .

2.5 User-defined Verbose Flags

Verbose flags may be defined by users within packages.

`declare verbose F, m;`

Given a verbose flag name F (without quotes), and a literal integer m , create the verbose flag F , with the maximal allowable level for the flag set to m . This directive may only be used within package files.

2.5.1 Examples

In this subsection we give examples which illustrate all of the above features.

Example H2E13

We illustrate how the predefined system attributes may be used. Note that the valid arguments for `AssertAttribute` and `HasAttribute` documented elsewhere now also work as system attributes so see the documentation for these functions for details as to the valid system attribute field names.

```
> // Create group G.
> G := PSL(3, 2);
> // Check whether order known.
> assigned G'Order;
false
> // Attempt to access order -- error since not assigned.
> G'Order;
```

```

>> G'Order;
~
Runtime error in ': Attribute 'Order' for this structure
is valid but not assigned
> // Force computation of order by intrinsic Order.
> Order(G);
168
> // Check Order field again.
> assigned G'Order;
true
> G'Order;
168
> G'"Order"; // String form for field
168
> o := "Order";
> G'o;
168
> // Create code C and set its minimum weight.
> C := QRCode(GF(2), 31);
> C'MinimumWeight := 7;
> C;
[31, 16, 7] Quadratic Residue code over GF(2)
...

```

Example H2E14

We illustrate how user attributes may be defined and used in an interactive session. This situation would arise rarely – more commonly, attributes would be used within packages.

```

> // Add attribute field MyStuff for matrix groups.
> AddAttribute(GrpMat, "MyStuff");
> // Create group G.
> G := GL(2, 3);
> // Try illegal field.
> G'silly;
>> G'silly;
~
Runtime error in ': Invalid attribute 'silly' for this structure
> // Try legal but unassigned field.
> G'MyStuff;
>> G'MyStuff;
~
Runtime error in ': Attribute 'MyStuff' for this structure is valid but not
assigned
> // Assign field and notice value.
> G'MyStuff := [1, 2];
> G'MyStuff;

```

[1, 2]

Example H2E15

We illustrate how user attributes may be used in packages. This is the most common usage of such attributes. We first give some (rather naive) MAGMA code to compute and store a permutation representation of a matrix group. Suppose the following code is stored in the file `permrep.m`.

```
declare attributes GrpMat: PermRep, PermRepMap;
intrinsic PermutationRepresentation(G::GrpMat) -> GrpPerm
{A permutation group representation P of G, with homomorphism f: G -> P};
    // Only compute rep if not already stored.
    if not assigned G'PermRep then
        G'PermRepMap, G'PermRep := CosetAction(G, sub<G|>);
    end if;
    return G'PermRep, G'PermRepMap;
end intrinsic;
```

Note that the information stored will be reused in subsequent calls of the intrinsic. Then the package can be attached within a MAGMA session and the intrinsic `PermutationRepresentation` called like in the following code (assumed to be run in the same directory).

```
> Attach("permrep.m");
> G := GL(2, 2);
> P, f := PermutationRepresentation(G);
> P;
Permutation group P acting on a set of cardinality 6
    (1, 2)(3, 5)(4, 6)
    (1, 3)(2, 4)(5, 6)
> f;
Mapping from: GrpMat: G to GrpPerm: P
```

Suppose the following line were also in the package file:

```
declare verbose MyAlgorithm, 3;
```

Then there would be a new verbose flag `MyAlgorithm` for use anywhere within MAGMA, with the maximum 3 for the level.

2.6 User-Defined Types

Since MAGMA V2.19, types may be defined by users within packages. This facility allows the user to declare new type names and create objects with such types and then supply some basic primitives and intrinsic functions for such objects.

The new types are known as *user-defined types*. The way these are typically used is that after declaring such a type T , the user supplies package intrinsics to: (1) create objects of type T and set relevant attributes to define the objects; (2) perform some basic primitives which are common to all objects in MAGMA; (3) perform non-trivial computations on objects of type T .

2.6.1 Declaring User-Defined Types

The following declarations are used to declare user-defined types. They **may only be placed in package files**, i.e., files that are included either by using `Attach` or a spec file (see above). Declarations may appear in any package file and at any place within the file at the top level (not in a function, etc.). In particular, it is not required that the declaration of a type appears before package code which refers to the type (as long as the type is declared before running the code). Examples below will illustrate how the basic declarations are used.

```
declare type  $T$ ;
```

Declare the given type name T (without quotes) to be a user-defined type.

```
declare type  $T : P_1, \dots, P_n$ ;
```

Declare the given type name T (without quotes) to be a user-defined type, and also declare T to inherit from the user types P_1, \dots, P_n (which must be declared separately). As a result, $\text{ISA}(T, P_i)$ will be true for each i and when intrinsic signatures are scanned at a function call, an object of type T will match an argument of a signature with type P_i for any i .

NB: currently one may not inherit from existing MAGMA internal types or virtual types (categories). It is hoped that this restriction will be removed in the future.

```
declare type  $T[E]$ ;
```

Declare the given type names T and E (both without quotes) to be user-defined types. This form also specifies that E is the *element type* corresponding to T ; i.e., if an object x has an element of type T for its parent, then x must have type E . This relationship is needed for the construction of sets and sequences which have objects of type T as a universe. The type E may also be declared separately, but this is not necessary.

```
declare type  $T[E] : P_1, \dots, P_n$ ;
```

This is a combination of the previous kinds two declarations: T and E are declared as user-defined types while E is also declared to be the element type of T , and T is declared to inherit from user-defined types P_1, \dots, P_n .

2.6.2 Creating an Object

New(*T*)

Create an empty object of type T , where T is a user-defined type. Typically, after setting X to the result of this function, the user should set attributes in X to define relevant properties of the object which are characteristic of objects of type T .

2.6.3 Special Ininsics Provided by the User

Let T be a user-defined type. Besides the declaration of T , the following special intrinsics are mostly required to be defined for type T (the requirements are specified for each kind of intrinsic). These intrinsics allow the internal MAGMA functions to perform some fundamental operations on objects of type T . Note that the special intrinsics need not be in one file or in the same file as the declaration.

```
intrinsic Print(X::T)
{Print X}
    // Code: Print X with no new line, via printf
end intrinsic;

intrinsic Print(X::T, L::MonStgElt)
{Print X at level L}
    // Code: Print X at level L with no new line, via printf
end intrinsic;
```

Exactly one of these intrinsics must be provided by the user for type T . Each is a procedure rather than a function (i.e., nothing is returned), and should contain one or more print statements. The procedure is called automatically by MAGMA whenever the object X of type T is to be printed. A new line should *not* occur at the end of the last (or only) line of printing: one should use `printf` (see examples below).

When the second form of the intrinsic is provided, it allows X to be printed differently depending on the print level L , which is a string equal to one of "Default", "Minimal", "Maximal", "Magma".

```
intrinsic Parent(X::T) -> .
{Parent of X}
    // Code: Return the parent of X
end intrinsic;
```

This intrinsic is only needed when T is an element type, so objects of type T have parents. It should be a user-provided package function, which takes an object X of type T (user-defined), and returns the parent of X , assuming it has one. In such a case, typically the attribute `Parent` will be defined for X and so `X.Parent` should simply be returned.

```

intrinsic 'in'(e::., X::T) -> BoolElt
{Return whether e is in X}
  // Code: Return whether e is in X
end intrinsic;

```

This intrinsic is only needed when objects of type T (user-defined) have elements, and should be a user-provided package function, which takes any object e and an object X of type T (user-defined), and returns whether e is an element of X .

```

intrinsic IsCoercible(X::T, y::.) -> BoolElt, .
{Return whether y is coercible into X and the result if so}
  // Code: do tests on the type of y to see whether coercible
  // On failure, do:
  //   return false, "Illegal coercion"; // Or more particular message
  // Assumed coercible now; set x to result of coercion into X
  return true, x;
end intrinsic;

```

Assuming that objects of type T (user-defined) have elements (and so coercion into such objects makes sense), this must be a user-provided package function, which takes an object X of type T (user-defined) and an object Y of any type. If Y is coercible into X , the function should return **true** and the result of the coercion (whose parent should be X). Otherwise, the function should return **false** and a string giving the reason for failure. If this package intrinsic is provided, then the coercion operation $X!y$ will also automatically work for an object X of type T (i.e., the internal coercion code in MAGMA will automatically call this function).

2.6.4 Examples

Some basic examples illustrating the general use of user-defined types are given here. Non-trivial examples can also be found in much of the standard MAGMA package code (one can search for "declare type" in the package .m files to see several typical uses).

Example H2E16

In this first simple example, we create a user-defined type **MyRat** which is used for a primitive representation of rational numbers. Of course, a serious version would keep the numerators & denominators always reduced, but for simplicity we skip such details. We define the operations $+$ and $*$ here; one would typically add other operations like $-$, **eq** and **IsZero**, etc.

```

declare type MyRat;
declare attributes MyRat: Numer, Denom;

intrinsic MyRational(n::RngIntElt, d::RngIntElt) -> MyRat
{Create n/d}

```

```

    require d ne 0: "Denominator must be non-zero";
    r := New(MyRat);
    r'Numer := n;
    r'Denom := d;
    return r;
end intrinsic;

intrinsic Print(r::MyRat)
{Print r}
    n := r'Numer;
    d := r'Denom;
    g := GCD(n, d);
    if d lt 0 then g := -g; end if;
    printf "%o/%o", n div g, d div g; // NOTE: no newline!
end intrinsic;

intrinsic '+'(r::MyRat, s::MyRat) -> MyRat
{Return r + s}
    rn := r'Numer;
    rd := r'Denom;
    sn := s'Numer;
    sd := s'Denom;
    return MyRational(rn*sd + sn*rd, rd*sd);
end intrinsic;

intrinsic '*'(r::MyRat, s::MyRat) -> MyRat
{Return r * s}
    rn := r'Numer;
    rd := r'Denom;
    sn := s'Numer;
    sd := s'Denom;
    return MyRational(rn*sn, rd*sd);
end intrinsic;

```

Assuming the above code is placed in a file `MyRat.m`, one could attach it in MAGMA and then do some simple operations, as follows.

```

> Attach("myrat.m");
> r := MyRational(3, -9);
> r;
-1/3
> s := MyRational(4, 7);
> s;
> r+s;
5/21
> r*s;
-4/21

```

Example H2E17

In this example, we define a type `DirProd` for direct products of rings, and a corresponding element type `DirProdElt` for their elements. Objects of type `DirProd` contain a tuple `Rings` with the rings making up the direct product, while objects of type `DirProdElt` contain a tuple `Element` with the elements of the corresponding rings, and also a reference to the parent direct product object.

```
/* Declare types and attributes */

// Note that we declare DirProdElt as element type of DirProd:
declare type DirProd[DirProdElt];
declare attributes DirProd: Rings;
declare attributes DirProdElt: Elements, Parent;

/* Special intrinsics for DirProd */

intrinsic DirectProduct(Rings::Tup) -> DirProd
{Create the direct product of given rings (a tuple)}
  require forall{R: R in Rings | ISA(Type(R), Rng)}:
    "Tuple entries are not all rings";
  D := New(DirProd);
  D'Rings := Rings;
  return D;
end intrinsic;

intrinsic Print(D::DirProd)
{Print D}
  Rings := D'Rings;
  printf "Direct product of %o", Rings; // NOTE: no newline!
end intrinsic;

function CreateElement(D, Elements)
  // Create DirProdElt with parent D and given Elements
  x := New(DirProdElt);
  x'Elements := Elements;
  x'Parent := D;
  return x;
end function;

intrinsic IsCoercible(D::DirProd, x::.) -> BoolElt, .
{Return whether x is coercible into D and the result if so}
  Rings := D'Rings;
  n := #Rings;
  if Type(x) ne Tup then
    return false, "Coercion RHS must be a tuple";
  end if;
  if #x ne n then
    return false, "Wrong length of tuple for coercion";
  end if;
```



```

    Elements := <>;
    for i := 1 to n do
        l, t := IsCoercible(Rings[i], x[i]);
        if not l then
            return false, Sprintf("Tuple entry %o not coercible", i);
        end if;
        Append(~Elements, t);
    end for;
    y := CreateElement(D, Elements);
    return true, y;
end intrinsic;

/* Special intrinsics for DirProdElt */

intrinsic Print(x::DirProdElt)
{Print x}
    printf "%o", x'Elements; // NOTE: no newline!
end intrinsic;

intrinsic Parent(x::DirProdElt) -> DirProd
{Parent of x}
    return x'Parent;
end intrinsic;

intrinsic '+'(x::DirProdElt, y::DirProdElt) -> DirProdElt
{Return x + y}
    D := Parent(x);
    require D cmpeq Parent(y): "Incompatible arguments";
    Ex := x'Elements;
    Ey := y'Elements;
    return CreateElement(D, <Ex[i] + Ey[i]: i in [1 .. #Ex]>);
end intrinsic;

intrinsic '*'(x::DirProdElt, y::DirProdElt) -> DirProdElt
{Return x * y}
    D := Parent(x);
    require D cmpeq Parent(y): "Incompatible arguments";
    Ex := x'Elements;
    Ey := y'Elements;
    return CreateElement(D, <Ex[i] * Ey[i]: i in [1 .. #Ex]>);
end intrinsic;

```

A sample MAGMA session using the above package is as follows. We create elements x, y of a direct product D and do simple operations on x, y . One would of course add other intrinsic functions for basic operations on the elements.

```

> Attach("DirProd.m");
> Z := IntegerRing();

```

```
> Q := RationalField();
> F8<a> := GF(2^3);
> F9<b> := GF(3^2);
> D := DirectProduct(<Z, Q, F8, F9>);
> x := D!<1, 2/3, a, b>;
> y := D!<2, 3/4, a+1, b+1>;
> x;
<1, 2/3, a, b>
> Parent(x);
Direct product of <Integer Ring, Rational Field, Finite field of
size 2^3, Finite field of size 3^2>
> y;
<2, 3/4, a^3, b^2>
> x+y;
<3, 17/12, 1, b^3>
> x*y;
<2, 1/2, a^4, b^3>
> D!x;
<1, 2/3, a, b>
> S := [x, y]; S;
[
    <1, 2/3, a, b>,
    <2, 3/4, a^3, b^2>
]
>
> &+S;
<3, 17/12, 1, b^3>
```

3 INPUT AND OUTPUT

3.1 Introduction	65	<code>print e, ..., e;</code>	72
3.2 Character Strings	65	<code>print e: -;</code>	72
3.2.1 <i>Representation of Strings</i>	65	3.3.2 <i>The printf and fprintf Statements</i>	73
3.2.2 <i>Creation of Strings</i>	66	<code>printf format, e, ..., e;</code>	73
"abc"	66	<code>fprintf file, format, e, ..., e;</code>	74
<code>BinaryString(s)</code>	66	3.3.3 <i>Verbose Printing (vprint, vprintf)</i>	75
<code>BString(s)</code>	66	<code>vprint flag: e, ..., e;</code>	75
<code>cat</code>	66	<code>vprint flag, n: e, ..., e;</code>	75
<code>*</code>	66	<code>vprintf flag: format, e, ..., e;</code>	75
<code>cat:=</code>	66	<code>vprintf flag, n: format, e, ..., e;</code>	75
<code>*:=</code>	66	3.3.4 <i>Automatic Printing</i>	76
<code>&cat s</code>	66	<code>ShowPrevious()</code>	76
<code>&* s</code>	66	<code>ShowPrevious(i)</code>	76
<code>^</code>	66	<code>ClearPrevious()</code>	76
<code>s[i]</code>	66	<code>SetPreviousSize(n)</code>	77
<code>s[i]</code>	67	<code>GetPreviousSize()</code>	77
<code>ElementToSequence(s)</code>	67	3.3.5 <i>Indentation</i>	78
<code>Eltseq(s)</code>	67	<code>IndentPush()</code>	78
<code>ElementToSequence(s)</code>	67	<code>IndentPop()</code>	78
<code>Eltseq(s)</code>	67	3.3.6 <i>Printing to a File</i>	78
<code>Substring(s, n, k)</code>	67	<code>PrintFile(F, x)</code>	78
3.2.3 <i>Integer-Valued Functions</i>	67	<code>Write(F, x)</code>	78
<code>#</code>	67	<code>WriteBinary(F, s)</code>	79
<code>Index(s, t)</code>	67	<code>PrintFile(F, x, L)</code>	79
<code>Position(s, t)</code>	67	<code>Write(F, x, L)</code>	79
3.2.4 <i>Character Conversion</i>	67	<code>PrintFileMagma(F, x)</code>	79
<code>StringToCode(s)</code>	67	3.3.7 <i>Printing to a String</i>	79
<code>CodeToString(n)</code>	67	<code>Sprint(x)</code>	79
<code>StringToInteger(s)</code>	68	<code>Sprint(x, L)</code>	79
<code>StringToInteger(s, b)</code>	68	<code>Sprintf(F, ...)</code>	79
<code>StringToIntegerSequence(s)</code>	68	3.3.8 <i>Redirecting Output</i>	80
<code>IntegerToString(n)</code>	68	<code>SetOutputFile(F)</code>	80
<code>IntegerToString(n, b)</code>	68	<code>UnsetOutputFile()</code>	80
3.2.5 <i>Boolean Functions</i>	68	<code>HasOutputFile()</code>	80
<code>eq</code>	68	3.4 External Files	80
<code>ne</code>	68	3.4.1 <i>Opening Files</i>	80
<code>in</code>	68	<code>Open(S, T)</code>	80
<code>notin</code>	69	3.4.2 <i>Operations on File Objects</i>	81
<code>lt</code>	69	<code>Flush(F)</code>	81
<code>le</code>	69	<code>Tell(F)</code>	81
<code>gt</code>	69	<code>Seek(F, o, p)</code>	81
<code>ge</code>	69	<code>Rewind(F)</code>	81
3.2.6 <i>Parsing Strings</i>	71	<code>Put(F, S)</code>	81
<code>Split(S, D)</code>	71	<code>Puts(F, S)</code>	81
<code>Split(S)</code>	71	<code>Getc(F)</code>	81
<code>Regexp(R, S)</code>	71	<code>Gets(F)</code>	81
3.3 Printing	72	<code>IsEof(S)</code>	81
3.3.1 <i>The print-Statement</i>	72	<code>Ungetc(F, c)</code>	81
<code>print e;</code>	72		

3.4.3 Reading a Complete File	82	read id;	88
Read(F)	82	read id, prompt;	88
ReadBinary(F)	82	readi id;	89
		readi id, prompt;	89
3.5 Pipes	83	3.8 Loading a Program File	89
3.5.1 Pipe Creation	83	load "filename";	89
POpen(C, T)	83	iload "filename";	89
Pipe(C, S)	83		
3.5.2 Operations on Pipes	84	3.9 Saving and Restoring Workspaces	89
Read(P : -)	84	save "filename";	89
ReadBytes(P : -)	84	restore "filename";	89
Write(P, s)	85		
WriteBytes(P, Q)	85	3.10 Logging a Session	90
3.6 Sockets	85	SetLogFile(F)	90
3.6.1 Socket Creation	85	UnsetLogFile()	90
Socket(H, P : -)	85	SetEchoInput(b)	90
Socket(: -)	86		
WaitForConnection(S)	86	3.11 Memory Usage	90
3.6.2 Socket Properties	86	GetMemoryUsage()	90
SocketInformation(S)	86	GetMaximumMemoryUsage()	90
3.6.3 Socket Predicates	86	ResetMaximumMemoryUsage()	90
IsServerSocket(S)	86		
3.6.4 Socket I/O	87	3.12 System Calls	90
Read(S : -)	87	Alarm(s)	90
ReadBytes(S : -)	87	ChangeDirectory(s)	90
Write(S, s)	87	GetCurrentDirectory()	90
WriteBytes(S, Q)	87	Getpid()	91
		Getuid()	91
3.7 Interactive Input	88	System(C)	91
		%! <i>shell-command</i>	91
		3.13 Creating Names	91
		Tempname(P)	91

Chapter 3

INPUT AND OUTPUT

3.1 Introduction

This chapter is concerned with the various facilities provided for communication between MAGMA and its environment. The first section describes character strings and their operations. Following this, the various forms of the `print`-statement are presented. Next the file type is introduced and its operations summarized. The chapter concludes with a section listing system calls. These include facilities that allow the user to execute an operating system command from within MAGMA or to run an external process.

3.2 Character Strings

Strings of characters play a central role in input/output so that the operations provided for strings to some extent reflect this. However, if one wishes, a more general set of operations are available if the string is first converted into a sequence. We will give some examples of this below.

MAGMA provides two kinds of strings: normal character strings, and *binary strings*. Character strings are an inappropriate choice for manipulating data that includes non-printable characters. If this is required, a better choice is the binary string type. This type is similar semantically to a sequence of integers, in which each character is represented by its ASCII value between 0 and 255. The difference between a binary string and a sequence of integers is that a binary string is stored internally as an array of bytes, which is a more space-efficient representation.

3.2.1 Representation of Strings

Character strings may consist of all ordinary characters appearing on your keyboard, including the blank (space). Two symbols have a special meaning: the double-quote `"` and the backslash `\`. The double-quote is used to delimit a character string, and hence cannot be used inside a string; to be able to use a double-quote in strings the backslash is designed to be an escape character and is used to indicate that the next symbol has to be taken literally; thus, by using `\"` inside a string one indicates that the symbol `"` has to be taken literally and is not to be interpreted as the end-of-string delimiter. Thus:

```
> "\"Print this line in quotes\"";  
"Print this line in quotes"
```

To obtain a literal backslash, one simply types two backslashes; for characters other than double-quotes and backslash it does not make a difference when a backslash precedes them

inside a string, with the exception of `n`, `r` and `t`. Any occurrence of `\n` or `\r` inside a string is converted into a `<new-line>` while `\t` is converted into a `<tab>`. For example:

```
> "The first line,\nthe second line, and then\r\n\tindented line";
The first line,
the second line, and then
an      indented line
```

Note that a backslash followed by a return allows one to conveniently continue the current construction on the next line; so `\<return>` inside a string will be ignored, except that input will continue on a new line on your screen.

Binary strings, on the hand, can consist of any character, whether printable or non-printable. Binary strings cannot be constructed using literals, but must be constructed either from a character string, or during a read operation from a file.

3.2.2 Creation of Strings

```
"abc"
```

Create a string from a succession of keyboard characters (`a`, `b`, `c`) enclosed in double quotes " ".

```
BinaryString(s)
```

```
BString(s)
```

Create a binary string from the character string s .

```
s cat t
```

```
s * t
```

Concatenate the strings s and t .

```
s cat:= t
```

```
s *:= t
```

Modification-concatenation of the string s with t : concatenate s and t and put the result in s .

```
&cat s
```

```
&* s
```

Given an enumerated sequence s of strings, return the concatenation of these strings.

```
s ^ n
```

Form the n -fold concatenation of the string s , for $n \geq 0$. If $n = 0$ this is the empty string, if $n = 1$ it equals s , etc.

```
s[i]
```

Returns the substring of s consisting of the i -th character.

`s[i]`

Returns the numeric value representing the i -th character of s .

`ElementToSequence(s)`

`Eltseq(s)`

Returns the sequence of characters of s (as length 1 strings).

`ElementToSequence(s)`

`Eltseq(s)`

Returns the sequence of numeric values representing the characters of s .

`Substring(s, n, k)`

Return the substring of s of length k starting at position n .

3.2.3 Integer-Valued Functions

`#s`

The length of the string s .

`Index(s, t)`

`Position(s, t)`

This function returns the position (an integer p with $0 < p \leq \#s$) in the string s where the beginning of a contiguous substring t occurs. It returns 0 if t is not a substring of s . (If t is the empty string, position 1 will always be returned, even if s is empty as well.)

3.2.4 Character Conversion

To perform more sophisticated operations, one may convert the string into a sequence and use the extensive facilities for sequences described in the next part of this manual; see the examples at the end of this chapter for details.

`StringToCode(s)`

Returns the code number of the first character of string s . This code depends on the computer system that is used; it is ASCII on most UNIX machines.

`CodeToString(n)`

Returns a character (string of length 1) corresponding to the code number n , where the code is system dependent (see previous entry).

StringToInteger(s)

Returns the integer corresponding to the string of decimal digits s . All non-space characters in the string s must be digits $(0, 1, \dots, 9)$, except the first character, which is also allowed to be $+$ or $-$. An error results if any other combination of characters occurs. Leading zeros are omitted.

StringToInteger(s, b)

Returns the integer corresponding to the string of digits s , all assumed to be written in base b . All non-space characters in the string s must be digits less than b (if b is greater than 10, 'A' is used for 10, 'B' for 11, etc.), except the first character, which is also allowed to be $+$ or $-$. An error results if any other combination of characters occurs.

StringToIntegerSequence(s)

Returns the sequence of integers corresponding to the string s of space-separated decimal numbers. All non-space characters in the string s must be digits $(0, 1, \dots, 9)$, except the first character after each space, which is also allowed to be $+$ or $-$. An error results if any other combination of characters occurs. Leading zeros are omitted. Each number can begin with a sign ($+$ or $-$) without a space.

IntegerToString(n)

Convert the integer n into a string of decimal digits; if n is negative the first character of the string will be $-$. (Note that leading zeros and a $+$ sign are ignored when MAGMA builds an integer, so the resulting string will never begin with $+$ or 0 characters.)

IntegerToString(n, b)

Convert the integer n into a string of digits with the given base (which must be in the range $[2 \dots 36]$); if n is negative the first character of the string will be $-$.

3.2.5 Boolean Functions**s eq t**

Returns **true** if and only if the strings s and t are identical. Note that blanks are significant.

s ne t

Returns **true** if and only if the strings s and t are distinct. Note that blanks are significant.

s in t

Returns **true** if and only if s appears as a contiguous substring of t . Note that the empty string is contained in every string.

<code>s notin t</code>

Returns **true** if and only if s does not appear as a contiguous substring of t . Note that the empty string is contained in every string.

<code>s lt t</code>

Returns **true** if s is lexicographically less than t , **false** otherwise. Here the ordering on characters imposed by their ASCII code number is used.

<code>s le t</code>

Returns **true** if s is lexicographically less than or equal to t , **false** otherwise. Here the ordering on characters imposed by their ASCII code number is used.

<code>s gt t</code>

Returns **true** if s is lexicographically greater than t , **false** otherwise. Here the ordering on characters imposed by their ASCII code number is used.

<code>s ge t</code>

Returns **true** if s is lexicographically greater than or equal to t , **false** otherwise. Here the ordering on characters imposed by their ASCII code number is used.

Example H3E1

```
> "Mag" cat "ma";
Magma
```

Omitting double-quotes usually has undesired effects:

```
> "Mag cat ma";
Mag cat ma
```

And note that there are two different equalities involved in the following!

```
> "73" * "9" * "42" eq "7" * "3942";
true
> 73 * 9 * 42 eq 7 * 3942;
true
```

The next line shows how strings can be concatenated quickly, and also that strings of blanks can be used for formatting:

```
> s := ("Mag" cat "ma? ")^2;
> s, " "^30, s[4]^12, "!";
Magma? Magma?                               mmmmmmmmmmmmm !
```

Here is a way to list (in a sequence) the first occurrence of each of the ten digits in the decimal expansion of π , using `IntegerToString` and `Position`.

```
> pi := Pi(RealField(1001));
> dec1000 := Round(10^1000*(pi-3));
> I := IntegerToString(dec1000);
> [ Position(I, IntegerToString(i)) : i in [0..9] ];
```

```
[ 32, 1, 6, 9, 2, 4, 7, 13, 11, 5 ]
```

Using the length `#` and string indexing `[]` it is also easy to count the number of occurrences of each digit in the string containing the first 1000 digits.

```
> [ #[i : i in [1..#I] | I[i] eq IntegerToString(j)] : j in [0..9] ];
[ 93, 116, 103, 102, 93, 97, 94, 95, 101, 106 ]
```

We would like to test if the ASCII-encoding of the string ‘Magma’ appears. This could be done as follows, using `StringToCode` and `in`, or alternatively, `Position`. To reduce the typing, we first abbreviate `IntegerToString` to `is` and `StringToCode` to `sc`.

```
> sc := StringToCode;
> its := IntegerToString;
> M := its(sc("M")) * its(sc("a")) * its(sc("g")) * its(sc("m")) * its(sc("a"));
> M;
779710310997
> M in I;
false
> Position(I, M);
0
```

So ‘Magma’ does not appear this way. However, we could be satisfied if the letters appear somewhere in the right order. To do more sophisticated operations (like this) on strings, it is necessary to convert the string into a sequence, because sequences constitute a more versatile data type, allowing many more advanced operations than strings.

```
> Iseq := [ I[i] : i in [1..#I] ];
> Mseq := [ M[i] : i in [1..#M] ];
> IsSubsequence(Mseq, Iseq);
false
> IsSubsequence(Mseq, Iseq: Kind := "Sequential");
true
```

Finally, we find that the string ‘magma’ lies in between ‘Pi’ and ‘pi’:

```
> "Pi" le "magma";
true
> "magma" lt "pi";
true
```

3.2.6 Parsing Strings

Split(*S*, *D*)

Split(*S*)

Given a string *S*, together with a string *D* describing a list of separator characters, return the sequence of strings obtained by splitting *S* at any of the characters contained in *D*. That is, *S* is considered as a sequence of fields, with any character in *D* taken to be a delimiter separating the fields. If *D* is omitted, it is taken to be the string consisting of the newline character alone (so *S* is split into the lines found in it). If *S* is desired to be split into space-separated words, the argument " \t\n" should be given for *D*.

Example H3E2

We demonstrate elementary uses of `Split`.

```
> Split("a b c d", " ");
[ a, b, c, d ]
> // Note that an empty field is included if the
> // string starts with the separator:
> Split(" a b c d", " ");
[ , a, b, c, d ]
> Split("abxcdyefzab", "xyz");
[ ab, cd, ef, ab ]
> // Note that no splitting happens if the delimiter
> // is empty:
> Split("abcd", "");
[ abcd ]
```

Regexp(*R*, *S*)

Given a string *R* specifying a regular expression, together with a string *S*, return whether *S* matches *R*. If so, return also the matched substring of *S*, together with the sequence of matched substrings of *S* corresponding to the parenthesized expressions of *R*. This function is based on the freely distributable reimplementations of the V8 regexp package by Henry Spencer. The syntax and interpretation of the characters `|`, `*`, `+`, `?`, `^`, `$`, `[]`, `\` is the same as in the UNIX command `egrep`. The parenthesized expressions are numbered in left-to-right order of their opening parentheses. Note that the parentheses should not have an initial backslash before them as the UNIX commands `grep` and `ed` require.

Example H3E3

We demonstrate some elementary uses of `Regexp`.

```
> Regexp("b.*d", "abcde");
true bcd []
> Regexp("b(.*?)d", "abcde");
true bcd [ c ]
> Regexp("b.*d", "xyz");
false
> date := "Mon Jun 17 10:27:27 EST 1996";
> _, _, f := Regexp("([0-9][0-9]):([0-9][0-9]):([0-9][0-9])", date);
> f;
[ 10, 27, 27 ]
> h, m, s := Explode(f);
> h, m, s;
10 27 27
```

3.3 Printing

3.3.1 The print-Statement

<code>print expression;</code>

<code>print expression, ..., expression;</code>

<code>print expression: parameters;</code>
--

Print the value of the expression. Some limited ways of formatting output are described in the section on strings. Four levels of printing (that may in specific cases coincide) exist, and may be indicated after the colon: **Default** (which is the same as the level obtained if no level is indicated), **Minimal**, **Maximal**, and **Magma**. The last of these produces output representing the value of the identifier as valid MAGMA-input (when possible).

3.3.2 The printf and fprintf Statements

```
printf format, expression, ..., expression;
```

Print values of the expressions under control of *format*. The first argument, the *format string*, must be a string which contains two types of objects: plain characters, which are simply printed, and conversion specifications (indicated by the % character), each of which causes conversion and printing of zero or more of the expressions. (Use %% to get a literal percent character.) Currently, the only conversion specifications allowed are: %o and %O, which stand for “object”, %m, which stands for “magma”, and %h, which stands for “hexadecimal”.

The hexadecimal conversion specification will print its argument in hexadecimal; currently, it only supports integer arguments. The object and magma conversion specifications each print the corresponding argument; they differ only in the printing mode used. The %o form uses the default printing mode, while the %O form uses the printing mode specified by the next argument (as a string). The “magma” conversion specification uses a printing mode of **Magma**. It is thus equivalent to (but shorter than) using %O and an extra argument of “Magma”.

For each of these conversion specifications, the object can be printed in a field of a particular width by placing extra characters immediately after the % character: digits describing a positive integer, specifying a field with width equal to that number and with right-justification; digits describing a negative integer, specifying a field with width equal to the absolute value of the number and with left-justification; or the character * specifying a field width given by the next appropriate expression argument (with justification determined by the sign of the number). This statement is thus like the C language function `printf()`, except that %o (and %O and %m) covers all kinds of objects — it is not necessary to have different conversion specifications for the different types of MAGMA objects. Note also that this statement does *not* print a newline character after its arguments while the `print` statement does (a \n character should be placed in the format string if this is desired). A newline character will be printed just before the next prompt, though, if there is an incomplete line at that point.

Example H3E4

The following statements demonstrate simple uses of *printf*.

```
> for i := 1 to 150 by 33 do printf "[%3o]\n", i; end for;
[ 1]
[34]
[67]
[100]
[133]
> for i := 1 to 150 by 33 do printf "[% -3o]\n", i; end for;
[1 ]
[34 ]
[67 ]
```



```
> delete F;
  37107316853453566312041115519 (2^109 mod p)
  70602400912917605986812821219 (2^102 mod p)
  74214633706907132624082231038 (2^110 mod p)
129638414606681695789005139447 (2^106 mod p)
141204801825835211973625642438 (2^103 mod p)
259276829213363391578010278894 (2^107 mod p)
267650600228229401496703205319 (2^100 mod p)
282409603651670423947251284876 (2^104 mod p)
518553658426726783156020557788 (2^108 mod p)
535301200456458802993406410638 (2^101 mod p)
564819207303340847894502569752 (2^105 mod p)
```

3.3.3 Verbose Printing (`vprint`, `vprintf`)

The following statements allow convenient printing of information conditioned by whether an appropriate verbose flag is turned on.

<code>vprint flag: expression, ..., expression;</code>
--

<code>vprint flag, n: expression, ..., expression;</code>

If the verbose flag *flag* (see the function `SetVerbose`) has a level greater than or equal to *n*, print the expressions to the right of the colon exactly as in the `print` statement. If the flag has level 0 (i.e. is not turned on), do nothing. In the first form of this statement, where a specific level is not given, *n* is taken to be 1. This statement is useful in MAGMA code found in packages where one wants to print verbose information if an appropriate verbose flag is turned on.

<code>vprintf flag: format, expression, ..., expression;</code>

<code>vprintf flag, n: format, expression, ..., expression;</code>
--

If the verbose flag *flag* (see the function `SetVerbose`) has a level greater than or equal to *n*, print using the format and the expressions to the right of the colon exactly as in the `printf` statement. If the flag has level 0 (i.e. is not turned on), do nothing. In the first form of this statement, where a specific level is not given, *n* is taken to be 1. This statement is useful in MAGMA code found in packages where one wants to print verbose information if an appropriate verbose flag is turned on.

3.3.4 Automatic Printing

MAGMA allows *automatic printing* of expressions: basically, a statement consisting of an expression (or list of expressions) alone is taken as a shorthand for the `print`-statement.

Some subtleties are involved in understanding the precise behaviour of MAGMA in interpreting lone expressions as statements. The rules MAGMA follows are outlined here. In the following, a *call-form* means any expression of the form $f(\textit{arguments})$; that is, anything which could be a procedure call or a function call.

- (a) Any single expression followed by a semicolon which is not a call-form is printed, just as if you had ‘print’ in front of it.
- (b) For a single call-form followed by a semicolon (which could be a function call or procedure call), the first signature which matches the input arguments is taken and if that is procedural, the whole call is taken as a procedure call, otherwise it is taken as function call and the results are printed.
- (c) A comma-separated list of any expressions is printed, just as if you had ‘print’ in front of it. Here any call-form is taken as a function call only so procedure calls are impossible.
- (d) A print level modifier is allowed after an expression list (whether the list has length 1 or more). Again any call-form is taken as a function call only so procedure calls are impossible.
- (e) Any list of objects printed, whether by any of the above rules or by the ‘print’ statement, is placed in the previous value buffer. `$1` gives the last printed list, `$2` the one before, etc. Note that multi-return values stay as a list of values in the previous value buffer. The only way to get at the individual values of such a list is by assignment to a list of identifiers, or by `where` (this is of course the only way to get the second result out of `Quotrem`, etc.). In other places, a `$1` expression is evaluated with principal value semantics.

MAGMA also provides procedures to manipulate the previous value buffer in which `$1`, etc. are stored.

ShowPrevious()

Show all the previous values stored. This does *not* change the contents of the previous value buffer.

ShowPrevious(i)

Show the i -th previous value stored. This does *not* change the contents of the previous value buffer.

ClearPrevious()

Clear all the previous values stored. This is useful for ensuring that no more memory is used than that referred to by the current identifiers.

SetPreviousSize(n)

Set the size of the previous value buffer (this is not how many values are defined in it at the moment, but the maximum number that will be stored). The default size is 3.

GetPreviousSize()

Return the size of the previous value buffer.

Example H3E7

Examples which illustrate point (a):

```
> 1;
1
> x := 3;
> x;
3
```

Examples which illustrate point (b):

```
> 1 + 1;          // really function call '+'(1, 1)
2
> Q := [ 0 ];
> Append(~Q, 1);   // first (in fact only) match is procedure call
> Append(Q, 1);    // first (in fact only) match is function call
[ 0, 1, 1 ]
> // Assuming fp is assigned to a procedure or function:
> fp(x);           // whichever fp is at runtime
> SetVerbose("Meataxe", true); // simple procedure call
```

Examples which illustrate point (c):

```
> 1, 2;
1 2
> // Assuming f assigned:
> f(x), 1;          // f only can be a function
> SetVerbose("Meataxe", true), 1; // type error in 'SetVerbose'
>                    // (since no function form)
```

Examples which illustrate point (d):

```
> 1: Magma;
1
> Sym(3), []: Maximal;
Symmetric group acting on a set of cardinality 3
Order = 6 = 2 * 3
[]
> SetVerbose("Meataxe", true): Magma; // type error as above
```

Examples which illustrate point (e):

```
> 1;
```

```

1
> $1;
1
> 2, 3;
2 3
> $1;
2 3
> Quotrem(124124, 123);
1009 17
> $1;
1009 17
> a, b := $1;
> a;
1009

```

3.3.5 Indentation

MAGMA has an indentation level which determines how many initial spaces should be printed before each line. The level can be increased or decreased. Each time the top level of Magma is reached (i.e. a prompt is printed), the level is reset to 0. The level is usually changed in verbose output of recursive functions and procedures. The functions `SetIndent` and `GetIndent` are used to control and examine the number of spaces used for each indentation level (default 4).

IndentPush()

Increase (push) the indentation level by 1. Thus the beginning of a line will have s more spaces than before, where s is the current number of indentation spaces.

IndentPop()

Decrease (pop) the indentation level by 1. Thus the beginning of a line will have s less spaces than before, where s is the current number of indentation spaces. If the current level is already 0, an error occurs.

3.3.6 Printing to a File

PrintFile(F , x)

Write(F , x)

Overwrite

BOOLELT

Default : false

Print x to the file specified by the string F . If this file already exists, the output will be appended, unless the optional parameter **Overwrite** is set to true, in which case the file is overwritten.

WriteBinary(*F*, *s*)

Overwrite

BOOLELT

Default : **false**

Write the binary string *s* to the file specified by the string *F*. If this file already exists, the output will be appended, unless the optional parameter **Overwrite** is set to true, in which case the file is overwritten.

PrintFile(*F*, *x*, *L*)

Write(*F*, *x*, *L*)

Overwrite

BOOLELT

Default : **false**

Print *x* in format defined by the string *L* to the file specified by the string *F*. If this file already exists, the output will be appended unless the optional parameter **Overwrite** is set to true, in which case the file is overwritten. The level *L* can be any of the print levels on the **print** command above (i.e., it must be one of the strings "Default", "Minimal", "Maximal", or "Magma").

PrintFileMagma(*F*, *x*)

Overwrite

BOOLELT

Default : **false**

Print *x* in Magma format to the file specified by the string *F*. If this file already exists, the output will be appended, unless the optional parameter **Overwrite** is set to true, in which case the file is overwritten.

3.3.7 Printing to a String

MAGMA allows the user to obtain the string corresponding to the output obtained when printing an object by means of the **Sprint** function. The **Sprintf** function allows formatted printing like the **printf** statement.

Sprint(*x*)

Sprint(*x*, *L*)

Given any MAGMA object *x*, this function returns a string containing the output obtained when *x* is printed. If a print level *L* is given also (a string), the printing is done according to that level (see the **print** statement for the possible printing levels).

Sprintf(*F*, ...)

Given a format string *F*, together with appropriate extra arguments corresponding to *F*, return the string resulting from the formatted printing of *F* and the arguments. The format string *F* and arguments should be exactly as for the **printf** statement – see that statement for details.

Example H3E8

We demonstrate elementary uses of `Sprintf`.

```
> Q := [Sprintf("{%4o<->%-4o}", x, x): x in [1,10,100,1000]];
> Q;
[ { 1<->1 }, { 10<->10 }, { 100<->100 }, {1000<->1000} ]
```

3.3.8 Redirecting Output**SetOutputFile(F)****Overwrite****BOOLELT***Default : false*

Redirect all MAGMA output to the file specified by the string F . By using `SetOutputFile(F: Overwrite := true)` the file F is emptied before output is written onto it.

UnsetOutputFile()

Close the output file, so that output will be directed to standard output again.

HasOutputFile()

If MAGMA currently has an output or log file F , return `true` and F ; otherwise return `false`.

3.4 External Files

MAGMA provides a special *file* type for the reading and writing of external files. Most of the standard C library functions can be applied to such files to manipulate them.

3.4.1 Opening Files**Open(S, T)**

Given a filename (string) S , together with a type indicator T , open the file named by S and return a MAGMA file object associated with it. Tilde expansion is performed on S . The standard C library function `fopen()` is used, so the possible characters allowed in T are the same as those allowed for that function in the current operating system, and have the same interpretation. Thus one should give the value `"r"` for T to open the file for reading, and give the value `"w"` for T to open the file for writing, etc. (Note that in the PC version of MAGMA, the character `"b"` should also be included in T if the file is desired to be opened in binary mode.) Once a file object is created, various I/O operations can be performed on it — see below. A file is closed by deleting it (i.e. by use of the `delete` statement or by reassigning the variable associated with the file); there is no `Fclose` function. This ensures that the file is not closed while there are still multiple references to it. (The function is called `Open` instead of `Fopen` to follow Perl-style conventions. The following functions also follow such conventions where possible.)

3.4.2 Operations on File Objects

Flush(*F*)

Given a file *F*, flush the buffer of *F*.

Tell(*F*)

Given a file *F*, return the offset in bytes of the file pointer within *F*.

Seek(*F*, *o*, *p*)

Perform `fseek(F, o, p)`; i.e. move the file pointer of *F* to offset *o* (relative to *p*: 0 means beginning, 1 means current, 2 means end).

Rewind(*F*)

Perform `rewind(F)`; i.e. move the file pointer of *F* to the beginning.

Put(*F*, *S*)

Put (write) the characters of the string *S* to the file *F*.

Puts(*F*, *S*)

Put (write) the characters of the string *S*, followed by a newline character, to the file *F*.

Getc(*F*)

Given a file *F*, get and return one more character from file *F* as a string. If *F* is at end of file, a special EOF marker string is returned; the function `IsEOF` should be applied to the character to test for end of file. (Thus the only way to loop over a file character by character is to get each character and test whether it is the EOF marker before processing it.)

Gets(*F*)

Given a file *F*, get and return one more line from file *F* as a string. The newline character is removed before the string is returned. If *F* is at end of file, a special EOF marker string is returned; the function `IsEOF` should be applied to the string to test for end of file.

IsEOF(*S*)

Given a string *S*, return whether *S* is the special EOF marker.

Ungetc(*F*, *c*)

Given a character (length one string) *C*, together with a file *F*, perform `ungetc(C, F)`; i.e. push the character *C* back into the input buffer of *F*.

Example H3E9

We write a function to count the number of lines in a file. Note the method of looping over the characters of the file: we must get the line and then test whether it is the special EOF marker.

```
> function LineCount(F)
>   FP := Open(F, "r");
>   c := 0;
>   while true do
>     s := Gets(FP);
>     if IsEof(s) then
>       break;
>     end if;
>     c += 1;
>   end while;
>   return c;
> end function;
> LineCount("/etc/passwd");
59
```

3.4.3 Reading a Complete File**Read(F)**

Function that returns the contents of the text-file with name indicated by the string F . Here F may be an expression returning a string.

ReadBinary(F)

Function that returns the contents of the text-file with name indicated by the string F as a binary string.

Example H3E10

In this example we show how **Read** can be used to import the complete output from a separate C program into a MAGMA session. We assume that a file **mystery.c** (of which the contents are shown below) is present in the current directory. We first compile it, from within MAGMA, and then use it to produce output for the MAGMA version of our **mystery** function.

```
> Read("mystery.c");
#include <stdio.h>
main(argc, argv)
int    argc;
char   **argv;
{
    int n, i;
    n = atoi(argv[1]);
    for (i = 1; i <= n; i++)
        printf("%d\n", i * i);
```

```
    return 0;
}
> System("cc mystery.c -o mystery");
> mysteryMagma := function(n)
>   System("./mystery " cat IntegerToString(n) cat " >outfile");
>   output := Read("outfile");
>   return StringToIntegerSequence(output);
> end function;
> mysteryMagma(5);
[ 1, 4, 9, 16, 25 ]
```

3.5 Pipes

Pipes are used to communicate with newly-created processes. Currently pipes are only available on UNIX systems.

