HANDBOOK OF MAGMA FUNCTIONS

Volume 1

Language and Data Structures

John Cannon Wieb Bosma
Claus Fieker Allan Steel

Editors

Version 2.22

Sydney

June 9, 2016

MAGMA
COMPUTER ALGEBRA

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 Roozemond, 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 as 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 V2.22 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 Roozemond, 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, May 2016

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.

Steve Donnelly, Ph.D. (Athens, Ga) [2005-]: Research interests are in arithmetic geometry, particularly elliptic curves and modular forms. Contributions include: many routines for elliptic curves over Q and number fields, including descent methods, Cassels-Tate pairings and integral points; Hilbert modular forms and fast algorithms for definite quaternion algebras; also developed a new implementation of the general class group algorithm. Currently continuing to work on class groups, elliptic curves and surfaces.

Andreas-Stephan Elsenhans, Ph.D. (Göttingen) [2012-]: Main research interests are in the areas of arithmetic and algebraic geometry, particularly cubic and K3 surfaces. Main contributions focus on cubic surfaces from the arithmetic and algebraic points of view. Currently working on the 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 including Kedlaya's algorithm and the modular parameter method of Mestre. Currently working on machinery for general surfaces and cohomology for projective varieties.

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, Ph.D. (Sydney), [1999-]: Works in the areas of arithmetic fields 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.

Brendan Creutz, [2011-2013]: 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.

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 (Magma)).

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.

David Howden, Ph.D. (Warwick) [2012-2014]: Primary research interests are in computational group theory. Main contributions focus on computing automorphism groups and isomorphism testing for soluble groups.

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 Roozemond, [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**.

A package for working with divisors on varieties has been developed by **Martin Bright** (American University of Beirut), **Gavin Brown** (Loughborough), **Mike Harrison** (Magma) and **Andrew Wilson** (Edinburgh). The functionality includes decomposition into irreducible components, Riemann-Roch spaces, canonical divisors and (surface) intersection numbers.

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** (Loughborough) and **Alexander Kasprzyk** (Imperial College), developed the toric geometry and polyhedra packages.

Functions for computing Shioda invariants for genus 3 hyperelliptic curves, reconstructing models for a curve from such invariants and computing geometric automorphism groups have been contributed by **Reynald Lercier** (DGA, Rennes) and **Christophe Ritzenthaler** (Luminy).

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** (Trento).

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.

A function that finds the intersection multiplicities for all intersection points of two plane curves was adapted into Magma from code provided by **Chris Smyth** (Edinburgh).

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

Arithmetic Geometry Over Characteristic 0 Fields

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 Q having conductor up to 300,000 constructed by **John Cremona** (Warwick) is also included.

John Cremona (Warwick) has contributed his code implementing the Cremona-Prickett-Siksek height bounds.

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.

Code for determining isomorphism of cubic surfaces has been contributed by **Andreas-Stephan Elsenhans** (Bayreuth).

A collection of tools that calculate information about the Picard rank of a surface has been developed by **Andreas-Stephan Elsenhans** (Bayreuth).

Code for calculating the invariants, covariants and contravariants of a cubic surface has been developed by **Andreas-Stephan Elsenhans** (Bayreuth).

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 3-descent on elliptic curves was mainly written by **Tom Fisher** (Cambridge). An earlier version as well as part of the current version were developed by **Michael Stoll** (Bremen).

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

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.

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

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).

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.

Arithmetic Geometry Over Finite Fields

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

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 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.

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).

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

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).

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.) provides an efficient means for constructing presentations for matrix algebras.

For matrix algebras defined over a finite field, **Jon Carlson** (Athens, Ga.) designed and implemented algorithms for the Jacobson radical and unit group which are faster than the Brooksbank-O'Brien algorithms for larger examples.

A substantial package for working with substructures and homomorphisms of basic algebras, developed by **Jon Carlson** (Athens, Ga.), was released as part of V2.19. Among other things, the package can compute the automorphism group of a basic algebra and test pairs of basic algebras for isomorphism.

Markus Kirschmer (Aachen) has written a number of optimized routines for definite quaternion algebras over number fields.

Markus Kirschmer has also contributed a package for quaternion algebras defined over the function fields $F_q[t]$, for q odd. The package includes calculation of the normaliser of an order and an efficient algorithm for computing the two-sided ideal classes of an order in a definite quaternion algebra (over $\mathbf{Z}$ or $\mathbf{F}_q[t]$).

Quaternion algebras over the rational field Q were originally implemented by **David Kohel** (Singapore-NUS, 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

A package for constructing linear codes associated with lattice points in a convex polytope has been contributed by **Gavin Brown** (Loughborough) and **Al Kasprzyk** (Imperial).