The MAGMA I/O module is currently undergoing revision, and the current pipe facilities are a mix of the old and new methods. A more uniform model will be available in future releases.

3.5.1 Pipe Creation

POpen(*C*, *T*)

Given a shell command line *C*, together with a type indicator *T*, open a pipe between the MAGMA process and the command to be executed. The standard C library function `popen()` is used, so the possible characters allowed in *T* are the same as those allowed for that function in the current operating system, and have the same interpretation. Thus one should give the value "r" for *T* so that MAGMA can read the output from the command, and give the value "w" for *T* so that MAGMA can write into the input of the command. See the **Pipe** intrinsic for a method for sending input to, and receiving output from, a single command.

Important: this function returns a **File** object, and the I/O functions for files described previously must be used rather than those described in the following.

Pipe(*C*, *S*)

Given a shell command *C* and an input string *S*, create a pipe to the command *C*, send *S* into the standard input of *C*, and return the output of *C* as a string. Note that for many commands, *S* should finish with a new line character if it consists of only one line.

Example H3E11

We write a function which returns the current time as 3 values: hour, minutes, seconds. The function opens a pipe to the UNIX command “date” and applies regular expression matching to the output to extract the relevant fields.

```
> function GetTime()
>   D := POpen("date", "r");
>   date := Gets(D);
>   _, _, f := Regexp("([0-9] [0-9]):([0-9] [0-9]):([0-9] [0-9])", date);
>   h, m, s := Explode(f);
>   return h, m, s;
> end function;
> h, m, s := GetTime();
> h, m, s;
14 30 01
> h, m, s := GetTime();
> h, m, s;
14 30 04
```

3.5.2 Operations on Pipes

When a read request is made on a pipe, the available data is returned. If no data is currently available, then the process waits until some does becomes available, and returns that. (It will also return if the pipe has been closed and hence no more data can be transmitted.) It does not continue trying to read more data, as it cannot tell whether or not there is some “on the way”.

The upshot of all this is that care must be exercised as reads may return less data than is expected.

Read (<i>P</i> : <i>parameters</i>)
--

Max

RNGINTELT

Default : 0

Waits for data to become available for reading from *P* and then returns it as a string. If the parameter **Max** is set to a positive value then at most that many characters will be read. Note that less than **Max** characters may be returned, depending on the amount of currently available data.

If the pipe has been closed then the special EOF marker string is returned.

ReadBytes (<i>P</i> : <i>parameters</i>)

Max

RNGINTELT

Default : 0

Waits for data to become available for reading from *P* and then returns it as a sequence of bytes (integers in the range 0..255). If the parameter **Max** is set to a positive value then at most that many bytes will be read. Note that less than **Max** bytes may be returned, depending on the amount of currently available data.

If the pipe has been closed then the empty sequence is returned.

Write(P, s)

Writes the characters of the string *s* to the pipe *P*.

WriteBytes(P, Q)

Writes the bytes in the byte sequence *Q* to the pipe *P*. Each byte must be an integer in the range 0..255.

3.6 Sockets

Sockets may be used to establish communication channels between machines on the same network. Once established, they can be read from or written to in much the same ways as more familiar I/O constructs like files. One major difference is that the data is not instantly available, so the I/O operations take much longer than with files. Currently sockets are only available on UNIX systems.

Strictly speaking, a *socket* is a communication endpoint whose defining information consists of a network address and a port number. (Even more strictly speaking, the communication protocol is also part of the socket. MAGMA only uses TCP sockets, however, so we ignore this point from now on.)

The network address selects on which of the available network interfaces communication will take place; it is a string identifying the machine on that network, in either domain name or dotted-decimal format. For example, both "localhost" and "127.0.0.1" identify the machine on the loopback interface (which is only accessible from the machine itself), whereas "foo.bar.com" or "10.0.0.3" might identify the machine in a local network, accessible from other machines on that network.

The port number is just an integer that identifies the socket on a particular network interface. It must be less than 65536. A value of 0 will indicate that the port number should be chosen by the operating system.

There are two types of sockets, which we will call client sockets and server sockets. The purpose of a client socket is to initiate a connection to a server socket, and the purpose of a server socket is to wait for clients to initiate connections to it. (Thus the server socket needs to be created before the client can connect to it.) Once a server socket accepts a connection from a client socket, a communication channel is established and the distinction between the two becomes irrelevant, as they are merely each side of a communication channel.

In the following descriptions, the network address will often be referred to as the *host*. So a socket is identified by a (*host*, *port*) pair, and an established communication channel consists of two of these pairs: (*local-host*, *local-port*), (*remote-host*, *remote-port*).

3.6.1 Socket Creation

Socket(H, P : parameters)

LocalHost	MONSTGELT	Default : none
LocalPort	RNGINTELT	Default : 0

Attempts to create a (client) socket connected to port *P* of host *H*. Note: these are the *remote* values; usually it does not matter which local values are used for client

sockets, but for those rare occasions where it does they may be specified using the parameters `LocalHost` and `LocalPort`. If these parameters are not set then suitable values will be chosen by the operating system. Also note that port numbers below 1024 are usually reserved for system use, and may require special privileges to be used as the local port number.

<code>Socket(: parameters)</code>

<code>LocalHost</code>	<code>MONSTGELT</code>	<i>Default : none</i>
<code>LocalPort</code>	<code>RNGINTELT</code>	<i>Default : 0</i>

Attempts to create a server socket on the current machine, that can be used to accept connections. The parameters `LocalHost` and `LocalPort` may be used to specify which network interface and port the socket will accept connections on; if either of these are not set then their values will be determined by the operating system. Note that port numbers below 1024 are usually reserved for system use, and may require special privileges to be used as the local port number.

<code>WaitForConnection(S)</code>

This may only be used on server sockets. It waits for a connection attempt to be made, and then creates a new socket to handle the resulting communication channel. Thus *S* may continue to be used to accept connection attempts, while the new socket is used for communication with whatever entity just connected. Note: this new socket is *not* a server socket.

3.6.2 Socket Properties

<code>SocketInformation(S)</code>

This routine returns the identifying information for the socket as a pair of tuples. Each tuple is a *<host, port>* pair — the first tuple gives the local information and the second gives the remote information. Note that this second tuple will be undefined for server sockets.

3.6.3 Socket Predicates

<code>IsServerSocket(S)</code>

Returns whether *S* is a server socket or not.

3.6.4 Socket I/O

Due to the nature of the network, it takes significant time to transmit data from one machine to another. Thus when a read request is begun it may take some time to complete, usually because the data to be read has not yet arrived. Also, data written to a socket may be broken up into smaller pieces for transmission, each of which may take different amounts of time to arrive. Thus, unlike files, there is no easy way to tell if there is still more data to be read; the current lack of data is no indicator as to whether more might arrive.

When a read request is made on a socket, the available data is returned. If no data is currently available, then the process waits until some does become available, and returns that. (It will also return if the socket has been closed and hence no more data can be transmitted.) It does not continue trying to read more data, as it cannot tell whether or not there is some “on the way”.

The upshot of all this is that care must be exercised as reads may return less data than is expected.

Read(*S* : *parameters*)

Max

RNGINTELT

Default : 0

Waits for data to become available for reading from *S* and then returns it as a string. If the parameter **Max** is set to a positive value then at most that many characters will be read. Note that less than **Max** characters may be returned, depending on the amount of currently available data.

If the socket has been closed then the special EOF marker string is returned.

ReadBytes(*S* : *parameters*)

Max

RNGINTELT

Default : 0

Waits for data to become available for reading from *S* and then returns it as a sequence of bytes (integers in the range 0..255). If the parameter **Max** is set to a positive value then at most that many bytes will be read. Note that less than **Max** bytes may be returned, depending on the amount of currently available data.

If the socket has been closed then the empty sequence is returned.

Write(*S*, *s*)

Writes the characters of the string *s* to the socket *S*.

WriteBytes(*S*, *Q*)

Writes the bytes in the byte sequence *Q* to the socket *S*. Each byte must be an integer in the range 0..255.

Example H3E12

Here is a trivial use of sockets to send a message from one MAGMA process to another running on the same machine. The first MAGMA process sets up a server socket and waits for another MAGMA to contact it.

```
> // First Magma process
> server := Socket(: LocalHost := "localhost");
> SocketInformation(server);
<localhost, 32794>
> S1 := WaitForConnection(server);
```

The second MAGMA process establishes a client socket connection to the first, writes a greeting message to it, and closes the socket.

```
> // Second Magma process
> S2 := Socket("localhost", 32794);
> SocketInformation(S2);
<localhost, 32795> <localhost, 32794>
> Write(S2, "Hello, other world!");
> delete S2;
```

The first MAGMA process is now able to continue; it reads and displays all data sent to it until the socket is closed.

```
> // First Magma process
> SocketInformation(S1);
<localhost, 32794> <localhost, 32795>
> repeat
>   msg := Read(S1);
>   msg;
> until IsEof(msg);
Hello, other world!
EOF
```

3.7 Interactive Input

<code>read identifier;</code>

<code>read identifier, prompt;</code>

This statement will cause MAGMA to assign to the given identifier the string of characters appearing (at run-time) on the following line. This allows the user to provide an input string at run-time. If the optional prompt is given (a string), that is printed first.

```
readi identifier;
```

```
readi identifier, prompt;
```

This statement will cause MAGMA to assign to the given identifier the literal integer appearing (at run-time) on the following line. This allows the user to specify integer input at run-time. If the optional prompt is given (a string), that is printed first.

3.8 Loading a Program File

```
load "filename";
```

Input the file with the name specified by the string. The file will be read in, and the text will be treated as MAGMA input. Tilde expansion of file names is allowed.

```
iload "filename";
```

(Interactive load.) Input the file with the name specified by the string. The file will be read in, and the text will be treated as MAGMA input. Tilde expansion of file names is allowed. In contrast to `load`, the user has the chance to interact as each line is read in:

As the line is read in, it is displayed and the system waits for user response. At this point, the user can skip the line (by moving “down”), edit the line (using the normal editing keys) or execute it (by pressing “enter”). If the line is edited, the new line is executed and the original line is presented again.

3.9 Saving and Restoring Workspaces

```
save "filename";
```

Copy all information present in the current MAGMA workspace onto a file specified by the string `"filename"`. The workspace is left intact, so executing this command does not interfere with the current computation.

```
restore "filename";
```

Copy a previously stored MAGMA workspace from the file specified by the string `"filename"` into central memory. Information present in the current workspace prior to the execution of this command will be lost. The computation can now proceed from the point it was at when the corresponding `save`-command was executed.

3.10 Logging a Session

SetLogFile(F)

Overwrite

BOOLELT

Default : false

Set the log file to be the file specified by the string F : all input and output will be sent to this log file as well as to the terminal. If a log file is already in use, it is closed and F is used instead. By using `SetLogFile(F: Overwrite := true)` the file F is emptied before input and output are written onto it. See also `HasOutputFile`.

UnsetLogFile()

Stop logging MAGMA's output.

SetEchoInput(b)

Set to true or false according to whether or not input from external files should also be sent to standard output.

3.11 Memory Usage

GetMemoryUsage()

Return the current memory usage of Magma (in bytes as an integer). This is the process data size, which does not include the executable code.

GetMaximumMemoryUsage()

Return the maximum memory usage of Magma (in bytes as an integer) which has been attained since last reset (see `ResetMaximumMemoryUsage`). This is the maximum process data size, which does not include the executable code.

ResetMaximumMemoryUsage()

Reset the value of the maximum memory usage of Magma to be the current memory usage of Magma (see `GetMaximumMemoryUsage`).

3.12 System Calls

Alarm(s)

A procedure which when used on UNIX systems, sends the signal SIGALRM to the MAGMA process after s seconds. This allows the user to specify that a MAGMA-process should self-destruct after a certain period.

ChangeDirectory(s)

Change to the directory specified by the string s . Tilde expansion is allowed.

GetCurrentDirectory()

Returns the current directory as a string.

Getpid()

Returns Magma's process ID (value of the Unix C system call `getpid()`).

Getuid()

Returns the user ID (value of the Unix C system call `getuid()`).

System(C)

Execute the system command specified by the string *C*. This is done by calling the C function `system()`.

This also returns the system command's return value as an integer. On most Unix systems, the lower 8 bits of this value give the process status while the next 8 bits give the value given by the command to the C function `exit()` (see the Unix manual entries for `system(3)` or `wait(2)`, for example). Thus one should normally divide the result by 256 to get the exit value of the program on success.

See also the `Pipe` intrinsic function.

%! *shell-command*

Execute the given command in the Unix shell then return to Magma. Note that this type of shell escape (contrary to the one using a `System` call) takes place entirely outside MAGMA and does not show up in MAGMA's history.

3.13 Creating Names

Sometimes it is necessary to create names for files from within MAGMA that will not clash with the names of existing files.

Tempname(P)

Given a prefix string *P*, return a unique temporary name derived from *P* (by use of the C library function `mktemp()`).

4 ENVIRONMENT AND OPTIONS

4.1 Introduction	95	SetMemoryLimit(n)	100
4.2 Command Line Options	95	GetMemoryLimit()	100
magma -b	95	SetNthreads(n)	100
magma -c <i>filename</i>	95	GetNthreads()	100
magma -d	96	SetOutputFile(F)	101
magma -n	96	UnsetOutputFile()	101
magma -q <i>name</i>	96	SetPath(s)	101
magma -r <i>workspace</i>	96	GetPath()	101
magma -s <i>filename</i>	96	SetPrintLevel(l)	101
magma -S <i>integer</i>	96	GetPrintLevel()	101
4.3 Environment Variables	97	SetPrompt(s)	101
MAGMA_STARTUP_FILE	97	GetPrompt()	101
MAGMA_PATH	97	SetQuitOnError(b)	101
MAGMA_MEMORY_LIMIT	97	SetRows(n)	101
MAGMA_LIBRARY_ROOT	97	GetRows()	101
MAGMA_LIBRARIES	97	GetTempDir()	102
MAGMA_SYSTEM_SPEC	97	SetTraceback(n)	102
MAGMA_USER_SPEC	97	GetTraceback()	102
MAGMA_HELP_DIR	97	SetSeed(s, c)	102
MAGMA_TEMP_DIR	97	GetSeed()	102
4.4 Set and Get	98	GetVersion()	102
SetAssertions(b)	98	SetViMode(b)	102
GetAssertions()	98	GetViMode()	102
SetAutoColumns(b)	98	4.5 Verbose Levels	102
GetAutoColumns()	98	SetVerbose(s, i)	102
SetAutoCompact(b)	98	SetVerbose(s, b)	102
GetAutoCompact()	98	GetVerbose(s)	102
SetBeep(b)	98	IsVerbose(s)	103
GetBeep()	98	IsVerbose(s, l)	103
SetColumns(n)	98	ListVerbose()	103
GetColumns()	98	ClearVerbose()	103
GetCurrentDirectory()	99	4.6 Other Information Procedures	103
SetEchoInput(b)	99	ShowMemoryUsage()	103
GetEchoInput()	99	ShowIdentifiers()	103
GetEnvironmentValue(s)	99	ShowValues()	103
GetEnv(s)	99	Traceback()	103
SetHistorySize(n)	99	ListSignatures(C)	103
GetHistorySize()	99	ListSignatures(F, C)	104
SetIgnorePrompt(b)	99	ListCategories()	104
GetIgnorePrompt()	99	ListTypes()	104
SetIgnoreSpaces(b)	99	4.7 History	104
GetIgnoreSpaces()	99	%p	104
SetIndent(n)	99	%pn	104
GetIndent()	99	%pn₁ n₂	104
SetLibraries(s)	100	%P	104
GetLibraries()	100	%Pn	104
SetLibraryRoot(s)	100	%Pn₁ n₂	104
GetLibraryRoot()	100	%s	104
SetLineEditor(b)	100	%sn	105
GetLineEditor()	100	%sn₁ n₂	105
SetLogFile(F)	100	%S	105
UnsetLogFile()	100	%Sn	105

<code>%Sn₁ n₂</code>	105	<code>;</code>	109
<code>%</code>	105	<code>,</code>	109
<code>%n</code>	105	<code>B</code>	109
<code>%n₁ n₂</code>	105	<code>b</code>	109
<code>%e</code>	105	<code>E</code>	109
<code>%en</code>	105	<code>e</code>	109
<code>%en₁ n₂</code>	105	<code>Fchar</code>	109
<code>%! shell-command</code>	105	<code>fchar</code>	109
4.8 The Magma Line Editor . . .	106	<code>h</code>	109
<code>SetViMode</code>	106	<code>H</code>	109
<code>SetViMode</code>	106	<code>l</code>	110
<i>4.8.1 Key Bindings (Emacs and VI mode)</i>	<i>106</i>	<code>L</code>	110
<code><Return></code>	106	<code>Tchar</code>	110
<code><Backspace></code>	106	<code>tchar</code>	110
<code><Delete></code>	106	<code>w</code>	110
<code><Tab></code>	106	<code>W</code>	110
<code><Ctrl>-A</code>	106	<code>A</code>	110
<code><Ctrl>-B</code>	106	<code>a</code>	110
<code><Ctrl>-C</code>	106	<code>C</code>	110
<code><Ctrl>-D</code>	106	<i>crange</i>	110
<code><Ctrl>-E</code>	107	<code>D</code>	110
<code><Ctrl>-F</code>	107	<i>drange</i>	110
<code><Ctrl>-H</code>	107	<code>I</code>	110
<code><Ctrl>-I</code>	107	<code>i</code>	110
<code><Ctrl>-J</code>	107	<code>j</code>	111
<code><Ctrl>-K</code>	107	<code>k</code>	111
<code><Ctrl>-L</code>	107	<code>P</code>	111
<code><Ctrl>-M</code>	107	<code>p</code>	111
<code><Ctrl>-N</code>	107	<code>R</code>	111
<code><Ctrl>-P</code>	107	<i>rchar</i>	111
<code><Ctrl>-U</code>	108	<code>S</code>	111
<code><Ctrl>-Vchar</code>	108	<code>s</code>	111
<code><Ctrl>-W</code>	108	<code>U</code>	111
<code><Ctrl>-X</code>	108	<code>u</code>	111
<code><Ctrl>-Y</code>	108	<code>X</code>	111
<code><Ctrl>-Z</code>	108	<code>x</code>	111
<code><Ctrl>-_</code>	108	<code>Y</code>	111
<code><Ctrl>-\<code>\</code></code>	108	<i>yrange</i>	111
<i>4.8.2 Key Bindings in Emacs mode only.</i>	<i>108</i>	4.9 The Magma Help System . . .	112
<code>Mb</code>	108	<code>SetHelpExternalBrowser(S, T)</code>	113
<code>MB</code>	108	<code>SetHelpExternalBrowser(S)</code>	113
<code>Mf</code>	108	<code>SetHelpUseExternalBrowser(b)</code>	113
<code>MF</code>	108	<code>SetHelpExternalSystem(s)</code>	113
<i>4.8.3 Key Bindings in VI mode only . . .</i>	<i>109</i>	<code>SetHelpUseExternalSystem(b)</code>	113
<code>0</code>	109	<code>GetHelpExternalBrowser()</code>	113
<code>\$</code>	109	<code>GetHelpExternalSystem()</code>	113
<code><Ctrl>-space</code>	109	<code>GetHelpUseExternal()</code>	113
<code>%</code>	109	<i>4.9.1 Internal Help Browser</i>	<i>113</i>

Chapter 4

ENVIRONMENT AND OPTIONS

4.1 Introduction

This chapter describes the environmental features of MAGMA, together with options which can be specified at start-up on the command line, or within MAGMA by the **Set-** procedures. The history and line-editor features of MAGMA are also described.

4.2 Command Line Options

When starting up MAGMA, various command-line options can be supplied, and a list of files to be automatically loaded can also be specified. These files may be specified by simply listing their names as normal arguments (i.e., without a **-** option) following the MAGMA command. For each such file name, a search for the specified file is conducted, starting in the current directory, and in directories specified by the environment variable **MAGMA_PATH** after that if necessary. It is also possible to have a *startup file*, in which one would usually store personal settings of parameters and variables. The startup file is specified by the **MAGMA_STARTUP_FILE** environment variable which should be set in the user's **.cshrc** file or similar. This environment variable can be overridden by the **-s** option, or cancelled by the **-n** option. The files specified by the arguments to MAGMA are loaded *after* the startup file. Thus the startup file is not cancelled by giving extra file arguments, which is what is usually desired.

MAGMA also allows one to set variables from the command line — if one of the arguments is of the form *var:=val*, where *var* is a valid identifier (consisting of letters, under-scores, or non-initial digits) and there is no space between *var* and the **:=**, then the variable *var* is assigned within MAGMA to the *string* value *val* at the point where that argument is processed. (Functions like **StringToInteger** should be used to convert the value to an object of another type once inside MAGMA.)

`magma -b`

If the **-b** argument is given to MAGMA, the opening banner and all other introductory messages are suppressed. The final “total time” message is also suppressed. This is useful when sending the whole output of a MAGMA process to a file so that extra removing of unwanted output is not needed.

`magma -c filename`

If the **-c** argument is given to MAGMA, followed by a filename, the filename is assumed to refer to a package source file and the package is compiled and MAGMA then exits straight away. This option is rarely needed since packages are automatically compiled when attached.

`magma -d`

If the `-d` option is supplied to MAGMA, the licence for the current `magmapassfile` is dumped. That is, the expiry date and the valid hostids are displayed. MAGMA then exits.

`magma -n`

If the `-n` option is supplied to MAGMA, any startup file specified by the environment variable `MAGMA_STARTUP_FILE` or by the `-s` option is cancelled.

`magma -q name`

If the `-q` option is supplied to MAGMA, then MAGMA operates in a special manner as a slave (with the given name) for the MPQS integer factorisation algorithm. Please see that function for more details.

`magma -r workspace`

If the `-r` option is supplied to MAGMA, together with a workspace file, that workspace is automatically restored by MAGMA when it starts up.

`magma -s filename`

If the `-s` option is supplied to MAGMA, the given filename is used for the startup file for MAGMA. This overrides the variable of the environment variable `MAGMA_STARTUP_FILE` if it has been set. This option should not be used (as it was before), for automatically loading files since that can be done by just listing them as arguments to the MAGMA process.

`magma -S integer`

When starting up MAGMA, it is possible to specify a seed for the generation of pseudo-random numbers. (Pseudo-random quantities are used in several MAGMA algorithms, and may also be generated explicitly by some intrinsics.) The seed should be in the range 0 to $(2^{32} - 1)$ inclusive. If `-S` is not followed by any number, or if the `-S` option is not used, MAGMA selects the seed itself.

Example H4E1

By typing the command

```
magma file1 x:=abc file2
```

MAGMA would start up, read the user's startup file specified by `MAGMA_STARTUP_FILE` if existent, then read the file `file1`, then assign the variable `x` to the string value `"abc"`, then read the file `file2`, then give the prompt.

4.3 Environment Variables

This section lists some environment variables used by MAGMA. These variables are set by an appropriate operating system command and are used to define various search paths and other run-time options.

MAGMA_STARTUP_FILE

The name of the default start-up file. It can be overridden by the `magma -s` command.

MAGMA_PATH

Search path for files that are loaded (a colon separated list of directories). It need not include directories for the libraries, just personal directories. This path is searched before the library directories.

MAGMA_MEMORY_LIMIT

Limit on the size of the memory that may be used by a MAGMA-session (in bytes).

MAGMA_LIBRARY_ROOT

The root directory for the MAGMA libraries (by supplying an absolute path name). From within MAGMA `SetLibraryRoot` and `GetLibraryRoot` can be used to change and view the value.

MAGMA_LIBRARIES

Give a list of MAGMA libraries (as a colon separated list of sub-directories of the library root directory). From within MAGMA `SetLibraries` and `GetLibraries` can be used to change and view the value.

MAGMA_SYSTEM_SPEC

The MAGMA system spec file containing the system packages automatically attached at start-up.

MAGMA_USER_SPEC

The personal user spec file containing the user packages automatically attached at start-up.

MAGMA_HELP_DIR

The root directory for the MAGMA help files.

MAGMA_TEMP_DIR

Optional variable containing the directory MAGMA is to use for temporary files. If not specified, this defaults to `/tmp` (on Unix-like systems) or the system-wide temporary directory (on Windows systems).

4.4 Set and Get

The **Set-** procedures allow the user to attach values to certain environment variables. The **Get-** functions enable one to obtain the current values of these variables.

SetAssertions(b)

GetAssertions()

Controls the checking of assertions (see the **assert** statement and related statements in the chapter on the language). Default is **SetAssertions(1)**. The relevant values are 0 for no checking at all, 1 for normal checks, 2 for debug checks and 3 for extremely stringent checking.

SetAutoColumns(b)

GetAutoColumns()

If enabled, the IO system will try to determine the number of columns in the window by using **ioctl()**; when a window change or a stop/cont occurs, the **Columns** variable (below) will be automatically updated. If disabled, the **Columns** variable will only be changed when explicitly done so by **SetColumns**. Default is **SetAutoColumns(true)**.

SetAutoCompact(b)

GetAutoCompact()

Control whether automatic compaction is performed. Normally the memory manager of MAGMA will compact all of its memory between each statement at the top level. This removes fragmentation and reduces excessive memory usage. In some very rare situations, the compactations may become very slow (one symptom is that an inordinate pause occurs between prompts when only a trivial operation or nothing is done). In such cases, turning the automatic compaction off may help (at the cost of possibly more use of memory). Default is **SetAutoCompact(true)**.

SetBeep(b)

GetBeep()

Controls ‘beeps’. Default is **SetBeep(true)**.

SetColumns(n)

GetColumns()

Controls the number of columns used by the IO system. This affects the line editor and the output system. (As explained above, if **AutoColumns** is on, this variable will be automatically determined.) The number of columns will determine how words are wrapped. If set to 0, word wrap is not performed. The default value is **SetColumns(80)** (unless **SetAutoColumns(true)**).

GetCurrentDirectory()

Returns the current directory as a string. (Use `ChangeDirectory(s)` to change the working directory.)

SetEchoInput(b)

GetEchoInput()

Set to `true` or `false` according to whether or not input from external files should also be sent to standard output.

GetEnvironmentValue(s)

GetEnv(s)

Returns the value of the external environment variable *s* as a string.

SetHistorySize(n)

GetHistorySize()

Controls the number of lines saved in the history. If the number is set to 0, no history is preserved.

SetIgnorePrompt(b)

GetIgnorePrompt()

Controls the option to ignore the prompt to allow the pasting of input lines back in. If enabled, any leading `'>'` characters (possibly separated by white space) are ignored by the history system when the input file is a terminal, *unless* the line consists of the `'>'` character alone (without a following space), which could not come from a prompt since in a prompt a space or another character follows a `'>'`. Default is `SetIgnorePrompt(false)`.

SetIgnoreSpaces(b)

GetIgnoreSpaces()

Controls the option to ignore spaces when searching in the line editor. If the user moves up or down in the line editor using `<Ctrl>-P` or `<Ctrl>-N` (see the line editor key descriptions) and if the cursor is not at the beginning of the line, a search is made forwards or backwards, respectively, to the first line which starts with the same string as the string consisting of all the characters before the cursor. While doing the search, spaces are ignored if and only if this option is on (value `true`). Default is `SetIgnoreSpaces(true)`.

SetIndent(n)

GetIndent()

Controls the indentation level for formatting output. The default is `SetIndent(4)`.

SetLibraries(s)

GetLibraries()

Controls the MAGMA library directories via environment variable `MAGMA_LIBRARIES`. The procedure `SetLibraries` takes a string, which will be taken as the (colon-separated) list of sub-directories in the library root directory for the libraries; the function `GetLibraryRoot` returns the current value as a string. These directories will be searched when you try to load a file; note however that first the directories indicated by the current value of your path environment variable `MAGMA_PATH` will be searched. See `SetLibraryRoot` for the root directory.

SetLibraryRoot(s)

GetLibraryRoot()

Controls the root directory for the MAGMA libraries, via the environment variable `MAGMA_LIBRARY_ROOT`. The procedure `SetLibraryRoot` takes a string, which will be the absolute pathname for the root of the libraries; the function `GetLibraryRoot` returns the current value as a string. See also `SetLibraries`

SetLineEditor(b)

GetLineEditor()

Controls the line editor. Default is `SetLineEditor(true)`.

SetLogFile(F)

Overwrite

BOOLELT

Default : false

UnsetLogFile()

Procedure. Set the log file to be the file specified by the string *F*: all input and output will be sent to this log file as well as to the terminal. If a log file is already in use, it is closed and *F* is used instead. The parameter `Overwrite` can be used to indicate that the file should be truncated before writing input and output on it; by default the file is appended.

SetMemoryLimit(n)

GetMemoryLimit()

Set the limit (in bytes) of the memory which the memory manager will allocate (no limit if 0). Default is `SetMemoryLimit(0)`.

SetNthreads(n)

GetNthreads()

Set the number of threads to be used in multi-threaded algorithms to be *n*, if POSIX threads are enabled in this version of Magma. Currently, this affects the coding theory minimum weight algorithm (`MinimumWeight`) and the F_4 Gröbner basis algorithm for medium-sized primes (`Groebner`).

SetOutputFile(F)

Overwrite

BOOLELT

Default : false

UnsetOutputFile()

Start/stop redirecting all MAGMA output to a file (specified by the string *F*). The parameter **Overwrite** can be used to indicate that the file should be truncated before writing output on it.

SetPath(s)

GetPath()

Controls the path by which the searching of files is done. The path consists of a colon separated list of directories which are searched in order (“.” implicitly assumed at the front). Tilde expansion is done on each directory. (May be overridden by the environment variable MAGMA_PATH.)

SetPrintLevel(l)

GetPrintLevel()

Controls the global printing level, which is one of "Minimal", "Magma", "Maximal", "Default". Default is SetPrintLevel("Default").

SetPrompt(s)

GetPrompt()

Controls the terminal prompt (a string). Expansion of the following % escapes occurs:

%% The character %

%h The current history line number.

%S The parser ‘state’: when a new line is about to be read while the parser has only seen incomplete statements, the state consists of a stack of words like “if”, “while”, indicating the incomplete statements.

%s Like %S except that only the topmost word is displayed.

Default is SetPrompt("%S> ").

SetQuitOnError(b)

Set whether Magma should quit on any error to *b*. If *b* is **true**, MAGMA will completely quit when any error (syntax, runtime, etc.) occurs. Default is SetQuitOnError(false).

SetRows(n)

GetRows()

Controls the number of rows in a page used by the IO system. This affects the output system. If set to 0, paging is not performed. Otherwise a prompt is given after the given number of rows for a new page. The default value is SetRows(0).

GetTempDir()

Returns the directory MAGMA uses for storing temporary files. May be influenced on startup via the MAGMA_TEMP_DIR environment variable (see Section 4.3).

SetTraceback(n)

GetTraceback()

Controls whether MAGMA should produce a traceback of user function calls before each error message. The default value is `SetTraceback(true)`.

SetSeed(s, c)

GetSeed()

Controls the initialization seed and step number for pseudo-random number generation. For details, see the section on random object generation in the chapter on statements and expressions.

GetVersion()

Return integers x , y and z such the current version of MAGMA is $Vx.y-z$.

SetViMode(b)

GetViMode()

Controls the type of line editor used: Emacs (`false`) or VI style. Default is `SetViMode(false)`.

4.5 Verbose Levels

By turning verbose printing on for certain modules within MAGMA, some information on computations that are performed can be obtained. For each option, the verbosity may have different levels. The default is level 0 for each option.

There are also 5 slots available for user-defined verbose flags. The flags can be set in user programs by `SetVerbose("User n ", true)` where n should be one of 1, 2, 3, 4, 5, and the current setting is returned by `GetVerbose("User n ")`.

SetVerbose(s, i)

SetVerbose(s, b)

Set verbose level for s to be level i or b . Here the argument s must be a string. The verbosity may have different levels. An integer i for the second argument selects the appropriate level. A second argument i of 0 or b of `false` means no verbosity. A boolean value for b of `true` for the second argument selects level 1. (See above for the valid values for the string s).

GetVerbose(s)

Return the value of verbose flag s as an integer. (See above for the valid values for the string s).

IsVerbose(*s*)

Return the whether the value of verbose flag *s* is non-zero. (See above for the valid values for the string *s*).

IsVerbose(*s*, *l*)

Return the whether the value of verbose flag *s* is greater than or equal to *l*. (See above for the valid values for the string *s*).

ListVerbose()

List all verbose flags. That is, print each verbose flag and its maximal level.

ClearVerbose()

Clear all verbose flags. That is, set the level for all verbose flags to 0.

4.6 Other Information Procedures

The following procedures print information about the current state of MAGMA.

ShowMemoryUsage()

(Procedure.) Show MAGMA's current memory usage.

ShowIdentifiers()

(Procedure.) List all identifiers that have been assigned to.

ShowValues()

(Procedure.) List all identifiers that have been assigned to with their values.

Traceback()

(Procedure.) Display a traceback of the current Magma function invocations.

ListSignatures(*C*)

Isa	BOOLELT	<i>Default : true</i>
Search	MONSTGELT	<i>Default : "Both"</i>
ShowSrc	BOOLELT	<i>Default : false</i>

List all intrinsic functions, procedures and operators having objects from category *C* among their arguments or return values. The parameter **Isa** may be set to false so that any categories which *C* inherit from are not considered. The parameter **Search**, with valid string values **Both**, **Arguments**, **ReturnValues**, may be used to specify whether the arguments, the return values, or both, are considered (default both). **ShowSrc** can be used to see where package intrinsics are defined. Use **ListCategories** for the names of the categories.

ListSignatures(F , C)

Isa	BOOLELT	<i>Default</i> : true
Search	MONSTGELT	<i>Default</i> : “Both”
ShowSrc	BOOLELT	<i>Default</i> : false

Given an intrinsic F and category C , list all signatures of F which match the category C among their arguments or return values. The parameters are as for the previous procedure.

ListCategories()

ListTypes()

Procedure to list the (abbreviated) names for all available categories in MAGMA.

4.7 History

Magma provides a history system which allows the recall and editing of previous lines. The history system is invoked by typing commands which begin with the history character ‘%’. Currently, the following commands are available.

%p

List the contents of the history buffer. Each line is preceded by its history line number.

%pn

List the history line n in %p format.

%pn₁ n_2

List the history lines in the range n_1 to n_2 in %p format.

%P

List the contents of the history buffer. The initial numbers are *not* printed.

%Pn

List the history line n in %P format.

%Pn₁ n_2

List the history lines in the range n_1 to n_2 in %P format.

%s

List the contents of the history buffer with an initial statement for each line to reset the random number seed to the value it was just before the line was executed. This is useful when one wishes to redo a computation using exactly the same seed as before but does not know what the seed was at the time.

%s*n*

Print the history line *n* in %s format.

%s*n*₁ *n*₂

Print the history lines in the range *n*₁ to *n*₂ in %s format.

%S

As for %s except that the statement to set the seed is only printed if the seed has changed since the previous time it was printed. Also, it is not printed if it would appear in the middle of a statement (i.e., the last line did not end in a semicolon).

%S*n*

Print the history line *n* in %S format.

%S*n*₁ *n*₂

Print the history lines in the range *n*₁ to *n*₂ in %S format.

%

Reenter the last line into the input stream.

%*n*

Reenter the line specified by line number *n* into the input stream.

%*n*₁ *n*₂

Reenter the history lines in the range *n*₁ to *n*₂ into the input stream.

%e

Edit the last line. The editor is taken to be the value of the EDITOR environment variable if it is set, otherwise “/bin/ed” is used. If after the editor has exited the file has not been changed then nothing is done. Otherwise the contents of the new file are reentered into the input stream.

%e*n*

Edit the line specified by line number *n*.

%e*n*₁ *n*₂

Edit the history lines in the range *n*₁ to *n*₂.

%! *shell-command*

Execute the given command in the Unix shell then return to Magma.

4.8 The Magma Line Editor

Magma provides a line editor with both Emacs and VI style key bindings. To enable the VI style of key bindings, type

```
SetViMode(true)
```

and type

```
SetViMode(false)
```

to revert to the Emacs style of key bindings. By default ViMode is **false**; that is, the Emacs style is in effect.

Many key bindings are the same in both Emacs and VI style. This is because some VI users like to be able to use some Emacs keys (like **<Ctrl>-P**) as well as the VI command keys. Thus key bindings in Emacs which are not used in VI insert mode can be made common to both.

4.8.1 Key Bindings (Emacs and VI mode)

<Ctrl>-key means hold down the Control key and press key.

```
<Return>
```

Accept the line and print a new line. This works in any mode.

```
<Backspace>
```

```
<Delete>
```

Delete the previous character.

```
<Tab>
```

Complete the word which the cursor is on or just after. If the word doesn't have a unique completion, it is first expanded up to the common prefix of all the possible completions. An immediately following Tab key will list all of the possible completions. Currently completion occurs for system functions and procedures, parameters, reserved words, and user identifiers.

```
<Ctrl>-A
```

Move to the beginning of the line ("alpha" = "beginning").

```
<Ctrl>-B
```

Move back a character ("back").

```
<Ctrl>-C
```

Abort the current line and start a new line.

```
<Ctrl>-D
```

On an empty line, send a EOF character (i.e., exit at the top level of the command interpreter). If at end of line, list the completions. Otherwise, delete the character under the cursor (“delete”).

`<Ctrl>-E`

Move to the end of the line (“end”).

`<Ctrl>-F`

Move forward a character (“forward”).

`<Ctrl>-H`

Same as Backspace.

`<Ctrl>-I`

Same as Tab.

`<Ctrl>-J`

Same as Return.

`<Ctrl>-K`

Delete all characters from the cursor to the end of the line (“kill”).

`<Ctrl>-L`

Redraw the line on a new line (helpful if the screen gets wrecked by programs like “write”, etc.).

`<Ctrl>-M`

Same as `<Return>`.

`<Ctrl>-N`

Go forward a line in the history buffer (“next”). If the cursor is not at the beginning of the line, go forward to the first following line which starts with the same string (ignoring spaces iff the ignore spaces option is on — see `SetIgnoreSpaces`) as the string consisting of all the characters before the cursor. Also, if `<Ctrl>-N` is typed initially at a new line and the last line entered was actually a recall of a preceding line, then the next line after that is entered into the current buffer. Thus to repeat a sequence of lines (with minor modifications perhaps to each), then one only needs to go back to the first line with `<Ctrl>-P` (see below), press `<Return>`, then successively press `<Ctrl>-N` followed by `<Return>` for each line.

`<Ctrl>-P`

Go back a line in the history buffer (“previous”). If the cursor is not at the beginning of the line, go back to the first preceding line which starts with the same string (ignoring spaces iff the ignore spaces option is on — see `SetIgnoreSpaces`) as the string consisting of all the characters before the cursor. For example, typing at a

new line `x:=` and then `<Ctrl>-P` will go back to the last line which assigned `x` (if a line begins with, say, `x :=`, it will also be taken).

`<Ctrl>-U`

Clear the whole of the current line.

`<Ctrl>-Vchar`

Insert the following character literally.

`<Ctrl>-W`

Delete the previous word.

`<Ctrl>-X`

Same as `<Ctrl>-U`.

`<Ctrl>-Y`

Insert the contents of the yank-buffer before the character under the cursor.

`<Ctrl>-Z`

Stop MAGMA.

`<Ctrl>-_`

Undo the last change.

`<Ctrl>-\`

Immediately quit MAGMA.

On most systems the arrow keys also have the obvious meaning.

4.8.2 Key Bindings in Emacs mode only

`Mkey` means press the Meta key and then `key`. (At the moment, the Meta key is only the Esc key.)

`Mb`

`MB`

Move back a word (“Back”).

`Mf`

`MF`

Move forward a word (“Forward”).

4.8.3 Key Bindings in VI mode only

In the VI mode, the line editor can also be in two modes: the insert mode and the command mode. When in the insert mode, any non-control character is inserted at the current cursor position. The command mode is then entered by typing the Esc key. In the command mode, various commands are given a *range* giving the extent to which they are performed. The following ranges are available:

0

Move to the beginning of the line.

\$

Move to the end of the line.

<Ctrl>-space

Move to the first non-space character of the line.

%

Move to the matching bracket. (Bracket characters are (,), [,], {, }, <, and >.)

;

Move to the next character. (See ‘F’, ‘f’, ‘T’, and ‘t’.)

,

Move to the previous character. (See ‘F’, ‘f’, ‘T’, and ‘t’.)

B

Move back a space-separated word (“Back”).

b

Move back a word (“back”).

E

Move forward to the end of the space-separated word (“End”).

e

Move forward to the end of the word (“end”).

F*char*

Move back to the first occurrence of *char*.

f*char*

Move forward to the first occurrence of *char*.

h

H

Move back a character (<Ctrl>-H = Backspace).

l

L

Move back a character (<Ctrl>-L = forward on some keyboards).

Tchar

Move back to just after the first occurrence of *char*.

tchar

Move forward to just before the first occurrence of *char*.

w

Move forward a space-separated word (“Word”).

W

Move forward a word (“word”).

Any range may be preceded by a number to multiply to indicate how many times the operation is done. The VI-mode also provides the *yank-buffer*, which contains characters which are deleted or “yanked” – see below.

The following keys are also available in command mode:

A

Move to the end of the line and change to insert mode (“Append”).

a

Move forward a character (if not already at the end of the line) and change to insert mode (“append”).

C

Delete all the characters to the end of line and change to insert mode (“Change”).

crange

Delete all the characters to the specified range and change to insert mode (“change”).

D

Delete all the characters to the end of line (“Delete”).

drange

Delete all the characters to the specified range (“delete”).

I

Move to the first non-space character in the line and change to insert mode (“Insert”).

i

Change to insert mode (“insert”).

j

Go forward a line in the history buffer (same as <Ctrl>-N).

k

Go back a line in the history buffer (same as <Ctrl>-P).

P

Insert the contents of the yank-buffer before the character under the cursor.

p

Insert the contents of the yank-buffer before the character after the cursor.

R

Enter over-type mode: typed characters replace the old characters under the cursor without insertion. Pressing Esc returns to the command mode.

r*char*

Replace the character the cursor is over with *char*.

S

Delete the whole line and change to insert mode (“Substitute”).

s

Delete the current character and change to insert mode (“substitute”).

U

u

Undo the last change.

X

Delete the character to the left of the cursor.

x

Delete the character under the cursor.

Y

“Yank” the whole line - i.e., copy the whole line into the yank-buffer (“Yank”).

y*range*

Copy all characters from the cursor to the specified range into the yank-buffer (“yank”).

4.9 The Magma Help System

Magma provides extensive online help facilities that can be accessed in different ways. The easiest way to access the documentation is by typing:

```
magmahelp
```

Which should start some browser (usually netscape) on the main page of the MAGMA documentation.

The easiest way to get some information about any MAGMA intrinsic is by typing: (Here we assume you to be interested in `FundamentalUnit`)

```
> FundamentalUnit;
```

Which now will list all signatures for this intrinsic (i.e. all known ways to use this function):

```
> FundamentalUnit;
Intrinsic 'FundamentalUnit'
Signatures:
  (<FldQuad> K) -> FldQuadElt
  (<RngQuad> O) -> RngQuadElt
    The fundamental unit of K or O
  (<RngQuad> R) -> RngQuadElt
    Fundamental unit of the real quadratic order.
```

Next, to get more detailed information, try

```
> ?FundamentalUnit
```

But now several things could happen depending on the installation. Using the default, you get

```
=====
PATH: /magma/ring-field-algebra/quadratic/operation/\
      class-group/FundamentalUnit
KIND: Intrinsic
=====
FundamentalUnit(K) : FldQuad -> FldQuadElt
FundamentalUnit(O) : RngQuad -> RngQuadElt
  A generator for the unit group of the order O or the
maximal order
  of the quadratic field K.
=====
```

Second, a WWW-browser could start on the part of the online help describing your function (or at least the index of the first character). Third, some arbitrary program could be called to provide you with the information.

If `SetVerbose("Help", true);` is set, MAGMA will show the exact command used and the return value obtained.

SetHelpExternalBrowser(S, T)

SetHelpExternalBrowser(S)

Defines the external browser to be used if `SetHelpUseExternalBrowser(true)` is in effect. The string has to be a valid command taking exactly one argument (`%s`) which will be replaced by a URL. In case two strings are provided, the second defines a fall-back system. Typical use for this is to first try to use an already running browser and if this fails, start a new one.

SetHelpUseExternalBrowser(b)

Tells MAGMA to actually use (or stop to use) the external browser. If both `SetHelpUseExternalSystem` and `SetHelpUseExternalBrowser` are set to `true`, the assignment made last will be effective.

SetHelpExternalSystem(s)

This will tell MAGMA to use a user defined external program to access the help. The string has to contain exactly one `%s` which will be replaced by the argument to `?`. The resulting string must be a valid command.

SetHelpUseExternalSystem(b)

Tells MAGMA to actually use (or stop to use) the external help system. If both `SetHelpUseExternalSystem` and `SetHelpUseExternalBrowser` are set to `true`, the assignment made last will be effective.

GetHelpExternalBrowser()

Returns the currently used command strings.

GetHelpExternalSystem()

Returns the currently used command string.

GetHelpUseExternal()

The first value is the currently used value from `SetHelpUseExternalBrowser`, the second reflects `SetHelpUseExternalSystem`.

4.9.1 Internal Help Browser

MAGMA has a very powerful internal help-browser that can be entered with

> ??

5 MAGMA SEMANTICS

5.1 Introduction	117	5.6.2 The ‘first use’ Rule	125
5.2 Terminology	117	5.6.3 Identifier Classes	126
5.3 Assignment	118	5.6.4 The Evaluation Process Revisited	126
5.4 Uninitialized Identifiers	118	5.6.5 The ‘single use’ Rule	127
5.5 Evaluation in Magma	119	5.7 Procedure Expressions	127
5.5.1 Call by Value Evaluation	119	5.8 Reference Arguments	129
5.5.2 Magma’s Evaluation Process	120	5.9 Dynamic Typing	130
5.5.3 Function Expressions	121	5.10 Traps for Young Players	131
5.5.4 Function Values Assigned to Identifiers	122	5.10.1 Trap 1	131
5.5.5 Recursion and Mutual Recursion	122	5.10.2 Trap 2	131
5.5.6 Function Application	123	5.11 Appendix A: Precedence	133
5.5.7 The Initial Context	124	5.12 Appendix B: Reserved Words	134
5.6 Scope	124		
5.6.1 Local Declarations	125		

Chapter 5

MAGMA SEMANTICS

5.1 Introduction

This chapter describes the semantics of MAGMA (how expressions are evaluated, how identifiers are treated, etc.) in a fairly informal way. Although some technical language is used (particularly in the opening few sections) the chapter should be easy and essential reading for the non-specialist. The chapter is descriptive in nature, describing how MAGMA works, with little attempt to justify why it works the way it does. As the chapter proceeds, it becomes more and more precise, so while early sections may gloss over or omit things for the sake of simplicity and learnability, full explanations are provided later.

It is assumed that the reader is familiar with basic notions like a function, an operator, an identifier, a type ...

And now for some buzzwords: MAGMA is an imperative, call by value, statically scoped, dynamically typed programming language, with an essentially functional subset. The remainder of the chapter explains what these terms mean, and why a user might want to know about such things.

5.2 Terminology

Some terminology will be useful. It is perhaps best to read this section only briefly, and to refer back to it when necessary.

The term *expression* will be used to refer to a textual entity. The term *value* will be used to refer to a run-time value denoted by an expression. To understand the difference between an expression and a value consider the expressions `1+2` and `3`. The expressions are textually different but they denote the same value, namely the integer 3.

A *function expression* is any expression of the form `function ... end function` or of the form `func< ... | ... >`. The former type of function expression will be said to be *in the statement form*, the latter *in the expression form*. A *function value* is the run-time value denoted by a function expression. As with integers, two function expressions can be textually different while denoting the same (i.e., extensionally equal) function value. To clearly distinguish function values from function expressions, the notation `FUNC(... : ...)` will be used to describe function *values*.

The *formal arguments* of a function in the statement form are the identifiers that appear between the brackets just after the `function` keyword, while for a function in the expression form they are the identifiers that appear before the `|`. The *arguments* to a function are the expressions between the brackets when a function is applied.

The *body* of a function in the statement form is the statements after the formal arguments. The body of a function in the expression form is the expression after the `|` symbol.

An identifier is said to occur *inside* a function expression when it occurs textually anywhere in the body of a function.

5.3 Assignment

An assignment is an association of an identifier to a *value*. The statement,

```
> a := 6;
```

establishes an association between the identifier *a* and the value 6 (6 is said to be *the value of a*, or to be *assigned to a*). A collection of such assignments is called a *context*.

When a value *V* is assigned to an identifier *I* one of two things happens:

- (1) if *I* has not been previously assigned to, it is added to the current context and associated with *V*. *I* is said to be *declared* when it is assigned to for the first time.
- (2) if *I* has been previously assigned to, the value associated with *I* is changed to *V*. *I* is said to be *re-assigned*.

The ability to assign and re-assign to identifiers is why MAGMA is called an *imperative* language.

One very important point about assignment is illustrated by the following example. Say we type,

```
> a := 6;  
> b := a+7;
```

After executing these two lines the context is [(a,6), (b,13)]. Now say we type,

```
> a := 0;
```

The context is now [(a,0), (b,13)]. Note that changing the value of *a* does *not* change the value of *b* because *b*'s value is statically determined at the point where it is assigned. Changing *a* does *not* produce the context [(a,0), (b,7)].

5.4 Uninitialized Identifiers

Before executing a piece of code MAGMA attempts to check that it is semantically well formed (i.e., that it will execute without crashing). One of the checks MAGMA makes is to check that an identifier is declared (and thus initialized) before it is used in an expression. So, for example assuming *a* had not been previously declared, then before executing either of the following lines MAGMA will raise an error:

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

MAGMA can determine that execution of either line will cause an error since *a* has no assigned value. The user should be aware that the checks made for semantic well-formedness are necessarily not exhaustive!

There is one important rule concerning uninitialized identifiers and assignment. Consider the line,

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

Now if a had been previously declared then this is re-assignment of a . If not then it is an error since a on the right hand side of the $:=$ has no value. To catch this kind of error MAGMA checks the expression on the right hand side of the $:=$ for semantic well formedness *before* it declares the identifiers on the left hand side of the $:=$. Put another way the identifiers on the left hand side are not considered to be declared in the right hand side, *unless* they were declared previously.

5.5 Evaluation in Magma

Evaluation is the process of computing (or constructing) a value from an expression. For example the value 3 can be computed from the expression $1+2$. Computing a value from an expression is also known as *evaluating an expression*.

There are two aspects to evaluation, namely *when* and *how* it is performed. This section discusses these two aspects.

5.5.1 Call by Value Evaluation

MAGMA employs call by value evaluation. This means that the arguments to a function are evaluated before the function is applied to those arguments. Assume f is a function value. Say we type,

```
> r := f( 6+7, true or false );
```

MAGMA evaluates the two arguments to 13 and true respectively, *before* applying f .

While knowing the exact point at which arguments are evaluated is not usually very important, there are cases where such knowledge is crucial. Say we type,

```
> f := function( n, b )
>       if b then return n else return 1;
> end function;
```

and we apply f as follows

```
> r := f( 4/0, false );
```

MAGMA treats this as an error since the $4/0$ is evaluated, and an error produced, *before* the function f is applied.

By contrast some languages evaluate the arguments to a function only if those arguments are encountered when executing the function. This evaluation process is known as call by name evaluation. In the above example r would be set to the value 1 and the expression $4/0$ would never be evaluated because b is false and hence the argument n would never be encountered.

Operators like $+$ and $*$ are treated as infix functions. So

```
> r := 6+7;
```

is treated as the function application,

```
> r := '+'(6,7);
```

Accordingly all arguments to an operator are evaluated before the operator is applied.

There are three operators, 'select', 'and' and 'or' that are exceptions to this rule and are thus not treated as infix functions. These operators use call by name evaluation and only evaluate arguments as need be. For example if we type,

```
> false and (4/0 eq 6);
```

MAGMA will reply with the answer false since MAGMA knows that `false and X` for all X is false.

5.5.2 Magma's Evaluation Process

Let us examine more closely how MAGMA evaluates an expression as it will help later in understanding more complex examples, specifically those using functions and maps. To evaluate an expression MAGMA proceeds by a process of identifier substitution, followed by simplification to a canonical form. Specifically expression evaluation proceeds as follows,

- (1) replace each identifier in the expression by its value in the current context.
- (2) simplify the resultant *value* to its canonical form.

The key point here is that the replacement step takes an expression and yields an unsimplified *value*! A small technical note: to avoid the problem of having objects that are part expressions, part values, all substitutions in step 1 are assumed to be done simultaneously for all identifiers in the expression. The examples in this chapter will however show the substitutions being done in sequence and will therefore be somewhat vague about what exactly these hybrid objects are!

To clarify this process assume that we type,

```
> a := 6;
```

```
> b := 7;
```

producing the context [(a,6), (b,7)]. Now say we type,

```
> c := a+b;
```

This produces the context [(a,6), (b,7), (c,13)]. By following the process outlined above we can see how this context is calculated. The steps are,

- (1) replace `a` in the expression `a+b` by its value in the current context giving `6+b`.
- (2) replace `b` in `6+b` by its value in the current context giving `6+7`.
- (3) simplify `6+7` to `13`

The result value of 13 is then assigned to `c` giving the previously stated context.

5.5.3 Function Expressions

MAGMA's evaluation process might appear to be an overly formal way of stating the obvious about calculating expression values. This formality is useful, however when it comes to function (and map) expressions.

Functions in MAGMA are first class values, meaning that MAGMA treats function values just like it treats any other type of value (e.g., integer values). A function value may be passed as an argument to another function, may be returned as the result of a function, and may be assigned to an identifier in the same way that any other type of value is. Most importantly however function expressions are evaluated *exactly* as are all other expressions. The fact that MAGMA treats functions as first class values is why MAGMA is said to have an essentially functional subset.

Take the preceding example. It was,

```
> a := 6;
> b := 7;
> c := a+b;
```

giving the context $[(a,6), (b,7), (c,13)]$. Now say I type,

```
> d := func< n | a+b+c+n >;
```

MAGMA uses the same process to evaluate the function expression `func< n | a+b+c+n >` on the right hand side of the assignment `d := ...` as it does to evaluate expression `a+b` on the right hand side of the assignment `c := ...`. So evaluation of this function expression proceeds as follows,

- (1) replace `a` in the expression `func< n | a+b+c+n >` by its value in the current context giving `func< n | 6+b+c+n >`.
- (2) replace `b` in `func< n | 6+b+c+n >` by its value in the current context giving `func< n | 6+7+c+n >`.
- (3) replace `c` in `func< n | 6+7+c+n >` by its value in the current context giving `FUNC(n : 6+7+13+n)`
- (4) simplify the resultant *value* `FUNC(n : 6+7+13+n)` to the *value* `FUNC(n : 26+n)`.

Note again that the process starts with an expression and ends with a value, and that throughout the function expression is evaluated just like any other expression. A small technical point: function simplification may not in fact occur but the user is guaranteed that the simplification process will at least produce a function extensionally equal to the function in its canonical form.

The resultant function value is now assigned to `d` just like any other type of value would be assigned to an identifier yielding the context $[(a,6), (b,7), (c,8), (d, \text{FUNC}(n : 26+n))]$.

As a final point note that changing the value of any of `a`, `b`, and `c`, does *not* change the value of `d`!

5.5.4 Function Values Assigned to Identifiers

Say we type the following,

```
> a := 1;
> b := func< n | a >;
> c := func< n | b(6) >;
```

The first line leaves a context of the form $[(a, 1)]$. The second line leaves a context of the form $[(a, 1), (b, \text{FUNC}(n : 1))]$.

The third line is evaluated as follows,

- (1) replace the value of **b** in the expression `func< n | b(6) >` by its value in the current context giving `FUNC(n : (FUNC(n : 1))(6))`.
- (2) simplify this value to `FUNC(n : 1)` since applying the function value `FUNC(n : 1)` to the argument 6 always yields 1.

The key point here is that identifiers whose assigned value is a function value (in this case *b*), are treated exactly like identifiers whose assigned value is any other type of value.

Now look back at the example at the end of the previous section. One step in the series of replacements was not mentioned. Remember that `+` is treated as a shorthand for an infix function. So `a+b` is equivalent to `'+'(a,b)`. `+` is an identifier (assigned a function value), and so in the replacement part of the evaluation process there should have been an extra step, namely,

- (4) replace `+` in `func< n : 6+7+13+n >` by its value in the current context giving `FUNC(n : A(A(A(6,7), 13), n))`.
- (5) simplify the resultant value to `FUNC(n : A(26, n))`. where *A* is the (function) value that is the addition function.

5.5.5 Recursion and Mutual Recursion

How do we write recursive functions? Function expressions have no names so how can a function expression apply *itself* to do recursion?

It is tempting to say that the function expression could recurse by using the identifier that the corresponding function value is to be assigned to. But the function value may not be being assigned at all: it may simply be being passed as an actual argument to some other function value. Moreover even if the function value were being assigned to an identifier the function expression cannot use that identifier because the assignment rules say that the identifiers on the left hand side of the `:=` in an assignment statement are not considered declared on the right hand side, unless they were previously declared.

The solution to the problem is to use the `$$` pseudo-identifier. `$$` is a placeholder for the function value denoted by the function expression inside which the `$$` occurs. An example serves to illustrate the use of `$$`. A recursive factorial function can be defined as follows,

```
> factorial := function(n)
>   if n eq 1 then
>     return 1;
```

```

>     else
>         return n * $$ (n-1);
>     end if;
> end function;

```

Here `$$` is a placeholder for the function value that the function expression `function(n) if n eq ... end function` denotes (those worried that the denoted function value appears to be defined in terms of itself are referred to the fixed point semantics of recursive functions in any standard text on denotational semantics).

A similar problem arises with mutual recursion where a function value f applies another function value g , and g likewise applies f . For example,

```

> f := function(...) ... a := g(...); ... end function;
> g := function(...) ... b := f(...); ... end function;

```

Again MAGMA's evaluation process appears to make this impossible, since to construct f MAGMA requires a value for g , but to construct g MAGMA requires a value for f . Again there is a solution. An identifier can be declared 'forward' to inform MAGMA that a function expression for the forward identifier will be supplied later. The functions f and g above can therefore be declared as follows,

```

> forward f, g;
> f := function(...) ... a := g(...); ... end function;
> g := function(...) ... b := f(...); ... end function;

```

(strictly speaking it is only necessary to declare g forward as the value of f will be known by the time the function expression `function(...) ... b := f(...); ... end function` is evaluated).

5.5.6 Function Application

It was previously stated that MAGMA employs call by value evaluation, meaning that the arguments to a function are evaluated before the function is applied. This subsection discusses how functions are applied once their arguments have been evaluated.

Say we type,

```

> f := func< a, b | a+b >;

```

producing the context `[(f, FUNC(a, b : a+b))]`.

Now say we apply f by typing,

```

> r := f( 1+2, 6+7 ).

```

How is the value to be assigned to r calculated? If we follow the evaluation process we will reach the final step which will say something like,

“simplify `(FUNC(a, b : A(a,b)))(3,13)` to its canonical form”

where as before A is the value that is the addition function. How is this simplification performed? How are function values applied to actual function arguments to yield result

values? Not unsurprisingly the answer is via a process of substitution. The evaluation of a function application proceeds as follows,

- (1) replace each formal argument in the function body by the corresponding actual argument.
- (2) simplify the function body to its canonical form.

Exactly what it means to “simplify the function body ...” is intentionally left vague as the key point here is the process of replacing formal arguments by values in the body of the function.

5.5.7 The Initial Context

The only thing that remains to consider with the evaluation semantics, is how to get the ball rolling. Where do the initial values for things like the addition function come from? The answer is that when MAGMA starts up it does so with an initial context defined. This initial context has assignments of all the built-in MAGMA function values to the appropriate identifiers. The initial context contains for example the assignment of the addition function to the identifier `+`, the multiplication function to the identifier `*`, etc.

If, for example, we start MAGMA and immediately type,

```
> 1+2;
```

then in evaluating the expression `1+2` MAGMA will replace `+` by its value in the initial context.

Users interact with this initial context by typing statements at the top level (i.e., statements not inside any function or procedure). A user can change the initial context through re-assignment or expand it through new assignments.

5.6 Scope

Say we type the following,

```
> temp := 7;
> f := function(a,b)
>   temp := a * b;
>   return temp^2;
> end function;
```

If the evaluation process is now followed verbatim, the resultant context will look like `[(temp,7), (f,FUNC(a,b : 7 := a*b; return 7^2;))]`, which is quite clearly not what was intended!

5.6.1 Local Declarations

What is needed in the previous example is some way of declaring that an identifier, in this case `temp`, is a ‘new’ identifier (i.e., distinct from other identifiers with the same name) whose use is confined to the enclosing function. MAGMA provides such a mechanism, called a local declaration. The previous example could be written,

```
> temp := 7;
> f := function(a,b)
>   local temp;
>   temp := a * b;
>   return temp^2;
> end function;
```

The identifier `temp` inside the body of `f` is said to be ‘(declared) local’ to the enclosing function. Evaluation of these two assignments would result in the context being `[(temp, 7), (f, FUNC(a,b : local temp := a*b; return local temp^2;))]` as intended.

It is very important to remember that `temp` and `local temp` are *distinct*! Hence if we now type,

```
> r := f(3,4);
```

the resultant context would be `[(temp,7), (f,FUNC(a,b : local temp := a*b; return local temp^2;)), (r,144)]`. The assignment to `local temp` inside the body of `f` does *not* change the value of `temp` outside the function. The effect of an assignment to a local identifier is thus localized to the enclosing function.

5.6.2 The ‘first use’ Rule

It can become tedious to have to declare all the local variables used in a function body. Hence MAGMA adopts a convention whereby an identifier can be implicitly declared according to how it is first used in a function body. The convention is that if the first use of an identifier inside a function body is on the left hand side of a `:=`, then the identifier is considered to be local, and the function body is considered to have an implicit local declaration for this identifier at its beginning. There is in fact no need therefore to declare `temp` as local in the previous example as the first use of `temp` is on the left hand side of a `:=` and hence `temp` is implicitly declared local.

It is very important to note that the term ‘first use’ refers to the first *textual* use of an identifier. Consider the following example,

```
> temp := 7;
> f := function(a,b)
>   if false then
>     temp := a * b;
>     return temp;
>   else
>     temp;
>     return 1;
```

```
>     end if;
> end function;
```

The first *textual* use of `temp` in this function body is in the line

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

Hence `temp` is considered as a local inside the function body. It is not relevant that the `if false ...` condition will never be true and so the first time `temp` will be encountered when *f* is applied to some arguments is in the line

```
> temp;
```

‘First use’ means ‘first textual use’, modulo the rule about examining the right hand side of a `:=` before the left!

5.6.3 Identifier Classes

It is now necessary to be more precise about the treatment of identifiers in MAGMA. Every identifier in a MAGMA program is considered to belong to one of three possible classes, these being:

- (a) the class of value identifiers
- (b) the class of variable identifiers
- (c) the class of reference identifiers

The class an identifier belongs to indicates how the identifier is used in a program.

The class of value identifiers includes all identifiers that stand as placeholders for values, namely:

- (a) all loop identifiers;
- (b) the `$$` pseudo-identifier;
- (c) all identifiers whose first use in a function expression is as a value (i.e., not on the left hand side of an `:=`, nor as an actual reference argument to a procedure).

Because value identifiers stand as placeholders for values to be substituted during the evaluation process, they are effectively constants, and hence they cannot be assigned to. Assigning to a value identifier would be akin to writing something like `7 := 8;!`

The class of variable identifiers includes all those identifiers which are declared as local, either implicitly by the first use rule, or explicitly through a local declaration. Identifiers in this class may be assigned to.

The class of reference identifiers will be discussed later.

5.6.4 The Evaluation Process Revisited

The reason it is important to know the class of an identifier is that the class of an identifier effects how it is treated during the evaluation process. Previously it was stated that the evaluation process was,

- (1) replace each identifier in the expression by its value in the current context.
- (2) simplify the resultant *value* to its canonical form.

Strictly speaking the first step of this process should read,

- (1') replace each *free* identifier in the expression by its value in the current context, where an identifier is said to be free if it is a value identifier which is not a formal argument, a loop identifier, or the \$\$ identifier.

This definition of the replacement step ensures for example that while computing the value of a function expression F , MAGMA does not attempt to replace F 's formal arguments with values from the current context!

5.6.5 The 'single use' Rule

As a final point on identifier classes it should be noted that an identifier may belong to only *one* class within an expression. Specifically therefore an identifier can only be used in one way inside a function body. Consider the following function,

```
> a := 7;
> f := function(n) a := a; return a; end function;
```

It is *not* the case that a is considered as a variable identifier on the left hand side of the $:=$, and as a value identifier on the right hand side of the $:=$. Rather a is considered to be a value identifier as its first use is as a value on the right hand side of the $:=$ (remember that MAGMA inspects the right hand side of an assignment, and hence sees a first as a value identifier, *before* it inspects the left hand side where it sees a being used as a variable identifier).

5.7 Procedure Expressions

To date we have only discussed function expressions, these being a mechanism for computing new values from the values of identifiers in the current context. Together with assignment this provides us with a means of changing the current context – to compute a new value for an identifier in the current context, we call a function and then re-assign the identifier with the result of this function. That is we do

```
> X := f(Y);
```

where Y is a list of arguments possibly including the current value of X .

At times however using re-assignment to change the value associated with an identifier can be both un-natural and inefficient. Take the problem of computing some reduced form of a matrix. We could write a function that looked something like this,

```
reduce :=
  function( m )
    local lm;
    ...
    lm := m;
    while not reduced do
```

```

...
  lm := some_reduction(m);
...
end while;
...
end function;

```

Note that the local `lm` is necessary since we cannot assign to the function's formal argument `m` since it stands for a value (and values cannot be assigned to). Note also that the function is inefficient in its space usage since at any given point in the program there are at least two different copies of the matrix (if the function was recursive then there would be more than two copies!).

Finally the function is also un-natural. It is perhaps more natural to think of writing a program that takes a given matrix and *changes* that matrix into its reduced form (i.e., the original matrix is lost). To accommodate for this style of programming, Magma includes a mechanism, the *procedure expression* with its *reference arguments*, for changing an association of an identifier and a value *in place*.

Before examining procedure expressions further, it is useful to look at a simple example of a procedure expression. Say we type:

```
> a := 5; b := 6;
```

giving the context `[(a,5), (b,6)]`. Say we now type the following:

```
> p := procedure( x, ~y ) y := x; end procedure;
```

This gives us a context that looks like `[(a,5), (b,6), (p, PROC(x,~y : y := x;))]`, using a notation analogous to the `FUNC` notation.

Say we now type the following *statement*,

```
> p(a, ~b);
```

This is known as a *call of the procedure p* (strictly it should be known as a call to the *procedure value* associated with the identifier `p`, since like functions, procedures in Magma are first class values!). Its effect is to *change* the current context to `[(a,5), (b,5), (p, PROC(a,~b : b := a;))]`. `a` and `x` are called *actual* and *formal value arguments* respectively since they are not prefixed by a `~`, while `b` and `y` are called *actual* and *formal reference arguments* respectively because they are prefixed by a `~`.

This example illustrates the defining attribute of procedures, namely that rather than returning a value, a procedure changes the context in which it is called. In this case the value of `b` was changed by the call to `p`. Observe however that *only b* was changed by the call to `p` as *only b* in the call, and its corresponding formal argument `y` in the definition, are reference arguments (i.e., prefixed with a `~`). A procedure may therefore only change that part of the context associated with its reference arguments! All other parts of the context are left unchanged. In this case `a` and `p` were left unchanged!

Note that apart from reference arguments (and the corresponding fact that that procedures do not return values), procedures are exactly like functions. In particular:

- a) procedures are first class values that can be assigned to identifiers, passed as arguments, returned from functions, etc.
- b) procedure expressions are evaluated in the same way that function expressions are.
- c) procedure value arguments (both formal and actual) behave exactly like function arguments (both formal and actual). Thus procedure value arguments obey the standard substitution semantics.
- d) procedures employ the same notion of scope as functions.
- e) procedure calling behaves like function application.
- f) procedures may be declared ‘forward’ to allow for (mutual) recursion.
- g) a procedure may be assigned to an identifier in the initial context.

The remainder of this section will thus restrict itself to looking at reference arguments, the point of difference between procedures and functions.

5.8 Reference Arguments

If we look at a context it consists of a set of pairs, each pair being a name (an identifier) and a value (that is said to be assigned to that identifier).

When a function is applied actual arguments are substituted for formal arguments, and the body of the function is evaluated. The process of evaluating an actual argument yields a value and any associated names are ignored. Magma’s evaluation semantics treats identifiers as ‘indexes’ into the context – when Magma wants the value of say x it searches through the context looking for a pair whose name component is x . The corresponding value component is then used as the value of x and the name part is simply ignored thereafter.

When we call a procedure with a reference argument, however, the name components of the context become important. When, for example we pass x as an actual reference argument to a formal reference argument y in some procedure, Magma remembers the name x . Then if y is changed (e.g., by assignment) in the called procedure, Magma, knowing the name x , finds the appropriate pair in the calling context and updates it by changing its corresponding value component. To see how this works take the example in the previous section. It was,

```
> a := 5; b := 6;
> p := procedure( x, ~y ) y := x; end procedure;
> p(a, ~b);
```

In the call Magma remembers the name b . Then when y is assigned to in the body of p , Magma knows that y is really b in the calling context, and hence changes b in the calling context appropriately. This example shows that an alternate way of thinking of reference arguments is as synonyms for the same part of (or pair in) the calling context.

5.9 Dynamic Typing

MAGMA is a dynamically typed language. In practice this means that:

- (a) there is no need to declare the type of identifiers (this is especially important for identifiers assigned function values!).
- (b) type violations are only checked for when the code containing the type violation is actually executed.

To make these ideas clearer consider the following two functions,

```
> f := func< a, b | a+b >;
> g := func< a, b | a+true >;
```

First note that there are no declarations of the types of any of the identifiers.

Second consider the use of `+` in the definition of function f . Which addition function is meant by the `+` in `a+b`? Integer addition? Matrix addition? Group addition? ... Or in other words what is the type of the identifier `+` in function f ? Is it integer addition, matrix addition, etc.? The answer to this question is that `+` here denotes all possible addition function values (`+` is said to denote a *family* of function values), and MAGMA will automatically chose the appropriate function value to apply when it knows the type of a and b .

Say we now type,

```
> f(1,2);
```

MAGMA now knows that a and b in f are both integers and thus `+` in f should be taken to mean the integer addition function. Hence it will produce the desired answer of 3.

Finally consider the definition of the function g . It is clear $X+\text{true}$ for all X is a type error, so it might be expected that MAGMA would raise an error as soon as the definition of g is typed in. MAGMA does not however raise an error at this point. Rather it is only when g is applied and the line `return a + true` is actually executed that an error is raised.

In general the exact point at which type checking is done is not important. Sometimes however it is. Say we had typed the following definition for g ,

```
> g := function(a,b)
>   if false then
>     return a+true;
>   else
>     return a+b;
>   end if;
> end function;
```

Now because the `if false` condition will never be true, the line `return a+true` will *never* be executed, and hence the type violation of adding a to `true` will *never* be raised!

One closing point: it should be clear now that where it was previously stated that the initial context “contains assignments of all the built-in MAGMA function values to the appropriate identifiers”, in fact the initial context contains assignments of all the built-in MAGMA function *families* to the appropriate identifiers.

5.10 Traps for Young Players

This section describes the two most common sources of confusion encountered when using MAGMA's evaluation strategy.

5.10.1 Trap 1

We boot MAGMA. It begins with an initial context something like $[\dots, ('+', A), ('-', S), \dots]$ where A is the (function) value that is the addition function, and S is the (function) value that is the subtraction function.

Now say we type,

```
> '+' := '-';
> 1 + 2;
```

MAGMA will respond with the answer -1.

To see why this is so consider the effect of each line on the current context. After the first line the current context will be $[\dots, ('+', S), ('-', S), \dots]$, where S is as before. The identifier $+$ has been re-assigned. Its new value is the value of the identifier $'-'$ in the current context, and the value of $'-'$ is the (function) value that is the subtraction function. Hence in the second line when MAGMA replaces the identifier $+$ with its value in the current context, the value that is substituted is therefore S , the subtraction function!

5.10.2 Trap 2

Say we type,

```
> f := func< n | n + 1 >;
> g := func< m | m + f(m) >;
```

After the first line the current context is $[(f, \text{FUNC}(n : n+1))]$. After the second line the current context is $[(f, \text{FUNC}(n : n+1)), (g, \text{FUNC}(m : m + \text{FUNC}(n : n+1)(m)))]$.

If we now type,

```
> g(6);
```

MAGMA will respond with the answer 13.

Now say we decide that our definition of f is wrong. So we now type in a new definition for f as follows,

```
> f := func< n | n + 2 >;
```

If we again type,

```
> g(6);
```

MAGMA will again reply with the answer 13!

To see why this is so consider how the current context changes. After typing in the initial definitions of f and g the current context is $[(f, \text{FUNC}(n : n+1)), (g, \text{FUNC}(m : m + \text{FUNC}(n : n+1)(m)))]$. After typing in the second definition of f the current

context is $[(f, \text{FUNC}(n : n+2)), (g, \text{FUNC}(m : m + \text{FUNC}(n : n+1)(m)))]$. Remember that changing the *value* of one identifier, in this case f , does *not* change the value of any other identifiers, in this case g ! In order to change the value of g to reflect the new value of f , g would have to be re-assigned.

5.11 Appendix A: Precedence

The table below defines the relative precedence of operators in MAGMA, with decreasing strength (so operators higher in the table bind more strongly). The column on the right indicates whether the operator is left-, right-, or non-associative.

' ''	<i>left</i>
(<i>left</i>
[<i>left</i>
assigned	<i>right</i>
~	<i>non</i>
#	<i>non</i>
&* &+ &and &cat &join &meet &or	<i>non-associative</i>
\$ \$\$	<i>non</i>
.	<i>left</i>
@ @@	<i>left</i>
! !!	<i>right</i>
^	<i>right</i>
unary-	<i>right</i>
cat	<i>left</i>
* / div mod	<i>left</i>
+ -	<i>left</i>
meet	<i>left</i>
sdiff	<i>left</i>
diff	<i>left</i>
join	<i>left</i>
adj in notadj notin notsubset subset	<i>non</i>
cmpeq cmpne eq ge gt le lt ne	<i>left</i>
not	<i>right</i>
and	<i>left</i>
or xor	<i>left</i>
^^	<i>non</i>
? else select	<i>right</i>
->	<i>left</i>
=	<i>left</i>
:= is where	<i>left</i>

5.12 Appendix B: Reserved Words

The list below contains all reserved words in the MAGMA language; these cannot be used as identifier names.

-	elif	is	require
adj	else	join	requirege
and	end	le	requirerange
assert	eq	load	restore
assert2	error	local	return
assert3	eval	lt	save
assigned	exists	meet	sdiff
break	exit	mod	select
by	false	ne	subset
case	for	not	then
cat	forall	notadj	time
catch	forward	notin	to
clear	fprintf	notsubset	true
cmpeq	freeze	or	try
cmpne	function	print	until
continue	ge	printf	vprint
declare	gt	procedure	vprintf
default	if	quit	vtime
delete	iload	random	when
diff	import	read	where
div	in	readi	while
do	intrinsic	repeat	xor

6 THE MAGMA PROFILER

6.1 Introduction	137	<code>ProfilePrintByTotalCount(G)</code>	140
6.2 Profiler Basics	137	<code>ProfilePrintByTotalTime(G)</code>	140
<code>SetProfile(b)</code>	137	<code>ProfilePrintChildrenByCount(G, n)</code>	140
<code>ProfileReset()</code>	137	<code>ProfilePrintChildrenByTime(G, n)</code>	140
<code>ProfileGraph()</code>	138	6.3.2 HTML Reports	141
6.3 Exploring the Call Graph . .	139	<code>ProfileHTMLOutput(G, prefix)</code>	141
6.3.1 Internal Reports	139	6.4 Recursion and the Profiler . .	141

Chapter 6

THE MAGMA PROFILER

6.1 Introduction

One of the most important aspects of the development cycle is optimization. It is often the case that during the implementation of an algorithm, a programmer makes erroneous assumptions about its run-time behavior. These errors can lead to performance which differs in surprising ways from the expected output. The unfortunate tendency of programmers to optimize code before establishing run-time bottlenecks tends to exacerbate the problem.

Experienced programmers will thus often be heard repeating the famous mantra “Premature optimization is the root of all evil”, coined by Sir Charles A. R. Hoare, the inventor of the Quick sort algorithm. Instead of optimizing during the initial implementation, it is generally better to perform an analysis of the run-time behaviour of the complete program, to determine what are the actual bottlenecks. In order to assist in this task, MAGMA provides a *profiler*, which gives the programmer a detailed breakdown of the time spent in a program. In this chapter, we provide an overview of how to use the profiler.

6.2 Profiler Basics

The MAGMA profiler records timing information for each function, procedure, map, and intrinsic call made by your program. When the profiler is switched on, upon the entry and exit to each such call the current system clock time is recorded. This information is then stored in a call graph, which can be viewed in various ways.

SetProfile(*b*)

Turns profiling on (if *b* is **true**) or off (if *b* is **false**). Profiling information is stored cumulatively, which means that in the middle of a profiling run, the profiler can be switched off during sections for which profiling information is not wanted. At startup, the profiler is off. Turning the profiler on will slow down the execution of your program slightly.

ProfileReset()

Clear out all information currently recorded by the profiler. It is generally a good idea to do this after the call graph has been obtained, so that future profiling runs in the same MAGMA session begin with a clean slate.

ProfileGraph()

Get the call graph based upon the information recorded up to this point by the profiler. This function will return an error if the profiler has not yet been turned on.

The call graph is a directed graph, with the nodes representing the functions that were called during the program's execution. There is an edge in the call graph from a function x to a function y if y was called during the execution of x . Thus, recursive calls will result in cycles in the call graph.

Each node in the graph has an associated label, which is a record with the following fields:

- (i) **Name:** the name of the function
- (ii) **Time:** the total time spent in the function
- (iii) **Count:** the number of times the function was called

Each edge $\langle x, y \rangle$ in the graph also has an associated label, which is a record with the following fields:

- (i) **Time:** the total time spent in function y when it was called from function x
- (ii) **Count:** the total number of times function y was called by function x

Example H6E1

We illustrate the basic use of the profiler in the following example. The code we test is a simple implementation of the Fibonacci sequence; this can be replaced by any MAGMA code that needs to be profiled.

```
> function fibonacci(n)
>   if n eq 1 or n eq 2 then
>     return 1;
>   else
>     return fibonacci(n - 1) + fibonacci(n - 2);
>   end if;
> end function;
>
> SetProfile(true);
> time assert fibonacci(27) eq Fibonacci(27);
Time: 10.940
> SetProfile(false);
> G := ProfileGraph();
> G;
Digraph
Vertex Neighbours
1      2 3 6 7 ;
2      2 3 4 5 ;
3      ;
4      ;
5      ;
```

```

6      ;
7      ;
> V := Vertices(G);
> Label(V!1);
rec<recformat<Name: Strings(), Time: RealField(), Count: IntegerRing()> |
    Name := <main>,
    Time := 10.93999999999999950262,
    Count := 1
>
> Label(V!2);
rec<recformat<Name: Strings(), Time: RealField(), Count: IntegerRing()> |
    Name := fibonacci,
    Time := 10.93999999999999950262,
    Count := 392835
>
> E := Edges(G);
> Label(E![1,2]);
rec<recformat<Time: RealField(), Count: IntegerRing()> |
    Time := 10.93999999999999950262,
    Count := 1
>

```

6.3 Exploring the Call Graph

6.3.1 Internal Reports

The above example demonstrates that while the call graph contains some useful information, it does not afford a particularly usable interface. The MAGMA profiler contains some profile report generators which can be used to study the call graph in a more intuitive way.

The reports are all tabular, and have a similar set of columns:

- (i) **Index:** The numeric identifier for the function in the vertex list of the call graph.
- (ii) **Name:** The name of the function. The function name will be followed by an asterisk if a recursive call was made through it.
- (iii) **Time:** The time spent in the function; depending on the report, the meaning might vary slightly.
- (iv) **Count:** The number of times the function was called; depending on the report, the meaning might vary slightly.

ProfilePrintByTotalCount(G)

Percentage	BOOLELT	<i>Default : false</i>
Max	RNGINTELT	<i>Default : -1</i>

Print the list of functions in the call graph, sorted in descending order by the total number of times they were called. The **Time** and **Count** fields of the report give the total time and total number of times the function was called. If **Percentage** is true, then the **Time** and **Count** fields represent their values as percentages of the total value. If **Max** is non-negative, then the report only displays the first **Max** entries.

ProfilePrintByTotalTime(G)

Percentage	BOOLELT	<i>Default : false</i>
Max	RNGINTELT	<i>Default : -1</i>

Print the list of functions in the call graph, sorted in descending order by the total time spent in them. Apart from the sort order, this function's behaviour is identical to that of **ProfilePrintByTotalCount**.

ProfilePrintChildrenByCount(G, n)

Percentage	BOOLELT	<i>Default : false</i>
Max	RNGINTELT	<i>Default : -1</i>

Given a vertex n in the call graph G , print the list of functions called by the function n , sorted in descending order by the number of times they were called by n . The **Time** and **Count** fields of the report give the time spent during calls by the function n and the number of times the function was called by the function n . If **Percentage** is true, then the **Time** and **Count** fields represent their values as percentages of the total value. If **Max** is non-negative, then the report only displays the first **Max** entries.

ProfilePrintChildrenByTime(G, n)

Percentage	BOOLELT	<i>Default : false</i>
Max	RNGINTELT	<i>Default : -1</i>

Given a vertex n in the call graph G , print the list of functions in the called by the function n , sorted in descending order by the time spent during calls by the function n . Apart from the sort order, this function's behaviour is identical to that of **ProfilePrintChildrenByCount**.

Example H6E2

Continuing with the previous example, we examine the call graph using profile reports.

```
> ProfilePrintByTotalTime(G);
```

Index	Name	Time	Count
1	<main>	10.940	1
2	fibonacci	10.940	392835
3	eq(<RngIntElt> x, <RngIntElt> y) -> BoolElt	1.210	710646


```

4      -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt           0.630   392834
5      +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt           0.250   196417
6      Fibonacci(<RngIntElt> n) -> RngIntElt                 0.000    1
7      SetProfile(<BoolElt> v)                               0.000    1
> ProfilePrintChildrenByTime(G, 2);
Function: fibonacci
Function Time: 10.940
Function Count: 392835
Index Name                                                    Time    Count
2      fibonacci (*)                                           182.430 392834
3      eq(<RngIntElt> x, <RngIntElt> y) -> BoolElt             1.210   710645
4      -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt           0.630   392834
5      +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt           0.250   196417
* A recursive call is made through this child

```

6.3.2 HTML Reports

While the internal reports are useful for casual inspection of a profile run, for detailed examination a text-based interface has serious limitations. MAGMA's profiler also supports the generation of HTML reports of the profile run. The HTML report can be loaded up in any web browser. If Javascript is enabled, then the tables in the report can be dynamically sorted by any field, by clicking on the column heading you wish to perform a sort with. Clicking the column heading multiple times will alternate between ascending and descending sorts.

ProfileHTMLOutput(G, prefix)

Given a call graph G , an HTML report is generated using the file prefix *prefix*. The index file of the report will be "*prefix.html*", and exactly n additional files will be generated with the given filename *prefix*, where n is the number of functions in the call graph.

6.4 Recursion and the Profiler

Recursive calls can cause some difficulty with profiler results. The profiler takes care to ensure that double-counting does not occur, but this can lead to unintuitive results, as the following example shows.

Example H6E3

In the following example, `recursive` is a recursive function which simply stays in a loop for half a second, and then recurses if not in the base case. Thus, the total running time should be approximately $(n + 1)/2$ seconds, where n is the parameter to the function.

```
> procedure delay(s)
>   t := Cputime();
>   repeat
>     _ := 1+1;
>   until Cputime(t) gt s;
> end procedure;
>
> procedure recursive(n)
>   if n ne 0 then
>     recursive(n - 1);
>   end if;
>
>   delay(0.5);
> end procedure;
>
> SetProfile(true);
> recursive(1);
> SetProfile(false);
> G := ProfileGraph();
```

Printing the profile results by total time yield no surprises:

```
> ProfilePrintByTotalTime(G);
```

Index	Name	Time	Count
1	<main>	1.020	1
2	recursive	1.020	2
5	delay	1.020	2
8	Cputime(<FldRElt> T) -> FldRElt	0.130	14880
7	+(<RngIntElt> x, <RngIntElt> y) -> RngIntElt	0.020	14880
9	gt(<FldRElt> x, <FldRElt> y) -> BoolElt	0.020	14880
3	ne(<RngIntElt> x, <RngIntElt> y) -> BoolElt	0.000	2
4	-(<RngIntElt> x, <RngIntElt> y) -> RngIntElt	0.000	1
6	Cputime() -> FldRElt	0.000	2
10	SetProfile(<BoolElt> v)	0.000	1

However, printing the children of `recursive`, and displaying the results in percentages, does yield a surprise:

```
> ProfilePrintChildrenByTime(G, 2 : Percentage);
```

Function: recursive
Function Time: 1.020
Function Count: 2

Index	Name	Time	Count
5	delay	100.00	33.33
2	recursive (*)	50.00	16.67

```

3      ne(<RngIntElt> x, <RngIntElt> y) -> BoolElt           0.00    33.33
4      -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt          0.00    16.67
* A recursive call is made through this child

```

At first glance, this doesn't appear to make sense, as the sum of the time column is 150%! The reason for this behavior is because some time is "double counted": the total time for the first call to `recursive` includes the time for the recursive call, which is also counted separately. In more detail:

```

> V := Vertices(G);
> E := Edges(G);
> Label(V!1)'Name;
<main>
> Label(V!2)'Name;
recursive
> Label(E![1,2])'Time;
1.019999999999999795718
> Label(E![2,2])'Time;
0.510000000000000000888
> Label(V!2)'Time;
1.019999999999999795718

```

As can be seen in the above, the total time for `recursive` is approximately one second, as expected. The double-counting of the recursive call can be seen in the values of `Time` for the edges `[1,2]` and `[2,2]`.

7 DEBUGGING MAGMA CODE

7.1 Introduction	147	7.2 Using the Debugger	147
SetDebugOnError(f)	147		

Chapter 7

DEBUGGING MAGMA CODE

7.1 Introduction

In order to facilitate the debugging of complex pieces of MAGMA code, MAGMA includes a debugger. *This debugger is very much a prototype, and can cause MAGMA to crash.*

`SetDebugOnError(f)`

If f is `true`, then upon an error MAGMA will break into the debugger. The usage of the debugger is described in the next section.

7.2 Using the Debugger

When use of the debugger is enabled and an error occurs, MAGMA will break into the command-line debugger. The syntax of the debugger is modelled on the GNU GDB debugger for C programs, and supports the following commands (acceptable abbreviations for the commands are given in parentheses):

- | | |
|------------------------------|--|
| backtrace (bt) | Print out the stack of function and procedure calls, from the top level to the point at which the error occurred. Each line in this trace gives a single <i>frame</i> , which consists of the function/procedure that was called, as well as all local variable definitions for that function. Each frame is numbered so that it can be referenced in other debugger commands. |
| frame (f) n | Change the current frame to the frame numbered n (the list of frames can be obtained using the backtrace command). The current frame is used by other debugger commands, such as print , to determine the context within which expressions should be evaluated. The default current frame is the top-most frame. |
| list (l) [n] | Print a source code listing for the current context (the context is set by the frame command). If n is specified, then the list command will print n lines of source code; the default value is 10. |
| print (p) <i>expr</i> | Evaluate the expression <i>expr</i> in the current context (the context is set by the frame command). The print command has semantics identical to evaluating the expression eval "expr" at the current point in the program. |
| help (h) | Print brief help on usage. |
| quit (q) | Quit the debugger and return to the MAGMA session. |

PART II

SETS, SEQUENCES, AND MAPPINGS

8	INTRODUCTION TO AGGREGATES	153
9	SETS	163
10	SEQUENCES	191
11	TUPLES AND CARTESIAN PRODUCTS	213
12	LISTS	221
13	ASSOCIATIVE ARRAYS	227
14	COPRODUCTS	233
15	RECORDS	239
16	MAPPINGS	245

8 INTRODUCTION TO AGGREGATES

8.1 Introduction	155	<i>8.2.3 Parents of Sets and Sequences. . .</i>	<i>159</i>
8.2 Restrictions on Sets and Sequences	155	8.3 Nested Aggregates	160
8.2.1 Universe of a Set or Sequence . . .	156	8.3.1 Multi-indexing	160
8.2.2 Modifying the Universe of a Set or Sequence	157		

Chapter 8

INTRODUCTION TO AGGREGATES

8.1 Introduction

This part of the Handbook comprises four chapters on aggregate objects in MAGMA as well as a chapter on maps.

Sets, sequences, tuples and lists are the four main types of aggregates, and each has its own specific purpose. *Sets* are used to collect objects that are elements of some common structure, and the most important operation is to test element membership. *Sequences* also contain objects of a common structure, but here the emphasis is on the ordering of the objects, and the most important operation is that of accessing (or modifying) elements at given positions. Sets will contain at most one copy of any element, whereas sequences may contain arbitrarily many copies of the same object. *Enumerated* sets and sequences are of arbitrary but finite length and will store all elements explicitly (with the exception of arithmetic progressions), while *formal* sets and sequences may be infinite, and use a Boolean function to test element membership. *Indexed* sets are a hybrid form of sets allowing indexing like sequences. Elements of *Cartesian products* of structures in MAGMA will be called *tuples*; they are of fixed length, and each coefficient must be in the corresponding structure of the defining Cartesian product. *Lists* are arbitrary finite ordered collections of objects of any type, and are mainly provided to the user to store assorted data to which access is not critical.

8.2 Restrictions on Sets and Sequences

Here we will explain the subtleties behind the mechanism dealing with sets and sequences and their universes and parents. Although the same principles apply to their formal counterparts, we will only talk about enumerated sets and sequences here, for two reasons: the enumerated versions are much more useful and common, and the very restricted number of operations on formal sets/sequences make issues of universe and overstructure of less importance for them.

In principle, every object e in MAGMA has some parent structure S such that $e \in S$; this structure can be used for type checking (are we allowed to apply function f to e ?), algorithm look-up etc. To avoid storing the structure with every element of a set or sequence and having to look up the structure of every element separately, only elements of a *common structure* are allowed in sets or sequences, and that common parent will only be stored once.

8.2.1 Universe of a Set or Sequence

This common structure is called the *universe* of the set or sequence. In the general constructors it may be specified up front to make clear what the universe for the set or sequence will be; the difference between the sets i and s in

```
> i := { IntegerRing() | 1, 2, 3 };
> s := { RationalField() | 1, 2, 3 };
```

lies entirely in their universes. The specification of the universe may be omitted if there is an obvious common overstructure for the elements. Thus the following provides a shorter way to create the set containing 1, 2, 3 and having the ring of integers as universe:

```
> i := { 1, 2, 3 };
```

Only empty sets and sequences that have been obtained directly from the constructions

```
> S := { };
> T := [ ];
```

do not have their universe defined – we will call them the *null* set or sequence. (There are two other ways in which empty sets and sequences arise: it is possible to create empty sequences with a prescribed universe, using

```
> S := { U | };
> T := [ U | ];
```

and it may happen that a non-empty set/sequence becomes empty in the course of a computation. In both cases these empty objects have their universe defined and will not be *null*).