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

The development of machinery for linear codes benefited greatly from the active involvement of **Markus Grassl** (Karlsruhe) over a long period. Of particular note is his contribution to the development of improved algorithms for computing the minimum weight and for the enumeration of codewords.

Routines implementing many different constructions for linear codes over finite fields were contributed by **Markus Grassl** (Karlsruhe).

Markus Grassl (Karlsruhe) played a key role in the design of Magma packages for Additive Codes and Quantum Error-Correcting Codes. The packages were implemented by Greg White (Magma).

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**.

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 intrinsics for constructing and computing properties of Z_4 codes has been contributed by **Jaume Pernas**, **Jaume Pujol** and **Merč 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 Symmetrica 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 `regexp` intrinsic function comes from the V8 regexp 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.

Galois Groups

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 arbitary high degree.

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

Galois Representations

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

Code for constructing Artin representations of the Galois group of the absolute extension of a number field was developed by **Tim Dokchitser** (Cambridge).

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

Geometry

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

Algorithms for testing whether two convex polytopes embedded in a lattice are isomorphic or equivalent have been implemented by **Al Kasprzyk** (Imperial College). Of particular note is Al's implementation of the PALP normal form algorithm.

Global Arithmetic Fields

Jean-Francois Biasse (Calgary) implemented a quadratic sieve for computing the class group of a quadratic field. He also developed a generalisation of the sieve for number fields having 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–NUS, Magma) has contributed to the machinery for binary quadratic forms and has implemented rings of Witt vectors.

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.

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

See also the subsection *Group Theory: Soluble Groups.*

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.

Versions of Magma from V2.8 onwards employ the Advanced Coset Enumerator designed by **George Havas** (UQ) and implemented by **Colin Ramsay** (UQ). 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.

An improved version of the Plesken-Fabianska algorithm for finding L2-quotients of a finitely presented group has been developed and implemented by **Sebastian Jambor** (Aachen).

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 Bäärnhielm** and **Charles Leedham-Green** has been coded with their assistance.

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.

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** (QMUL).

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

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

Lifting-style algorithms have been developed by **Derek Holt** (Warwick) for computing structural information in groups given in terms of the Composition Tree data structure. The operations include centralisers, conjugacy classes, normalizers, subgroup conjugacy and maximal subgroups.

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 on code written by **Dimitri Leemans** (Brussels).

Csaba Schneider (Lisbon) has implemented code which allows the user to write an arbitrary element of a classical group as an SLP in terms of its standard generators.

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.

An algorithm for determining the conjugacy of any pair of matrices in $GL(2, Z)$ was developed and implemented by **D. Husert** (University of Paderborn). In particular, this allows the conjugacy of elements having infinite order to be determined.

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.

A much improved algorithm for computing the normaliser or centraliser of a finite subgroup of $GL(n, Z)$ has been implemented by **Markus Kirschmer** (Aachen). Markus has also implemented an algorithm that tests finite subgroups for conjugacy.

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 an algorithm of Babai et al which identifies a quasisimple group of Lie type.

Magma incorporates a database of the maximal finite rational subgroups of $GL(n, \mathbf{Q})$ up to dimension 31. This database as constructed by **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 $\mathrm{GL}(n, q)$.

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

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: Soluble Groups

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

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

A new approach to the more efficient calculation of the automorphism group of a finite soluble group has been developed and implemented **David Howden** (Warwick). A slight variation of the algorithm is used to test isomorphism.

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).

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.

Derek Holt (Warwick) has constructed a table of irreducible representations of quasisimple groups (up to degree 100). Some representations were contributed by **Allan Steel, Volker Gebhardt** and **Bill Unger** (all Magma).

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).

Homological Algebra

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

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.

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.

In 2011, **Derek Holt** (Warwick) implemented an alternative algorithm for finding the dimension of the cohomology group $H^n(G, K)$, for G a finite group, and K a finite field. In this approach the dimension is found using projective covers and dimension shifting.

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.

Gael Collinet (Strasbourg) has contributed the basis of the package for lattices over number fields.

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). Subsequently, Bob supplied code for working with directed W-graphs.

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 Murray** (Sydney).

The current version of Lie groups in MAGMA has been implemented by **Scott 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 Roozemond** (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) created by **R. Clint Whaley et al.** (UTSA) are used for some fundamental matrix algorithms over finite fields GF(p), where p is about the size of a machine integer.

Local Arithmetic Fields

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.

Modular Forms

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 (V2.19) 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** (Magma).