Usually (but not always: the exception will be explained below) the universe of a set or sequence is the parent for all its elements; thus the ring of integers is the parent of 2 in the set $i = \{1, 2, 3\}$, rather than that set itself. The universe is not static, and it is not necessarily the same structure as the parent of the elements *before* they were put in the set or sequence. To illustrate this point, suppose that we try to create a set containing integers and rational numbers, say $T = \{1, 2, 1/3\}$; then we run into trouble with the rule that the universe must be common for all elements in T ; the way this problem is solved in MAGMA is by automatic coercion: the obvious universe for T is the field of rational numbers of which $1/3$ is already an element and into which any integer can be coerced in an obvious way. Hence the assignment

```
> T := { 1, 2, 1/3 }
```

will result in a set with universe the field of rationals (which is also present when MAGMA is started up). Consequently, when we take the element 1 of the set T , it will have the rational field as its parent rather than the integer ring! It will now be clear that

```
> s := { 1/1, 2, 3 };
```

is a shorter way to specify the set of rational numbers 1, 2, 3 than the way we saw before, but in general it is preferable to declare the universe beforehand using the $\{ U \mid \}$ notation.

Of course

```
> T := { Integers() | 1, 2, 1/3 }
```

would result in an error because $1/3$ cannot be coerced into the ring of integers.

So, usually not every element of a given structure can be coerced into another structure, and even if it can, it will not always be done automatically. The possible (automatic) coercions are listed in the descriptions of the various MAGMA modules. For instance, the table in the introductory chapter on rings shows that integers can be coerced automatically into the rational field.

In general, every MAGMA structure is valid as a universe. This includes enumerated sets and sequences themselves, that is, it is possible to define a set or sequence whose elements are confined to be elements of a given set or sequence. So, for example,

```
> S := [ [ 1..10 ] | x^2+x+1 : x in { -3 .. 2 by 1 } ];
```

produces the sequence $[7, 3, 1, 1, 3, 7]$ of values of the polynomial $x^2 + x + 1$ for $x \in \mathbf{Z}$ with $-3 \leq x \leq 2$. However, an entry of S will in fact have the ring of integers as its parent (and *not* the sequence $[1..10]$), because the effect of the above assignment is that the values after the $|$ are calculated and coerced into the universe, which is $[1..10]$; but coercing an element into a sequence or set means that it will in fact be coerced into the *universe* of that sequence/set, in this case the integers. So the main difference between the above assignment and

```
> T := [ Integers() | x^2+x+1 : x in { -3 .. 2 by 1 } ];
```

is that in the first case it is checked that the resulting values y satisfy $1 \leq y \leq 10$, and an error would occur if this is violated:

```
> S := [ [ 1..10 ] | x^2+x+1 : x in { -3 .. 3 by 1 } ];
```

leads to a run-time error.

In general then, the parent of an element of a set or sequence will be the universe of the set or sequence, unless that universe is itself a set or sequence, in which case the parent will be the universe of this universe, and so on, until a non-set or sequence is encountered.

8.2.2 Modifying the Universe of a Set or Sequence

Once a (non-null) set or sequence S has been created, the universe has been defined. If one attempts to *modify* S (that is, to add elements, change entries etc. using a procedure that will not reassign the result to a new set or sequence), the universe will not be changed, and the modification will only be successful if the new element can be coerced into the current universe. Thus,

```
> Z := Integers();
> T := [ Z | 1, 2, 3/3 ];
> T[2] := 3/4;
```

will result in an error, because $3/4$ cannot be coerced into Z .

The universe of a set or sequence S can be explicitly modified by creating a *parent* for S with the desired universe and using the `!` operator for the coercion; as we will see in the next subsection, such a parent can be created using the `PowerSet` and `PowerSequence` commands. Thus, for example, the set $\{1, 2\}$ can be made into a sequence of rationals as follows:

```
> I := { 1, 2 };
> P := PowerSet( RationalField() );
> J := P ! I;
```

The coercion will be successful if every element of the sequence can be coerced into the new universe, and it is *not* necessary that the old universe could be coerced completely into the new one: the set $\{3/3\}$ of rationals can be coerced into `PowerSet(Integers())`. As a consequence, the empty set (or sequence) with any universe can be coerced into the power set (power sequence) of any other universe.

Binary functions on sets or sequences (like `join` or `cat`) can only be applied to sets and sequences that are *compatible*: the operation on S with universe A and T with universe B can only be performed if a common universe C can be found such that the elements of S and T are all elements of C . The compatibility conditions are dependent on the particular MAGMA module to which A and B belong (we refer to the corresponding chapters of this manual for further information) and do also apply to elements of $a \in A$ and $b \in B$ — that is, the compatibility conditions for S and T are the same as the ones that determine whether binary operations on $a \in A$ and $b \in B$ are allowed. For example, we are able to join a set of integers and a set of rationals:

```
> T := { 1, 2 } join { 1/3 };
```

for the same reason that we can do

```
> c := 1 + 1/3;
```

(automatic coercion for rings). The resulting set T will have the rationals as universe. The basic rules for compatibility of two sets or sequences are then:

- (1) every set/sequence is compatible with the null set/sequence (which has no universe defined (see above));
- (2) two sets/sequences with the same universe are compatible;
- (3) a set/sequence S with universe A is compatible with set/sequence T with universe B if the elements of A can be automatically coerced into B , or vice versa;
- (4) more generally, a set/sequence S with universe A is also compatible with set/sequence T with universe B if MAGMA can automatically find an *over-structure* for the parents A and B (see below);
- (5) nested sets and sequences are compatible only when they are of the same ‘depth’ and ‘type’ (that is, sets and sequences appear in exactly the same recursive order in both) and the universes are compatible.

The possibility of finding an overstructure C for the universe A and B of sets or sequences S and T (such that $A \subset C \supset B$), is again module-dependent. We refer the reader for

details to the Introductions of Parts III–VI, and we give some examples here; the next subsection contains the rules for parents of sets and sequences.

8.2.3 Parents of Sets and Sequences

The universe of a set or sequence S is the common parent for all its elements; but S itself is a MAGMA object as well, so it should have a parent too.

The parent of a set is a *power set*: the set of all subsets of the universe of S . It can be created using the `PowerSet` function. Similarly, `PowerSequence(A)` creates the parent structure for a sequence of elements from the structure A – that is, the elements of `PowerSequence(A)` are all sequences of elements of A .

The rules for finding a common overstructure for structures A and B , where either A or B is a set/sequence or the parent of a set/sequence, are as follows. (If neither A nor B is a set, sequence, or its parent we refer to the Part of this manual describing the operations on A and B .)

- (1) The overstructure of A and B is the same as that of B and A .
- (2) If A is the null set or sequence (empty, and no universe specified) the overstructure of A and B is B .
- (3) If A is a set or sequence with universe U , the overstructure of A and B is the overstructure of U and B ; in particular, the overstructure of A and A will be the universe U of A .
- (4) If A is the parent of a set (a power set), then A and B can only have a common overstructure if B is also the parent of a set, in which case the overstructure is the power set of the overstructure of the universes U and V of A and B respectively. Likewise for sequences instead of sets.

We give two examples to illustrate rules (3) and (4). It is possible to create a set with a set as its universe:

```
> S := { { 1..100 } | x^3 : x in [ 0..3 ] };
```

If we wish to intersect this set with some set of integers, say the formal set of odd integers

```
> T := {! x : x in Integers() | IsOdd(x) !};
> W := S meet T;
```

then we can only do that if we can find a universe for W , which must be the common overstructure of the universe $U = \{1, 2, \dots, 100\}$ of S and the universe ‘ring of integers’ of T . By rule (3) above, this overstructure of $U = \{1, 2, \dots, 100\}$ will be the overstructure of the universe of U and the ring of integers; but the universe of U is the ring of integers (because it is the default for the set $\{1, 2, \dots, 100\}$), and hence the overstructure we are looking for (and the universe for W) will be the ring of integers.

For the second example we look at sequences of sequences:

```
> a := [ [ 1 ], [ 1, 2, 3 ] ];
```

```
> b := [ [ 2/3 ] ];
```

so a is a sequence of sequences of integers, and b is a sequence of sequences of rationals. If we wish to concatenate a and b ,

```
> c := a cat b;
```

we will only succeed if we find a universe for c . This universe must be the common overstructure of the universes of a and b , which are the ‘power sequence of the integers’ and the ‘power sequence of the rationals’ respectively. By rule (4), the overstructure of these two power sequences is the power sequence of the common overstructure of the rationals and the integers, which is the rationals themselves. Hence c will be a sequence of sequences of rationals, and the elements of a will have to be coerced.

8.3 Nested Aggregates

Enumerated sets and sequences can be arbitrarily nested (that is, one may create sets of sets, as well as sequences of sets etc.); tuples can also be nested and may be freely mixed with sets and sequences (as long as the proper Cartesian product parent can be created). Lists can be nested, and one may create lists of sets or sequences or tuples.

8.3.1 Multi-indexing

Since sequences (and lists) can be nested, assignment functions and mutation operators allow you to use *multi-indexing*, that is, one can use a multi-index i_1, i_2, \dots, i_r rather than a single i to reach r levels deep. Thus, for example, if $S = [[1, 2], [2, 3]]$, instead of

```
> S[2][2] := 4;
```

one may use the multi-index 2,2 to obtain the same effect of changing the 3 into a 4:

```
> S[2,2] := 4;
```

All i_j in the multi-index i_1, i_2, \dots, i_r have to be greater than 0, and an error will also be flagged if any i_j indexes beyond the length at level j , that is, if $i_j > \#S[i_1, \dots, i_{j-1}]$, (which means $i_1 > \#S$ for $j = 1$). There is one exception: the last index i_r is allowed to index beyond the current length of the sequence at level r if the multi-index is used on the left-hand side of an assignment, in which case any intermediate terms will be undefined. This generalizes the possibility to assign beyond the length of a ‘flat’ sequence. In the above example the following assignments are allowed:

```
> S[2,5] := 7;
```

(and the result will be $S = [[1, 2], [2, 3, \text{undef}, \text{undef}, 7]]$)

```
> S[4] := [7];
```

(and the result will be $S = [[1, 2], [2, 3], \text{undef}, [7]]$). But the following results in an error:

```
> S[4,1] := 7;
```

Finally we point out that multi-indexing should not be confused with the use of sequences as

indexes to create subsequences. For example, to create a subsequence of $S = [5, 13, 17, 29]$ consisting of the second and third terms, one may use

```
> S := [ 5, 13, 17, 29 ];  
> T := S[ [2, 3] ];
```

To obtain the second term of this subsequence one could have done:

```
> x := S[ [2, 3] ][2];
```

(so x now has the value $S[3] = 17$), but it would have been more efficient to index the indexing sequence, since it is rather expensive to build the subsequence $[S[2], S[3]]$ first, so:

```
> x := S[ [2, 3][2] ];
```

has the same effect but is better (of course $x := S[3]$ would be even better in this simple example.) To add to the confusion, it is possible to mix the above constructions for indexing, since one can use lists of sequences and indices for indexing; continuing our example, there is now a third way to do the same as above, using an indexing list that first takes out the subsequence consisting of the second and third terms and then extracts the second term of that:

```
> x := S[ [2, 3], 2 ];
```

Similarly, the construction

```
> x := S[ [2, 3], [2] ];
```

pulls out the subsequence consisting of the second term of the subsequence of terms two and three of S , in other words, this assigns the *sequence* consisting of the element 17, not just the element itself!

9 SETS

9.1 Introduction	165	$\{ U \mid i \dots j \text{ by } k \}$	173
9.1.1 Enumerated Sets	165	9.3 Power Sets	173
9.1.2 Formal Sets	165	PowerSet(R)	173
9.1.3 Indexed Sets	165	PowerIndexedSet(R)	173
9.1.4 Multisets	165	PowerMultiset(R)	174
9.1.5 Compatibility	166	in	174
9.1.6 Notation	166	PowerFormalSet(R)	174
9.2 Creating Sets	166	in	174
9.2.1 The Formal Set Constructor . . .	166	!	174
$\{! \ x \text{ in } F \mid P(x) \ !\}$	166	!	174
9.2.2 The Enumerated Set Constructor .	167	9.3.1 The Cartesian Product Constructors	175
$\{ \}$	167	9.4 Sets from Structures	175
$\{ U \mid \}$	167	Set(M)	175
$\{ e_1, e_2, \dots, e_n \}$	167	FormalSet(M)	175
$\{ U \mid e_1, e_2, \dots, e_n \}$	167	9.5 Accessing and Modifying Sets .	176
$\{ e(x) : x \text{ in } E \mid P(x) \}$	168	9.5.1 Accessing Sets and their Associated	
$\{ U \mid e(x) : x \text{ in } E \mid P(x) \}$	168	Structures	176
$\{ e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k$		#	176
$\text{ in } E_k \mid P(x_1, \dots, x_k) \}$	168	Category(S)	176
$\{ U \mid e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots,$		Type(S)	176
$x_k \text{ in } E_k \mid P(x_1, \dots, x_k) \}$	168	Parent(R)	176
9.2.3 The Indexed Set Constructor . . .	169	Universe(R)	176
$\{ @ \ @ \}$	169	Index(S, x)	176
$\{ @ \ U \mid @ \}$	169	Position(S, x)	176
$\{ @ \ e_1, e_2, \dots, e_n \ @ \}$	169	S[i]	176
$\{ @ \ U \mid e_1, e_2, \dots, e_m \ @ \}$	169	S[I]	176
$\{ @ \ e(x) : x \text{ in } E \mid P(x) \ @ \}$	169	9.5.2 Selecting Elements of Sets	177
$\{ @ \ U \mid e(x) : x \text{ in } E \mid P(x) \ @ \}$	169	Random(R)	178
$\{ @ \ e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k$		random $\{ e(x) : x \text{ in } E \mid P(x) \}$	178
$\text{ in } E_k \mid P(x_1, \dots, x_k) \ @ \}$	170	random $\{ e(x_1, \dots, x_k) : x_1 \text{ in } E_1,$	
$\{ @ \ U \mid e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots,$		$\dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k) \}$	178
$x_k \text{ in } E_k \mid P(x_1, \dots, x_k) @ \}$	170	Representative(R)	178
9.2.4 The Multiset Constructor	170	Rep(R)	178
$\{ * \ * \}$	170	ExtractRep($\sim R, \sim r$)	179
$\{ * \ U \mid * \}$	170	rep $\{ e(x) : x \text{ in } E \mid P(x) \}$	179
$\{ * \ e_1, e_2, \dots, e_n \ * \}$	171	rep $\{ e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots,$	
$\{ * \ U \mid e_1, e_2, \dots, e_m \ * \}$	171	$x_k \text{ in } E_k \mid P(x_1, \dots, x_k) \}$	179
$\{ * \ e(x) : x \text{ in } E \mid P(x) \ * \}$	171	Minimum(S)	180
$\{ * \ U \mid e(x) : x \text{ in } E \mid P(x) \ * \}$	171	Min(S)	180
$\{ * \ e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k$		Maximum(S)	180
$\text{ in } E_k \mid P(x_1, \dots, x_k) \ * \}$	171	Max(S)	180
$\{ * \ U \mid e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots,$		Hash(x)	180
$x_k \text{ in } E_k \mid P(x_1, \dots, x_k) * \}$	171	9.5.3 Modifying Sets	180
9.2.5 The Arithmetic Progression Construc-		Include($\sim S, x$)	180
tors	172	Include(S, x)	180
$\{ i..j \}$	172	Exclude($\sim S, x$)	180
$\{ U \mid i..j \}$	172	Exclude(S, x)	180
$\{ i \dots j \text{ by } k \}$	173	ChangeUniverse($\sim S, V$)	181

ChangeUniverse(S, V)	181	9.6.3 Other Set Operations	185
CanChangeUniverse(S, V)	181	Multiplicity(S, x)	185
SetToIndexedSet(E)	182	Multiplicities(S)	185
IndexedSetToSet(S)	182	Subsets(S)	185
Isetset(S)	182	Subsets(S, k)	186
IndexedSetToSequence(S)	182	RandomSubset(S, k)	186
Isetseq(S)	182	Multisets(S, k)	186
MultisetToSet(S)	182	Subsequences(S, k)	186
SetToMultiset(E)	182	Permutations(S)	186
SequenceToMultiset(Q)	182	Permutations(S, k)	186
9.6 Operations on Sets	183	9.7 Quantifiers	186
9.6.1 Boolean Functions and Operators .	183	exists(t){ e(x): x in E P(x) }	186
IsNull(R)	183	exists(t ₁ , ..., t _r){ e(x) :	
IsEmpty(R)	183	x in E P(x) }	186
eq	183	exists(t){e(x ₁ , ..., x _k): x ₁ in E ₁ ,	
ne	183	..., x _k in E _k P(x ₁ , ..., x _k)}	187
in	183	exists(t ₁ , ..., t _r){ e(x ₁ , ..., x _k) :	
notin	183	x ₁ in E ₁ , ..., x _k in E _k P }	187
subset	184	forall(t){ e(x) : x in E P(x) }	188
notsubset	184	forall(t ₁ , ..., t _r){ e(x) :	
eq	184	x in E P(x) }	188
ne	184	forall(t){e(x ₁ , ..., x _k): x ₁ in E ₁ ,	
IsDisjoint(R, S)	184	..., x _k in E _k P(x ₁ , ..., x _k)}	188
9.6.2 Binary Set Operators	184	forall(t ₁ , ..., t _r){ e(x ₁ , ..., x _k) :	
join	184	x ₁ in E ₁ , ..., x _k in E _k P }	188
meet	185	9.8 Reduction and Iteration over Sets	189
diff	185	x in S	189
sdiff	185	&	189

Chapter 9

SETS

9.1 Introduction

A *set* in MAGMA is a (usually unordered) collection of objects belonging to some common structure (called the *universe* of the set). There are four basic types of sets: *enumerated sets*, whose elements are all stored explicitly (with one exception, see below); *formal sets*, whose elements are stored implicitly by means of a predicate that allows for testing membership; *indexed sets*, which are restricted enumerated sets having a numbering on elements; and *multisets*, which are enumerated sets with possible repetition of elements. In particular, enumerated and indexed sets and multisets are always finite, and formal sets are allowed to be infinite.

9.1.1 Enumerated Sets

Enumerated sets are finite, and can be specified in three basic ways (see also section 2 below): by listing all elements; by an expression involving elements of some finite structure; and by an arithmetic progression. If an arithmetic progression is specified, the elements are not calculated explicitly until a modification of the set necessitates it; in all other cases all elements of the enumerated set are stored explicitly.

9.1.2 Formal Sets

A formal set consists of the subset of elements of some carrier set (structure) on which a certain predicate assumes the value ‘true’.

The only set-theoretic operations that can be performed on formal sets are union, intersection, difference and symmetric difference, and element membership testing.

9.1.3 Indexed Sets

For some purposes it is useful to be able to access elements of a set through an index map, which numbers the elements of the set. For that purpose MAGMA has indexed sets, on which a very few basic set operations are allowed (element membership testing) as well as some sequence-like operations (such as accessing the i -th term, getting the index of an element, appending and pruning).

9.1.4 Multisets

For some purposes it is useful to construct a set with some of its members repeated. For that purpose MAGMA has multisets, which take into account the repetition of members. The number of times an object x occurs in a multiset S is called the *multiplicity* of x in S . MAGMA has the \wedge operator to specify a multiplicity: the expression $x \wedge n$ means the object x with multiplicity n . In the following, whenever any multiset constructor or function expects an element y , the expression $x \wedge n$ may usually be used.

9.1.5 Compatibility

The binary operators for sets do not allow mixing of the four types of sets (so one cannot take the intersection of an enumerated set and a formal set, for example), but it is easy to convert an enumerated set into a formal set – see the section on binary operators below – and there are functions provided for making an enumerated set out of an indexed set or a multiset (and vice versa).

By the limitation on their construction formal sets can only contain elements from one structure in MAGMA. The elements of enumerated sets are also restricted, in the sense that either some universe must be specified upon creation, or MAGMA must be able to find such universe automatically. The rules for compatibility of elements and the way MAGMA deals with these universes are the same for sequences and sets, and are described in the previous chapter. The restrictions on indexed sets are the same as those for enumerated sets.

9.1.6 Notation

Certain expressions appearing in the sections below (possibly with subscripts) have a standard interpretation:

U the universe: any MAGMA structure;

E the carrier set for enumerated sets: any enumerated structure (it must be possible to loop over its elements – see the Introduction to this Part (Chapter 8));

F the carrier set for formal sets: any structure for which membership testing using `in` is defined – see the Introduction to this Part (Chapter 8));

x a free variable which successively takes the elements of E (or F in the formal case) as its values;

P a Boolean expression that usually involves the variable(s) x, x_1, \dots, x_k ;

e an expression that also usually involves the variable(s) x, x_1, \dots, x_k .

9.2 Creating Sets

The customary braces $\{$ and $\}$ are used to define enumerated sets. Formal sets are delimited by the composite braces $\{!$ and $!\}$. For indexed sets $\{@$ and $@\}$ are used. For multisets $\{*$ and $*\}$ are used.

9.2.1 The Formal Set Constructor

The formal set constructor has the following fixed format (the expressions appearing in the construct are defined above):

 $\{! \quad x \text{ in } F \mid P(x) \quad !\}$

Form the formal set consisting of the subset of elements x of F for which $P(x)$ is true. If $P(x)$ is true for every element of F , the set constructor may be abbreviated to $\{! \quad x \text{ in } F \quad !\}$. Note that the universe of a formal set will always be equal to the carrier set F .

9.2.2 The Enumerated Set Constructor

Enumerated sets can be constructed by expressions enclosed in braces, provided that the values of all expressions can be automatically coerced into some common structure, as outlined in the Introduction, (Chapter 8). All general constructors have an optional universe (U in the list below) up front, that allows the user to specify into which structure all terms of the sets should be coerced.

$\{ \}$

The null set: an empty set that does not have its universe defined.

$\{ U \mid \}$

The empty set with universe U .

$\{ e_1, e_2, \dots, e_n \}$

Given a list of expressions e_1, \dots, e_n , defining elements a_1, a_2, \dots, a_n all belonging to (or automatically coercible into) a single algebraic structure U , create the set $\{ a_1, a_2, \dots, a_n \}$ of elements of U .

Example H9E1

We create a set by listing its elements explicitly.

```
> S := { (7^2+1)/5, (8^2+1)/5, (9^2-1)/5 };
> S;
{ 10, 13, 16 }
> Parent(S);
Set of subsets of Rational Field
```

Thus S was created as a set of rationals, because $/$ on integers has a rational result. If one wishes to obtain a set of integers, one could specify the universe (or one could use `div`, or one could use `!` on every element to coerce it into the ring of integers):

```
> T := { Integers() | (7^2+1)/5, (8^2+1)/5, (9^2-1)/5 };
> T;
{ 10, 13, 16 }
> Parent(T);
Set of subsets of Integer Ring
```

$\{ U \mid e_1, e_2, \dots, e_n \}$

Given a list of expressions e_1, \dots, e_n , which define elements a_1, a_2, \dots, a_n that are all coercible into U , create the set $\{ a_1, a_2, \dots, a_n \}$ of elements of U .

$\{ e(x) : x \text{ in } E \mid P(x) \}$

Form the set of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8) (in particular, E must be a finite structure that can be enumerated).

If $P(x)$ is true for every value of x in E , then the set constructor may be abbreviated to $\{ e(x) : x \text{ in } E \}$.

$\{ U \mid e(x) : x \text{ in } E \mid P(x) \}$

Form the set of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before.

If P is always true, it may be omitted (including the \mid).

$\{ e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k) \}$

The set consisting of those elements $e(x_1, \dots, x_k)$, in some common structure, for which x_1, \dots, x_k in E_1, \dots, E_k have the property that $P(x_1, \dots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8).

Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to x_i, x_{i+1} in E_i .

Also, if $P(x_1, \dots, x_k)$ is always true, it may be omitted (including the \mid).

$\{ U \mid e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k) \}$

As in the previous entry, the set consisting of those elements $e(x_1, \dots, x_k)$ for which $P(x_1, \dots, x_k)$ is true, is formed, as a set of elements of U (an error occurs if not all $e(x_1, \dots, x_k)$ are elements of or coercible into U).

Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example H9E2

Now that Fermat's last theorem may have been proven, it may be of interest to find integers that almost satisfy $x^n + y^n = z^n$. In this example we find all $2 < x, y, z < 1000$ such that $x^3 + y^3 = z^3 + 1$. First we build a set of cubes, then two sets of pairs for which the sum of cubes differs from a cube by 1. Note that we build a *set* rather than a sequence of cubes because we only need fast membership testing. Also note that the resulting sets of pairs do not have their elements in the order in which they were found.

```
> cubes := { Integers() | x^3 : x in [1..1000] };
> plus := { <a, b> : a in [2..1000], b in [2..1000] | \
>   b ge a and (a^3+b^3-1) in cubes };
> plus;
{
```

```

    < 9, 10 >,
    < 135, 235 >
    < 334, 438 >,
    < 73, 144 >,
    < 64, 94 >,
    < 244, 729 >
}

```

Note that we spend a lot of time cubing integers this way. For a more efficient approach, see a subsequent example.

9.2.3 The Indexed Set Constructor

The creation of indexed sets is similar to that of enumerated sets.

$\{\emptyset \emptyset\}$

The null set: an empty indexed set that does not have its universe defined.

$\{\emptyset U \mid \emptyset\}$

The empty indexed set with universe U .

$\{\emptyset e_1, e_2, \dots, e_n \emptyset\}$

Given a list of expressions e_1, \dots, e_n , defining elements a_1, a_2, \dots, a_n all belonging to (or automatically coercible into) a single algebraic structure U , create the indexed set $Q = \{a_1, a_2, \dots, a_n\}$ of elements of U .

$\{\emptyset U \mid e_1, e_2, \dots, e_m \emptyset\}$

Given a list of expressions e_1, \dots, e_m , which define elements a_1, a_2, \dots, a_n that are all coercible into U , create the indexed set $Q = \{a_1, a_2, \dots, a_n\}$ of elements of U .

$\{\emptyset e(x) : x \text{ in } E \mid P(x) \emptyset\}$

Form the indexed set of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8) (in particular, E must be a finite structure that can be enumerated).

If P is always true, it may be omitted (including the \mid).

$\{\emptyset U \mid e(x) : x \text{ in } E \mid P(x) \emptyset\}$

Form the indexed set of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before.

If P is always true, it may be omitted (including the \mid).

$$\{\textcircled{e}(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k) \textcircled{}\}$$

The indexed set consisting of those elements $e(x_1, \dots, x_k)$ (in some common structure), for which x_1, \dots, x_k in $E_1 \times \dots \times E_k$ have the property that $P(x_1, \dots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8).

Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to \mathbf{x}_i , \mathbf{x}_{i+1} in E_i .

Also, if $P(x_1, \dots, x_k)$ is always true, it may be omitted.

$$\{\textcircled{U} \mid e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k) \textcircled{}\}$$

As in the previous entry, the indexed set consisting of those elements $e(x_1, \dots, x_k)$ for which $P(x_1, \dots, x_k)$ is true is formed, as an indexed set of elements of U (an error occurs if not all $e(x_1, \dots, x_k)$ are elements of or coercible into U).

Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example H9E3

In the previous example we found pairs x, y such that $x^3 + y^3$ differs by one from some cube z^3 . Using indexed sets it is somewhat easier to retrieve the integer z as well. We give a small example. Note also that it is beneficial to know here that evaluation of expressions proceeds left to right.

```
> cubes := { @ Integers() | z^3 : z in [1..25] @ };
> plus := { <x, y, z> : x in [-10..10], y in [-10..10], z in [1..25] |
>   y ge x and Abs(x) gt 1 and Abs(y) gt 1 and (x^3+y^3-1) in cubes
>   and (x^3+y^3-1) eq cubes[z] };
> plus;
{ <-6, 9, 8>, <9, 10, 12>, <-8, 9, 6> }
```

9.2.4 The Multiset Constructor

The creation of multisets is similar to that of enumerated sets. An important difference is that repetitions are significant and the operator $\hat{}$ (mentioned above) may be used to specify the multiplicity of an element.

$$\{ * \}$$

The null set: an empty multiset that does not have its universe defined.

$$\{ * U \mid * \}$$

The empty multiset with universe U .

$$\{ * e_1, e_2, \dots, e_n * \}$$

Given a list of expressions e_1, \dots, e_n , defining elements a_1, a_2, \dots, a_n all belonging to (or automatically coercible into) a single algebraic structure U , create the multiset $Q = \{ * a_1, a_2, \dots, a_n * \}$ of elements of U .

$$\{ * U \mid e_1, e_2, \dots, e_m * \}$$

Given a list of expressions e_1, \dots, e_m , which define elements a_1, a_2, \dots, a_n that are all coercible into U , create the multiset $Q = \{ * a_1, a_2, \dots, a_n * \}$ of elements of U .

$$\{ * e(x) : x \text{ in } E \mid P(x) * \}$$

Form the multiset of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8) (in particular, E must be a finite structure that can be enumerated).

If P is always true, it may be omitted (including the $|$).

$$\{ * U \mid e(x) : x \text{ in } E \mid P(x) * \}$$

Form the multiset of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as before.

If P is always true, it may be omitted (including the $|$).

$$\{ * e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k) * \}$$

The multiset consisting of those elements $e(x_1, \dots, x_k)$ (in some common structure), for which x_1, \dots, x_k in $E_1 \times \dots \times E_k$ have the property that $P(x_1, \dots, x_k)$ is true. The expressions appearing in this construct have the interpretation given in the Introduction (Chapter 8).

Note that if two successive allowable structures E_i and E_{i+1} are identical, then the specification of the carrier sets for x_i and x_{i+1} may be abbreviated to $\mathbf{x}_i, \mathbf{x}_{i+1}$ in E_i .

Also, if $P(x_1, \dots, x_k)$ is always true, it may be omitted.

$$\{ * U \mid e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k) * \}$$

As in the previous entry, the multiset consisting of those elements $e(x_1, \dots, x_k)$ for which $P(x_1, \dots, x_k)$ is true is formed, as an multiset of elements of U (an error occurs if not all $e(x_1, \dots, x_k)$ are elements of or coercible into U).

Again, identical successive structures may be abbreviated, and a predicate that is always true may be omitted.

Example H9E4

Here we demonstrate the use of the multiset constructors.

```
> M := { * 1, 1, 1, 3, 5 * };
> M;
{ * 1^3, 3, 5 * }
> M := { * 1^4, 2^5, 1/2^3 * };
> M;
> // Count frequency of digits in first 1000 digits of pi:
> pi := Pi(RealField(1001));
> dec1000 := Round(10^1000*(pi-3));
> I := IntegerToString(dec1000);
> F := { * I[i]: i in [1 .. #I] * };
> F;
{ * 7^95, 3^102, 6^94, 2^103, 9^106, 5^97,
  1^116, 8^101, 4^93, 0^93 * }
> for i := 0 to 9 do i, Multiplicity(F, IntegerToString(i)); end for;
0 93
1 116
2 103
3 102
4 93
5 97
6 94
7 95
8 101
9 106
```

9.2.5 The Arithmetic Progression Constructors

Some special constructors exist to create and store enumerated sets of integers in arithmetic progression efficiently. This only works for arithmetic progressions of elements of the ring of integers.

$$\boxed{\{ i..j \}}$$

$$\boxed{\{ U \mid i..j \}}$$

The enumerated set whose elements form the arithmetic progression $i, i+1, i+2, \dots, j$, where i and j are (expressions defining) integers. If j is less than i then the empty set will be created.

The only universe U that is legal here is the ring of integers.

$\{ i \dots j \text{ by } k \}$

$\{ U \mid i \dots j \text{ by } k \}$

The enumerated set consisting of the integers forming the arithmetic progression $i, i+k, i+2*k, \dots, j$, where i, j and k are (expressions defining) integers (but $k \neq 0$).

If k is positive then the last element in the progression will be the greatest integer of the form $i + n*k$ that is less than or equal to j . If j is less than i , the empty set will be constructed.

If k is negative then the last element in the progression will be the least integer of the form $i + n*k$ that is greater than or equal to j . If j is greater than i , the empty set will be constructed.

As for the previous constructor, only the ring of integers is allowed as a legal universe U .

Example H9E5

It is possible to use the arithmetic progression constructors to save typing in the creation of ‘arithmetic progressions’ of elements of other structures than the ring of integers, but it should be kept in mind that the result will not be treated especially efficiently like the integer case. Here is the ‘wrong’ way, as well as two correct ways to create a set of 10 finite field elements.

```
> S := { FiniteField(13) | 1..10 };
Runtime error in { .. }: Invalid set universe
> S := { FiniteField(13) | x : x in { 1..10 } };
> S;
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
> G := PowerSet(FiniteField(13));
> S := G ! { 1..10 };
> S;
{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

9.3 Power Sets

The `PowerSet` constructor returns a structure comprising the subsets of a given structure R ; it is mainly useful as a parent for other set and sequence constructors. The only operations that are allowed on power sets are printing, testing element membership, and coercion into the power set (see the examples below).

`PowerSet(R)`

The structure comprising all enumerated subsets of structure R .

`PowerIndexedSet(R)`

The structure comprising all indexed subsets of structure R .

PowerMultiset(*R*)

The structure consisting of all submultisets of the structure *R*.

S* in *P

Returns **true** if enumerated set *S* is in the power set *P*, that is, if all elements of the set *S* are contained in or coercible into *R*, where *P* is the power set of *R*; **false** otherwise.

PowerFormalSet(*R*)

The structure comprising all formal subsets of structure *R*.

S* in *P

Returns **true** if indexed set *S* is in the power set *P*, that is, if all elements of the set *S* are contained in or coercible into *R*, where *P* is the power set of *R*; **false** otherwise.

S* in *P

Returns **true** if multiset *S* is in the power set *P*, that is, if all elements of the set *S* are contained in or coercible into *R*, where *P* is the power set of *R*; **false** otherwise.

P* ! *S

Return a set with universe *R* consisting of the elements of the set *S*, where *P* is the power set of *R*. An error results if not all elements of *S* can be coerced into *R*.

P* ! *S

Return an indexed set with universe *R* consisting of the elements of the set *S*, where *P* is the power set of *R*. An error results if not all elements of *S* can be coerced into *R*.

P* ! *S

Return a multiset with universe *R* consisting of the elements of the set *S*, where *P* is the power set of *R*. An error results if not all elements of *S* can be coerced into *R*.

Example H9E6

```

> S := { 1 .. 10 };
> P := PowerSet(S);
> P;
Set of subsets of { 1 .. 10 }
> F := { 6/3, 12/4 };
> F in P;
true
> G := P ! F;
> Parent(F);
Set of subsets of Rational Field
> Parent(G);
Set of subsets of { 1 .. 10 }

```

9.3.1 The Cartesian Product Constructors

Using `car< >` and `CartesianProduct()`, it is possible to create the Cartesian product of sets (or, in fact, of any combination of structures), but the result will be of type ‘Cartesian product’ rather than set, and the elements are tuples – we refer the reader to Chapter 11 for details.

9.4 Sets from Structures

Set(M)

Given a finite structure that allows explicit enumeration of its elements, return the set containing its elements (having M as its universe).

FormalSet(M)

Given a structure M , return the formal set consisting of its elements.

9.5 Accessing and Modifying Sets

Enumerated sets can be modified by inserting or removing elements. Indexed sets allow some sequence-like operators for modification and access.

9.5.1 Accessing Sets and their Associated Structures

#R

Cardinality of the enumerated, indexed, or multi- set R . Note that for a multiset, repetitions are significant, so the result may be greater than the underlying set.

Category(S)

Type(S)

The category of the object S . For a set this will be one of `SetEnum`, `SetIndx`, `SetMulti`, or `SetFormal`. For a power set the type is one of `PowSetEnum`, `PowSetIndx`, `PowSetMulti`.

Parent(R)

Returns the parent structure of R , that is, the structure consisting of all (enumerated) sequences over the universe of R .

Universe(R)

Returns the ‘universe’ of the (enumerated or indexed or multi- or formal) set R , that is, the common structure to which all elements of the set belong. An error is signalled when R is the null set.

Index(S, x)

Position(S, x)

Given an indexed set S , and an element x , returns the index i such that $S[i] = x$ if such index exists, or return 0 if x is not in S . If x is not in the universe of S , an attempt will be made to coerce it; an error occurs if this fails.

S[i]

Return the i -th entry of indexed set S . If $i < 1$ or $i > \#S$ an error occurs. Note that indexing is *not* allowed on the left hand side.

S[I]

The indexed set $\{S[i_1], \dots, S[i_r]\}$ consisting of terms selected from the indexed set S , according to the terms of the integer sequence I . If any term of I lies outside the range 1 to $\#S$, then an error results. If I is the empty sequence, then the empty set with universe the same as that of S is returned.

Example H9E7

We build an indexed set of sets to illustrate the use of the above functions.

```
> B := { @ { i : i in [1..k] } : k in [1..5] @ };
> B;
{ @
  { 1 },
  { 1, 2 },
  { 1, 2, 3 },
  { 1, 2, 3, 4 },
  { 1, 2, 3, 4, 5 },
@}
> #B;
5
> Universe(B);
Set of subsets of Integer Ring
> Parent(B);
Set of indexed subsets of Set of subsets of Integer Ring
> Category(B);
SetIndx
> Index(B, { 2, 1 });
2
> #B[2];
2
> Universe(B[2]);
Integer Ring
```

9.5.2 Selecting Elements of Sets

Most finite structures in MAGMA, including enumerated sets, allow one to obtain a random element using **Random**. There is an alternative (and often preferable) option for enumerated sets in the **random**{ } constructor. This makes it possible to choose a random element of the set without generating the whole set first.

Likewise, **rep**{ } is an alternative to the general **Rep** function returning a representative element of a structure, having the advantage of aborting the construction of the set as soon as one element has been found.

Here, E will again be an enumerable structure, that is, a structure that allows enumeration of its elements (see the Appendix for an exhaustive list).

Note that **random**{ $e(x) : x \text{ in } E \mid P(x)$ } does *not* return a random element of the set of values $e(x)$, but rather a value of $e(x)$ for a random x in E which satisfies P (and *mutatis mutandis* for **rep**).

See the subsection on Notation in the Introduction (Chapter 8) for conventions regarding e, x, E, P .

Random(*R*)

A random element chosen from the enumerated, indexed or multi- set R . Every element has an equal probability of being chosen for enumerated or indexed sets, and a weighted probability in proportion to its multiplicity for multisets. Successive invocations of the function will result in independently chosen elements being returned as the value of the function. If R is empty an error occurs.

random{ $e(x) : x \text{ in } E \mid P(x)$ }

Given an enumerated structure E and a Boolean expression P , return the value of the expression $e(y)$ for a randomly chosen element y of E for which $P(y)$ is true.

P may be omitted if it is always true.

random{ $e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k)$ }

Given enumerated structures E_1, \dots, E_k , and a Boolean expression $P(x_1, \dots, x_k)$, return the value of the expression $e(y_1, \dots, y_k)$ for a randomly chosen element $\langle y_1, \dots, y_k \rangle$ of $E_1 \times \dots \times E_k$, for which $P(y_1, \dots, y_k)$ is true.

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation x_i, x_{i+1} in E_i may be used.

Example H9E8

Here are two ways to find a ‘random’ primitive element for a finite field.

```
> p := 10007;
> F := FiniteField(p);
> proots := { z : z in F | IsPrimitive(z) };
> #proots;
5002
> Random(proots);
5279
```

This way, a set of 5002 elements is built (and primitivity is checked for all elements of F), and a random choice is made. Alternatively, we use **random**.

```
> random{ x : x in F | IsPrimitive(x) };
4263
```

In this case random elements in F are chosen until one is found that is primitive. Since almost half of F ’s elements are primitive, only very few primitivity tests will be done before success occurs.

Representative(*R*)**Rep(*R*)**

An arbitrary element chosen from the enumerated, indexed, or multi- set R .

ExtractRep($\sim R$, $\sim r$)

Assigns an arbitrary element chosen from the enumerated set R to r , and removes it from R . Thus the set R is modified, as well as the element r . An error occurs if R is empty.

rep{ $e(x) : x \text{ in } E \mid P(x)$ }

Given an enumerated structure E and a Boolean expression P , return the value of the expression $e(y)$ for the first element y of E for which $P(y)$ is true. If $P(x)$ is false for every element of E , an error will occur.

rep{ $e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k)$ }

Given enumerated structures E_1, \dots, E_k , and a Boolean expression $P(x_1, \dots, x_k)$, return the value of the expression $e(y_1, \dots, y_k)$ for the first element $\langle y_1, \dots, y_k \rangle$ of $E_1 \times \dots \times E_k$, for which $P(y_1, \dots, y_k)$ is true. An error occurs if no element of $E_1 \times \dots \times E_k$ satisfies P .

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation $\mathbf{x}_i, \mathbf{x}_{i+1}$ in E_i may be used.

Example H9E9

As an illustration of the use of **ExtractRep**, we modify an earlier example, and find cubes satisfying $x^3 + y^3 = z^3 - 1$ (with $x, y, z \leq 1000$).

```
> cubes := { Integers() | x^3 : x in [1..1000] };
> cc := cubes;
> min := { };
> while not IsEmpty(cc) do
>   ExtractRep(~cc, ~a);
>   for b in cc do
>     if a+b+1 in cubes then
>       min join:= { <a, b> };
>     end if;
>   end for;
> end while;
> { < Iroot(x[1], 3), Iroot(x[2], 3) > : x in min };
{ <138, 135>, <823, 566>, <426, 372>, <242, 720>,
  <138, 71>, <426, 486>, <6, 8> }
```

Note that instead of taking cubes over again, we only have to take cube roots in the last line (on the small resulting set) once.

Minimum(<i>S</i>)

Min(<i>S</i>)

Given a non-empty enumerated, indexed, or multi- set *S*, such that **lt** and **eq** are defined on the universe of *S*, this function returns the minimum of the elements of *S*. If *S* is an indexed set, the position of the minimum is also returned.

Maximum(<i>S</i>)

Max(<i>S</i>)

Given a non-empty enumerated, indexed, or multi- set *S*, such that **lt** and **eq** are defined on the universe of *S*, this function returns the maximum of the elements of *S*. If *S* is an indexed set, the position of the maximum is also returned.

Hash(<i>x</i>)

Given a Magma object *x* which can be placed in a set, return the hash value of *x* used by the set machinery. This is a fixed but arbitrary non-negative integer (whose maximum value is the maximum value of a C unsigned long on the particular machine). The crucial property is that if *x* and *y* are objects and *x* equals *y* then the hash values of *x* and *y* are equal (even if *x* and *y* have different internal structures). Thus one could implement sets manually if desired by the use of this function.

9.5.3 Modifying Sets

Include($\sim S$, <i>x</i>)

Include(<i>S</i> , <i>x</i>)

Create the enumerated, indexed, or multi- set obtained by putting the element *x* in *S* (*S* is unchanged if *S* is not a multiset and *x* is already in *S*). If *S* is an indexed set, the element will be appended at the end. If *S* is a multiset, the multiplicity of *x* will be increased accordingly. If *x* is not in the universe of *S*, an attempt will be made to coerce it; an error occurs if this fails.

There are two versions of this: a procedure, where *S* is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to *S* as an argument.

Note that the procedural version is much more efficient since the set *S* will not be copied.

Exclude($\sim S$, <i>x</i>)

Exclude(<i>S</i> , <i>x</i>)

Create a new set by removing the element *x* from *S*. If *S* is an enumerated set, nothing happens if *x* is not in *S*. If *S* is a multiset, the multiplicity of *x* will be decreased accordingly. If *x* is not in the universe of *S*, an attempt will be made to coerce it; an error occurs if this fails.

There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the set S will not be copied.

<code>ChangeUniverse($\sim S$, V)</code>
--

<code>ChangeUniverse(S, V)</code>

Given an enumerated, indexed, or multi- set S with universe U and a structure V which contains U , construct a new set of the same type which consists of the elements of S coerced into V .

There are two versions of this: a procedure, where S is replaced by the new set, and a function, which returns the new set. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the set S will not be copied.

<code>CanChangeUniverse(S, V)</code>
--

Given an enumerated, indexed, or multi- set S with universe U and a structure V which contains U , attempt to construct a new set T of the same type which consists of the elements of S coerced into V ; if successful, return `true` and T , otherwise return `false`.

Example H9E10

This example uses `Include` and `Exclude` to find a set (if it exists) of cubes of integers such that the elements of a given set R can be expressed as the sum of two of those.

```
> R := { 218, 271, 511 };
> x := 0;
> cubes := { 0 };
> while not IsEmpty(R) do
>   x += 1;
>   c := x^3;
>   Include(~cubes, c);
>   Include(~cubes, -c);
>   for z in cubes do
>     Exclude(~R, z+c);
>     Exclude(~R, z-c);
>   end for;
> end while;
```

We did not record how the elements of R were obtained as sums of a pair of cubes. For that, the following suffices.