David Kohel (Singapore-NUS, 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.

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

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 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) produced the package for Bianchi modular forms.

Primality and Factorisation

The factorisation of integers of the form $p^n \pm 1$, for small primes p, makes use of tables compiled by **Richard Brent** that extend tables developed by the Cunningham project. In addition MAGMA uses Richard's 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

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).

Derek Holt (Warwick) has made a number of important contributions to the design of the module theory algorithms employed in MAGMA.

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).

In 2011, **Derek Holt** (Warwick) and **John Cannon** (Magma) developed a package for computing the projective indecomposable KG-modules for a finite group G.

The algorithms used in Magma for finding the lattice of submodules and the endomorphism ring of a KG-module (K a finite field) were developed by **Charles Leedham-Green** (QMW, London) and **Allan Steel** (Magma).

Topology

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

Nathan Dunfield (Cornell) and **William Thurston** (Cornell) 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 V2.22 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 over Number Fields • *M. Watkins*
32 Lattices With Group Action • *B. Souvignier* ◦ *M. Kirschmer*
33 Quadratic Forms • *S. Donnelly*
34 Binary Quadratic Forms • *D. Kohel*
35 Number Fields • *C. Fieker* ◦ *W. Bosma, N. Sutherland*
36 Quadratic Fields • *W. Bosma*
37 Cyclotomic Fields • *W. Bosma, C. Fieker*
38 Number Fields and Orders • *C. Fieker* ◦ *W. Bosma, N. Sutherland*

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

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

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

153 Incidence Structures and Designs • *J. Cannon*
154 Hadamard Matrices • *G. Bailey*
155 Graphs • *J. Cannon, P. Lieby* ○ *G. Bailey*
156 Multigraphs • *J. Cannon, P. Lieby*
157 Networks • *P. Lieby*
158 Linear Codes over Finite Fields • *J. Cannon, A. Steel* ○ *G. White*
159 Algebraic-geometric Codes • *J. Cannon, G. White*
160 Low Density Parity Check Codes • *G. White*
161 Linear Codes over Finite Rings • *A. Steel* ○ *G. White*
162 Linear Codes over the Integer Residue Ring $\mathbf{Z}_4$ • *A. Steel* ○ *G. White, M. Villanueva*
163 Additive Codes • *G. White*
164 Quantum Codes • *G. White*
165 Pseudo-random Bit Sequences • *S. Contini*
166 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   *Default :* `true`

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

VOLUME 2: OVERVIEW

VOLUME 3: OVERVIEW

VOLUME 4: OVERVIEW

VOLUME 5: OVERVIEW

VOLUME 6: OVERVIEW

VOLUME 7: OVERVIEW

VOLUME 8: OVERVIEW

VOLUME 9: OVERVIEW

VOLUME 10: OVERVIEW

VOLUME 11: OVERVIEW

VOLUME 12: OVERVIEW

VOLUME 13: OVERVIEW

VOLUME 1: CONTENTS

VOLUME 2: CONTENTS

VOLUME 3: CONTENTS

VOLUME 4: CONTENTS

VOLUME 5: CONTENTS

VOLUME 6: CONTENTS

VOLUME 7: CONTENTS

VOLUME 8: CONTENTS

VOLUME 9: CONTENTS

VOLUME 10: CONTENTS

VOLUME 11: CONTENTS

VOLUME 12: CONTENTS

VOLUME 13: CONTENTS

PART I
THE MAGMA LANGUAGE

1 STATEMENTS AND EXPRESSIONS

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 := `*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 `+:=`).

$\mathtt{x}_1$`, `$\mathtt{x}_2$`, ..., `$\mathtt{x}_n$` := `*expression*`;`

Assignment of $n \geq 1$ values, returned by the expression on the right hand side. Here the $\mathtt{x}_i$ are identifiers, and the right hand side expression must return $m \geq n$ values; the first n of these will be assigned to $\mathtt{x}_1$, $\mathtt{x}_2$, ..., $\mathtt{x}_n$ respectively.