```
> R := { 218, 271, 511 }; // it has been emptied !
> { { x, y } : x, y in cubes | x+y in R };
```

```
{
  { -729, 1000 },
  { -125, 343 },
  { -1, 512 },
}
```

SetToIndexedSet(E)

Given an enumerated set E , this function returns an indexed set with the same elements (and universe) as E .

IndexedSetToSet(S)

Isetset(S)

Given an indexed set S , this function returns an enumerated set with the same elements (and universe) as E .

IndexedSetToSequence(S)

Isetseq(S)

Given an indexed set S , this function returns a sequence with the same elements (and universe) as E .

MultisetToSet(S)

Given a multiset S , this function returns an enumerated set with the same elements (and universe) as S .

SetToMultiset(E)

Given an enumerated set E , this function returns a multiset with the same elements (and universe) as E .

SequenceToMultiset(Q)

Given an enumerated sequence E , this function returns a multiset with the same elements (and universe) as E .

9.6 Operations on Sets

9.6.1 Boolean Functions and Operators

As explained in the Introduction (Chapter 8), when elements are taken out of a set their parent will be the universe of the set (or, if the universe is itself a set, the universe of the universe, etc.); in particular, the set itself is not the parent. Hence equality testing on set elements is in fact equality testing between two elements of certain algebraic structures, and the sets are irrelevant. We only list the (in)equality operator for convenience here.

Element membership testing is of critical importance for all types of sets.

Testing whether or not R is a subset of S can be done if R is an enumerated or indexed set and S is any set; hence (in)equality testing is only possible between sets that are not formal sets.

IsNull(R)

Returns **true** if and only if the enumerated, indexed, or multi- set R is empty and does not have its universe defined.

IsEmpty(R)

Returns **true** if and only if the enumerated, indexed or multi- set R is empty.

$x \text{ eq } y$

Given an element x of a set R with universe U and an element y of a set S with universe V , where a common overstructure W can be found with $U \subset W \supset V$ (see the Introduction (Chapter 8) for details on overstructures), return **true** if and only if x and y are equal as elements of W .

$x \text{ ne } y$

Given an element x of a set R with universe U and an element y of a set S with universe V , where a common overstructure W can be found with $U \subset W \supset V$ (see the Introduction (Chapter 8) for details on overstructures), return **true** if and only if x and y are distinct as elements of W .

$x \text{ in } R$

Returns **true** if and only if the element x is a member of the set R . If x is not an element of the universe U of R , it is attempted to coerce x into U ; if this fails, an error occurs.

$x \text{ notin } R$

Returns **true** if and only if the element x is not a member of the set R . If x is not an element of the parent structure U of R , it is attempted to coerce x into U ; if this fails, an error occurs.

R subset S

Returns **true** if the enumerated, indexed or multi- set R is a subset of the set S , **false** otherwise. For multisets, if an element x of R has multiplicity n in R , the multiplicity of x in S must be at least n . Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

R notsubset S

Returns **true** if the enumerated, indexed, or multi- set R is a not a subset of the set S , **false** otherwise. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

R eq S

Returns **true** if and only if R and S are identical sets, where R and S are enumerated, indexed or multi- sets For indexed sets, the index function is irrelevant for deciding equality. For multisets, matching multiplicities must also be equal. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

R ne S

Returns **true** if and only if R and S are distinct sets, where R and S are enumerated indexed, or multi- sets. For indexed sets, the index function is irrelevant for deciding equality. For multisets, matching multiplicities must also be equal. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

IsDisjoint(R, S)

Returns **true** iff the enumerated, indexed or multi- sets R and S are disjoint. Coercion of the elements of R into S is attempted if necessary, and an error occurs if this fails.

9.6.2 Binary Set Operators

For each of the following operators, R and S are sets of the same type. If R and S are both formal sets, then an error will occur unless both have been constructed with the same carrier structure F in the definition. If R and S are both enumerated, indexed, or multisets, then an error occurs unless the universes of R and S are compatible, as defined in the Introduction to this Part (Chapter 8).

Note that

$Q := \{ ! x \text{ in } R ! \}$

converts an enumerated set R into a formal set Q .

R join S

Union of the sets R and S (see above for the restrictions on R and S). For multisets, matching multiplicities are added in the union.

R meet S

Intersection of the sets R and S (see above for the restrictions on R and S). For multisets, the minimum of matching multiplicities is stored in the intersection.

R diff S

Difference of the sets R and S . i.e., the set consisting of those elements of R which are not members of S (see above for the restrictions on R and S). For multisets, the difference contains any elements of R remaining after removing the corresponding elements of S the appropriate number of times.

R sdiff S

Symmetric difference of the sets R and S . i.e., the set consisting of those elements which are members of either R or S but not both (see above for the restrictions on R and S). Alternatively, it is the union of the difference of R with S and the difference of S with R .

Example H9E11

```
> R := { 1, 2, 3 };
> S := { 1, 1/2, 1/3 };
> R join S;
{ 1/3, 1/2, 1, 2, 3 }
> R meet S;
{ 1 }
> R diff S;
{ 2, 3 }
> S diff R;
{ 1/3, 1/2 }
> R sdiff S;
{ 1/3, 1/2, 2, 3 }
```

9.6.3 Other Set Operations**Multiplicity(S, x)**

Return the multiplicity in multiset S of element x . If x is not in S , zero is returned.

Multiplicities(S)

Returns the sequence of multiplicities of distinct elements in the multiset S . The order is the same as the internal enumeration order of the elements.

Subsets(S)

The set of all subsets of S .

Subsets(S , k)

The set of subsets of S of size k . If k is larger than the cardinality of S then the result will be empty.

RandomSubset(S , k)

A random subset of S of size k . It is an error if k is larger than the size of S .

Multisets(S , k)

The set of multisets consisting of k not necessarily distinct elements of S .

Subsequences(S , k)

The set of sequences of length k with elements from S .

Permutations(S)

The set of permutations (stored as sequences) of the elements of S .

Permutations(S , k)

The set of permutations (stored as sequences) of each of the subsets of S of cardinality k .

9.7 Quantifiers

To test whether some enumerated set is empty or not, one may use the `IsEmpty` function. However, to use `IsEmpty`, the set has to be created in full first. The existential quantifier `exists` enables one to do the test and abort the construction of the set as soon as an element is found; moreover, the element found will be assigned to a variable.

Likewise, `forall` enables one to abort the construction of the set as soon as an element not satisfying a certain property is encountered.

Note that `exists(t){ e(x) : x in E | P(x) }` is *not* designed to return true if an element of the set of values $e(x)$ satisfies P , but rather if there is an $x \in E$ satisfying $P(x)$ (in which case $e(x)$ is assigned to t).

For the notation used here, see the beginning of this chapter.

`exists(t){ e(x): x in E | P(x) }`

`exists(t1, ..., tr){ e(x) : x in E | P(x) }`

Given an enumerated structure E and a Boolean expression $P(x)$, the Boolean value true is returned if E contains at least one element x for which $P(x)$ is true. If $P(x)$ is not true for any element x of E , then the Boolean value false is returned.

Moreover, if $P(x)$ is found to be true for the element y , say, of E , then in the first form of the `exists` expression, variable t will be assigned the value of the expression $e(y)$. If $P(x)$ is never true for an element of E , t will be left unassigned. In the second form, where r variables t_1, \dots, t_r are given, the result $e(y)$ should be a tuple of length r ; each variable will then be assigned to the corresponding component of

the tuple. Similarly, all the variables will be left unassigned if $P(x)$ is never true. The clause (t) may be omitted entirely.

P may be omitted if it is always true.

$\text{exists}(t)\{e(x_1, \dots, x_k): x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k)\}$

$\text{exists}(t_1, \dots, t_r)\{ e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P \}$

Given enumerated structures E_1, \dots, E_k , and a Boolean expression $P(x_1, \dots, x_k)$, the Boolean value true is returned if there is an element $\langle y_1, \dots, y_k \rangle$ in the Cartesian product $E_1 \times \dots \times E_k$, such that $P(y_1, \dots, y_k)$ is true. If $P(x_1, \dots, x_k)$ is not true for any element (y_1, \dots, y_k) of $E_1 \times \dots \times E_k$, then the Boolean value false is returned.

Moreover, if $P(x_1, \dots, x_k)$ is found to be true for the element $\langle y_1, \dots, y_k \rangle$ of $E_1 \times \dots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \dots, y_k)$. If $P(x_1, \dots, x_k)$ is never true for an element of $E_1 \times \dots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables t_1, \dots, t_r are given, the result $e(y_1, \dots, y_k)$ should be a tuple of length r ; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x_1, \dots, x_k)$ is never true. The clause (t) may be omitted entirely.

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation $\mathbf{x}_i, \mathbf{x}_{i+1}$ in E_i may be used.

Example H9E12

As a variation on an earlier example, we check whether or not some integers can be written as sums of cubes (less than 10^3 in absolute value):

```
> exists(t){ <x, y> : x, y in [ t^3 : t in [-10..10] ] | x + y eq 218 };
true
> t;
<-125, 343>
> exists(t){ <x, y> : x, y in [ t^3 : t in [1..10] ] | x + y eq 218 };
false
> t;
>> t;
```

User error: Identifier 't' has not been declared

forall(<i>t</i>) { <i>e</i> (<i>x</i>) : <i>x</i> in <i>E</i> <i>P</i> (<i>x</i>) }

forall(<i>t</i> ₁ , ..., <i>t</i> _{<i>r</i>}) { <i>e</i> (<i>x</i>) : <i>x</i> in <i>E</i> <i>P</i> (<i>x</i>) }
--

Given an enumerated structure E and a Boolean expression $P(x)$, the Boolean value true is returned if $P(x)$ is true for every element x of E .

If $P(x)$ is not true for at least one element x of E , then the Boolean value false is returned.

Moreover, if $P(x)$ is found to be false for the element y , say, of E , then in the first form of the exists expression, variable t will be assigned the value of the expression $e(y)$. If $P(x)$ is true for every element of E , t will be left unassigned. In the second form, where r variables t_1, \dots, t_r are given, the result $e(y)$ should be a tuple of length r ; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x)$ is always true. The clause (*t*) may be omitted entirely.

P may be omitted if it is always true.

forall(<i>t</i>) { <i>e</i> (<i>x</i> ₁ , ..., <i>x</i> _{<i>k</i>}) : <i>x</i> ₁ in <i>E</i> ₁ , ..., <i>x</i> _{<i>k</i>} in <i>E</i> _{<i>k</i>} <i>P</i> (<i>x</i> ₁ , ..., <i>x</i> _{<i>k</i>}) }

forall(<i>t</i> ₁ , ..., <i>t</i> _{<i>r</i>}) { <i>e</i> (<i>x</i> ₁ , ..., <i>x</i> _{<i>k</i>}) : <i>x</i> ₁ in <i>E</i> ₁ , ..., <i>x</i> _{<i>k</i>} in <i>E</i> _{<i>k</i>} <i>P</i> }
--

Given sets E_1, \dots, E_k , and a Boolean expression $P(x_1, \dots, x_k)$, the Boolean value true is returned if $P(x_1, \dots, x_k)$ is true for every element (x_1, \dots, x_k) in the Cartesian product $E_1 \times \dots \times E_k$.

If $P(x_1, \dots, x_k)$ fails to be true for some element (y_1, \dots, y_k) of $E_1 \times \dots \times E_k$, then the Boolean value false is returned.

Moreover, if $P(x_1, \dots, x_k)$ is false for the element $\langle y_1, \dots, y_k \rangle$ of $E_1 \times \dots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \dots, y_k)$. If $P(x_1, \dots, x_k)$ is true for every element of $E_1 \times \dots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables t_1, \dots, t_r are given, the result $e(y_1, \dots, y_k)$ should be a tuple of length r ; each variable will then be assigned to the corresponding component of the tuple. Similarly, all the variables will be left unassigned if $P(x_1, \dots, x_k)$ is never true. The clause (*t*) may be omitted entirely.

P may be omitted if it is always true.

If successive structures E_i and E_{i+1} are identical, then the abbreviation \mathbf{x}_i , \mathbf{x}_{i+1} in E_i may be used.

Example H9E13

This example shows that **forall** and **exists** may be nested.

It is well known that every prime that is 1 modulo 4 can be written as the sum of two squares, but not every integer m congruent to 1 modulo 4 can. In this example we explore for small m whether perhaps $m \pm \epsilon$ (with $|\epsilon| \leq 1$) is always a sum of squares.

```
> forall(u) { m : m in [5..1000 by 4] |
>     exists { <x, y, z> : x, y in [0..30], z in [-1, 0, 1] |
>         x^2+y^2+z eq m } };
```



```
false
> u;
77
```

9.8 Reduction and Iteration over Sets

Both enumerated and indexed sets allow enumeration of their elements; formal sets do not. For indexed sets the enumeration will occur according to the order given by the indexing.

Instead of using a loop to apply the same binary associative operator to all elements of an enumerated or indexed set, it is in certain cases possible to use the *reduction operator* `&`.

x in S

Enumerate the elements of an enumerated or indexed set S . This can be used in *loops*, as well as in the set and sequence *constructors*.

&o S

Given an enumerated or indexed set $S = \{a_1, a_2, \dots, a_n\}$ of elements belonging to an algebraic structure U , and an (associative) operator $\circ : U \times U \rightarrow U$, form the element $a_{i_1} \circ a_{i_2} \circ a_{i_3} \circ \dots \circ a_{i_n}$, for some permutation i_1, \dots, i_n of $1, \dots, n$.

Currently, the following operators may be used to reduce enumerated sets: `+`, `*`, `and`, `or`, `join`, `meet` and `+`, `*`, `and`, `or` to reduce indexed sets. An error will occur if the operator is not defined on U .

If S contains a single element a , then the value returned is a . If S is the null set (empty and no universe specified) or S is empty with universe U (and the operation is defined in U), then the result (or error) depends on the operation and upon U . The following table defines the return value:

	<i>empty</i>	<i>null</i>
<code>&+</code>	$U \neq 0$	error
<code>&*</code>	$U \neq 1$	error
<code>&and</code>	true	true
<code>&or</code>	false	false
<code>&join</code>	<i>empty</i>	<i>null</i>
<code>&meet</code>	error	error

Warning: since the reduction may take place in an arbitrary order on the arguments a_1, \dots, a_n , the result is not unambiguously defined if the operation is not commutative on the arguments!

Example H9E14

The function `choose` defined below takes a set S and an integer k as input, and produces a set of all subsets of S with cardinality k .

```
> function choose(S, k)
>   if k eq 0 then
>     return { { } };
>   else
>     return &join{{ s join { x } : s in choose(S diff { x }, k-1) } : x in S};
>   end if;
> end function;
```

So, for example:

```
> S := { 1, 2, 3, 4 };
> choose(S, 2);
{
  { 1, 3 },
  { 1, 4 },
  { 2, 4 },
  { 2, 3 },
  { 1, 2 },
  { 3, 4 }
}
```

Try to guess what happens if $k < 0$.

10 SEQUENCES

10.1 Introduction	193		
10.1.1 Enumerated Sequences	193	Position(S, x, f)	199
10.1.2 Formal Sequences	193	Representative(R)	199
10.1.3 Compatibility	194	Rep(R)	199
10.2 Creating Sequences	194	Random(R)	200
10.2.1 The Formal Sequence Constructor .	194	Explode(R)	200
[! x in F P(x) !]	194	Eltseq(R)	200
10.2.2 The Enumerated Sequence Construc-		10.4.3 Modifying Enumerated Sequences .	200
tor	195	Append(~S, x)	200
[]	195	Append(S, x)	200
[U]	195	Exclude(~S, x)	200
[e ₁ , e ₂ , ..., e _n]	195	Exclude(S, x)	200
[U e ₁ , e ₂ , ..., e _m]	195	Include(~S, x)	201
[e(x) : x in E P(x)]	195	Include(S, x)	201
[U e(x) : x in E P(x)]	195	Insert(~S, i, x)	201
[e(x ₁ , ..., x _k) : x ₁ in E ₁ , ..., x _k		Insert(S, i, x)	201
in E _k P(x ₁ , ..., x _k)]	195	Insert(~S, k, m, T)	201
[U e(x ₁ , ..., x _k) : x ₁ in E ₁ , ...,		Insert(S, k, m, T)	201
x _k in E _k P(x ₁ , ..., x _k)]	196	Prune(~S)	202
10.2.3 The Arithmetic Progression Construc-		Prune(S)	202
tors	196	Remove(~S, i)	202
[i..j]	196	Remove(S, i)	202
[U i..j]	196	Reverse(~S)	202
[i .. j by k]	196	Reverse(S)	202
[U i .. j by k]	196	Rotate(~S, p)	202
10.2.4 Literal Sequences	197	Rotate(S, p)	202
\[m ₁ , ..., m _n]	197	Sort(~S)	203
10.3 Power Sequences	197	Sort(S)	203
PowerSequence(R)	197	Sort(~S, C)	203
in	197	Sort(~S, C, ~p)	203
!	197	Sort(S, C)	203
10.4 Operators on Sequences	198	ParallelSort(~S, ~T)	203
10.4.1 Access Functions	198	Undefine(~S, i)	203
#	198	Undefine(S, i)	203
Parent(S)	198	ChangeUniverse(S, V)	204
Universe(S)	198	ChangeUniverse(S, V)	204
S[i]	198	CanChangeUniverse(S, V)	204
10.4.2 Selection Operators on Enumerated		10.4.4 Creating New Enumerated Sequences	
Sequences	199	from Existing Ones	205
S[I]	199	cat	205
Minimum(S)	199	cat:=	205
Min(S)	199	Partition(S, p)	205
Maximum(S)	199	Partition(S, P)	206
Max(S)	199	Setseq(S)	206
Index(S, x)	199	SetToSequence(S)	206
Index(S, x, f)	199	Seqset(S)	206
Position(S, x)	199	SequenceToSet(S)	206
		And(S, T)	207
		And(~S, T)	207
		Or(S, T)	207
		Or(~S, T)	207
		Xor(S, T)	207
		Xor(~S, T)	207
		Not(S)	207
		Not(~S)	207

10.5 Predicates on Sequences . . .	208		
IsComplete(S)	208	ge	210
IsDefined(S, i)	208	gt	210
IsEmpty(S)	208	10.6 Recursion, Reduction, and Iteration	210
IsNull(S)	208	10.6.1 Recursion	210
10.5.1 Membership Testing	208	Self(n)	210
in	208	Self()	210
notin	208	10.6.2 Reduction	211
IsSubsequence(S, T)	209	&	211
IsSubsequence(S, T: Kind := o)	209	10.7 Iteration	211
eq	209	for x in S do st; end for;	211
ne	209	10.8 Bibliography	212
10.5.2 Testing Order Relations	209		
lt	209		
le	209		

Chapter 10

SEQUENCES

10.1 Introduction

A *sequence* in MAGMA is a linearly ordered collection of objects belonging to some common structure (called the *universe* of the sequence).

There are two types of sequence: *enumerated sequences*, of which the elements are all stored explicitly (with one exception, see below); and *formal sequences*, of which elements are stored implicitly by means of a predicate that allows for testing membership. In particular, enumerated sequences are always finite, and formal sequences are allowed to be infinite. In this chapter a *sequence* will be either a formal or an enumerated sequence.

10.1.1 Enumerated Sequences

An *enumerated sequence of length l* is an array of indefinite length of which only finitely many terms – including the l -th term, but no term of bigger index — have been defined to be elements of some common structure. Such sequence is called *complete* if all of the terms (from index 1 up to the length l) are defined.

In practice the length of any sequence is bounded by the constant integer `_beta` (usually 2^{29}).

Incomplete enumerated sequences are allowed as a convenience for the programmer in building complete enumerated sequences. Some sequence functions require their arguments to be complete; if that is the case, it is mentioned explicitly in the description below. However, all functions using sequences in *other* MAGMA modules always assume that a sequence that is passed in as an argument is complete. Note that the following line converts a possibly incomplete sequence S into a complete sequence T :

```
T := [ s : s in S ];
```

because the enumeration using the `in` operator simply ignores undefined terms.

Enumerated sequences of *Booleans* are highly optimized (stored as bit-vectors).

10.1.2 Formal Sequences

A formal sequence consists of elements of some range set on which a certain predicate assumes the value ‘true’.

There is only a very limited number of operations that can be performed on them.

10.1.3 Compatibility

The binary operators for sequences do not allow mixing of the formal and enumerated sequence types (so one cannot take the concatenation of an enumerated sequence and a formal sequence, for example); but it is easy to convert an enumerated sequence into a formal sequence – see the section on binary operators below.

By the limitation on their construction formal sequences can only contain elements from one structure in MAGMA. The elements of enumerated sequences are also restricted, in the sense that either some common structure must be specified upon creation, or MAGMA must be able to find such universe automatically. The rules for compatibility of elements and the way MAGMA deals with these parents is the same for sequences and sets, and is outlined in Chapter 8.

10.2 Creating Sequences

Square brackets are used for the definition of enumerated sequences; formal sequences are delimited by the composite brackets `[!` and `!]`.

Certain expressions appearing below (possibly with subscripts) have the standard interpretation:

U the universe: any MAGMA structure;

E the range set for enumerated sequences: any enumerated structure (it must be possible to loop over its elements – see the Introduction to this Part);

F the range set for formal sequences: any structure for which membership testing using `in` is defined – see the Introduction to this Part);

x a free variable which successively takes the elements of E (or F in the formal case) as its values;

P a Boolean expression that usually involves the variable(s) x, x_1, \dots, x_k ;

e an expression that also usually involves the variable(s) x, x_1, \dots, x_k .

10.2.1 The Formal Sequence Constructor

The formal sequence constructor has the following fixed format (the expressions appearing in the construct are defined above):

`[! x in F | P(x) !]`

Create the formal sequence consisting of the subsequence of elements x of F for which $P(x)$ is true. If $P(x)$ is true for every element of F , the sequence constructor may be abbreviated to `[! x in F !]`

10.2.2 The Enumerated Sequence Constructor

Sequences can be constructed by expressions enclosed in square brackets, provided that the values of all expressions can be automatically coerced into some common structure, as outlined in the Introduction. All general constructors have the universe U optionally up front, which allows the user to specify into which structure all terms of the sequences should be coerced.

[]

The null sequence (empty, and no universe specified).

[U |]

The empty sequence with universe U .

[e_1, e_2, \dots, e_n]

Given a list of expressions e_1, \dots, e_n , defining elements a_1, a_2, \dots, a_n all belonging to (or automatically coercible into) a single algebraic structure U , create the sequence $Q = [a_1, a_2, \dots, a_n]$ of elements of U .

As for multisets, one may use the expression x^n to specify the object x with multiplicity n : this is simply interpreted to mean x repeated n times (i.e., no internal compaction of the repetition is done).

[U | e_1, e_2, \dots, e_m]

Given a list of expressions e_1, \dots, e_m , which define elements a_1, a_2, \dots, a_n that are all coercible into U , create the sequence $Q = [a_1, a_2, \dots, a_n]$ of elements of U .

[$e(x) : x \text{ in } E \mid P(x)$]

Form the sequence of elements $e(x)$, all belonging to some common structure, for those $x \in E$ with the property that the predicate $P(x)$ is true. The expressions appearing in this construct have the interpretation given at the beginning of this section.

If $P(x)$ is true for every element of E , the sequence constructor may be abbreviated to [$e(x) : x \text{ in } E$] .

[U | $e(x) : x \text{ in } E \mid P(x)$]

Form the sequence of elements of U consisting of the values $e(x)$ for those $x \in E$ for which the predicate $P(x)$ is true (an error results if not all $e(x)$ are coercible into U). The expressions appearing in this construct have the same interpretation as above.

[$e(x_1, \dots, x_k) : x_1 \text{ in } E_1, \dots, x_k \text{ in } E_k \mid P(x_1, \dots, x_k)$]

The sequence consisting of those elements $e(x_1, \dots, x_k)$, in some common structure, for which $x_1, \dots, x_k \text{ in } E_1, \dots, E_k$ have the property that $P(x_1, \dots, x_k)$ is true.

The expressions appearing in this construct have the interpretation given at the beginning of this section.

Note that if two successive ranges E_i and E_{i+1} are identical, then the specification of the ranges for x_i and x_{i+1} may be abbreviated to $\mathbf{x}_i, \mathbf{x}_{i+1} \text{ in } E_i$.

Also, if $P(x_1, \dots, x_k)$ is always true, it may be omitted.

$[U \mid e(\mathbf{x}_1, \dots, \mathbf{x}_k) : \mathbf{x}_1 \text{ in } E_1, \dots, \mathbf{x}_k \text{ in } E_k \mid P(\mathbf{x}_1, \dots, \mathbf{x}_k)]$

As in the previous entry, the sequence consisting of those elements $e(x_1, \dots, x_k)$ for which $P(x_1, \dots, x_k)$ is true is formed, as a sequence of elements of U (an error occurs if not all $e(x_1, \dots, x_k)$ are coercible into U).

10.2.3 The Arithmetic Progression Constructors

Since enumerated sequences of integers arise so often, there are a few special constructors to create and handle them efficiently in case the entries are in arithmetic progression. The universe must be the ring of integers. Some effort is made to preserve the special way of storing arithmetic progressions under sequence operations.

$[i..j]$

$[U \mid i..j]$

The enumerated sequence of integers whose elements form the arithmetic progression $i, i+1, i+2, \dots, j$, where i and j are (expressions defining) arbitrary integers. If j is less than i then the empty sequence of integers will be created.

The universe U , if it is specified, has to be the ring of integers; any other universe will lead to an error.

$[i \dots j \text{ by } k]$

$[U \mid i \dots j \text{ by } k]$

The enumerated sequence consisting of the integers forming the arithmetic progression $i, i+k, i+2*k, \dots, j$, where i, j and k are (expressions defining) arbitrary integers (but $k \neq 0$).

If k is positive then the last element in the progression will be the greatest integer of the form $i + n * k$ that is less than or equal to j ; if j is less than i , the empty sequence of integers will be constructed.

If k is negative then the last element in the progression will be the least integer of the form $i + n * k$ that is greater than or equal to j ; if j is greater than i , the empty sequence of integers will be constructed.

The universe U , if it is specified, has to be the ring of integers; any other universe will lead to an error.

Example H10E1

As in the case of sets, it is possible to use the arithmetic progression constructors to save some typing in the creation of sequences of elements of rings other than the ring of integers, but the result will not be treated especially efficiently.

```
> s := [ IntegerRing(200) | x : x in [ 25..125 ] ];
```

10.2.4 Literal Sequences

A literal sequence is an enumerated sequence all of whose terms are from the same structure and all of these are ‘typed in’ literally. The sole purpose of literal sequences is to load certain enumerated sequences very fast and very space-efficiently; this is only useful when reading in very large sequences (all of whose elements must have been specified literally, that is, not as some expression other than a literal), but then it may save a lot of time. The result will be an enumerated sequence, that is, not distinguished in any way from other such sequences.

At present, only literal sequences of integers are supported.

$\backslash [m_1, \dots, m_n]$

Given a succession of literal integers m_1, \dots, m_n , build the enumerated sequence $[m_1, \dots, m_n]$, in a time and space efficient way.

10.3 Power Sequences

The **PowerSequence** constructor returns a structure comprising the enumerated sequences of a given structure R ; it is mainly useful as a parent for other set and sequence constructors. The only operations that are allowed on power sequences are printing, testing element membership, and coercion into the power sequence (see the examples below).

PowerSequence(R)

The structure comprising all enumerated sequences of elements of structure R . If R itself is a sequence (or set) then the power structure of its universe is returned.

S in P

Returns **true** if enumerated sequence S is in the power sequence P , that is, if all elements of the sequence S are contained in or coercible into R , where P is the power sequence of R ; **false** otherwise.

P ! S

Return a sequence with universe R consisting of the entries of the enumerated sequence S , where P is the power sequence of R . An error results if not all elements of S can be coerced into R .

Example H10E2

```
> S := [ 1 .. 10 ];
> P := PowerSequence(S);
> P;
Set of sequences over [ 1 .. 10 ]
> F := [ 6/3, 12/4 ];
> F in P;
true
> G := P ! F;
```

```
> Parent(F);
Set of sequences over Rational Field
> Parent(G);
Set of sequences over [ 1 .. 10 ]
```

10.4 Operators on Sequences

This section lists functions for obtaining information about existing sequences, for modifying sequences and for creating sequences from others. Most of these operators only apply to enumerated sequences.

10.4.1 Access Functions

#S

Returns the length of the enumerated sequence S , which is the index of the last term of S whose value is defined. The length of the empty sequence is zero.

Parent(S)

Returns the parent structure for a sequence S , that is, the structure consisting of all (enumerated) sequences over the universe of S .

Universe(S)

Returns the ‘universe’ of the sequence S , that is, the common structure to which all elements of the sequence belong. This universe may itself be a set or sequence. An error is signalled when S is the null sequence.

$S[i]$

The i -th term s_i of the sequence S . If $i \leq 0$, or $i > \#S + 1$, or $S[i]$ is not defined, then an error results. Here i is allowed to be a multi-index (see Introduction for the interpretation). This can be used as the left hand side of an assignment: $S[i] := x$ redefines the i -th term of the sequence S to be x . If $i \leq 0$, then an error results. If $i > n$, then the sequence $[s_1, \dots, s_n, s_{n+1}, \dots, s_{i-1}, x]$ replaces S , where s_{n+1}, \dots, s_{i-1} are all undefined. Here i is allowed to be a multi-index.

An error occurs if x cannot be coerced into the universe of S .

10.4.2 Selection Operators on Enumerated Sequences

Here, S denotes an enumerated sequence $[s_1, \dots, s_n]$. Further, i and j are integers or multi-indices (see Introduction).

S[I]

The sequence $[s_{i_1}, \dots, s_{i_r}]$ consisting of terms selected from the sequence S , according to the terms of the integer sequence I . If any term of I lies outside the range 1 to $\#S$, then an error results. If I is the empty sequence, then the empty set with universe the same as that of S is returned.

The effect of $T := S[I]$ differs from that of $T := [S[i] : i \text{ in } I]$: if in the first case an undefined entry occurs for $i \in I$ between 1 and $\#S$ it will be copied over; in the second such undefined entries will lead to an error.

Minimum(S)

Min(S)

Given a non-empty, complete enumerated sequence S such that **lt** and **eq** are defined on the universe of S , this function returns two values: a minimal element s in S , as well as the first position i such that $s = S[i]$.

Maximum(S)

Max(S)

Given a non-empty, complete enumerated sequence S such that **gt** and **eq** are defined on the universe of S , this function returns two values: a maximal element s in S , as well as the first position i such that $s = S[i]$.

Index(S, x)

Index(S, x, f)

Position(S, x)

Position(S, x, f)

Returns either the position of the first occurrence of x in the sequence S , or zero if S does not contain x . The second variants of each function starts the search at position f . This can save time in second (and subsequent) searches for the same entry further on. If no occurrence of x in S from position f onwards is found, then zero is returned.

Representative(R)

Rep(R)

An (arbitrary) element chosen from the enumerated sequence R

Random(<i>R</i>)

A random element chosen from the enumerated sequence R . Every element has an equal probability of being chosen. Successive invocations of the function will result in independently chosen elements being returned as the value of the function. If R is empty an error occurs.

Explode(<i>R</i>)

Given an enumerated sequence R of length r this function returns the r entries of the sequence (in order).

Eltseq(<i>R</i>)

The enumerated sequence R itself. This function is just included for completeness.

10.4.3 Modifying Enumerated Sequences

The operations given here are available as both procedures and functions. In the procedure version, the given sequence is destructively modified ‘in place’. This is very efficient, since it is not necessary to make a copy of the sequence. In the function version, the given sequence is not changed, but a modified version of it is returned. This is more suitable if the old sequence is still required. Some of the functions also return useful but non-obvious values.

Here, S denotes an enumerated sequence, and x an element of some structure V . The modifications involving S and x will only be successful if x can be coerced into the universe of S ; an error occurs if this fails. (See the Introduction to this Part).

Append($\sim S$, x)

Append(S , x)

Create an enumerated sequence by adding the object x to the end of S , i.e., the enumerated sequence $[s_1, \dots, s_n, x]$.

There are two versions of this: a procedure, where S is replaced by the appended sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

Exclude($\sim S$, x)

Exclude(S , x)

Create an enumerated sequence obtained by removing the first occurrence of the object x from S , i.e., the sequence $[s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_n]$, where s_i is the first term of S that is equal to x . If x is not in S then this is just S .

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

<code>Include($\sim S$, x)</code>

<code>Include(S, x)</code>
--

Create a sequence by adding the object x to the end of S , provided that no term of S is equal to x . Thus, if x does not occur in S , the enumerated sequence $[s_1, \dots, s_n, x]$ is created.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

<code>Insert($\sim S$, i, x)</code>
--

<code>Insert(S, i, x)</code>

Create the sequence formed by inserting the object x at position i in S and moving the terms $S[i], \dots, S[n]$ down one place, i.e., the enumerated sequence $[s_1, \dots, s_{i-1}, x, s_i, \dots, s_n]$. Note that i may be bigger than the length n of S , in which case the new length of S will be i , and the entries $S[n+1], \dots, S[i-1]$ will be undefined.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

<code>Insert($\sim S$, k, m, T)</code>
--

<code>Insert(S, k, m, T)</code>

Create the sequence $[s_1, \dots, s_{k-1}, t_1, \dots, t_l, s_{m+1}, \dots, s_n]$. If $k \leq 0$ or $k > m+1$, then an error results. If $k = m+1$ then the terms of T will be inserted into S immediately before the term s_k . If $k > n$, then the sequence $[s_1, \dots, s_n, s_{n+1}, \dots, s_{k-1}, t_1, \dots, t_l]$ is created, where s_{n+1}, \dots, s_{k-1} are all undefined. In the case where T is the empty sequence, terms s_k, \dots, s_m are deleted from S .

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

Prune($\sim S$)

Prune(S)

Create the enumerated sequence formed by removing the last term of the sequence S , i.e., the sequence $[s_1, \dots, s_{n-1}]$. An error occurs if S is empty.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

Remove($\sim S$, i)

Remove(S , i)

Create the enumerated sequence formed by removing the i -th term from S , i.e., the sequence $[s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_n]$. An error occurs if $i < 1$ or $i > n$.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

Reverse($\sim S$)

Reverse(S)

Create the enumerated sequence formed by reversing the order of the terms in the complete enumerated sequence S , i.e., the sequence $[s_n, \dots, s_1]$.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

Rotate($\sim S$, p)

Rotate(S , p)

Given a complete sequence S and an integer p , create the enumerated sequence formed by cyclically rotating the terms of the sequence p terms: if p is positive, rotation will be to the right; if p is negative, S is cyclically rotated $-p$ terms to the left; if p is zero nothing happens.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

Sort($\sim S$)

Sort(S)

Given a complete enumerated sequence S whose terms belong to a structure on which **lt** and **eq** are defined, create the enumerated sequence formed by (quick-)sorting the terms of S into increasing order.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

Sort($\sim S$, C)

Sort($\sim S$, C , $\sim p$)

Sort(S , C)

Given a complete enumerated sequence S and a comparison function C which compares elements of S , create the enumerated sequence formed by sorting the terms of S into increasing order with respect to C . The comparison function C must take two arguments and return an integer less than, equal to, or greater than 0 according to whether the first argument is less than, equal to, or greater than the second argument (e.g.: **func**< x , y | $x - y$ >).

There are three versions of this: a procedure, where S is replaced by the new sequence, a procedure, where S is replaced by the new sequence and the corresponding permutation p is set, and a function, which returns the new sequence and the corresponding permutation. The procedural version takes a reference $\sim S$ to S as an argument. Note that the procedural version is much more efficient since the sequence S will not be copied.

ParallelSort($\sim S$, $\sim T$)

Given a complete enumerated sequence S , sorts it in place and simultaneously sorts T in the same order. That is, whenever the sorting process would swap the two elements $S[i]$ and $S[j]$ then the two elements $T[i]$ and $T[j]$ are also swapped.

Undefine($\sim S$, i)

Undefine(S , i)

Create the sequence which is the same as the enumerated sequence S but with the i -th term of S undefined; i may be bigger than $\#S$, but $i \leq 0$ produces an error.

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

ChangeUniverse(S, V)

ChangeUniverse(S, V)

Given a sequence S with universe U and a structure V which contains U , construct a sequence which consists of the elements of S coerced into V .

There are two versions of this: a procedure, where S is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the sequence S will not be copied.

CanChangeUniverse(S, V)

Given a sequence S with universe U and a structure V which contains U , attempt to construct a sequence T which consists of the elements of S coerced into V ; if successful, return **true** and T , otherwise return **false**.

Example H10E3

We present three ways to obtain the Farey series F_n of degree n .

The Farey series F_n of degree n consists of all rational numbers with denominator less than or equal to n , in order of magnitude. Since we will need numerator and denominator often, we first abbreviate those functions.

```
> D := Denominator;
> N := Numerator;
```

The first method calculates the entries in order. It uses the fact that for any three consecutive Farey fractions $\frac{p}{q}, \frac{p'}{q'}, \frac{p''}{q''}$ of degree n :

$$p'' = \lfloor \frac{q+n}{q'} \rfloor p' - p, \quad q'' = \lfloor \frac{q+n}{q'} \rfloor q' - q.$$

```
> farey := function(n)
>   f := [ RationalField() | 0, 1/n ];
>   p := 0;
>   q := 1;
>   while p/q lt 1 do
>     p := ( D(f[#f-1]) + n) div D(f[#f]) * N(f[#f]) - N(f[#f-1]);
>     q := ( D(f[#f-1]) + n) div D(f[#f]) * D(f[#f]) - D(f[#f-1]);
>     Append(~f, p/q);
>   end while;
>   return f;
> end function;
```

The second method calculates the Farey series recursively. It uses the property that F_n may be obtained from F_{n-1} by inserting a new fraction (namely $\frac{p+p'}{q+q'}$) between any two consecutive rationals $\frac{p}{q}$ and $\frac{p'}{q'}$ in F_{n-1} for which $q+q'$ equals n .

```
> function farey(n)
```



```

>   if n eq 1 then
>     return [RationalField() | 0, 1 ];
>   else
>     f := farey(n-1);
>     i := 0;
>     while i lt #f-1 do
>       i += 1;
>       if D(f[i]) + D(f[i+1]) eq n then
>         Insert( ~f, i+1, (N(f[i]) + N(f[i+1]))/(D(f[i]) + D(f[i+1])));
>       end if;
>     end while;
>     return f;
>   end if;
> end function;

```

The third method is very straightforward, and uses `Sort` and `Setseq` (defined above).

```

> farey := func< n |
>   Sort(Setseq({ a/b : a in { 0..n}, b in { 1..n} | a le b }));
> farey(6);
[ 0, 1/6, 1/5, 1/4, 1/3, 2/5, 1/2, 3/5, 2/3, 3/4, 4/5, 5/6, 1 ]

```

10.4.4 Creating New Enumerated Sequences from Existing Ones

S cat T

The enumerated sequence formed by concatenating the terms of S with the terms of T , i.e. the sequence $[s_1, \dots, s_n, t_1, \dots, t_m]$.

If the universes of S and T are different, an attempt to find a common overstructure is made; if this fails an error results (see the Introduction).

S cat:= T

Mutation assignment: change S to be the concatenation of S and T . Functionally equivalent to `S := S cat T`.

If the universes of S and T are different, an attempt to find a common overstructure is made; if this fails an error results (see the Introduction).

Partition(S, p)

Given a complete non-empty sequence S as well as an integer p that divides the length n of S , construct the sequence whose terms are the sequences formed by taking p terms of S at a time.

Partition(S, P)

Given a complete non-empty sequence S as well as a complete sequence of positive integers P , such that the sum of the entries of P equals the length of S , construct the sequence whose terms are the sequences formed by taking $P[i]$ terms of S , for $i = 1, \dots, \#P$.

Setseq(S)**SetToSequence(S)**

Given a set S , construct a sequence whose terms are the elements of S taken in some arbitrary order.

Seqset(S)**SequenceToSet(S)**

Given a sequence S , create a set whose elements are the distinct terms of S .

Example H10E4

The following example illustrates several of the access, creation and modification operations on sequences.

Given a rational number r , this function returns a sequence of different integers d_i such that $r = \sum 1/d_i$ [Bee93].

```
> egyptian := function(r)
>   n := Numerator(r);
>   d := Denominator(r);
>   s := [d : i in [1..n]];
>   t := { d};
>   i := 2;
>   while i le #s do
>     c := s[i];
>     if c in t then
>       Remove(~s, i);
>       s cat:= [c+1, c*(c+1)];
>     else
>       t join:= { c};
>       i := i+1;
>     end if;
>   end while;
>   return s;
> end function;
```

Note that the result may be rather larger than necessary:

```
> e := egyptian(11/13);
> // Check the result!
> &+[1/d : d in e];
11/13
```

```
> #e;
2047
> #IntegerToString(Maximum(e));
1158
```

while instead of this sequence of 2047 integers, the biggest of the entries having 1158 decimal digits, the following equation also holds:

$$\frac{1}{3} + \frac{1}{4} + \frac{1}{6} + \frac{1}{12} + \frac{1}{78} = \frac{11}{13}.$$

10.4.4.1 Operations on Sequences of Booleans

The following operation work pointwise on sequences of booleans of equal length.

And(S, T)

And(~S, T)

The sequence whose i th entry is the logical and of the i th entries of S and T . The result is placed in S if it is given by reference (\sim).

Or(S, T)

Or(~S, T)

The sequence whose i th entry is the logical or of the i th entries of S and T . The result is placed in S if it is given by reference.

Xor(S, T)

Xor(~S, T)

The sequence whose i th entry is the logical xor of the i th entries of S and T . The result is placed in S if it is given by reference.

Not(S)

Not(~S)

The sequence whose i th entry is the logical not of the i th entry of S . The result is placed in S if it is given by reference.

10.5 Predicates on Sequences

Boolean valued operators and functions on enumerated sequences exist to test whether entries are defined (see previous section), to test for membership and containment, and to compare sequences with respect to an ordering on its entries. On formal sequences, only element membership can be tested.

IsComplete(S)

Boolean valued function, returning **true** if and only if each of the terms $S[i]$ for $1 \leq i \leq \#S$ is defined, for an enumerated sequence S .

IsDefined(S, i)

Given an enumerated sequence S and an index i , this returns **true** if and only if $S[i]$ is defined. (Hence the result is **false** if $i > \#S$, but an error results if $i < 1$.) Note that the index i is allowed to be a multi-index; if $i = [i_1, \dots, i_r]$ is a multi-index and $i_j > \#S[i_1, \dots, i_{j-1}]$ the function returns false, but if S is s levels deep and $r > s$ while $i_j \leq \#S[i_1, \dots, i_{j-1}]$ for $1 \leq j \leq s$, then an error occurs.

IsEmpty(S)

Boolean valued function, returning **true** if and only if the enumerated sequence S is empty.

IsNull(S)

Boolean valued function, returning **true** if and only if the enumerated sequence S is empty and its universe is undefined, **false** otherwise.

10.5.1 Membership Testing

Here, S and T denote sequences. The element x is always assumed to be compatible with S .

x in S

Returns **true** if the object x occurs as a term of the enumerated or formal sequence S , **false** otherwise. If x is not in the universe of S , coercion is attempted. If that fails, an error results.

x notin S

Returns **true** if the object x does not occur as a term of the enumerated or formal sequence S , **false** otherwise. If x is not in the universe of S , coercion is attempted. If that fails, an error results.

IsSubsequence(<i>S</i> , <i>T</i>)

IsSubsequence(<i>S</i> , <i>T</i> : <i>Kind</i> := <i>option</i>)

Kind

MONSTGELE

Default : “*Consecutive*”

Returns **true** if the enumerated sequence *S* appears as a subsequence of consecutive elements of the enumerated sequence *T*, **false** otherwise.

By changing the default value “*Consecutive*” of the parameter *Kind* to “*Sequential*” or to “*Setwise*”, this returns **true** if and only if the elements of *S* appear in order (but not necessarily consecutively) in *T*, or if and only if all elements of *S* appear as elements of *T*; so in the latter case the test is merely whether the set of elements of *S* is contained in the set of elements of *T*.

If the universes of *S* and *T* are not the same, coercion is attempted.

<i>S</i> eq <i>T</i>

Returns **true** if the enumerated sequences *S* and *T* are equal, **false** otherwise. If the universes of *S* and *T* are not the same, coercion is attempted.

<i>S</i> ne <i>T</i>

Returns **true** if the enumerated sequences *S* and *T* are not equal, **false** otherwise. If the universes of *S* and *T* are not the same, coercion is attempted.

10.5.2 Testing Order Relations

Here, *S* and *T* denote complete enumerated sequences with universe *U* and *V* respectively, such that a common overstructure *W* for *U* and *V* can be found (as outlined in the Introduction), and such that on *W* an ordering on the elements is defined allowing the MAGMA operators **eq** (=), **le** (≤), **lt** (<), **gt** (>), and **ge** (≥) to be invoked on its elements.

With these comparison operators the *lexicographical* ordering is used to order complete enumerated sequences. Sequences *S* and *T* are equal (**S eq T**) if and only if they have the same length and all terms are the same. A sequence *S* precedes *T* (**S lt T**) in the ordering imposed by that of the terms if at the first index *i* where *S* and *T* differ then *S*[*i*] < *T*[*i*]. If the length of *T* exceeds that of *S* and *S* and *T* agree in all places where *S* until after the length of *S*, then **S lt T** is true also. In all other cases where *S* ≠ *T* one has **S gt T**.

<i>S</i> lt <i>T</i>

Returns **true** if the sequence *S* precedes the sequence *T* under the ordering induced from *S*, **false** otherwise. Thus, **true** is returned if and only if either *S*[*k*] < *T*[*k*] and *S*[*i*] = *T*[*i*] (for 1 ≤ *i* < *k*) for some *k*, or *S*[*i*] = *T*[*i*] for 1 ≤ *i* ≤ #*S* and #*S* < #*T*.

<i>S</i> le <i>T</i>

Returns **true** if the sequence *S* either precedes the sequence *T*, under the ordering induced from *S*, or is equal to *T*, **false** otherwise. Thus, **true** is returned if and only if either *S*[*k*] < *T*[*k*] and *S*[*i*] = *T*[*i*] (for 1 ≤ *i* < *k*) for some *k*, or *S*[*i*] = *T*[*i*] for 1 ≤ *i* ≤ #*S* and #*S* ≤ #*T*.

S ge T

Returns **true** if the sequence S either comes after the sequence T , under the ordering induced from S , or is equal to T , **false** otherwise. Thus, **true** is returned if and only if either $S[k] > T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k , or $S[i] = T[i]$ for $1 \leq i \leq \#T$ and $\#S \geq \#T$.

S gt T

Returns **true** if the sequence S comes after the sequence T under the ordering induced from S , **false** otherwise. Thus, **true** is returned if and only if either $S[k] > T[k]$ and $S[i] = T[i]$ (for $1 \leq i < k$) for some k , or $S[i] = T[i]$ for $1 \leq i \leq \#T$ and $\#S > \#T$.

10.6 Recursion, Reduction, and Iteration

10.6.1 Recursion

It is often very useful to be able to refer to a sequence currently under construction, for example to define the sequence recursively. For this purpose the **Self** operator is available.

Self(n)

Self()

This operator enables the user to refer to an already defined previous entry $s[n]$ of the enumerated sequence s inside the sequence constructor, or the sequence s itself.

Example H10E5

The example below shows how the sequence of the first 100 Fibonacci numbers can be created recursively, using **Self**. Next it is shown how to use reduction on these 100 integers.