`_ := `*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

x[*expression*$_1$][*expression*$_2$]...[*expression*$_n$] := *expression*;

x[*expression*$_1$, *expression*$_2$, ..., *expression*$_n$] := *expression*;

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 `expr`$_i$. 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<x`$_1$`, x`$_2$`, ...x`$_n$`> :=` *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`$_2$`, ..., x`$_n$. Naming of generators usually has two aspects; firstly, the *strings* `x`$_1$`, x`$_2$`, ...x`$_n$ are used for printing of the generators, and secondly, to the *identifiers* `x`$_1$`, x`$_2$`, ...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:

`E :=` *expression*;
`x`$_1$` := E.1; x`$_2$` := E.2; ... x`$_n$` := E.`n;

`E<[x]> :=` *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(~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(~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

$$x = y \qquad \Longleftrightarrow \qquad (x \wedge y) \vee (\neg x \wedge \neg y).$$

```
> 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 ! 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
   statements_1
else
   statements_2
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 expression_1 then
   statements_1
elif Boolean expression_2 then
   statements_2
else
   statements_3
end if;
```

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

```
if Boolean expression then
   statements_1
elif Boolean expression_2 then
   statements_2
else
   statements_3
end if;
```

is equivalent to

```
if Boolean expression_1 then
    statements_1
else
    if Boolean expression_2 then
```

```
        statements_2
    else
        statements_3
    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` *expression*$_1$ `else` *expression*$_2$

This is an expression, of which the value is that of *expression*$_1$ or *expression*$_2$, 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 |
      expression_left,1 : expression_right,1 ,
         ⋮
      expression_left,n : expression_right,n ,
      default : expression_def >
```

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

`assert` *Boolean expression*;

`assert2` *Boolean expression*;

`assert3` *Boolean expression*;

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 be 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

```
try
    statements_1
catch e
    statements_2
end try;
```

The `try/catch` statement lets users handle raised errors. The semantics of a `try/catch` statement are as follows: the block of statements *statements*$_1$ is executed. If no error is raised during its execution, then the block of statements *statements*$_2$ is not executed; if an error is raised at any point in *statements*$_1$, execution *immediately* transfers to *statements*$_2$ (the remainder of *statements*$_1$ is not executed). When transfer is controlled to the `catch` block, the variable named *e* is initialized to the error that was raised by *statements*$_1$; 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 := expression_1 to expression_2 by expression_3 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 $k = 0, 1, 2, \ldots$ in succession, as long as $b + k \cdot s \le e$ (for $e > 0$) or $b + k \cdot s \ge e$ (for $e < 0$).

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

```
for i := expression_1 to expression_2 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 x_11, ..., x_1n_1 in S_1, ..., x_m1, ..., x_mn_m in S_m 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 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.

`SetShowRealTime(v)`

Sets whether or not to print real time as well as CPU time when a `time` or `vtime` statement is executed. By default this flag is false, meaning that only the CPU time is printed.

`GetShowRealTime()`

Returns the current value of the flag that determines if real time is additionally printed by the `time` and `vtime` statements.

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 **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 **Z** is `RngIntElt` since that is the type of the integers which lie in **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 `-S`n 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 over 10^{2857} times as long as the previously touted longest-period one. Preprint, 2000.

2 FUNCTIONS, PROCEDURES AND PACKAGES

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 $x_1, \ldots, x_n$) 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(m`$_1$`, ..., m`$_n$`)`, where $m_1, \ldots, m_n$ 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(y`$_1$`, ..., y`$_m$`)`, where $y_1, \ldots, y_m$ 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 $[*y_n, \ldots, y_m*]$.

```
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 $x_1, \ldots, x_n$). 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, Al := "Simple")
>    return <x, y, Proof, Al>;
> end function;
>
> f(1, 2);
<1, 2, true, Simple>
> f(1, 2: Proof := false);
<1, 2, false, Simple>
> f(1, 2: Al := "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(x1, ..., xn)`, 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 := proc< x`$_1$`, ..., x`$_n$`:` *parameters* `|` *expression*`>;`

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 $x_1, \ldots, x_n$). Optional parameters are permitted as in the main procedure constructor.

`p := proc< x`$_1$`, ..., x`$_n$`, ...:` *parameters* `|` *expression*`>;`

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 $a^{n-1} \equiv 1 \bmod n$, 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 *type*
{*type*}	Sets over *type*
{`[`*type*`]`}	Sets or sequences over *type*
{@*type*@}	Indexed sets over *type*
{**type**}	Multisets over *type*

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 $\langle p_1, \ldots, p_r \rangle$. Let the tuple of the types of these parameters be $\langle t_1, \ldots, t_r \rangle$, 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 $p \in \{p_1, \ldots, p_r\}$, Magma finds the set $S_i \subseteq S$ of participating intrinsics which are the best matches for that parameter. More specifically, an intrinsic $s = \langle u_1, \ldots, u_r \rangle$ is included in S_i if and only if t_i is a u_i, and no participating intrinsic $s' = \langle v_1, \ldots, v_r \rangle$ 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. The bounds L and U may be any expressions each yielding an integer value. If v is not in the range $[L, \ldots, U]$, 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. The bound 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`:` $F_1, \ldots, F_n$`;`

Given a category C, and a comma-separated list of identifiers $F_1, \ldots, F_n$ 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` $T : P_1, \ldots, P_n$`;`

Declare the given type name T (without quotes) to be a user-defined type, and also declare T to inherit from the user types $P_1, \ldots, P_n$ (which must be declared separately). As a result, `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` $T[E] : P_1, \ldots, P_n$`;`

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 $P_1, \ldots, P_n$.