```
> s := [ i gt 2 select Self(i-2)+Self(i-1) else 1 : i in [1..100] ];
> &+s;
927372692193078999175
```

10.6.2 Reduction

Instead of using a loop to apply the same binary associative operator to all elements of a complete enumerated sequence, it is possible to use the *reduction operator* `&`.

`&◦ S`

Given a complete enumerated sequence $S = [a_1, a_2, \dots, a_n]$ of elements belonging to an algebraic structure U , and an (associative) operator $\circ : U \times U \rightarrow U$, form the element $a_1 \circ a_2 \circ a_3 \circ \dots \circ a_n$.

Currently, the following operators may be used to reduce sequences: `+`, `*`, `and`, `or`, `join`, `meet`, `cat`. An error will occur if the operator is not defined on U .

If S contains a single element a , then the value returned is a . If S is the null sequence (empty and no universe specified), then reduction over S leads to an error; if S is empty with universe U in which the operation is defined, then the result (or error) depends on the operation and upon U . The following table defines the return value:

	<i>empty</i>	<i>null</i>
<code>&+</code>	$U \neq 0$	error
<code>&*</code>	$U \neq 1$	error
<code>&and</code>	true	true
<code>&or</code>	false	false
<code>&join</code>	<i>empty</i>	<i>null</i>
<code>&meet</code>	error	error
<code>&cat</code>	<i>empty</i>	<i>null</i>

10.7 Iteration

Enumerated sequences allow iteration over their elements. In particular, they can be used as the range set in the sequence and set constructors, and as domains in `for` loops.

When multiple range sequences are used, it is important to know in which order the range are iterated over; the rule is that the repeated iteration takes place as nested loops where the first range forms the innermost loop, etc. See the examples below.

`for x in S do statements; end for;`

An enumerated sequence S may be the range for the `for`-statement. The iteration only enumerates the defined terms of the sequence.

Example H10E6

The first example shows how repeated iteration inside a sequence constructor corresponds to nesting of loops.

```
> [<number, letter> : number in [1..5], letter in ["a", "b", "c"]];
```

```
[ <1, a>, <2, a>, <3, a>, <4, a>, <5, a>, <1, b>, <2, b>, <3, b>, <4, b>, <5,
b>, <1, c>, <2, c>, <3, c>, <4, c>, <5, c> ]
> r := [];
> for letter in ["a", "b", "c"] do
>   for number in [1..5] do
>     Append(~r, <number, letter>);
>   end for;
> end for;
> r;
[ <1, a>, <2, a>, <3, a>, <4, a>, <5, a>, <1, b>, <2, b>, <3, b>, <4, b>, <5,
b>, <1, c>, <2, c>, <3, c>, <4, c>, <5, c> ]
```

This explains why the first construction below leads to an error, whereas the second leads to the desired sequence.

```
> // The following produces an error:
> [ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 16 ];
```

User error: Identifier 'x' has not been declared

```
> [ <x, y> : x in [0..y], y in [0..5] | x^2+y^2 lt 16 ];
[ <0, 0>, <0, 1>, <1, 1>, <0, 2>, <1, 2>, <2, 2>, <0, 3>, <1, 3>, <2, 3> ]
```

Note the following! In the last line below there are two different things with the name x . One is the (inner) loop variable, the other just an identifier with value 1000 that is used in the bound for the other (outer) loop variable y : the limited scope of the inner loop variable x makes it invisible to y , whence the error in the first case.

```
> // The following produces an error:
> #[ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 100 ];
```

User error: Identifier 'x' has not been declared

```
> x := 1000;
> #[ <x, y> : x in [0..5], y in [0..x] | x^2+y^2 lt 100 ];
59
```

10.8 Bibliography

[Bee93] L. Beeckmans. The splitting algorithm for Egyptian fractions. *J. Number Th.*, 43:173–185, 1993.

11 TUPLES AND CARTESIAN PRODUCTS

11.1 Introduction	215	Append(T, x)	216
11.2 Cartesian Product Constructor and Functions	215	Append(~T, x)	217
car< >	215	Prune(T)	217
CartesianProduct(R, S)	215	Prune(~T)	217
CartesianProduct(L)	215	Flat(T)	217
CartesianPower(R, k)	215	11.4 Tuple Access Functions . . .	218
Flat(C)	215	Parent(T)	218
NumberOfComponents(C)	216	#	218
Component(C, i)	216	T[i]	218
C[i]	216	Explode(T)	218
#	216	TupleToList(T)	218
Rep(C)	216	Tuplist(T)	218
Random(C)	216	11.5 Equality	218
11.3 Creating and Modifying Tuples	216	eq	218
elt< >	216	ne	218
!	216	11.6 Other operations	219
< a ₁ , a ₂ , ..., a _k >	216	&*	219

Chapter 11

TUPLES AND CARTESIAN PRODUCTS

11.1 Introduction

A cartesian product may be constructed from a finite number of factors, each of which may be a set or algebraic structure. The term *tuple* will refer to an element of a cartesian product.

Note that the rules for tuples are quite different to those for sequences. Sequences are elements of a cartesian product of n copies of a fixed set (or algebraic structure) while tuples are elements of cartesian products where the factors may be different sets (structures). The semantics for tuples are quite different to those for sequences. In particular, the parent cartesian product of a tuple is fixed once and for all. This is in contrast to a sequence, which may grow and shrink during its life (thus implying a varying parent cartesian product).

11.2 Cartesian Product Constructor and Functions

The special constructor `car< ... >` is used for the creation of cartesian products of structures.

`car< R1, ..., Rk >`

Given a list of sets or algebraic structures R_1, \dots, R_k , construct the cartesian product set $R_1 \times \dots \times R_k$.

`CartesianProduct(R, S)`

Given structures R and S , construct the cartesian product set $R \times S$. This is the same as calling the `car` constructor with the two arguments R and S .

`CartesianProduct(L)`

Given a sequence or tuple L of structures, construct the cartesian product of the elements of L .

`CartesianPower(R, k)`

Given a structure R and an integer k , construct the cartesian power set R^k .

`Flat(C)`

Given a cartesian product C of structures which may themselves be cartesian products, return the cartesian product of the base structures, considered in depth-first order (see `Flat` for the element version).

NumberOfComponents(C)

Given a cartesian product C , return the number of components of C .

Component(C, i)

C[i]

The i -th component of C .

#C

Given a cartesian product C , return the cardinality of C .

Rep(C)

Given a cartesian product C , return a representative of C .

Random(C)

Given a cartesian product C , return a random element of C .

Example H11E1

We create the product of \mathbf{Q} and \mathbf{Z} .

```
> C := car< RationalField(), Integers() >;
> C;
Cartesian Product<Rational Field, Ring of Integers>
```

11.3 Creating and Modifying Tuples

elt< C | a₁, a₂, ..., a_k >

C ! < a₁, a₂, ..., a_k >

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a sequence of elements a_1, a_2, \dots, a_k , such that a_i belongs to the set R_i ($i = 1, \dots, k$), create the tuple $T = \langle a_1, a_2, \dots, a_k \rangle$ of C .

< a₁, a₂, ..., a_k >

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a list of elements a_1, a_2, \dots, a_k , such that a_i belongs to the set R_i , ($i = 1, \dots, k$), create the tuple $T = \langle a_1, a_2, \dots, a_k \rangle$ of C . Note that if C does not already exist, it will be created at the time this expression is evaluated.

Append(T, x)

Return the tuple formed by adding the object x to the end of the tuple T . Note that the result lies in a new cartesian product of course.

Append($\sim T$, x)

(Procedure.) Destructively add the object x to the end of the tuple T . Note that the new T lies in a new cartesian product of course.

Prune(T)

Return the tuple formed by removing the last term of the tuple T . The length of T must be greater than 1. Note that the result lies in a new cartesian product of course.

Prune($\sim T$)

(Procedure.) Destructively remove the last term of the tuple T . The length of T must be greater than 1. Note that the new T lies in a new cartesian product of course.

Flat(T)

Construct the flattened version of the tuple T . The flattening is done in the same way as Flat, namely depth-first.

Example H11E2

We build a set of pairs consisting of primes and their reciprocals.

```
> C := car< Integers(), RationalField() >;
> C ! < 26/13, 13/26 >;
<2, 1/2>
> S := { C | <p, 1/p> : p in [1..25] | IsPrime(p) };
> S;
{ <5, 1/5>, <7, 1/7>, <2, 1/2>, <19, 1/19>, <17, 1/17>, <23, 1/23>, <11, 1/11>,
<13, 1/13>, <3, 1/3> }
```

11.4 Tuple Access Functions

Parent(T)

The cartesian product to which the tuple T belongs.

T

Number of components of the tuple T .

$T[i]$

Return the i -th component of tuple T . Note that this indexing can also be used on the left hand side for modification of T .

Explode(T)

Given a tuple T of length n , this function returns the n entries of T (in order).

TupleToList(T)

Tuplist(T)

Given a tuple T return a list containing the entries of T .

Example H11E3

```
> f := < 11/2, 13/3, RootOfUnity(3, CyclotomicField(3)) >;
> f;
<11/2, 13/3, (zeta_3)>
> #f;
3
> Parent(f);
Cartesian Product<Rational Field, Rational Field, Cyclotomic field Q(zeta_3)>
> f[1]+f[2]+f[3];
(1/6) * (59 + 6*zeta_3)
> f[3] := 7;
> f;
<11/2, 13/3, 7>
```

11.5 Equality

$T \text{ eq } U$

Return **true** if and only if the tuples T and U are equal.

$T \text{ ne } U$

Return **true** if and only if the tuples T and U are distinct.

11.6 Other operations

&*T

For a tuple T where each component lies in a structure that supports multiplication and such there exists a common over structure, return the product of the entries.

12 LISTS

12.1 Introduction	223	SequenceToList(Q)	224
12.2 Construction of Lists	223	SeqList(Q)	224
[* *]	223	TupleToList(T)	224
[* e ₁ , e ₂ , ..., e _n *]	223	TupList(T)	224
12.3 Creation of New Lists	223	Reverse(L)	224
cat	223	12.4 Access Functions	224
cat:=	223	#	224
Append(S, x)	223	IsEmpty(S)	224
Append(~S, x)	223	S[i]	224
Insert(~S, i, x)	224	S[I]	225
Insert(S, i, x)	224	IsDefined(L, i)	225
Prune(S)	224	12.5 Assignment Operator	225
Prune(~S)	224	S[i] := x	225

Chapter 12

LISTS

12.1 Introduction

A *list* in MAGMA is an ordered finite collection of objects. Unlike sequences, lists are not required to consist of objects that have some common parent. Lists are not stored compactly and the operations provided for them are not extensive. They are mainly provided to enable the user to gather assorted objects temporarily together.

12.2 Construction of Lists

Lists can be constructed by expressions enclosed in special brackets $[* \text{ and } *]$.

$[* \ *]$

The empty list.

$[* \ e_1, \ e_2, \ \dots, \ e_n \ *]$

Given a list of expressions e_1, \dots, e_n , defining elements a_1, a_2, \dots, a_n , create the list containing a_1, a_2, \dots, a_n .

12.3 Creation of New Lists

Here, S denotes the list $[* \ s_1, \dots, s_n \ *]$, while T denotes the list $[* \ t_1, \dots, t_m \ *]$.

$S \text{ cat } T$

The list formed by concatenating the terms of the list S with the terms of the list T , i.e. the list $[* \ s_1, \dots, s_n, t_1, \dots, t_m \ *]$.

$S \text{ cat} := T$

(Procedure.) Destructively concatenate the terms of the list T to S ; i.e. so S becomes the list $[* \ s_1, \dots, s_n, t_1, \dots, t_m \ *]$.

$\text{Append}(S, \ x)$

The list formed by adding the object x to the end of the list S , i.e. the list $[* \ s_1, \dots, s_n, x \ *]$.

$\text{Append}(\sim S, \ x)$

(Procedure.) Destructively add the object x to the end of the list S ; i.e. so S becomes the list $[* \ s_1, \dots, s_n, x \ *]$.

Insert($\sim S$, i , x)

Insert(S , i , x)

Create the list formed by inserting the object x at position i in S and moving the terms $S[i], \dots, S[n]$ down one place, i.e., the list $[* s_1, \dots, s_{i-1}, x, s_i, \dots, s_n *]$. Note that i must not be bigger than $n + 1$ where n is the length of S .

There are two versions of this: a procedure, where S is replaced by the new list, and a function, which returns the new list. The procedural version takes a reference $\sim S$ to S as an argument.

Note that the procedural version is much more efficient since the list S will not be copied.

Prune(S)

The list formed by removing the last term of the list S , i.e. the list $[* s_1, \dots, s_{n-1} *]$.

Prune($\sim S$)

(Procedure.) Destructively remove the last term of the list S ; i.e. so S becomes the list $[* s_1, \dots, s_{n-1} *]$.

SequenceToList(Q)

Seqlist(Q)

Given a sequence Q , construct a list whose terms are the elements of Q taken in the same order.

TupleToList(T)

Tuplist(T)

Given a tuple T , construct a list whose terms are the elements of T taken in the same order.

Reverse(L)

Given a list L return the same list, but in reverse order.

12.4 Access Functions

S

The length of the list S .

IsEmpty(S)

Return whether S is empty (has zero length).

$S[i]$

Return the i -th term of the list S . If either $i \leq 0$ or $i > \#S + 1$, then an error results. Here i is allowed to be a multi-index (see Section 8.3.1 for the interpretation).

S[I]

Return the sublist of S given by the indices in the sequence I . Each index in I must be in the range $[1..l]$, where l is the length of S .

IsDefined(L, i)

Checks whether the i th item in L is defined or not, that is it returns **true** if i is at most the length of L and **false** otherwise.

12.5 Assignment Operator

S[i] := x

Redefine the i -th term of the list S to be x . If $i \leq 0$, then an error results. If $i = \#S + 1$, then x is appended to S . Otherwise, if $i > \#S + 1$, an error results. Here i is allowed to be a multi-index.

13 ASSOCIATIVE ARRAYS

13.1 Introduction	229	<code>A[x]</code>	229
13.2 Operations	229	<code>IsDefined(A, x)</code>	229
<code>AssociativeArray()</code>	229	<code>Remove(~ A, x)</code>	229
<code>AssociativeArray(I)</code>	229	<code>Universe(A)</code>	229
<code>A[x] := y</code>	229	<code>Keys(A)</code>	230

Chapter 13

ASSOCIATIVE ARRAYS

13.1 Introduction

An *associative array* in MAGMA is an array which may be indexed by arbitrary elements of an index structure I . The indexing may thus be by objects which are not integers. These objects are known as the *keys*. For each current key there is an associated value. The *values* associated with the keys need not lie in a fixed universe but may be of any type.

13.2 Operations

AssociativeArray()

Create the null associative array with no index universe. The first assignment to the array will determine its index universe.

AssociativeArray(I)

Create the empty associative array with index universe I .

A[x] := y

Set the value in A associated with index x to be y . If x is not coercible into the current index universe I of A , then an attempt is first made to lift the index universe of A to contain both I and x .

A[x]

Given an index x coercible into the index universe I of A , return the value associated with x . If x is not in the keys of A , then an error is raised.

IsDefined(A, x)

Given an index x coercible into the index universe I of A , return whether x is currently in the keys of A and if so, return also the value $A[x]$.

Remove(\sim A, x)

(Procedure.) Destructively remove the value indexed by x from the array A . If x is not present as an index, then nothing happens (i.e., an error is not raised).

Universe(A)

Given an associative array A , return the index universe I of A , in which the keys of A currently lie.

Keys(A)

Given an associative array A , return the current keys of A as a set. Warning: this constructs a new copy of the set of keys, so should only be called when that is needed. It is not meant to be used as a quick access function.

Example H13E1

This example shows simple use of associative arrays. First we create an array indexed by rationals.

```
> A := AssociativeArray();
> A[1/2] := 7;
> A[3/8] := "abc";
> A[3] := 3/8;
> A[1/2];
7
> IsDefined(A, 3);
true 3/8
> IsDefined(A, 4);
false
> IsDefined(A, 3/8);
true abc
> Keys(A);
{ 3/8, 1/2, 3 }
> for x in Keys(A) do x, A[x]; end for;
1/2 7
3/8 abc
3 3/8
> Remove(~A, 3/8);
> IsDefined(A, 3/8);
false
> Keys(A);
{ 1/2, 3 }
> Universe(A);
Rational Field
```

We repeat that an associative array can be indexed by elements of any structure. We now index an array by elements of the symmetric group S_3 .

```
> G := Sym(3);
> A := AssociativeArray(G);
> v := 1; for x in G do A[x] := v; v += 1; end for;
> A;
Associative Array with index universe GrpPerm: G, Degree 3, Order 2 * 3
> Keys(A);
{
  (1, 3, 2),
  (2, 3),
  (1, 3),
  (1, 2, 3),
```

```
      (1, 2),  
      Id(G)  
}  
> A[G!(1,3,2)];  
3
```

14 COPRODUCTS

14.1 Introduction	235	#	236
14.2 Creation Functions	235	Constituent(C, i)	236
14.2.1 Creation of Coproducts	235	Index(x)	236
cop< >	235	14.4 Retrieve	236
cop< >	235	Retrieve(x)	236
14.2.2 Creation of Coproduct Elements	235	14.5 Flattening	237
m(e)	235	Flat(C)	237
!	235	14.6 Universal Map	237
14.3 Accessing Functions	236	UniversalMap(C, S, [n ₁ , ..., n _m])	237
Injections(C)	236		

Chapter 14

COPRODUCTS

14.1 Introduction

Coproducts can be useful in various situations, as they may contain objects of entirely different types. Although the coproduct structure will serve as a single parent for such diverse objects, the proper parents of the elements are recorded internally and restored whenever the element is retrieved from the coproduct.

14.2 Creation Functions

There are two versions of the coproduct constructor. Ordinarily, coproducts will be constructed from a list of structures. These structures are called the **constituents** of the coproduct. A single sequence argument is allowed as well to be able to create coproducts of parameterized families of structures conveniently.

14.2.1 Creation of Coproducts

<code>cop< S₁, S₂, ..., S_k ></code>
<code>cop< [S₁, S₂, ..., S_k] ></code>

Given a list or a sequence of two or more structures S_1, S_2, \dots, S_k , this function creates and returns their coproduct C as well as a sequence of maps $[m_1, m_2, \dots, m_k]$ that provide the injections $m_i : S_i \rightarrow C$.

14.2.2 Creation of Coproduct Elements

Coproduct elements are usually created by the injections returned as the second return value from the `cop<>` constructor. The bang (!) operator may also be used but only if the type of the relevant constituent is unique for the particular coproduct.

<code>m(e)</code>

Given a coproduct injection map m and an element of one of the constituents of the coproduct C , create the coproduct element version of e .

<code>C ! e</code>

Given a coproduct C and an element e of one of the constituents of C such that the type of that constituent is unique within that coproduct, create the coproduct element version of e .

14.3 Accessing Functions

Injection(C)

Given a coproduct C , return the sequence of injection maps returned as the second argument from the `cop<>` constructor.

#C

Given a coproduct C , return the length (number of constituents) of C .

Constituent(C, i)

Given a coproduct C and an integer i between 1 and the length of C , return the i -th constituent of C .

Index(x)

Given an element x from a coproduct C , return the constituent number of C to which x belongs.

14.4 Retrieve

The function described here restores an element of a coproduct to its original state.

Retrieve(x)

Given an element x of some coproduct C , return the element as an element of the structure that formed its parent before it was mapped into C .

Example H14E1

We illustrate basic uses of the coproduct constructors and functions.

```
> C := cop<IntegerRing(), Strings()>;
> x := C ! 5;
> y := C ! "abc";
> x;
5
> y;
abc
> Parent(x);
Coproduct<Integer Ring, String structure>
> x eq 5;
true
> x eq y;
false
> Retrieve(x);
5
> Parent(Retrieve(x));
Integer Ring
```

14.5 Flattening

The function described here enables the ‘concatenation’ of coproducts into a single one.

Flat(C)

Given a coproduct C of structures which may themselves be coproducts, return the coproduct of the base structures, considered in depth-first order.

14.6 Universal Map

UniversalMap(C , S , [n_1 , ..., n_m])

Given maps n_1, \dots, n_m from structures S_1, \dots, S_m that compose the coproduct C , to some structure S , this function returns the universal map $C \rightarrow S$.

15 RECORDS

15.1 Introduction	241	<code>Format(r)</code>	243
15.2 The Record Format Constructor	241	<code>Names(F)</code>	243
<code>recformat< ></code>	241	<code>Names(r)</code>	243
15.3 Creating a Record	242	<code>r'fieldname</code>	243
<code>rec< ></code>	242	<code>r'fieldname:= e;</code>	243
15.4 Access and Modification		<code>delete</code>	243
Functions	243	<code>assigned</code>	243
		<code>r's</code>	243

Chapter 15

RECORDS

15.1 Introduction

In a *record* several objects can be collected. The objects in a record are stored in *record fields*, and are accessed by using *fieldnames*. Records are like tuples (and unlike sets or sequences) in that the objects need not all be of the same kind. Though records and tuples are somewhat similar, there are several differences too. The components of tuples are indexed by integers, and every component must be defined. The fields of records are indexed by fieldnames, and it is possible for some (or all) of the fields of a record not to be assigned; in fact, a field of a record may be assigned or deleted at any time. A record must be constructed according to a pre-defined *record format*, whereas a tuple may be constructed without first giving the Cartesian product that is its parent, since MAGMA can deduce the parent from the tuple.

In the definition of a record format, each field is given a fieldname. If the field is also given a parent magma or a category, then in any record created according to this format, that field must conform to this requirement. However, if the field is not given a parent magma or category, there is no restriction on the kinds of values stored in that field; different records in the format may contain disparate values in that field. By contrast, every component of a Cartesian product is a magma, and the components of all tuples in this product must be elements of the corresponding magma.

Because of the flexibility of records, with respect to whether a field is assigned and what kind of value is stored in it, Boolean operators are not available for comparing records.

15.2 The Record Format Constructor

The special constructor `recformat< ... >` is used for the creation of record formats. A record format must be created before records in that format are created.

<code>recformat< L ></code>

Construct the record format corresponding to the non-empty fieldname list L . Each term of L must be one of the following:

- (a) *fieldname* in which case there is no restriction on values that may be stored in this field of records having this format;
- (b) *fieldname:expression* where the expression evaluates to a magma which will be the parent of values stored in this field of records having this format; or
- (c) *fieldname:expression* where the expression evaluates to a category which will be the category of values stored in this field of records having this format;

where *fieldname* consists of characters that would form a valid identifier name. Note that it is not a string.

Example H15E1

We create a record format with these fields: `n`, an integer; `misc`, which has no restrictions; and `seq`, a sequence (with any universe possible).

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> RF;
recformat<n: IntegerRing(), misc, seq: SeqEnum>
> Names(RF);
[ n, misc, seq ]
```

15.3 Creating a Record

Before a record is created, its record format must be defined. A record may be created by assigning as few or as many of the record fields as desired.

`rec< F | L >`

Given a record format F , construct the record format corresponding to the field assignment list L . Each term of L must be of the form $fieldname := expression$ where $fieldname$ is in F and the value of the expression conforms (directly or by coercion) to any restriction on it. The list L may be empty, and there is no fixed order for the fieldnames.

Example H15E2

We build some records having the record format `RF`.

```
> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> r := rec< RF | >;
> r;
rec<RF | >
> s := rec< RF | misc := "adsifaj", n := 42, seq := [ GF(13) | 4, 8, 1 ]>;
> s;
rec<RF | n := 42, misc := adsifaj, seq := [ 4, 8, 1 ]>
> t := rec< RF | seq := [ 4.7, 1.9 ], n := 51/3 >;
> t;
rec<RF | n := 17, seq := [ 4.7, 1.9 ]>
> u := rec< RF | misc := RModule(PolynomialRing(Integers(7)), 4) >;
> u;
rec<RF | misc := RModule of dimension 4 with base ring Univariate Polynomial
Algebra over Integers(7)>
```

15.4 Access and Modification Functions

Fields of records may be inspected, assigned and deleted at any time.

Format(*r*)

The format of record *r*.

Names(*F*)

The fieldnames of the record format *F* returned as a sequence of strings.

Names(*r*)

The fieldnames of record *r* returned as a sequence of strings.

r*'*fieldname

Return the field of record *r* with this fieldname. The format of *r* must include this fieldname, and the field must be assigned in *r*.

***r*'*fieldname*:= *expression*;**

Reassign the given field of *r* to be the value of the expression. The format of *r* must include this fieldname, and the expression's value must satisfy (directly or by coercion) any restriction on the field.

delete *r*'*fieldname*

(Statement.) Delete the current value of the given field of record *r*.

assigned *r*'*fieldname*

Returns true if and only if the given field of record *r* currently contains a value.

r*'*s

Given an expression *s* that evaluates to a string, return the field of record *r* with the fieldname corresponding to this string. The format of *r* must include this fieldname, and the field must be assigned in *r*.

This syntax may be used anywhere that *r*'*fieldname* may be used, including in left hand side assignment, **assigned** and **delete**.

Example H15E3

```

> RF := recformat< n : Integers(), misc, seq : SeqEnum >;
> r := rec< RF | >;
> s := rec< RF | misc := "adsifaj", n := 42, seq := [ GF(13) | 4, 8, 1 ]>;
> t := rec< RF | seq := [ 4.7, 1.9 ], n := 51/3 >;
> u := rec< RF | misc := RModule(PolynomialRing(Integers(7)), 4) >;
> V4 := u'misc;
> assigned r'seq;
false
> r'seq := Append(t'seq, t'n); assigned r'seq;
true
> r;
rec<RF | seq := [ 4.7, 1.9, 17 ]>
> // The following produces an error:
> t''(s'misc);
>> t''(s'misc);
      ^
Runtime error in ': Field 'adsifaj' does not exist in this record
> delete u''("m" cat "isc"); u;
rec<RF | >

```

16 MAPPINGS

16.1 Introduction	247	*	251
16.1.1 The Map Constructors	247	Components(f)	251
16.1.2 The Graph of a Map	248	16.3.2 (Co)Domain and (Co)Kernel . . .	252
16.1.3 Rules for Maps	248	Domain(f)	252
16.1.4 Homomorphisms	248	Codomain(f)	252
16.1.5 Checking of Maps	248	Image(f)	252
		Kernel(f)	252
16.2 Creation Functions	249	16.3.3 Inverse	252
16.2.1 Creation of Maps	249	Inverse(m)	252
map< >	249	16.3.4 Function	252
map< >	249	Function(f)	252
map< >	249	16.4 Images and Preimages	253
16.2.2 Creation of Partial Maps	250	@	253
pmap< >	250	f(a)	253
pmap< >	250	@	253
pmap< >	250	f(S)	253
16.2.3 Creation of Homomorphisms	250	@	253
hom< >	250	f(C)	253
hom< >	250	@@	253
hom< >	250	@@	253
hom< >	251	@@	253
hom< >	251	HasPreimage(x, f)	253
16.2.4 Coercion Maps	251	16.5 Parents of Maps	254
Coercion(D, C)	251	Parent(m)	254
Bang(D, C)	251	Domain(P)	254
		Codomain(P)	254
16.3 Operations on Mappings	251	Maps(D, C)	254
16.3.1 Composition	251	Iso(D, C)	254
		Aut(S)	254

Chapter 16

MAPPINGS

16.1 Introduction

Mappings play a fundamental role in algebra and, indeed, throughout mathematics. Reflecting this importance, mappings are one of the fundamental datatypes in our language. The most general way to define a mapping $f : A \rightarrow B$ in a programming language is to write a *function* which, given any element of A , will return its image under f in B . While this approach to the definition of mappings is completely general, it is desirable to have mappings as an independent datatype. It is then possible to provide a very compact notation for specifying important classes of mappings such as homomorphisms. Further, a range of operations peculiar to the mapping type can be provided.

Mappings are created either through use of *mapping constructors* as described in this Chapter, or through use of certain standard functions that return mappings as either primary or secondary values.

All mappings are objects in the MAGMA category `Map`.

16.1.1 The Map Constructors

There are three main mapping constructors: the general map constructor `map< >`, the homomorphism constructor `hom< >`, and the partial map constructor `pmap< >`. The general form of all constructors is the same: inside the angle brackets there are two components separated by a pipe `|`. To the left the user specifies a *domain* A and a *codomain* B , separated by `->`; to the right of the pipe the user specifies how images are obtained for elements of the domain. The latter can be done in one of several ways: one specifies either the *graph* of the map, or a *rule* describing how images are to be formed, or for homomorphisms, one specifies generator images. We will describe each in the next subsections. The result is something like `map< A -> B | expression>`.

The domain and codomain of the map can be arbitrary magmas. When a full map (as opposed to a partial map) is constructed by use of a graph, the domain is necessarily finite.

The main difference between maps and partial maps is that a partial map need not be defined for every element of the domain. The main difference between these two types of map and homomorphisms is that the latter are supposed to provide *structure-preserving* maps between algebraic structures. On the one hand this makes it possible to allow the specification of images for homomorphisms in a different fashion: homomorphism can be given via *images* for *generators* of the domain. On the other hand homomorphisms are restricted to cases where domain and (image in the) codomain have a similar structure. The generator image form only makes sense for domains that are *finitely presented*. Homomorphisms are described in more detail below.

16.1.2 The Graph of a Map

Let A and B be structures. A *subgraph* of the cartesian product $C = A \times B$ is a subset G of C such that each element of A appears at most once among the first components of the pairs $\langle a, b \rangle$ of G . A subgraph having the additional property that every element of A appears as the first component of some pair $\langle a, b \rangle$ of G is called a *graph* of $A \times B$.

A mapping between A and B can be identified with a graph G of $A \times B$, a partial map can be identified with a subgraph. We now describe how a graph may be represented in the context of the map constructor. An element of the graph of $A \times B$ can be given either as a *tuple* $\langle a, b \rangle$, or as an *arrow pair* $a \rightarrow b$. The specification of a (sub)graph in a map constructor should then consist of either a (comma separated) list, a sequence, or a set of such tuples or arrow pairs (a mixture is permitted).

16.1.3 Rules for Maps

The specification of a rule in the map constructor involves a free variable and an expression, usually involving the free variable, separated by $:->$, for example $x :-> 3*x - 1$. The scope of the free variable is restricted to the map constructor (so the use of x does not interfere with values of x outside the constructor). A general expression is allowed in the rule, which may involve intrinsic or user functions, and even in-line definitions of such functions.

16.1.4 Homomorphisms

Probably the most useful form of the map-constructor is the version for homomorphisms. Most interesting mappings in algebra are homomorphisms, and if an algebraic structure A belongs to a family of algebraic structures which form a variety we have the fundamental result that a homomorphism is uniquely determined by the images of any generating set. This provides us with a particularly compact way of defining and representing homomorphisms. While the syntax of the homomorphism constructor is similar to that of the general mapping constructor, the semantics are sometimes different.

The kind of homomorphism built by the *hom*-constructor is determined entirely by the domain: thus, a *group* homomorphism results from applying *hom* to a domain A that is one of the types of group in MAGMA, a *ring* homomorphism results when A is a ring, etc. As a consequence, the requirements on the specification of homomorphisms are dependent on the category to which A belongs. Often, the codomain of a homomorphism is required to belong to the same variety. But even within a category the specification may depend on the type of structure; for details we refer the reader to the specific chapters.

A homomorphism can be specified using either a rule map or by generator images. In the latter case the processor will seek to express an element as a word in the generators of A when asked to compute its image. Thus A needs to be finitely presented.

16.1.5 Checking of Maps

It should be pointed out that checking the ‘correctness’ of mappings can be done to a limited extent only. If the mapping is given by means of a graph, MAGMA will check that no multiple images are specified, and that an image is given for every element of the

domain (unless a partial map is defined). If a rule is given, it cannot be checked that it is defined on all of the domain. Also, it is in general the responsibility of the user to ensure that the images provided for a `hom` constructor do indeed define a homomorphism.

16.2 Creation Functions

In this section we describe the creation of maps, partial maps, and homomorphisms via the various forms of the constructors, as well as maps that define coercions between algebraic structures.

16.2.1 Creation of Maps

Maps between structures A and B may be specified either by providing the full graph (as defined in the previous section) or by supplying an expression rule for finding images.

`map< A -> B | G >`

Given a finite structure A , a structure B and a graph G of $A \times B$, construct the mapping $f : A \rightarrow B$, as defined by G . The graph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter. Note that G must be a full graph, i.e., every element of A must occur exactly once as a first component.

`map< A -> B | x :-> e(x) >`

Given a set or structure A , a set or structure B , a variable x and an expression $e(x)$, usually involving x , construct the mapping $f : A \rightarrow B$, as defined by $e(x)$. It is the user's responsibility to ensure that a value is defined for every $x \in A$. The scope of the variable x is restricted to the map-constructor.

`map< A -> B | x :-> e(x), y :-> i(y) >`

Given a set or structure A , a set or structure B , a variable x , an expression $e(x)$, usually involving x , a variable y , and an expression $i(y)$, usually involving y , construct the mapping $f : A \rightarrow B$, as defined by $x \mapsto e(x)$, with corresponding inverse $f^{-1} : B \rightarrow A$, as defined by $y \mapsto i(y)$. It is the user's responsibility to ensure that a value $e(x)$ is defined for every $x \in A$, a value $i(y)$ is defined for every $y \in B$, and that $i(y)$ is the true inverse of $e(x)$. The scope of the variables x and y is restricted to the map-constructor.

16.2.2 Creation of Partial Maps

Partial mappings are quite different to both general mappings and homomorphisms, in that images need not be defined for every element of the domain.

`pmap< A -> B | G >`

Given a finite structure A of cardinality n , a structure B and a subgraph G of $A \times B$, construct the partial map $f : A \rightarrow B$, as defined by G . The subgraph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

`pmap< A -> B | x :-> e(x) >`

Given a set A , a set B , a variable x and an expression $e(x)$, construct the partial map $f : A \rightarrow B$, as defined by $e(x)$. This form of the map constructor is a special case of the previous one whereby the image of x can be defined using a single expression. Again the scope of x is restricted to the map-constructor.

`pmap< A -> B | x :-> e(x), y :-> i(y) >`

This constructor is the same as the map constructor above which allows the inverse map $i(y)$ to be specified, except that the result is marked to be a partial map.

16.2.3 Creation of Homomorphisms

The principal construction for homomorphisms consists of the generator image form, where the images of the generators of the domain are listed. Note that the kind of homomorphism and the kind and number of generators for which images are expected, depend entirely on the type of the domain. Moreover, some features of the created homomorphism, e.g. whether checking of the homomorphism is done during creation or whether computing preimages is possible, depend on the types of the domain and the codomain. We refer to the appropriate handbook chapters for further information.

`hom< A -> B | G >`

Given a finitely generated algebraic structure A and a structure B , as well as a graph G of $A \times B$, construct the homomorphism $f : A \rightarrow B$ defined by extending the map of the generators of A to all of A . The graph G may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

The detailed requirements on the specification are module-dependent, and can be found in the chapter describing the domain A .

`hom< A -> B | y1, ..., yn >`

`hom< A -> B | x1 -> y1, ..., xn -> yn >`

This is a module-dependent constructor for homomorphisms between structures A and B ; see the chapter describing the functions for A . In general after the bar the images for all generators of the structure A must be specified.

hom< A -> B x :-> e(x) >

Given a structure A , a structure B , a variable x and an expression $e(x)$, construct the homomorphism $f : A \rightarrow B$, as defined by $e(x)$. This form of the map constructor is a special case of the previous one whereby the image of x can be defined using a single expression. Again the scope of x is restricted to the map-constructor.

hom< A -> B x :-> e(x), y :-> i(y) >
--

This constructor is the same as the map constructor above which allows the inverse map $i(y)$ to be specified, except that the result is marked to be a homomorphism.

16.2.4 Coercion Maps

MAGMA has a sophisticated machinery for coercion of elements into structures other than the parent. Non-automatic coercion is usually performed via the `!` operator. To obtain the coercion map corresponding to `!` in a particular instance the `Coercion` function can be used.

Coercion(D, C)

Bang(D, C)

Given structures D and C such that elements from D can be coerced into C , return the map m that performs this coercion. Thus the domain of m will be D and the codomain will be C .

16.3 Operations on Mappings

16.3.1 Composition

Although compatible maps can be composed by repeated application, say $g(f(x))$, it is also possible to create a composite map.

f * g

Given a mapping $f : A \rightarrow B$, and a mapping $g : B \rightarrow C$, construct the composition h of the mappings f and g as the mapping $h = g \circ f : A \rightarrow C$.

Components(f)

Returns the maps which were composed to form f .

16.3.2 (Co)Domain and (Co)Kernel

The domain and codomain of any map can simply be accessed. Only for some intrinsic maps and for maps with certain domains and codomains, also the formation of image, kernel and cokernel is available.

Domain(f)

The domain of the mapping f .

Codomain(f)

The codomain of the mapping f .

Image(f)

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 supported only for some intrinsic maps and for maps with certain domains and codomains.

Kernel(f)

Given the homomorphism f with domain A and codomain B , return the kernel of f as a substructure of A . This function is currently supported only for some intrinsic maps and for maps with certain domains and codomains.

16.3.3 Inverse

Inverse(m)

The inverse map of the map m .

16.3.4 Function

For a map given by a rule, it is possible to get access to the rule as a user defined function.

Function(f)

The function underlying the mapping f . Only available if f has been defined by the user by means of a rule map (i.e., an expression for the image under f of an arbitrary element of the domain).

16.4 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.

$a @ f$

$f(a)$

Given a mapping f with domain A and codomain B , and an element a belonging to A , return the image of a under f as an element of B .

$S @ f$

$f(S)$

Given a mapping f with domain A and codomain B , and a finite enumerated set, indexed set, or sequence S of elements belonging to A , return the image of S under f as an enumerated set, indexed set, or sequence of elements of B .

$C @ f$

$f(C)$

Given a homomorphism f with domain A and codomain B , and a substructure C of A , return the image of C under f as a substructure of B .

$y @@ f$

Given a mapping f with domain A and codomain B , where f supports preimages, and an element y belonging to B , return the preimage of y under f as an element of A .

If the mapping f is a homomorphism, then a single element is returned as the preimage of y . In order to obtain the full preimage of y , it is necessary to form the coset $K * y @@ f$, where K is the kernel of f .

$R @@ f$

Given a mapping f with domain A and codomain B , where f supports preimages, and a finite enumerated set, indexed set, or sequence of elements R belonging to B , return the preimage of R under f as an enumerated set, indexed set, or sequence of elements of A .

$D @@ f$

Given a mapping f with domain A and codomain B , where f supports preimages and the kernel of f is known or can be computed, and a substructure D of B , return the preimage of D under f as a substructure of A .

<code>HasPreimage(x, f)</code>

Return whether the preimage of x under f can be taken and the preimage as a second argument if it can.

16.5 Parents of Maps

Parents of maps are structures knowing a domain and a codomain. They are often used in automorphism group calculations where a map is returned from an automorphism group into the set of all automorphisms of some structure. Parents of maps all inherit from the type `PowMap`. The type `PowMapAut` which inherits from `PowMap` is type which the parents of automorphisms inherit from.

There is also a power structure of maps (of type `PowStr`, similar to that of other structures) which is used as a common overstructure of the different parents.

<code>Parent(m)</code>

The parent of m .

<code>Domain(P)</code>

<code>Codomain(P)</code>

The domain and codomain of the maps for which P is the parent.

<code>Maps(D, C)</code>

<code>Iso(D, C)</code>

The parent of maps (or isomorphisms) from D to C . `Iso` will only return a different structure to `Maps` if it has been specifically implemented for such maps.

<code>Aut(S)</code>

The parent of automorphisms of S .

INDEX OF INTRINSICS

- !, 1-13, 1-174, 1-197, 1-216, 1-235,
2-269, 2-283, 2-336, 2-342, 2-353,
354, 2-370, 2-397, 2-413, 2-447,
2-478, 2-588, 3-653, 3-737, 3-754,
3-760, 3-780, 3-875, 876, 3-950,
3-990, 3-1037, 3-1059, 3-1127-1129,
3-1152, 3-1157, 3-1197, 4-1227,
4-1279, 4-1313, 4-1324, 4-1349,
4-1369, 4-1399, 4-1413, 4-1434,
5-1462, 5-1504, 5-1506, 5-1521,
1522, 5-1534, 5-1641, 1642, 5-1645,
5-1806, 1807, 5-1817, 5-1869, 5-1872,
5-2001, 2002, 6-2050, 6-2063, 6-2081,
6-2250, 6-2258, 6-2297-2299, 6-2348,
6-2366, 6-2378, 6-2383, 6-2388,
6-2408, 7-2421, 7-2432, 7-2456,
7-2469, 7-2507, 2508, 7-2547, 7-2551,
7-2630, 7-2691, 7-2705, 2706, 7-2709,
7-2757, 2758, 8-2981, 8-3007, 3008,
8-3042, 8-3080, 8-3113, 9-3308,
9-3404, 9-3427, 9-3488, 9-3500,
9-3503, 9-3644, 9-3657, 9-3677,
9-3688, 9-3703, 9-3739, 10-3959,
10-4133, 10-4150, 4151, 10-4196,
11-4336, 4337, 11-4364, 11-4389,
11-4429, 11-4479, 11-4493, 11-4535,
11-4583, 12-4710, 4711, 12-4787,
12-4808, 4809, 12-4811, 4812, 12-4816,
4817, 12-4844, 4845, 12-4852, 12-4871,
4872, 12-4926, 4927, 12-5000, 5001,
13-5076, 13-5192, 13-5206, 13-5253
!!, 3-931, 3-1140, 3-1216, 5-1534,
11-4436
~, 12-4850
(,), 2-590, 4-1402, 5-1464, 5-1535,
5-1650, 5-1869, 6-2083, 6-2350,
6-2368, 7-2445, 13-5078, 13-5193,
13-5207
(, ,), 5-1464, 5-1535, 5-1650, 5-1808,
5-2002, 6-2083, 6-2252, 6-2350,
6-2368
(,), 8-3116
(), 1-235, 1-253, 2-604, 4-1414,
6-2086, 6-2099, 6-2172, 6-2331,
7-2764, 9-3314, 10-3950
*, 1-66, 1-251, 2-269, 2-273, 2-287,
2-310, 2-314, 2-337, 2-339, 2-346,
2-357, 2-377, 2-397, 2-417, 2-434,
2-449, 2-481, 2-539, 2-572, 573,
2-589, 2-604, 3-654, 3-661, 3-755,
3-763, 3-794, 3-808, 3-903, 3-931,
3-939, 3-950, 3-954, 3-974, 3-990,
3-1012, 3-1045, 3-1061, 3-1130,
3-1140, 1141, 3-1154, 3-1157, 3-1159,
3-1176, 3-1196, 1197, 3-1202, 3-1220,
4-1228, 4-1283, 4-1315, 4-1326,
4-1342, 4-1352, 4-1369, 4-1382,
4-1400, 4-1414, 4-1427, 4-1435,
4-1437, 4-1449, 5-1464, 5-1534,
5-1537, 5-1598, 5-1650, 5-1676,
5-1808, 5-1869, 5-2002, 6-2052,
6-2064, 6-2083, 6-2171, 6-2251,
6-2310, 6-2349, 6-2367, 6-2378,
6-2388, 6-2409, 7-2426, 2427, 7-2456,
7-2458, 7-2460, 7-2471, 7-2481,
7-2486, 7-2517, 7-2549, 7-2554,
7-2569, 7-2574, 7-2586, 7-2630,
7-2649, 7-2691, 7-2763, 8-2981,
8-3012, 8-3032, 8-3043, 8-3066,
8-3080, 8-3115, 8-3125, 8-3147,
9-3224, 9-3278, 9-3288, 9-3309,
9-3314, 9-3320, 9-3411, 9-3431,
9-3501, 9-3532, 9-3580, 9-3677,
9-3694, 9-3701, 9-3706, 9-3863,
9-3881, 10-3962, 10-4105, 10-4139,
10-4154, 10-4197, 10-4251, 11-4336,
11-4339, 4340, 11-4364, 11-4367,
11-4394, 11-4479, 11-4499, 11-4562,
11-4584, 11-4611, 12-4692, 12-4774,
4775, 12-4788, 12-4809, 12-4812,
12-4827, 12-4849, 13-5077, 13-5193,
13-5206, 13-5255
*:=, 1-66, 2-270, 2-287, 2-337, 2-357,
2-377, 2-397, 2-417, 2-449, 2-481,
3-654, 3-1045, 4-1228, 4-1315,
5-1808, 6-2251, 6-2310, 7-2471,
8-3147, 10-3962, 10-4154, 12-4849
+, 2-269, 2-273, 2-287, 2-310, 2-314,
2-337, 2-339, 2-357, 2-377, 2-397,
2-417, 2-434, 2-449, 2-481, 2-539,
2-572, 2-589, 2-601, 3-653, 3-664,
3-737, 3-794, 3-808, 3-869, 3-903,
3-940, 3-950, 3-954, 3-974, 3-1045,
3-1061, 3-1093, 3-1130, 3-1141,
3-1154, 3-1159, 3-1176, 3-1197,
3-1202, 3-1220, 4-1228, 4-1283,
4-1315, 4-1326, 4-1342, 4-1352,
4-1369, 4-1382, 4-1400, 4-1405,
4-1427, 4-1435, 4-1449, 6-2052,
6-2064, 7-2427, 7-2455, 2456, 7-2460,
7-2471, 7-2481, 7-2486, 7-2516,
7-2553, 7-2569, 7-2630, 7-2691,
7-2694, 7-2706, 7-2763, 8-2844,
8-2887, 8-2981, 8-3032, 8-3043,

- 8-3066, 8-3080, 8-3146, 3147, 9-3224,
9-3278, 9-3288, 9-3309, 9-3320,
9-3411, 9-3431, 9-3501, 9-3580,
9-3694, 9-3701, 9-3706, 9-3881,
10-3961, 10-4103, 10-4154, 11-4340,
11-4364, 11-4394, 11-4479, 11-4499,
11-4562, 4563, 11-4574, 11-4586,
11-4611, 11-4620, 12-4690, 12-4774,
12-4787, 4788, 12-4849, 12-4932-4934,
12-5012-5014, 12-5049, 5050, 13-5077,
13-5083, 13-5188, 13-5193, 13-5206,
13-5210, 13-5255
+:=, 2-270, 2-287, 2-337, 2-357, 2-377,
2-397, 2-417, 2-449, 2-481, 3-654,
3-1045, 4-1228, 4-1315, 7-2471,
8-3146, 3147, 10-3962, 10-4154,
12-4849, 12-4933, 4934, 12-5013, 5014,
12-5050
-, 2-269, 2-287, 2-310, 2-314, 2-337,
2-357, 2-377, 2-397, 2-417, 2-449,
2-481, 2-539, 2-572, 2-589, 3-653,
3-794, 3-808, 3-903, 3-950, 3-954,
3-974, 3-1045, 3-1061, 3-1130,
3-1154, 3-1159, 3-1176, 3-1197,
3-1202, 3-1220, 4-1228, 4-1283,
4-1315, 4-1326, 4-1342, 4-1352,
4-1369, 4-1382, 4-1400, 4-1435,
6-2052, 7-2427, 7-2456, 7-2471,
7-2516, 2517, 7-2553, 2554, 7-2630,
7-2691, 7-2763, 8-2981, 8-3032,
8-3043, 8-3066, 8-3080, 9-3309,
9-3411, 9-3431, 9-3501, 9-3580,
9-3694, 9-3701, 9-3706, 9-3881,
10-3961, 3962, 10-4135, 10-4150,
10-4154, 10-4197, 11-4340, 11-4364,
11-4394, 11-4479, 11-4499, 11-4563,
11-4611, 12-4775, 12-4788, 12-4849,
12-4933, 12-4935, 12-5013, 12-5015,
5016, 13-5077, 13-5193, 13-5206,
13-5255
-:=, 2-270, 2-287, 2-337, 2-357, 2-377,
2-397, 2-417, 2-449, 2-481, 3-654,
3-1045, 4-1228, 4-1315, 7-2471,
10-3962, 10-4154, 12-4849, 12-4933,
12-4935, 12-5013, 12-5016
-A, 2-572
-x, 8-3066
., 2-342, 343, 2-370, 371, 2-413, 2-435,
2-447, 2-478, 2-599, 3-653, 3-782,
3-795, 3-882, 3-905, 3-974, 3-1038,
3-1059, 3-1127, 3-1197, 3-1200,
4-1274, 4-1313, 4-1324, 4-1340,
4-1348, 4-1369, 4-1397, 4-1425,
5-1480, 5-1524, 5-1645, 5-1797,
5-1870, 5-2001, 6-2044, 6-2048,
6-2098, 6-2264, 6-2297, 2298, 6-2346,
6-2363, 6-2378, 6-2391, 6-2405,
7-2423, 7-2456, 7-2469, 7-2485,
7-2510, 7-2568, 7-2630, 7-2687,
8-3015, 8-3042, 8-3065, 8-3080,
9-3287, 9-3404, 9-3427, 9-3484,
9-3494, 9-3500, 9-3677, 9-3877,
11-4389, 11-4479, 11-4493, 11-4578,
11-4619, 12-4710, 4711, 12-4787,
12-4808, 12-4811, 12-4843, 12-4871,
12-4926, 4927, 13-5072, 13-5165,
13-5205
/, 2-270, 2-273, 2-287, 2-310, 2-314,
2-337, 2-346, 2-353, 2-357, 2-377,
2-397, 2-417, 2-449, 2-481, 2-589,
2-596, 3-654, 3-661, 3-794, 3-903,
3-940, 3-950, 3-1045, 3-1061,
3-1130, 3-1141, 3-1176, 4-1228,
4-1284, 4-1326, 4-1342, 4-1369,
4-1400, 4-1435, 5-1464, 5-1472,
5-1534, 5-1561, 5-1650, 5-1673,
5-1808, 5-1828, 6-2055, 6-2088,
6-2252, 6-2259, 6-2310, 6-2349,
6-2367, 7-2422, 7-2427, 7-2457,
7-2471, 7-2481, 7-2484, 7-2549,
7-2630, 7-2692, 8-3009, 8-3147,
9-3224, 9-3285, 9-3320, 9-3412,
9-3501, 9-3694, 10-3962, 10-4251,
11-4340, 11-4364, 11-4394, 11-4575,
11-4589, 11-4620, 12-4788
/:=, 2-270, 2-287, 2-337, 2-357, 2-481,
5-1808, 6-2252, 6-2310, 8-3147,
10-3962
< >, 1-216
=, 6-2042, 6-2085, 6-2390
@, 1-253, 6-2099, 6-2172, 6-2331,
7-2764, 9-3435, 9-3490, 9-3679,
9-3689, 9-3867, 10-4139, 11-4558,
11-4615
@@, 1-253, 6-2100, 6-2331, 9-3538,
9-3540, 9-3679, 10-4139, 11-4558,
11-4615
[...], 1-66, 67, 1-176, 1-195, 196, 1-198,
199, 1-216, 1-218, 1-224, 225, 1-229,
2-531, 2-564, 2-593, 3-800, 3-910,
3-990, 4-1400, 6-2086, 7-2434,
7-2473, 7-2523, 7-2692, 7-2764,
8-3033, 3034, 9-3489, 9-3644, 9-3658,
10-3961, 10-4135, 10-4154, 10-4197,
12-4722, 4723, 13-5079, 13-5208
[* *], 1-223
[], 2-530, 2-564, 2-592, 3-756, 4-1400,
6-2042, 6-2085, 2086, 7-2434, 7-2523,
7-2554, 7-2692, 8-3033, 3034, 9-3309,
9-3314, 12-4809, 13-5195
" ", 1-66
#, 1-11, 1-67, 1-176, 1-198, 1-216,
1-218, 1-224, 1-236, 2-266, 2-335,
2-375, 2-416, 2-448, 3-703, 3-709,
3-737, 3-827, 3-989, 3-1183, 4-1276,
4-1315, 5-1481, 5-1504, 5-1526,

- 5-1599, 5-1656, 5-1753, 5-1798,
- 5-1954, 5-1958, 5-1970, 1971, 5-1973,
- 5-1975, 5-1984, 5-1997, 6-2061,
- 6-2081, 6-2105, 6-2172, 6-2265,
- 6-2303, 6-2325, 6-2347, 6-2350,
- 6-2364, 6-2368, 6-2378, 6-2388,
- 6-2407, 6-2409, 7-2422, 7-2469,
- 7-2705, 7-2764, 2765, 8-2914, 8-3014,
- 8-3110, 10-3948, 10-3972, 10-4051,
- 10-4136, 10-4157, 11-4623, 12-4718,
- 12-4809, 12-4812, 12-4878, 12-4881,
- 12-4928, 12-4984, 13-5071, 13-5165,
- 13-5205
- #A, 11-4524
- #N, 2-399
- &, 1-189, 1-211, 9-3493
- &*, 1-66, 1-219, 3-939
- &cat, 1-66
- &meet, 3-941, 3-990
- &meet S, 2-601, 9-3226, 9-3279
- \[...], 1-197
- ~, 1-66, 2-270, 2-287, 2-310, 2-314,
- 2-337, 2-346, 2-357, 2-377, 2-397,
- 2-417, 2-424, 2-449, 2-481, 2-539,
- 2-572, 3-755, 3-794, 3-903, 3-940,
- 3-950, 3-1045, 3-1061, 3-1130,
- 3-1141, 3-1197, 3-1202, 4-1228,
- 4-1283, 4-1315, 4-1326, 4-1342,
- 4-1352, 4-1369, 4-1414, 5-1464,
- 5-1488, 1489, 5-1492, 5-1534, 1535,
- 5-1550, 1551, 5-1567, 5-1650, 5-1667,
- 1668, 5-1676, 5-1688, 5-1808,
- 5-1813, 5-1819, 5-1869, 5-2002,
- 6-2083, 6-2159, 2160, 6-2251, 2252,
- 6-2270, 6-2310, 6-2349, 2350, 6-2368,
- 6-2378, 6-2388, 6-2409, 7-2426, 2427,
- 7-2455, 7-2457, 7-2471, 7-2517,
- 7-2569, 7-2642, 7-2690, 7-2715,
- 7-2737, 7-2763, 7-2765, 8-3012,
- 8-3043, 8-3080, 3081, 8-3116, 8-3125,
- 9-3224, 9-3278, 9-3288, 9-3355, 3356,
- 9-3412, 9-3431, 9-3501, 9-3677,
- 9-3863, 11-4336, 11-4394, 11-4562,
- 11-4584, 12-4732, 12-4787, 12-4849,
- 12-4896, 12-4938, 13-5130
- ~-1, 2-572
- ~:=, 2-270, 2-287, 2-337, 2-357, 2-481,
- 5-1808, 6-2251, 2252, 6-2310
- ‘, 1-52, 1-243
- ‘‘, 1-52
- ‘‘, 1-243
- { }, 1-167, 1-172, 173
- {* *}, 1-170, 171
- {@ @}, 1-169
- A, 11-4590
- AbelianBasis, 5-1494, 5-1831
- AbelianExtension, 3-1007, 1008, 3-1010,
- 3-1192
- AbelianGroup, 2-343, 5-1467, 5-1473,
- 5-1530, 5-1792, 5-1856, 6-2043, 2044,
- 6-2049, 6-2055, 6-2094, 6-2260,
- 6-2263, 10-3981, 10-4004, 10-4163,
- 11-4622
- AbelianInvariants, 5-1494, 5-1705, 5-1831
- AbelianLieAlgebra, 8-2978
- AbelianNormalQuotient, 5-1597
- AbelianNormalSubgroup, 5-1597
- AbelianpExtension, 3-1008
- AbelianQuotient, 5-1562, 5-1674, 5-1829,
- 6-2055, 6-2123, 6-2278
- AbelianQuotientInvariants, 5-1829, 6-2123,
- 2124, 6-2278
- AbelianSubfield, 3-1014
- AbelianSubgroups, 5-1499, 5-1560, 5-1824
- Abs, 2-290, 2-314, 2-359, 2-427, 2-467,
- 2-484, 11-4365
- AbsoluteAffineAlgebra, 3-1049
- AbsoluteAlgebra, 10-4067
- AbsoluteBasis, 2-355, 3-791, 3-896
- AbsoluteCartanMatrix, 7-2752
- AbsoluteCharacteristicPolynomial, 3-798,
- 3-908
- AbsoluteDegree, 2-356, 3-788, 3-891,
- 3-1016, 3-1100, 4-1273
- AbsoluteDiscriminant, 2-356, 3-788,
- 3-892, 3-1016, 3-1101
- AbsoluteField, 3-783, 3-883
- AbsoluteFunctionField, 3-1096
- AbsoluteGaloisGroup, 3-1020
- AbsoluteInertiaDegree, 4-1272
- AbsoluteInertiaIndex, 4-1272
- AbsoluteInvariants, 10-4127
- AbsoluteLogarithmicHeight, 3-796, 3-906
- AbsolutelyIrreducibleConstituents, 7-2741
- AbsolutelyIrreducibleModule, 7-2696
- AbsolutelyIrreducibleModules, 7-2738
- AbsolutelyIrreducibleModulesBurnside,
- 7-2741
- AbsolutelyIrreducibleModulesInit, 7-2744
- AbsolutelyIrreducibleModulesSchur, 5-1850,
- 7-2742
- AbsolutelyIrreducibleRepresentationProc-
- essDelete, 7-2744
- AbsolutelyIrreducibleRepresentationsInit,
- 7-2744
- AbsolutelyIrreducibleRepresentationsSchur,
- 5-1850
- AbsoluteMinimalPolynomial, 3-799, 3-909,
- 3-1132
- AbsoluteModuleOverMinimalField, 7-2731
- AbsoluteModulesOverMinimalField, 7-2732
- AbsoluteNorm, 2-379, 3-798, 3-908, 3-933
- AbsoluteOrder, 3-883, 3-1096
- AbsolutePolynomial, 3-1049
- AbsolutePrecision, 4-1286, 4-1327, 4-1342
- AbsoluteQuotientRing, 3-1049

- AbsoluteRamificationDegree, 4-1273
- AbsoluteRamificationIndex, 4-1273
- AbsoluteRank, 8-2865
- AbsoluteRationalScroll, 9-3487
- AbsoluteRepresentation, 5-1687
- AbsoluteRepresentationMatrix, 3-799, 3-909
- AbsoluteTotallyRamifiedExtension, 4-1271
- AbsoluteTrace, 2-379, 3-798, 3-908
- AbsoluteValue, 2-290, 2-314, 2-359,
2-427, 2-467, 2-484, 11-4365
- AbsoluteValues, 3-796, 3-906
- Absolutize, 3-1049
- ActingGroup, 5-2030, 8-3104
- ActingWord, 5-1602
- Action, 5-1565, 5-1572, 6-2172, 7-2583,
7-2688, 12-4733, 12-4893, 12-4976
- ActionGenerator, 3-729, 7-2582, 7-2688,
7-2729
- ActionGenerators, 7-2729
- ActionGroup, 7-2729
- ActionImage, 5-1572, 12-4733, 12-4893,
12-4977
- ActionKernel, 5-1572, 12-4733, 12-4894,
12-4977
- ActionMatrix, 7-2614, 7-2715
- AdamsOperator, 8-3153
- AddAttribute, 1-52
- AddColumn, 2-535, 2-568, 7-2525
- AddConstraints, 13-5278
- AddCubics, 10-4028, 10-4103
- AddEdge, 12-4934, 12-5014, 5015, 12-5050
- AddEdges, 12-4934, 4935, 12-5015, 12-5050
- AddGenerator, 3-1031, 6-2204, 6-2394
- AdditiveCode, 13-5200, 5201
- AdditiveCyclicCode, 13-5217, 5218
- AdditiveGroup, 2-285, 2-335, 2-373,
4-1274
- AdditiveHilbert90, 2-380
- AdditiveOrder, 8-2843, 8-2883, 8-2919,
8-3122
- AdditivePolynomialFromRoots, 3-1200
- AdditiveQuasiCyclicCode, 13-5218
- AdditiveRepetitionCode, 13-5202
- AdditiveUniverseCode, 13-5203
- AdditiveZeroCode, 13-5202
- AdditiveZeroSumCode, 13-5203
- AddNormalizingGenerator, 5-1620
- AddRedundantGenerators, 6-2379
- AddRelation, 3-918, 6-2204, 6-2393
- AddRelator, 6-2212
- AddRepresentation, 8-3147
- AddRow, 2-534, 2-568, 7-2525
- AddScaledMatrix, 2-539, 540
- AddSimplex, 12-4693
- Addsimplex, 12-4693
- AddSubgroupGenerator, 6-2213
- AddVectorToLattice, 12-4790
- AddVertex, 12-4933, 12-5013
- AddVertices, 12-4933, 12-5013
- adj, 12-4943, 12-5021
- AdjacencyMatrix, 3-705, 12-4957
- Adjoin, 7-2455
- Adjoint, 2-548, 7-2520, 9-3434
- AdjointAlgebra, 7-2666
- AdjointIdeal, 9-3654
- AdjointIdealForNodalCurve, 9-3654
- AdjointLinearSystem, 9-3655
- AdjointLinearSystemForNodalCurve, 9-3654
- AdjointLinearSystemFromIdeal, 9-3654
- AdjointMatrix, 8-3033
- AdjointPreimage (G, g), 5-1907
- AdjointRepresentation, 8-3132, 8-3140,
8-3145
- AdjointRepresentationDecomposition, 8-3139
- Adjoints, 9-3655
- AdjointVersion, 8-2889
- AdmissableTriangleGroups, 11-4372
- AdmissiblePair, 11-4676
- Advance, 5-1629, 5-1944, 5-1963, 5-1968,
5-1982
- AffineAction, 5-1591
- AffineAlgebra, 3-1044, 9-3286
- AffineAlgebraMapKernel, 9-3290
- AffineDecomposition, 9-3519, 9-3548
- AffineGammaLinearGroup, 5-1621
- AffineGeneralLinearGroup, 5-1620, 5-1881
- AffineGroup, 2-401, 5-1626
- AffineImage, 5-1591
- AffineKernel, 5-1591
- AffineLieAlgebra, 8-3063
- AffinePatch, 9-3517, 3518, 9-3669
- AffinePlane, 9-3643
- AffineSigmaLinearGroup, 5-1621
- AffineSpace, 9-3483, 3484, 9-3643
- AffineSpecialLinearGroup, 5-1621, 5-1881
- AFRNumber, 9-3839
- AGammaL, 5-1621
- AGCode, 13-5140
- AGDecode, 13-5143
- AGDualCode, 13-5140
- Agemo, 5-1833, 6-2065
- AGL, 5-1620, 5-1881
- AGM, 2-509
- AHom, 7-2588, 7-2709
- AInfinityRecord, 7-2612
- aInvariants, 10-3942, 10-4105
- Alarm, 1-90
- AlgComb, 4-1382
- Algebra, 2-355, 3-786, 3-889, 4-1442,
7-2420, 7-2431, 2432, 7-2442, 2443,
7-2452, 7-2459, 7-2486, 7-2550,
7-2583, 7-2640, 7-2715, 8-2979,
8-3041, 9-3373
- AlgebraGenerators, 7-2537
- AlgebraicClosure, 3-1036
- AlgebraicGenerators, 8-3109

- AlgebraicGeometricCode, 13-5140
- AlgebraicGeometricDualCode, 13-5140
- AlgebraicPowerSeries, 4-1377
- AlgebraicToAnalytic, 3-1209
- AlgebraMap, 9-3536
- AlgebraOverCenter, 7-2443
- AlgebraStructure, 7-2537
- AlgorithmicFunctionField, 9-3692
- AllCliques, 12-4962, 4963
- AllCompactChainMaps, 7-2607
- AllCones, 9-3864
- AllDefiningPolynomials, 9-3536
- Alldeg, 12-4945, 12-4947, 12-5023, 12-5025
- AllExtensions, 4-1308
- AllFaces, 4-1238
- AllHomomorphisms, 6-2068
- AllInformationSets, 13-5074
- AllInverseDefiningPolynomials, 9-3536
- AllIrreduciblePolynomials, 2-382
- AllLinearRelations, 2-491
- AllNilpotentLieAlgebras, 8-3048
- AllPairsShortestPaths, 12-5036
- AllParallelClasses, 12-4886
- AllParallelisms, 12-4886
- AllPartitions, 5-1576
- AllPassants, 12-4727
- AllRays, 9-3866
- AllResolutions, 12-4885
- AllRoots, 2-381
- AllSecants, 12-4727
- AllSlopes, 4-1241
- AllSolvableLieAlgebras, 8-3048
- AllSqrts, 2-338
- AllSquareRoots, 2-338
- AllTangents, 12-4727, 12-4729
- AllVertices, 4-1238
- AlmostSimpleGroupDatabase, 5-1957
- Alphabet, 13-5072, 13-5165, 13-5203
- AlphaBetaData, 10-4220
- Alt, 5-1473, 5-1530, 6-2094
- AlternantCode, 13-5102
- AlternatingCharacter, 7-2782
- AlternatingCharacterTable, 7-2782
- AlternatingCharacterValue, 7-2782
- AlternatingDominant, 8-3158, 3159
- AlternatingElementToWord (G, g), 5-1611, 5-1892
- AlternatingGroup, 5-1473, 5-1530, 6-2094
- AlternatingPower, 8-3153
- AlternatingSquarePreimage (G, g), 5-1907
- AlternatingSum, 2-511
- AlternatingWeylSum, 8-3160
- Ambient, 9-3307, 9-3496, 9-3570, 9-3866, 12-4785
- AmbientMatrix, 9-3314
- AmbientModule, 11-4481
- AmbientSpace, 3-657, 9-3496, 9-3570, 9-3649, 11-4397, 13-5072, 13-5165, 13-5204
- AmbientVariety, 11-4623
- AmbiguousForms, 3-759
- AModule, 7-2582, 7-2606
- AnalyticDrinfeldModule, 3-1205
- AnalyticHomomorphisms, 10-4204
- AnalyticInformation, 10-4086
- AnalyticJacobian, 10-4200
- AnalyticModule, 3-1208
- AnalyticRank, 10-4044, 10-4068, 10-4086
- And, 1-207
- and, 1-11
- Angle, 11-4341, 11-4366
- AnisotropicSubdatum, 8-2865
- Annihilator, 7-2575, 9-3322
- AntiAutomorphismTau, 8-3086
- Antipode, 8-3085
- AntisymmetricForms, 3-730, 5-1779
- AntisymmetricMatrix, 2-526, 527
- ApparentCodimension, 9-3827, 9-3836
- ApparentEquationDegrees, 9-3827, 9-3836
- ApparentSyzygyDegrees, 9-3827, 9-3836
- Append, 1-200, 1-216, 217, 1-223
- Apply, 9-3435
- ApplyContravariant, 9-3812
- ApplyTransformation, 10-4105
- ApproximateByTorsionGroup, 11-4617
- ApproximateByTorsionPoint, 11-4616
- ApproximateOrder, 11-4612
- ApproximateStabiliser, 5-1681
- AQInvariants, 5-1829, 6-2123, 2124, 6-2278
- Arccos, 2-495, 4-1334
- Arccosec, 2-496
- Arccot, 2-496
- Arcsec, 2-496
- Arcsin, 2-495, 4-1334
- Arctan, 2-496, 4-1334
- Arctan2, 2-496
- AreCohomologous, 5-2031
- AreIdentical, 6-2314
- AreInvolutionsConjugate, 5-1711
- AreLinearlyEquivalent, 9-3885
- AreProportional, 12-4788
- ArfInvariant, 2-623
- Arg, 2-482
- Argcosech, 2-498
- Argcosh, 2-498, 4-1335
- Argcoth, 2-499
- Argsech, 2-498
- Argsinh, 2-498, 4-1334
- Argtanh, 2-498, 4-1335
- Argument, 2-482, 11-4365
- ArithmeticGenus, 9-3512, 9-3667, 9-3756
- ArithmeticGenusOfDesingularization, 9-3782
- ArithmeticGeometricMean, 2-509
- ArithmeticTriangleGroup, 11-4372

- ArithmeticVolume, 11-4360, 11-4366
- Arrows, 9-3749
- ArtinMap, 3-1018
- ArtinRepresentation, 3-1219, 10-4223
- ArtinRepresentations, 3-1215
- ArtinSchreierExtension, 3-1181
- ArtinSchreierImage, 3-1198
- ArtinSchreierMap, 3-1198
- ArtinTateFormula, 10-4276
- AsExtensionOf, 3-869, 3-1093
- ASigmaL, 5-1621
- ASL, 5-1621, 5-1881
- AssertAttribute, 2-305, 2-369, 4-1323, 5-1543, 5-1616, 5-1665, 5-1701, 5-1703, 7-2769
- AssertEmbedding, 11-4540
- AssignCapacities, 12-5004, 5005
- AssignCapacity, 12-5004
- assigned, 1-6, 1-52, 1-243
- AssignEdgeLabels, 12-5005
- AssignLabel, 12-5003, 5004
- AssignLabels, 12-5003, 5004
- AssignLDPCMatrix, 13-5148
- AssignNamePrefix, 3-1036
- AssignNames, 1-9, 2-342, 2-369, 2-413, 2-446, 2-476, 3-782, 3-835, 3-882, 3-1058, 3-1087, 3-1152, 3-1158, 4-1276, 4-1312, 4-1324, 4-1340, 4-1348, 4-1366, 7-2468, 7-2621, 8-3041, 8-3079, 9-3403, 9-3426, 9-3484, 9-3494, 9-3500, 9-3877
- AssignVertexLabels, 12-5003
- AssignWeight, 12-5004
- AssignWeights, 12-5004, 5005
- AssociatedEllipticCurve, 10-4014, 10-4020
- AssociatedGradedAlgebra, 7-2577
- AssociatedHyperellipticCurve, 10-4020
- AssociatedNewSpace, 11-4438
- AssociatedPrimitiveCharacter, 2-344, 3-812
- AssociatedPrimitiveGrossencharacter, 3-820
- AssociativeAlgebra, 7-2420, 7-2441, 2442
- AssociativeArray, 1-229
- AteqPairing, 10-3984
- AteTPairing, 10-3984
- AtkinLehner, 11-4446
- AtkinLehnerInvolution, 11-4293, 11-4310
- AtkinLehnerOperator, 11-4401, 11-4486, 11-4501, 11-4629, 4630, 11-4652
- AtkinModularPolynomial, 11-4287
- ATLASGroup, 5-1984
- ATLASGroupNames, 5-1984
- Attach, 1-47
- AttachSpec, 1-49
- Augmentation, 7-2554
- AugmentationIdeal, 7-2551
- AugmentationMap, 7-2550
- AugmentCode, 13-5106, 13-5219
- Aut, 1-254, 9-3551, 10-4138, 12-4892, 13-5132
- AutoCorrelation, 13-5268
- AutomaticGroup, 6-2358, 2359
- Automorphism, 8-3125, 9-3545, 9-3548, 9-3551, 9-3672, 10-3923, 3924, 10-3953
- AutomorphismGroup, 2-355, 2-375, 2-401, 3-719, 3-721, 3-727, 3-803, 3-962, 963, 3-1018, 3-1110, 3-1113, 1114, 4-1302, 4-1368, 5-1602, 5-1694, 5-1836, 5-1841, 5-1994, 5-1996, 6-2070, 7-2580, 7-2712, 8-3125, 9-3551, 9-3676, 10-3959, 10-4141, 12-4731, 12-4756, 12-4779, 12-4890, 12-4896, 12-4968, 13-5131, 13-5221, 13-5250
- AutomorphismGroupMatchingIdempotents, 7-2579
- AutomorphismGroupOverCyclotomicExtension, 11-4311
- AutomorphismGroupOverExtension, 11-4311
- AutomorphismGroupOverQ, 11-4310
- AutomorphismGroupSolubleGroup, 5-1839
- AutomorphismGroupStabilizer, 12-4891, 13-5132
- AutomorphismOmega, 8-3086
- Automorphisms, 3-962, 3-1110, 3-1113, 4-1302, 9-3676, 10-3959
- AutomorphismSubgroup, 12-4891, 13-5131
- AutomorphismTalpha, 8-3086
- AutomorphousClasses, 3-703
- AuxiliaryLevel, 11-4496
- BachBound, 3-802, 3-914
- BadPlaces, 10-4054, 10-4079
- BadPrimes, 10-3913, 10-3997, 10-4171
- BaerDerivation, 12-4738
- BaerSubplane, 12-4738
- Ball, 12-4956
- Bang, 1-251
- BarAutomorphism, 8-3086
- BarycentricSubdivision, 12-4694
- Base, 5-1617, 5-1703
- BaseChange, 3-660, 7-2790, 9-3515, 3516, 9-3648, 10-3936, 3937, 10-4117, 10-4146, 10-4196
- BaseChangeMatrix, 7-2594
- BaseComponent, 9-3571
- BaseCurve, 11-4292
- BaseElement, 6-2325
- BaseExtend, 2-342, 3-660, 8-3109, 9-3515, 3516, 10-3936, 3937, 10-4117, 10-4146, 10-4196, 11-4388, 11-4478, 11-4541, 11-4570
- BaseField, 2-355, 2-367, 2-599, 3-783, 3-883, 3-1016, 3-1043, 3-1095, 1096, 3-1183, 3-1196, 1197, 4-1273, 4-1397, 7-2632, 8-2834, 9-3405, 9-3496,

- 9-3649, 10-3907, 10-4130, 10-4145,
10-4195, 10-4201, 11-4649, 11-4651,
11-4665
- BaseImage, 5-1618
- BaseImageWordStrip, 5-1619
- BaseLocus, 9-3581
- BaseModule, 7-2510, 8-3034, 3035
- BaseMPolynomial, 2-324
- BasePoint, 5-1617, 5-1703
- BasePoints, 9-3542, 9-3573
- BaseRing, 2-343, 344, 2-415, 2-447,
2-530, 2-563, 3-660, 3-883, 3-974,
3-1016, 3-1059, 3-1095, 1096, 3-1197,
3-1200, 3-1202, 4-1273, 4-1325,
4-1339, 4-1348, 4-1364, 4-1397,
4-1424, 5-1645, 7-2422, 7-2452,
7-2469, 7-2510, 7-2551, 2552, 7-2568,
7-2632, 7-2687, 8-2834, 8-2865,
8-2980, 8-2989, 8-3014, 8-3041,
8-3064, 8-3107, 8-3109, 9-3307,
9-3405, 9-3428, 9-3496, 9-3649,
9-3876, 10-3907, 10-3945, 10-3948,
10-4100, 10-4130, 10-4145, 10-4195,
10-4201, 11-4332, 11-4358, 11-4397,
11-4481, 11-4496, 11-4521, 11-4649,
11-4665, 12-4846
- BaseScheme, 9-3542, 9-3571
- BasicAlgebra, 7-2561-2563
- BasicAlgebraOfBlockAlgebra, 7-2564
- BasicAlgebraOfEndomorphismAlgebra, 7-2563
- BasicAlgebraOfExtAlgebra, 7-2564, 7-2604
- BasicAlgebraOfGroupAlgebra, 7-2563
- BasicAlgebraOfHeckeAlgebra, 7-2563
- BasicAlgebraOfMatrixAlgebra, 7-2563
- BasicAlgebraOfPrincipalBlock, 7-2564
- BasicAlgebraOfSchurAlgebra, 7-2563
- BasicCodegrees, 8-2911, 8-2959
- BasicDegrees, 8-2911, 8-2959
- BasicOrbit, 5-1617, 5-1703
- BasicOrbitLength, 5-1617, 5-1703
- BasicOrbitLengths, 5-1617, 5-1703
- BasicOrbits, 5-1617
- BasicRootMatrices, 8-2955
- BasicStabiliser, 5-1617, 5-1704
- BasicStabiliserChain, 5-1617, 5-1704
- BasicStabilizer, 5-1617, 5-1704
- BasicStabilizerChain, 5-1617, 5-1704
- Basis, 2-355, 2-602, 3-659, 3-790,
3-895, 3-935, 3-1100, 3-1147,
3-1164, 4-1403, 4-1428, 4-1437,
7-2423, 7-2453, 7-2459, 7-2475,
7-2522, 7-2568, 7-2632, 7-2715,
7-2760, 8-2990, 8-3015, 9-3190,
9-3275, 9-3317, 9-3712, 10-4081,
11-4390, 11-4431, 11-4481, 11-4495,
11-4578, 12-4787, 4788, 13-5072,
13-5165, 13-5205
- BasisChange, 8-2874
- BasisDenominator, 3-659
- BasisElement, 2-602, 7-2423, 7-2476,
7-2522, 8-3015, 9-3190, 9-3275,
9-3317
- BasisMatrix, 2-602, 3-659, 3-738, 3-896,
3-935, 3-1100, 3-1147, 7-2459,
7-2552, 7-2640, 9-3317, 13-5072,
13-5205
- BasisOfDifferentialsFirstKind, 3-1175,
9-3693
- BasisOfHolomorphicDifferentials, 3-1175,
9-3693
- BasisProduct, 7-2432, 8-3008
- BasisProducts, 7-2433, 8-3008
- BasisReduction, 3-672, 673
- Basket, 9-3833, 9-3835
- BBSModulus, 13-5268
- BCHBound, 13-5120
- BCHCode, 13-5100
- BDLC, 13-5123
- BDLCLowerBound, 13-5118
- BDLCUpperBound, 13-5118
- Bell, 2-296, 12-4800
- BerlekampMassey, 13-5265
- BernoulliApproximation, 2-509, 12-4800
- BernoulliNumber, 2-509, 12-4800
- BernoulliPolynomial, 2-438, 12-4800
- BesselFunction, 2-507
- BesselFunctionSecondKind, 2-508
- BestApproximation, 2-490
- BestDimensionLinearCode, 13-5123
- BestKnownLinearCode, 13-5122
- BestKnownQuantumCode, 13-5247
- BestLengthLinearCode, 13-5122
- BestTranslation, 2-326
- BettiNumber, 9-3331, 10-4087, 12-4698
- BettiNumbers, 9-3331, 9-3836
- BettiTable, 9-3331
- BFSTree, 12-4958, 12-5029
- BianchiCuspForms, 11-4665
- Bicomponents, 12-4949, 12-5026
- Big0, 4-1280, 4-1325
- BigPeriodMatrix, 10-4200
- BinaryForms, 9-3384
- BinaryQuadraticForms, 3-753
- BinaryResidueCode, 13-5174
- BinaryString, 1-66
- BinaryTorsionCode, 13-5174
- Binomial, 2-296, 12-4799
- bInvariants, 10-3943, 10-4105
- BipartiteGraph, 12-4921
- Bipartition, 12-4946, 12-5022
- BiquadraticResidueSymbol, 3-841
- BitFlip, 13-5259
- BitPrecision, 2-480, 2-483
- BKLC, 13-5122
- BKLCLowerBound, 13-5118
- BKLCUpperBound, 13-5118

- BKQC, 13-5247
- BLLC, 13-5122
- BLLCLowerBound, 13-5118
- BLLCUpperBound, 13-5118
- Block, 12-4871, 12-4882
- BlockDegree, 12-4879, 12-4881
- BlockDegrees, 12-4879
- BlockGraph, 12-4895, 12-4941
- BlockGroup, 12-4891
- BlockMatrix, 2-537
- Blocks, 5-1714, 7-2774, 12-4878
- BlocksAction, 5-1576
- BlockSet, 12-4871
- BlocksImage, 5-1576, 5-1714
- BlockSize, 12-4879, 12-4881
- BlockSizes, 12-4879
- BlocksKernel, 5-1576
- Blowup, 9-3660, 9-3863, 9-3888
- BlumBlumShub, 13-5267
- BlumBlumShubModulus, 13-5268
- BogomolovNumber, 9-3845
- BooleanPolynomialRing, 9-3199, 3200
- Booleans, 1-11
- BorderedDoublyCirculantQRCode, 13-5104
- Borel, 12-4752
- BorelSubgroup, 12-4752
- Bottom, 3-989, 5-1504, 7-2705
- Bound, 3-977
- Boundary, 12-4689
- BoundaryIntersection, 11-4367
- BoundaryMap, 4-1443, 11-4442
- BoundaryMaps, 4-1443
- BoundaryMatrix, 12-4698
- BoundaryPoints, 12-4778
- BoundedFSubspace, 11-4595
- BQPlotkinSum, 13-5177
- BraidGroup, 6-2094, 6-2296, 8-2930
- Branch, 8-3151
- BranchVertexPath, 12-4959
- BrandtModule, 11-4477, 4478, 11-4487, 11-4497
- BrandtModuleDimension, 11-4486, 4487
- BrandtModuleDimensionOfNewSubspace, 11-4487
- BrauerCharacter, 7-2774
- BrauerClass, 11-4596
- BravaisGroup, 5-1781
- BreadthFirstSearchTree, 12-4958, 12-5029
- Bruhat, 8-3117
- BruhatDescendants, 8-2912
- BruhatLessOrEqual, 8-2911
- BSGS, 5-1613, 5-1701
- BString, 1-66
- BuildHomomorphismFromGradedCap, 7-2577
- BureauRepresentation, 6-2335
- BurnsideMatrix, 5-1825
- CalabiYau, 9-3846
- CalculateCanonicalClass, 9-3744
- CalculateMultiplicities, 9-3744
- CalculateTransverseIntersections, 9-3745
- CalderbankShorSteaneCode, 13-5232
- CambridgeMatrix, 7-2508
- CanChangeRing, 11-4541
- CanChangeUniverse, 1-181, 1-204
- CanContinueEnumeration, 6-2215
- CanDetermineIsomorphism, 11-4528
- CanIdentifyGroup, 5-1945
- CanNormalize, 3-1208
- CanonicalBasis, 8-3083
- CanonicalClass, 9-3745, 9-3881
- CanonicalCoordinateIdeal, 9-3768
- CanonicalCurve, 10-4224
- CanonicalDissidentPoints, 9-3834
- CanonicalDivisor, 3-1158, 9-3577, 9-3706, 9-3881
- CanonicalElements, 8-3091
- CanonicalFactorRepresentation, 6-2303
- CanonicalGraph, 12-4972
- CanonicalHeight, 10-4007, 10-4167
- CanonicalImage, 9-3713
- CanonicalInvolution, 11-4293
- CanonicalLength, 6-2304
- CanonicalLinearSystem, 9-3655
- CanonicalLinearSystemFromIdeal, 9-3654
- CanonicalMap, 9-3713
- CanonicalModularPolynomial, 11-4287
- CanonicalScheme, 10-4224
- CanonicalSheaf, 9-3600
- CanonicalWeightedModel, 9-3767
- CanRedoEnumeration, 6-2215
- CanSignNormalize, 3-1209
- CanteautChabaudsAttack, 13-5116
- Capacities, 12-5006
- Capacity, 12-5006
- car, 1-215
- Cardinality, 2-399
- CarlitzModule, 3-1203
- CarmichaelLambda, 2-293
- CartanInteger, 8-2895
- CartanMatrix, 7-2538, 7-2752, 8-2807, 2808, 8-2815, 8-2833, 8-2863, 8-2910, 8-2958, 8-3064, 8-3111, 9-3746
- CartanName, 8-2818, 8-2833, 8-2863, 8-2909, 8-2958, 8-3016, 8-3064, 8-3110
- CartanSubalgebra, 8-3025
- CartesianPower, 1-215
- CartesianProduct, 1-215, 12-4938
- Cartier, 3-1179, 9-3695, 9-3882
- CartierRepresentation, 3-1179, 9-3696
- CasimirValue, 8-3150
- CasselsMap, 10-4058
- CasselsTatePairing, 10-4017
- cat, 1-66, 1-205, 1-223, 13-5110, 13-5190, 13-5220

- cat:=, 1-66, 1-205, 1-223
- Catalan, 2-483, 12-4799
- Category, 1-28, 1-176, 2-266, 2-268,
 - 2-285, 2-287, 2-335, 2-337, 2-354,
 - 2-357, 2-373, 2-377, 2-397, 2-415,
 - 2-417, 2-447, 2-479, 480, 3-657,
 - 3-757, 3-782, 3-793, 3-882, 3-903,
 - 3-1043, 3-1045, 3-1060, 3-1095,
 - 3-1128, 3-1140, 3-1154, 4-1228,
 - 4-1314, 4-1316, 4-1325, 1326, 7-2469,
 - 7-2762, 9-3405, 9-3411, 9-3428,
 - 9-3431, 10-3907, 10-3945, 10-3948,
 - 10-3951, 10-3961, 11-4480, 12-4846,
 - 4847
- CayleyGraph, 12-4939
- Ceiling, 2-290, 2-314, 2-359, 2-483
- Cell, 5-1628
- CellNumber, 5-1627
- CellSize, 5-1628
- Center, 2-266, 2-285, 2-335, 4-1228,
 - 5-1491, 5-1584, 5-1688, 5-1830,
 - 6-2056, 6-2274, 8-3024, 11-4367
- CenterDensity, 3-682
- CenterPolynomials, 8-3114
- CentralCharacter, 3-817, 3-821, 11-4649,
 - 11-4674
- CentralCollineationGroup, 12-4736
- CentralEndomorphisms, 3-731, 5-1780
- CentralExtension, 5-1853
- CentralExtensionProcess, 5-1853
- CentralExtensions, 5-1853
- CentralIdempotents, 7-2447
- Centraliser, 5-1488, 1489, 5-1506, 5-1551,
 - 5-1819, 6-2056, 6-2270, 2271, 7-2443,
 - 7-2445, 7-2552, 7-2555, 8-3024
- CentraliserOfInvolution, 5-1710, 1711
- CentralisingMatrix, 5-1716
- Centralizer, 5-1488, 1489, 5-1506, 5-1551,
 - 5-1668, 5-1819, 6-2056, 6-2270, 2271,
 - 7-2443, 7-2445, 7-2516, 7-2552,
 - 7-2555, 7-2575, 8-3024
- CentralizerGLZ, 5-1781, 5-1783
- CentralizerOfNormalSubgroup, 5-1551
- CentralOrder, 5-1654
- CentralValue, 10-4249
- Centre, 2-266, 2-373, 3-784, 3-887,
 - 3-1043, 4-1314, 5-1491, 5-1584,
 - 5-1688, 5-1830, 6-2056, 6-2274,
 - 7-2443, 7-2516, 7-2575, 7-2762,
 - 8-3024
- CentredAffinePatch, 9-3519
- CentreDensity, 3-682
- CentreOfEndomorphismAlgebra, 5-1780
- CentreOfEndomorphismRing, 3-731, 5-1780,
 - 7-2712
- CentrePolynomials, 8-3114
- CFP, 6-2303
- Chabauty, 10-4063, 4064, 10-4183
- Chabauty0, 10-4183
- ChainComplex, 12-4699
- ChainMap, 4-1448
- ChainmapToCohomology, 7-2614
- ChangeAlgebra, 7-2583
- ChangeAmbient, 12-4785
- ChangeBase, 5-1620
- ChangeBasis, 7-2432, 7-2442, 8-2978
- ChangeDerivation, 9-3415, 9-3437
- ChangeDifferential, 9-3416, 9-3437
- ChangeDirectory, 1-90
- ChangeField, 3-1216
- ChangeIdempotents, 7-2577
- ChangeModel, 3-1211
- ChangeOfBasisMatrix, 5-1699
- ChangeOrder, 9-3214, 3215, 9-3282
- ChangePrecision, 2-483, 3-951, 4-1277,
 - 4-1287, 4-1325, 4-1328, 4-1338,
 - 7-2789, 9-3410
- ChangeRepresentationType, 7-2551
- ChangeRing, 2-415, 2-448, 2-538, 2-570,
 - 3-660, 4-1325, 4-1348, 4-1384,
 - 4-1398, 5-1644, 7-2423, 7-2483,
 - 7-2516, 7-2690, 7-2730, 8-3015,
 - 8-3041, 8-3079, 8-3109, 9-3214,
 - 9-3281, 9-3324, 10-3936, 10-4097,
 - 10-4117, 11-4541
- ChangeSupport, 12-4920, 12-4999
- ChangeUniverse, 1-181, 1-204, 4-1398
- ChangGraphs, 12-4942
- Character, 3-1217
- CharacterDegrees, 5-1508, 5-1849, 7-2760,
 - 2761
- CharacterDegreesPGroup, 5-1849, 7-2761
- Characteristic, 2-266, 2-286, 2-335,
 - 2-356, 2-375, 2-416, 2-448, 2-479,
 - 3-788, 3-891, 3-1044, 3-1060,
 - 3-1099, 4-1228, 4-1276, 4-1315,
 - 4-1326, 7-2469
- CharacteristicPolynomial, 2-379, 2-546,
 - 3-798, 3-908, 3-1131, 1132, 4-1290,
 - 5-1654, 7-2458, 7-2520, 7-2631,
 - 11-4565, 12-4942, 13-5265
- CharacteristicPolynomialFromTraces,
 - 10-4087
- CharacteristicSeries, 5-1997
- CharacteristicVector, 2-588, 4-1399
- CharacterMultiset, 8-3160, 8-3164
- CharacterRing, 7-2757
- CharacterTable, 5-1508, 5-1606, 5-1698,
 - 5-1849, 6-2068, 7-2759
- CharacterTableConlon, 5-1849, 7-2760
- CharacterTableDS, 7-2759
- CharacterWithSchurIndex, 7-2769
- ChebyshevFirst, 2-436
- ChebyshevSecond, 2-436
- ChebyshevT, 2-436
- ChebyshevU, 2-436

- CheckCodimension, 9-3836
- CheckFunctionalEquation, 10-4256
- CheckPolynomial, 13-5075
- CheckWeilPolynomial, 10-4277
- ChernNumber, 9-3756
- ChevalleyBasis, 8-3019, 3020
- ChevalleyGroup, 5-1878
- ChevalleyGroupOrder, 5-1880
- ChevalleyOrderPolynomial, 5-1879
- chi, 2-345
- ChiefFactors, 5-1587, 5-1691
- ChiefSeries, 5-1587, 5-1691, 5-1831
- ChienChoyCode, 13-5102
- ChineseRemainderTheorem, 2-312, 2-332, 2-424, 3-944, 3-1141
- Cholesky, 3-700
- ChromaticIndex, 12-4959
- ChromaticNumber, 12-4959
- ChromaticPolynomial, 12-4959
- cInvariants, 10-3943, 10-4105
- Class, 5-1494, 5-1500, 5-1539, 5-1662, 5-1813
- ClassCentraliser, 5-1542, 5-1663
- Classes, 5-1495, 5-1539, 5-1662, 5-1813, 12-4981
- ClassField, 4-1307
- ClassFunctionSpace, 7-2757
- ClassGroup, 2-285, 2-355, 3-758, 3-801, 3-836, 3-912, 3-1121, 3-1170, 9-3710
- ClassGroupAbelianInvariants, 3-1121, 3-1170, 9-3711
- ClassGroupCyclicFactorGenerators, 3-914
- ClassGroupExactSequence, 3-1121, 3-1172
- ClassGroupGenerationBound, 3-1169
- ClassGroupGetUseMemory, 3-918
- ClassGroupPRank, 3-1123, 3-1173, 9-3711
- ClassGroupPrimeRepresentatives, 3-913
- ClassGroupSetUseMemory, 3-918
- ClassGroupStructure, 3-758
- ClassicalConstructiveRecognition, 5-1731
- ClassicalCovariantsOfCubicSurface, 9-3811
- ClassicalElementToWord, 5-1732
- ClassicalForms, 5-1897
- ClassicalIntersection, 7-2674
- ClassicalMaximals, 5-1919
- ClassicalModularPolynomial, 11-4287
- ClassicalPeriod, 11-4462
- ClassicalStandardGenerators, 5-1731
- ClassicalStandardPresentation (type, d, q : -), 5-1732
- ClassicalSyLOW, 5-1921
- ClassicalSyLOWConjugation, 5-1921
- ClassicalSyLOWNormaliser, 5-1921
- ClassicalSyLOWToPC, 5-1921
- ClassicalType, 5-1902
- ClassifyRationalSurface, 9-3785
- ClassInvariants, 5-1664
- ClassMap, 5-1494, 5-1542, 5-1662, 5-1813
- ClassNumber, 3-757, 3-802, 3-837, 3-913, 3-1122, 3-1171, 9-3710
- ClassNumberApproximation, 3-1169
- ClassNumberApproximationBound, 3-1170
- ClassPowerCharacter, 7-2765
- ClassRepresentative, 2-332, 3-945, 5-1496, 5-1541, 5-1663, 5-1813
- ClassRepresentativeFromInvariants, 5-1664
- ClassTwo, 5-1848
- CleanCompositionTree, 5-1739
- ClearDenominator, 11-4566
- ClearDenominators, 2-462
- ClearPrevious, 1-76
- ClearVerbose, 1-103
- ClebschGraph, 12-4942
- ClebschInvariants, 10-4125, 4126
- ClebschSalmonInvariants, 9-3810
- ClebschToIgusaClebsch, 10-4127
- CliffordIndexOne, 9-3726
- CliqueComplex, 12-4686
- CliqueNumber, 12-4962
- ClockCycles, 1-27
- ClosestVectors, 3-684
- ClosestVectorsMatrix, 3-684
- CloseVectors, 3-686
- CloseVectorsMatrix, 3-687
- CloseVectorsProcess, 3-691
- Closure, 8-3165
- ClosureGraph, 12-4940
- Cluster, 9-3492, 9-3507
- cmpeq, 1-12
- cmpne, 1-12
- CMPoints, 11-4374
- CMTwists, 11-4525
- CO, 5-1883
- CoblesRadicand, 9-3809
- CoboundaryMapImage, 5-2020
- CocycleMap, 5-2032
- CodeComplement, 13-5106, 13-5219
- CodeToString, 1-67
- Codifferent, 3-945, 3-1147
- Codimension, 9-3512, 9-3835
- Codomain, 1-252, 1-254, 2-604, 4-1414, 5-1528, 5-1647, 6-2100, 6-2331, 7-2589, 8-3126, 9-3313, 9-3536, 9-3607, 10-4140, 11-4566, 11-4577
- Coefficient, 2-418, 2-451, 4-1282, 4-1293, 4-1328, 4-1353, 7-2554, 9-3432, 11-4392, 12-4851
- CoefficientField, 2-599, 3-783, 3-883, 3-1016, 3-1095, 1096, 3-1183, 4-1273, 4-1397, 7-2762, 9-3354, 9-3496, 11-4649, 11-4665, 13-5203
- CoefficientHeight, 3-797, 3-907, 3-933, 3-1138
- CoefficientIdeals, 3-896, 3-935, 3-1100, 3-1147, 4-1436

- CoefficientLength, 3-797, 3-907, 3-933, 3-1138
- CoefficientMap, 9-3573
- CoefficientRing, 2-415, 2-447, 2-530, 2-563, 3-660, 3-783, 3-883, 3-974, 3-1016, 3-1059, 3-1095, 1096, 4-1273, 4-1325, 4-1339, 4-1348, 4-1364, 4-1397, 4-1424, 5-1645, 7-2422, 7-2452, 7-2469, 7-2485, 7-2510, 7-2551, 2552, 7-2568, 7-2687, 7-2715, 7-2789, 7-2791, 8-2980, 8-2989, 8-3014, 8-3041, 8-3064, 8-3079, 8-3107, 8-3109, 9-3287, 9-3307, 9-3354, 9-3428, 9-3496, 9-3649, 9-3876, 10-3945, 10-3948, 10-4130, 10-4145, 10-4195, 10-4201, 11-4397, 11-4649, 11-4665, 12-4846
- Coefficients, 2-418, 2-450, 4-1282, 4-1328, 4-1342, 4-1353, 7-2472, 7-2554, 8-3043, 8-3081, 9-3310, 9-3432, 10-3942
- CoefficientsAndMonomials, 2-452, 9-3310
- CoefficientsNonSpiral, 4-1355
- CoefficientSpace, 9-3573
- Coercion, 1-251
- Cofactor, 2-545
- Cofactors, 2-545
- CohenCoxeterName, 8-2955
- CohomologicalDimension, 5-1507, 5-1604, 5-2014, 2015, 7-2753
- CohomologicalDimensions, 5-2014, 7-2753
- Cohomology, 5-2032
- CohomologyClass, 5-2031
- CohomologyDimension, 9-3345, 9-3615
- CohomologyElementToChainMap, 7-2607
- CohomologyElementToCompactChainMap, 7-2607
- CohomologyGeneratorToChainMap, 7-2599
- CohomologyGroup, 5-2014
- CohomologyLeftModuleGenerators, 7-2599
- CohomologyModule, 3-1014, 5-2012, 2013
- CohomologyRightModuleGenerators, 7-2598
- CohomologyRing, 7-2608
- CohomologyRingGenerators, 7-2598
- CohomologyRingQuotient, 7-2614
- CohomologyToChainmap, 7-2614
- CoisogenyGroup, 8-2868, 8-2911, 8-2959, 8-3112
- Cokernel, 2-605, 4-1414, 4-1448, 7-2589, 9-3314, 9-3608, 11-4559, 11-4589
- Collect, 6-2233, 8-3151
- CollectRelations, 6-2231
- CollineationGroup, 12-4731
- CollineationGroupStabilizer, 12-4731
- CollineationSubgroup, 12-4731
- Colon, 7-2460
- ColonIdeal, 3-941, 3-1141, 9-3225, 9-3322
- ColonIdealEquivalent, 9-3225
- ColonModule, 9-3322
- Column, 9-3310, 12-4823
- ColumnLength, 12-4824
- Columns, 12-4823
- ColumnSkewLength, 12-4823
- ColumnSubmatrix, 2-532, 533, 2-567
- ColumnSubmatrixRange, 2-533, 2-567
- ColumnWeight, 2-564
- ColumnWeights, 2-564, 9-3307
- ColumnWord, 12-4825
- CombineIdealFactorisation, 9-3578
- CombineInvariants, 3-976
- COMinus, 5-1885
- CommonComplement, 2-627
- CommonEigenspaces, 7-2530
- CommonModularStructure, 11-4534
- CommonOverfield, 2-367
- CommonZeros, 3-1135, 9-3700
- Commutator, 8-3116
- CommutatorGraph, 8-2989
- CommutatorIdeal, 7-2444, 7-2644
- CommutatorModule, 7-2443
- CommutatorSubgroup, 5-1488, 5-1550, 5-1583, 5-1668, 5-1688, 5-1819, 5-1830, 6-2056, 6-2139, 6-2270
- comp, 2-275, 3-786, 3-889
- CompactInjectiveResolution, 7-2595
- CompactPart, 12-4781
- CompactPresentation, 5-1862
- CompactProjectiveResolution, 7-2591, 7-2606
- CompactProjectiveResolutionPGroup, 7-2606
- CompactProjectiveResolutionsOfSimpleModules, 7-2591
- CompanionMatrix, 2-433, 7-2508, 9-3444
- Complement, 2-601, 9-3573, 11-4442, 11-4598, 12-4873, 12-4935
- ComplementaryDivisor, 3-1168, 9-3709
- ComplementaryErrorFunction, 2-510
- ComplementBasis, 5-1823
- ComplementOfImage, 11-4598
- Complements, 5-1595, 5-1834, 7-2702
- Complete, 6-2105, 6-2326
- CompleteClassGroup, 3-918
- CompleteDigraph, 12-4923
- CompleteGraph, 12-4922
- CompleteKArc, 12-4726
- CompleteTheSquare, 10-4097
- CompleteUnion, 12-4938
- CompleteWeightEnumerator, 13-5093, 13-5185, 13-5215, 5216
- Completion, 2-275, 2-353, 3-786, 3-889, 3-1157, 4-1304, 9-3418, 9-3439, 9-3689
- Complex, 4-1441
- ComplexCartanMatrix, 8-2955
- ComplexConjugate, 2-289, 2-358, 2-484, 3-792, 3-841, 3-850, 3-900
- ComplexEmbeddings, 11-4410

- ComplexField, 2-477
- ComplexReflectionGroup, 8-2951, 2952
- ComplexRootDatum, 8-2957
- ComplexRootMatrices, 8-2954
- ComplexToPolar, 2-482
- ComplexValue, 11-4339, 11-4364
- Component, 1-216, 9-3744, 12-4948, 4949, 12-5026
- ComponentGroup, 9-3723
- ComponentGroupOfIntersection, 11-4587
- ComponentGroupOfKernel, 11-4556
- ComponentGroupOrder, 11-4466, 11-4638
- Components, 1-251, 3-1015, 9-3532, 12-4948, 12-5026
- ComposeTransformations, 10-4105
- Composite, 4-1272
- CompositeFields, 3-778, 3-864
- Composition, 3-755, 3-1115, 4-1330, 7-2771
- CompositionFactors, 5-1493, 5-1588, 5-1691, 5-1831, 7-2424, 7-2697, 8-3027
- CompositionSeries, 5-1583, 5-1831, 6-2065, 7-2424, 7-2697, 8-3027
- CompositionTree, 5-1736
- CompositionTreeCBM, 5-1738
- CompositionTreeElementToWord, 5-1738
- CompositionTreeFactorNumber, 5-1738
- CompositionTreeFastVerification, 5-1737
- CompositionTreeNiceGroup, 5-1737
- CompositionTreeNiceToUser, 5-1737
- CompositionTreeOrder, 5-1738
- CompositionTreeReductionInfo, 5-1738
- CompositionTreeSeries, 5-1738
- CompositionTreeSLPGroup, 5-1737
- CompositionTreeVerify, 5-1737
- Compositum, 3-778, 3-864
- ComputePrimeFactorisation, 9-3579
- ComputeReducedFactorisation, 9-3578
- Comultiplication, 8-3085
- ConcatenatedCode, 13-5110
- CondensationMatrices, 7-2538
- CondensedAlgebra, 7-2534
- ConditionalClassGroup, 3-802, 3-913
- ConditionedGroup, 5-1858
- Conductor, 2-344, 2-356, 3-755, 3-812, 3-820, 3-836, 3-850, 3-894, 3-1016, 3-1194, 1195, 3-1217, 7-2641, 10-3997, 10-4054, 10-4070, 10-4079, 10-4261, 11-4481, 11-4487, 11-4524, 11-4674
- ConductorRange, 10-4051
- Cone, 9-3864, 12-4695, 12-4771
- ConeIndices, 9-3865
- ConeInSublattice, 12-4772
- ConeIntersection, 9-3865
- ConeQuotientByLinearSubspace, 12-4772
- Cones, 9-3864
- ConesOfCodimension, 9-3864
- ConesOfMaximalDimension, 9-3865
- ConeToPolyhedron, 12-4773
- ConeWithInequalities, 12-4771
- ConformalHamiltonianLieAlgebra, 8-3004
- ConformalOrthogonalGroup, 5-1883
- ConformalOrthogonalGroupMinus, 5-1885
- ConformalOrthogonalGroupPlus, 5-1884
- ConformalSpecialLieAlgebra, 8-3003
- ConformalSymplecticGroup, 5-1882
- ConformalUnitaryGroup, 5-1881
- CongruenceGroup, 11-4411, 11-4459
- CongruenceGroupAnemic, 11-4412
- CongruenceImage, 5-1760
- CongruenceModulus, 11-4464, 11-4605
- CongruenceSubgroup, 11-4331
- Conic, 9-3647, 10-3906, 10-3920, 12-4726
- ConjecturalRegulator, 10-4045, 10-4069
- ConjecturalSha, 10-4069
- ConjugacyClasses, 5-1495, 5-1539, 5-1662, 5-1813, 7-2647, 8-2910
- Conjugate, 2-289, 2-358, 2-484, 3-755, 3-797, 3-841, 3-843, 3-851, 3-906, 5-1488, 5-1550, 5-1667, 5-1819, 6-2159, 6-2270, 7-2463, 7-2631, 7-2649, 12-4827
- ConjugateIntoBorel, 8-3117
- ConjugateIntoTorus, 8-3117
- ConjugatePartition, 12-4822
- Conjugates, 3-796, 3-906, 3-1047, 5-1494, 5-1500, 5-1539, 5-1662, 5-1813
- ConjugatesToPowerSums, 3-988
- ConjugateTranspose, 2-622
- ConjugationClassLength, 8-3169
- Connect, 9-3744
- ConnectedKernel, 11-4556
- ConnectingHomomorphism, 4-1452
- ConnectionNumber, 12-4882
- ConnectionPolynomial, 13-5265
- Consistency, 6-2231
- ConstaCyclicCode, 13-5099
- ConstantCoefficient, 2-418, 3-1202
- ConstantField, 3-1095, 9-3405
- ConstantFieldExtension, 3-1099, 9-3417, 9-3438
- ConstantMap, 9-3529
- ConstantRing, 9-3405, 9-3428
- ConstantWords, 13-5096
- Constituent, 1-236
- Constituents, 3-737, 7-2698
- ConstituentsWithMultiplicities, 7-2698
- Constraint, 13-5279
- Construction, 5-1972-1975
- ConstructionX, 13-5111
- ConstructionX3, 13-5111
- ConstructionX3u, 13-5111
- ConstructionXChain, 13-5111
- ConstructionXX, 13-5112
- ConstructionY1, 13-5114

- ConstructTable, 7-2551
- ContactLieAlgebra, 8-3005
- ContainsQuadrangle, 12-4725
- Content, 2-426, 2-462, 3-658, 3-843, 3-934, 3-941, 12-4809, 12-4812, 12-4825
- ContentAndPrimitivePart, 2-426, 2-462
- Contents, 4-1425
- Continuations, 4-1303
- ContinuedFraction, 2-490
- ContinueEnumeration, 6-2215
- Contpp, 2-426, 2-462
- Contract, 12-4936, 12-5017
- Contraction, 12-4873
- Contravariants, 10-4106
- ContravariantsOfCubicSurface, 9-3811
- ControlledNot, 13-5259
- Convergents, 2-490
- Converse, 12-4941, 12-5019
- ConvertFromManinSymbol, 11-4429
- ConvertToCWIFormat, 2-323
- Convolution, 4-1330
- ConwayPolynomial, 2-382
- Coordelt, 3-653
- Coordinate, 9-3489, 9-3658
- CoordinateLattice, 3-647
- CoordinateMatrix, 9-3198
- CoordinateRing, 3-660, 9-3487, 9-3497, 9-3644, 9-3649
- Coordinates, 2-602, 3-655, 656, 4-1403, 7-2435, 7-2522, 7-2552, 7-2631, 8-3034, 9-3198, 9-3310, 9-3489, 9-3644, 9-3658, 12-4723, 13-5078, 13-5193, 13-5207
- CoordinateSpace, 3-657
- CoordinatesToElement, 3-653
- CoordinateVector, 3-656
- cop, 1-235
- COPlus, 5-1884
- CoprimeBasis, 2-308, 3-943
- CoprimeBasisInsert, 3-944
- CoprimeRepresentative, 3-945
- CordaroWagnerCode, 13-5068
- Core, 5-1489, 5-1551, 5-1668, 5-1819, 6-2056, 6-2159, 6-2271
- CoreflectionGroup, 8-2929
- CoreflectionMatrices, 8-2839, 8-2879, 8-2924, 8-2966
- CoreflectionMatrix, 8-2839, 8-2879, 8-2924, 8-2966
- CorestrictCocycle, 5-2020
- CorestrictionMapImage, 5-2020
- Coroot, 8-2837, 8-2874, 8-2917, 8-2964, 8-3119
- CorootAction, 8-2929
- CorootGSet, 8-2928
- CorootHeight, 8-2841, 8-2882, 8-2921, 8-3122
- CorootLattice, 8-2872
- CorootNorm, 8-2842, 8-2882, 8-2921, 8-3122
- CorootNorms, 8-2841, 8-2882, 8-2921, 8-3122
- CorootPosition, 8-2837, 8-2874, 8-2917, 8-2964, 8-3119
- Coroots, 8-2837, 8-2874, 8-2917, 8-2963, 8-3119
- CorootSpace, 8-2836, 8-2872, 8-2916, 8-2963, 8-3119
- Correlation, 13-5180
- CorrelationGroup, 12-4756
- Cos, 2-494, 4-1334
- Cosec, 2-494, 495
- Cosech, 2-497
- CosetAction, 5-1477, 5-1488, 5-1580, 5-1686, 5-1835, 6-2178, 6-2226, 6-2268
- CosetDistanceDistribution, 13-5097
- CosetEnumerationProcess, 6-2209
- CosetGeometry, 12-4748, 12-4753
- CosetImage, 5-1477, 5-1488, 5-1580, 5-1686, 5-1835, 6-2178, 6-2179, 6-2226, 6-2269
- CosetKernel, 5-1477, 5-1488, 5-1580, 5-1686, 5-1835, 6-2179, 6-2226, 6-2269
- CosetLeaders, 13-5080
- CosetRepresentatives, 11-4333, 11-4342
- CosetSatisfying, 6-2177, 6-2216
- CosetSpace, 6-2171, 6-2227
- CosetsSatisfying, 6-2177, 6-2216
- CosetTable, 5-1487, 5-1599, 5-1693, 5-1835, 6-2168, 6-2217, 6-2267
- CosetTableToPermutationGroup, 6-2169
- CosetTableToRepresentation, 6-2169
- Cosh, 2-497, 4-1334
- Cot, 2-494
- Coth, 2-497
- Counit, 8-3085
- CountPGroups, 5-1948
- Covalence, 12-4879
- CoverAlgebra, 7-2576
- CoveringCovariants, 10-4106
- CoveringRadius, 3-697, 13-5097
- CoveringStructure, 1-29
- CoweightLattice, 8-2884, 8-2922, 8-2967, 8-3123
- CoxeterDiagram, 8-2819, 8-2833, 8-2863, 8-2909, 8-2958, 8-3111
- CoxeterElement, 8-2915, 8-2960, 8-3114
- CoxeterForm, 8-2839, 8-2879, 8-2919
- CoxeterGraph, 8-2805, 8-2814, 8-2833, 8-2863, 8-2910, 8-2958, 8-3111
- CoxeterGroup, 6-2092, 6-2094, 8-2822, 8-2846, 8-2897, 8-2902-2907, 8-2936, 8-2969

- CoxeterGroupFactoredOrder, 8-2804, 2805,
 8-2809, 8-2811, 8-2817
 CoxeterGroupOrder, 8-2804, 2805, 8-2809,
 8-2811, 8-2817, 8-2834, 8-2867
 CoxeterLength, 8-2914, 8-2967
 CoxeterMatrix, 8-2804, 8-2814, 8-2833,
 8-2863, 8-2910, 8-2958, 8-3111
 CoxeterNumber, 8-2915, 8-2960, 8-3111
 CoxMonomialLattice, 9-3872, 9-3878
 CoxRing, 9-3874, 9-3876
 Cputime, 1-26
 CreateCharacterFile, 2-320
 CreateCycleFile, 2-320
 CreateK3Data, 9-3847
 CremonaDatabase, 10-4050
 CremonaReference, 10-4052
 CriticalStrip, 11-4633
 CrossCorrelation, 13-5269
 CrossPolytope, 12-4770
 CRT, 2-312, 2-424, 3-944, 3-1141
 CryptographicCurve, 10-3979
 CrystalGraph, 8-3090
 CSp, 5-1882
 CSSCode, 13-5232
 CU, 5-1881
 CubicFromPoint, 10-4097
 CubicSurfaceByHexahedralCoefficients,
 9-3809
 CubicSurfaceFromClebschSalmon, 9-3810
 Cunningham, 2-305
 Current, 5-1944, 5-1963, 5-1968, 5-1982
 CurrentLabel, 5-1944, 5-1963, 5-1968,
 5-1982
 Curve, 9-3497, 9-3503, 3504, 9-3645,
 3646, 9-3657, 3658, 9-3678, 9-3688,
 9-3694, 9-3698, 9-3701-3703, 9-3831,
 10-3948, 10-3951, 10-3961, 10-4100,
 10-4145, 13-5143
 CurveDifferential, 9-3693
 CurveDivisor, 9-3693
 CurvePlace, 9-3693
 CurveQuotient, 9-3683
 Curves, 9-3833
 Cusp, 11-4314
 CuspForms, 11-4385
 CuspidalInducingDatum, 11-4675
 CuspidalProjection, 11-4399
 CuspidalSubgroup, 11-4626
 CuspidalSubspace, 11-4399, 11-4441,
 11-4483, 11-4494
 CuspIsSingular, 11-4314
 CuspPlaces, 11-4314
 Cusps, 11-4334, 11-4342
 CuspWidth, 11-4334
 CutVertices, 12-4949, 12-5026
 Cycle, 5-1567, 6-2311
 CycleCount, 2-320
 CycleDecomposition, 5-1567
 CycleStructure, 5-1535
 CyclicCode, 13-5069, 13-5098, 13-5163
 CyclicGroup, 5-1473, 5-1530, 5-1792,
 6-2095, 6-2261
 CyclicPolytope, 12-4771
 CyclicSubgroups, 5-1499, 5-1560, 5-1824
 CyclicToRadical, 3-985
 CyclotomicAutomorphismGroup, 3-850
 CyclotomicData, 10-4220
 CyclotomicFactors, 13-5163
 CyclotomicField, 3-847
 CyclotomicOrder, 3-850
 CyclotomicPolynomial, 3-848
 CyclotomicRelativeField, 3-850
 CyclotomicUnramifiedExtension, 4-1268
 Cylinder, 12-4696
 D), 10-4151
 Darstellungsgruppe, 6-2096
 Data, 5-1946
 DawsonIntegral, 2-509
 Decimation, 13-5269
 Decode, 13-5127
 DecodingAttack, 13-5116
 DecomposeAutomorphism, 8-3127
 DecomposeCharacter, 8-3149
 DecomposeUsing, 11-4592
 DecomposeVector, 2-601
 Decomposition, 2-332, 2-345, 2-355,
 3-807, 808, 3-812, 3-911, 3-942,
 3-953, 3-1145, 3-1151, 3-1218,
 7-2702, 7-2771, 9-3705, 11-4437,
 11-4483, 11-4494, 11-4590
 DecompositionField, 3-964, 3-1016
 DecompositionGroup, 3-810, 3-956, 3-963,
 3-1016, 1017, 4-1368
 DecompositionMatrix, 7-2752
 DecompositionMultiset, 8-3160, 8-3164
 DecompositionType, 3-942, 3-1017, 3-1145,
 3-1151, 3-1196
 DecompositionTypeFrequency, 3-1017
 Decycle, 6-2312
 DedekindEta, 2-502
 DedekindTest, 2-427
 DeepHoles, 3-697
 DefinesAbelianSubvariety, 11-4519
 DefinesHomomorphism, 6-2105
 DefiningConstantField, 3-1095
 DefiningEquations, 9-3536, 10-4099
 DefiningIdeal, 9-3497, 9-3649, 10-3907
 DefiningMap, 4-1273
 DefiningMatrix, 12-4792
 DefiningModularSymbolsSpace, 11-4674
 DefiningMonomial, 9-3885
 DefiningPoints, 4-1236
 DefiningPolynomial, 2-356, 2-375, 3-789,
 3-893, 3-1100, 3-1218, 4-1273,
 4-1340, 4-1364, 4-1381, 9-3497,

- 9-3649, 10-3907, 10-3946, 10-4133, 10-4195
- DefiningPolynomials, 3-1100, 9-3497, 9-3536, 10-4220
- DefiningSubschemePolynomial, 10-3948
- DefiniteGramMatrix, 11-4358
- DefiniteNorm, 11-4358
- DefRing, 8-3107
- DegeneracyMap, 11-4435
- DegeneracyMatrix, 11-4436
- DegeneracyOperator, 11-4652
- Degree, 2-332, 2-356, 2-375, 2-419, 2-455, 2-599, 3-658, 3-788, 3-810, 3-827, 3-891, 3-933, 3-955, 3-990, 3-1016, 3-1043, 3-1062, 3-1099, 3-1134, 3-1150, 3-1155, 3-1162, 3-1183, 3-1196, 3-1202, 3-1217, 4-1273, 4-1315, 4-1329, 4-1364, 4-1424, 4-1448, 5-1524, 5-1535, 5-1565, 5-1627, 5-1645, 5-1652, 7-2435, 7-2452, 7-2510, 7-2765, 8-3034, 8-3043, 8-3081, 9-3186, 9-3307, 9-3310, 9-3315, 9-3433, 9-3507, 9-3512, 9-3571, 9-3581, 9-3607, 9-3649, 9-3674, 9-3689, 9-3702, 9-3706, 9-3748, 9-3831, 9-3835, 10-3957, 10-4099, 10-4124, 10-4220, 11-4397, 11-4481, 11-4496, 11-4566, 11-4612, 12-4851, 12-4945, 12-4947, 12-5023, 5024
- Degree6DelPezzoType2_1, 9-3800
- Degree6DelPezzoType2_2, 9-3800
- Degree6DelPezzoType2_3, 9-3800
- Degree6DelPezzoType3, 9-3800
- Degree6DelPezzoType4, 9-3800
- Degree6DelPezzoType6, 9-3800
- DegreeMap, 11-4596
- DegreeOfExactConstantField, 3-1101, 3-1195
- DegreeOfFieldExtension, 5-1716
- DegreeOnePrimeIdeals, 3-912
- DegreeRange, 12-4904
- DegreeReduction, 5-1581
- Degrees, 4-1442, 12-4904
- DegreeSequence, 12-4946, 12-4948, 12-5023, 12-5025
- DegreesOfCohomologyGenerators, 7-2599
- Delaunay, 10-4210
- delete, 1-10, 1-243, 5-1939
- DeleteCapacities, 12-5007
- DeleteCapacity, 12-5007
- DeleteData, 5-1599
- DeleteEdgeLabels, 12-5007
- DeleteGenerator, 6-2204, 6-2394
- DeleteHeckePrecomputation, 11-4653
- DeleteLabel, 12-5003, 12-5007
- DeleteLabels, 12-5004, 12-5007
- DeleteRelation, 6-2204, 2205, 6-2394
- DeleteVertexLabels, 12-5004
- DeleteWeight, 12-5007
- DeleteWeights, 12-5007
- DelPezzoSurface, 9-3795, 3796
- DelsarteGoethalsCode, 13-5169
- Delta, 2-503, 504
- DeltaPreimage (G, g), 5-1908
- Demazure, 8-3155
- Denominator, 2-285, 2-357, 3-794, 3-904, 3-932, 3-1062, 3-1133, 3-1146, 3-1163, 7-2460, 9-3294, 9-3501, 9-3706, 11-4566
- Density, 2-530, 2-563, 3-682
- DensityEvolutionBinarySymmetric, 13-5153
- DensityEvolutionGaussian, 13-5155
- Depth, 2-589, 4-1402, 5-1859, 6-2251, 9-3376
- DepthFirstSearchTree, 12-4958, 12-5030
- Derivation, 9-3407, 9-3428
- Derivative, 2-422, 2-457, 458, 3-974, 3-1063, 4-1293, 4-1329, 4-1357, 9-3415
- DerivedGroup, 5-1491, 5-1583, 5-1688, 5-1830, 6-2139, 6-2275
- DerivedGroupMonteCarlo (G : -), 5-1712
- DerivedLength, 5-1491, 5-1583, 5-1688, 5-1831, 6-2274
- DerivedSeries, 5-1491, 5-1583, 5-1688, 5-1831, 6-2275, 8-3028
- DerivedSubgroup, 5-1491, 5-1583, 5-1688, 5-1830, 6-2056, 6-2139, 6-2275
- DerksonIdeal, 9-3385, 9-3391
- Descendants, 5-1846
- DescentInformation, 10-4002, 10-4055
- DescentMaps, 10-4058
- Design, 12-4738, 12-4868, 12-4889
- Detach, 1-47
- DetachSpec, 1-49
- Determinant, 2-544, 2-574, 3-658, 3-702, 3-704, 3-1216, 4-1425, 5-1654, 7-2519, 9-3746
- Development, 12-4877
- DFSTree, 12-4958, 12-5030
- DiagonalAutomorphism, 8-3036, 8-3126
- DiagonalForm, 2-460
- Diagonalisation, 7-2531
- Diagonalization, 3-699, 7-2531
- DiagonalJoin, 2-538, 2-570, 7-2524, 2525
- DiagonalMatrix, 2-525, 7-2508, 8-3008
- DiagonalModel, 10-4097
- DiagonalSparseMatrix, 2-562
- DiagonalSum, 12-4827
- Diagram, 12-4759
- DiagramAutomorphism, 8-3036, 8-3087, 8-3126
- Diameter, 2-485, 12-4955, 13-5097
- DiameterPath, 12-4955
- DickmanRho, 2-293
- DicksonFirst, 2-386, 2-437

- DicksonInvariant, 2-624
- DicksonNearfield, 2-395
- DicksonPairs, 2-393
- DicksonSecond, 2-386, 2-437
- DicksonTriples, 2-393
- DicyclicGroup, 5-1473
- diff, 1-185
- Difference, 9-3493
- DifferenceSet, 12-4876
- Different, 3-894, 3-911, 3-945, 3-1104, 3-1138, 3-1147
- DifferentDivisor, 3-1158
- Differential, 3-1174, 9-3407, 9-3415, 9-3428, 9-3694
- DifferentialBasis, 3-1168, 3-1175, 9-3694, 9-3712
- DifferentialFieldExtension, 9-3419
- DifferentialIdeal, 9-3424
- DifferentialLaurentSeriesRing, 9-3403
- DifferentialOperator, 9-3452
- DifferentialOperatorRing, 9-3426
- DifferentialRing, 9-3402
- DifferentialRingExtension, 9-3419
- DifferentialSpace, 3-1097, 3-1169, 3-1174, 1175, 9-3693, 3694, 9-3712
- Differentiation, 3-1137
- DifferentiationSequence, 3-1137
- Digraph, 12-4918
- DihedralForms, 11-4405
- DihedralGroup, 5-1473, 5-1530, 5-1792, 6-2095, 6-2261
- DihedralSubspace, 11-4399
- Dilog, 2-492
- Dimension, 2-599, 2-602, 3-658, 3-704, 3-1163, 4-1425, 4-1436, 5-2013, 7-2422, 7-2452, 7-2486, 7-2522, 7-2568, 7-2583, 7-2706, 7-2716, 7-2789, 8-2834, 8-2865, 8-2910, 8-2990, 8-3014, 8-3064, 8-3110, 9-3242, 9-3282, 9-3290, 9-3512, 9-3571, 9-3712, 9-3829, 9-3832, 9-3835, 10-4145, 10-4201, 11-4397, 11-4481, 11-4496, 11-4521, 11-4546, 11-4578, 11-4649, 11-4665, 12-4686, 12-4785, 12-4787, 13-5071, 13-5204, 13-5252
- DimensionByFormula, 11-4397
- DimensionCuspForms, 11-4471
- DimensionCuspFormsGamma0, 11-4471
- DimensionCuspFormsGamma1, 11-4471
- DimensionNewCuspFormsGamma0, 11-4471
- DimensionNewCuspFormsGamma1, 11-4471
- DimensionOfCentreOfEndomorphismRing, 3-731, 5-1780
- DimensionOfEndomorphismRing, 3-731, 5-1780
- DimensionOfExactConstantField, 3-1101
- DimensionOfFieldOfGeometricIrreducibility, 9-3690
- DimensionOfGlobalSections, 9-3615
- DimensionOfHomology, 4-1443
- DimensionOfKernelZ2, 13-5179
- DimensionOfSpanZ2, 13-5179
- DimensionsEstimate, 8-2996
- DimensionsOfHomology, 4-1443
- DimensionsOfInjectiveModules, 7-2569
- DimensionsOfProjectiveModules, 7-2569
- DimensionsOfTerms, 4-1443
- DirectProduct, 5-1475, 5-1532, 5-1648, 5-1802, 6-2096, 6-2260, 6-2393, 8-2927, 8-3124, 9-3486, 9-3643, 11-4584, 13-5106, 13-5190, 13-5219
- DirectSum, 3-664, 4-1398, 4-1443, 6-2056, 7-2433, 7-2513, 7-2515, 7-2690, 7-2716, 7-2735, 7-2789, 8-2844, 8-2887, 8-3023, 8-3161, 8-3164, 9-3321, 9-3605, 11-4584, 12-4787, 13-5106, 13-5189, 13-5219, 13-5245
- DirectSumDecomposition, 7-2447, 7-2702, 8-2845, 8-2888, 8-3023, 8-3161, 8-3164
- DirichletCharacter, 3-815, 3-1219, 11-4398, 11-4521, 11-4649
- DirichletCharacterOverNF, 3-818
- DirichletCharacterOverQ, 3-818
- DirichletCharacters, 11-4397, 11-4522
- DirichletGroup, 2-342, 3-811
- DirichletRestriction, 3-814
- Disconnect, 9-3744
- Discriminant, 2-356, 2-432, 2-467, 2-623, 3-754, 3-788, 3-836, 3-843, 3-892, 3-1015, 3-1101, 4-1274, 4-1366, 7-2452, 7-2633, 7-2640, 10-3911, 10-3943, 10-4106, 10-4124, 11-4481, 11-4487, 11-4577
- DiscriminantDivisor, 3-1195
- DiscriminantFromShiodaInvariants, 10-4129
- DiscriminantOfHeckeAlgebra, 11-4449
- DiscriminantRange, 3-827
- DiscToPlane, 11-4367
- Display, 6-2232
- DisplayBurnsideMatrix, 5-1825
- DisplayCompTreeNodees, 5-1737
- DisplayFareySymbolDomain, 11-4345
- DisplayPolygons, 11-4343
- Distance, 2-485, 4-1300, 11-4340, 11-4366, 12-4955, 12-5034, 13-5077, 13-5193, 13-5207
- DistanceMatrix, 12-4957
- DistancePartition, 12-4956
- Distances, 12-5034
- DistinctDegreeFactorization, 2-432
- DistinctExtensions, 5-2024
- DistinguishedOrbitsOnSimples, 8-2865
- div, 2-287, 2-337, 2-417, 2-423, 2-449, 2-459, 3-654, 3-808, 3-903, 3-940, 3-954, 3-1130, 3-1154, 3-1159,

- 4-1228, 4-1283, 4-1293, 4-1316,
4-1326, 4-1342, 7-2457, 7-2471,
9-3309, 9-3412, 9-3701, 9-3706
- div:=, 2-287, 2-449, 4-1284, 7-2471
- DivideOutIntegers, 11-4552
- DivisionPoints, 10-3962
- DivisionPolynomial, 10-3946
- Divisor, 3-808, 3-953, 954, 3-1134,
3-1147, 3-1157, 3-1177, 9-3576,
9-3695, 9-3703-3705, 9-3880, 13-5143
- DivisorClassGroup, 9-3879
- DivisorClassLattice, 9-3872, 9-3879
- DivisorGroup, 3-807, 3-952, 3-1097,
3-1153, 3-1157, 9-3576, 9-3702, 3703,
9-3880
- DivisorIdeal, 7-2485, 9-3287
- DivisorMap, 9-3608, 9-3713
- DivisorOfDegreeOne, 3-1158, 9-3692
- Divisors, 2-308, 2-311, 3-911, 3-942
- DivisorSigma, 2-293
- DivisorToSheaf, 9-3609
- Dodecacode, 13-5232
- Domain, 1-252, 1-254, 2-604, 3-812,
4-1381, 4-1414, 5-1528, 5-1646,
6-2100, 6-2331, 7-2589, 8-3126,
9-3313, 9-3536, 9-3607, 10-4140,
11-4566, 11-4577
- DominantCharacter, 8-3150
- DominantDiagonalForm, 3-726
- DominantLSPath, 8-3088
- DominantWeight, 8-2885, 8-2922, 8-2968,
8-3123
- DotProduct, 2-611
- DotProductMatrix, 2-611
- Double, 10-4197
- DoubleCoset, 5-1598, 6-2176
- DoubleCosetRepresentatives, 5-1598
- DoubleCosets, 6-2176
- DoubleGenusOneModel, 10-4103
- DoublePlotkinSum, 13-5178
- DoublyCirculantQRCode, 13-5103
- DoublyCirculantQRCodeGF4, 13-5104
- Dual, 3-662, 4-1429, 4-1442, 6-2070,
7-2594, 7-2736, 8-2845, 8-2889,
8-2927, 8-2962, 8-3124, 9-3605,
11-4599, 12-4717, 12-4772, 12-4786,
12-4873, 13-5073, 13-5083, 13-5188,
13-5205, 13-5210
- DualAtkinLehner, 11-4446
- DualBasisLattice, 3-663
- DualCoxeterForm, 8-2839, 8-2879, 8-2919
- DualEuclideanWeightDistribution, 13-5184
- DualFaceInDualFan, 9-3866
- DualFan, 9-3862
- DualHeckeOperator, 11-4445
- DualIsogeny, 10-3956
- DualityAutomorphism, 8-3127
- DualKroneckerZ4, 13-5178
- DualLeeWeightDistribution, 13-5183
- DualMorphism, 8-2893
- DualQuotient, 3-663
- DualStarInvolution, 11-4446
- DualVectorSpace, 11-4434
- DualWeightDistribution, 13-5092, 13-5182,
13-5215
- DuvalPuisseuxExpansion, 4-1249
- DynkinDiagram, 8-2819, 8-2833, 8-2863,
8-2909, 8-2958, 8-3110
- DynkinDigraph, 8-2811, 8-2815, 8-2833,
8-2863, 8-2910, 8-2958, 8-3111
- E, 2-478
- e, 2-478
- E . i, 12-5001
- E2NForm, 11-4316
- E4Form, 11-4316
- E6Form, 11-4316
- Ealpha, 8-3088, 3089
- EARNs, 5-1590
- EasyBasis, 9-3197
- EasyIdeal, 9-3197
- EchelonForm, 2-548, 7-2525
- EcheloniseWord, 6-2233
- ECM, 2-307
- ECMFactoredOrder, 2-308
- ECMOrder, 2-308
- ECMSteps, 2-307
- EdgeCapacities, 12-5007
- EdgeConnectivity, 12-4953, 12-5028
- EdgeDeterminant, 9-3748
- EdgeGroup, 12-4972
- EdgeIndices, 12-4778, 12-5000
- EdgeLabels, 9-3748, 12-5007
- EdgeMultiplicity, 12-5000
- Edges, 12-4777, 12-4926, 12-5000
- EdgeSeparator, 12-4953, 12-5028
- EdgeSet, 12-4926
- EdgeUnion, 12-4938, 12-5018
- EdgeWeights, 12-5007
- EFAModuleMaps, 6-2279
- EFAModules, 6-2280
- EFASeries, 6-2275
- EffectiveSubcanonicalCurves, 9-3838
- EhrhartCoefficient, 12-4779
- EhrhartCoefficients, 12-4779
- EhrhartPolynomial, 12-4779
- EhrhartSeries, 12-4779
- EichlerInvariant, 7-2642
- Eigenform, 11-4416, 11-4451, 11-4656
- Eigenforms, 11-4656
- Eigenspace, 2-547, 7-2521, 9-3829
- Eigenvalues, 2-547, 7-2521
- EightDescent, 10-4022
- Eisenstein, 2-499, 500, 3-761
- EisensteinData, 11-4403
- EisensteinProjection, 11-4399
- EisensteinSeries, 11-4403

- EisensteinSubspace, 11-4399, 11-4441, 11-4483, 11-4494
- EisensteinTwo, 10-4012
- Element, 2-397, 11-4616
- ElementaryAbelianGroup, 6-2261
- ElementaryAbelianNormalSubgroup, 5-1597
- ElementaryAbelianQuotient, 5-1562, 5-1674, 5-1829, 6-2060, 6-2123, 6-2278
- ElementaryAbelianSeries, 5-1594, 5-1690, 5-1831
- ElementaryAbelianSeriesCanonical, 5-1594, 5-1690, 5-1832
- ElementaryAbelianSubgroups, 5-1499, 5-1560, 5-1824
- ElementaryDivisors, 2-552, 2-575, 4-1429, 7-2526
- ElementaryPhiModule, 7-2789
- ElementarySymmetricPolynomial, 9-3262, 9-3394
- ElementaryToHomogeneousMatrix, 12-4862
- ElementaryToMonomialMatrix, 12-4861
- ElementaryToPowerSumMatrix, 12-4862
- ElementaryToSchurMatrix, 12-4861
- Elements, 2-342, 6-2325, 11-4619, 12-4751
- ElementSequence, 5-1853
- ElementSet, 5-1537
- ElementToSequence, 1-67, 2-310, 2-344, 2-360, 2-372, 2-397, 2-418, 2-530, 2-563, 2-589, 3-655, 3-756, 3-800, 3-910, 3-950, 3-1129, 4-1282, 4-1313, 4-1328, 4-1342, 4-1400, 4-1435, 5-1522, 5-1642, 5-1806, 6-2051, 6-2081, 6-2250, 6-2303, 6-2350, 6-2368, 6-2395, 6-2409, 7-2435, 7-2457, 7-2523, 7-2554, 7-2631, 7-2691, 8-3034, 10-3942, 10-3961, 10-4135, 10-4154, 10-4197, 12-4723, 12-4809
- ElementType, 1-29
- EliasAsymptoticBound, 13-5120
- EliasBound, 13-5118
- Eliminate, 6-2184, 6-2206, 6-2395
- EliminateGenerators, 6-2184
- EliminateRedundancy, 6-2232
- Elimination, 9-3530
- EliminationIdeal, 9-3234
- EllipticCurve, 9-3670, 10-3932-3934, 10-4051, 10-4223, 11-4416, 11-4469, 11-4640
- EllipticCurveDatabase, 10-4050
- EllipticCurveFromjInvariant, 10-3932
- EllipticCurveFromPeriods, 10-4042
- EllipticCurves, 10-4053
- EllipticCurveSearch, 10-4069, 10-4080
- EllipticCurveWithGoodReductionSearch, 10-4069
- EllipticCurveWithjInvariant, 10-3932
- EllipticExponential, 10-4043
- EllipticInvariants, 11-4361, 11-4641
- EllipticLogarithm, 10-4043
- EllipticPeriods, 11-4641
- EllipticPoints, 11-4334
- elt, 1-216, 2-282, 283, 2-336, 2-354, 2-370, 371, 2-413, 2-447, 2-478, 2-587, 3-653, 3-754, 3-780, 3-875, 876, 3-1059, 3-1127, 1128, 4-1279, 1280, 4-1324, 4-1350, 4-1399, 5-1462, 5-1521, 5-1641, 7-2432, 7-2469, 7-2507, 7-2547, 7-2690, 7-2757, 8-3008, 8-3065, 8-3113, 10-4133, 10-4150, 13-5076, 13-5192, 13-5206
- elt< >, 10-3959
- Eltlist, 8-3114
- Eltseq, 1-67, 1-200, 2-285, 2-310, 2-360, 2-372, 2-418, 2-530, 2-563, 2-589, 3-655, 3-756, 3-800, 3-910, 3-950, 3-1129, 3-1197, 3-1203, 4-1282, 4-1313, 4-1328, 4-1342, 4-1370, 4-1400, 4-1435, 5-1522, 5-1642, 5-1806, 6-2051, 6-2081, 6-2250, 6-2303, 6-2350, 6-2368, 6-2395, 6-2409, 7-2435, 7-2457, 7-2523, 7-2554, 7-2631, 7-2691, 8-3034, 9-3309, 9-3413, 9-3432, 10-3942, 10-3961, 10-4100, 10-4135, 10-4154, 10-4197, 11-4336, 11-4398, 11-4480, 11-4496, 11-4560, 11-4616, 12-4723, 12-4809
- EltTup, 8-3066
- Embed, 2-368, 3-784, 3-887, 3-1097, 7-2463, 7-2636, 2637
- Embedding, 12-4966, 12-5031
- EmbeddingMap, 3-784, 3-887, 3-990
- EmbeddingMatrix, 7-2640
- Embeddings, 11-4539
- EmbeddingSpace, 4-1425
- EmbedPlaneCurveInP3, 9-3562
- EModule, 9-3305, 3306
- EmptyBasket, 9-3833
- EmptyDigraph, 12-4923
- EmptyGraph, 12-4922
- EmptyPolyhedron, 12-4773
- EmptyScheme, 9-3493
- EmptySubscheme, 9-3493
- End, 11-4570
- EndomorphismAlgebra, 4-1409, 5-1780, 7-2712
- EndomorphismRing, 3-730, 5-1780, 7-2712, 10-4205
- Endomorphisms, 3-730, 5-1780
- EndpointWeight, 8-3089
- EndVertices, 4-1239, 12-4929, 12-5001
- Enumerate, 7-2463, 7-2652
- EnumerationCost, 3-694

EnumerationCostArray, 3-694