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 Intrinsics 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).

```
intrinsic SubConstructor(X::T, t::.) -> T
{Return the substructure of X generated by elements of the tuple t}
    // This corresponds to the constructor call sub<X | r1, r2, ..., rn>
    // t is ALWAYS a tuple of the form <r1, r2, ..., rn>
    // Code: do tests on the elements in t to see whether valid and then
    //       set S to the substructure of T generated by r1, r2, ..., rn
    // Use standard require statements for error checking
    // Possibly use "t := Flat(t);" to make it easy to loop over t if
    //     any of the ri are sequences
    return S;
end intrinsic;
```

Assuming that objects of type T (user-defined) have elements, this must be a user-provided package function, which takes an object X of type T (user-defined) and a tuple t. The user call `sub<X | r1, r2, ..., rn>` (where X has type T) will cause this intrinsic to be called with X and the tuple $t = \langle$`r1,...,rn`$\rangle$. The function should create the substructure S of X generated by `r1,...,rn` and return S alone (the inclusion map from X to S is automatically handled by Magma via coercion).

```
intrinsic Hash(X::T) -> RngIntElt
{Return a hash value for the object x (should be between 0 and 2^31-1)}
   // Code: determine a hash value for the given object
   // NOTE: Objects X and Y of type T for which X eq Y is true
   //       MUST have the same hash value
   return hash;
end intrinsic;
```

Providing this intrinsic can greatly speed the checking of equality of objects of type T, and in particular if you wish to work with sets of reasonable cardinality (more than 1000 elements) it should be made available. The requirement is that if X and Y are equal, then their hashes should be the same, regardless of their internal representation.

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

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\ran\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 $(0, 1, \ldots, 9)$, 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 $(0, 1, \ldots, 9)$, 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 $[2 \ldots 36]$); 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.

`s notin t`

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.

`s lt t`

Returns `true` if *s* is lexicographically less than *t*, `false` otherwise. Here the ordering on characters imposed by their ASCII code number is used.

`s le t`

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.

`s gt t`

Returns `true` if *s* is lexicographically greater than *t*, `false` otherwise. Here the ordering on characters imposed by their ASCII code number is used.

`s ge t`

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?                              mmmmmmmmmmmm !
```

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 `its` 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 reimplementation 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

`print` *expression*;

`print` *expression*, ..., *expression*;

`print` *expression*: *parameters*;

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 ]
```

```
[100]
[133]
> for w := 1 to 5 do printf "[%*o]", w, 1; end for;
[1][ 1][  1][   1][    1]
```

Example H3E5

Some further uses of the `printf` statement are illustrated below.

```
> x := 3;
> y := 4;
> printf "x = %o, y = %o\n", x, y;
x = 3, y = 4
> printf "G'"; printf "day";
G'day
> p := 53.211;
> x := 123.2;
> printf "%.3o%% of %.2o is %.3o\n", p, x, p/100.0 * x;
53.211% of 123.20 is 65.556
> Zx<x> := PolynomialRing(Integers());
> printf "%O\n", x, "Magma";
Polynomial(\[0, 1])
```

`fprintf` *file*, *format*, *expression*, ..., *expression*;

Print values of the expressions under control of *format* into the file given by *file*. The first argument *file* must be either a string specifying a file which can be opened for appending (tilde expansion is performed on the filename), or a file object (see the section below on external files) opened for writing. The rest of the arguments are exactly as in the `printf` statement. In the string (filename) case, the file is opened for appending, the string obtained from the formatted printing of the other arguments is appended to the file, and the file is closed. In the file object case, the string obtained from the formatted printing of the other arguments is simply appended to the file. Note that this statement, like `printf`, does *not* print a newline character after its arguments (a `\n` character should be placed in the format string if this is desired).

Example H3E6

The following statements demonstrate a (rather contrived) use of `fprintf` with a file pipe.

```
> p := 1000000000000000000000000000057;
> F := POpen("sort -n", "w");
> for i := 100 to 110 do
>    fprintf F, "%30o (2^%o mod p)\n", 2^i mod p, i;
> end for;
> // Close F and then see output on standard output:
```

```
> 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.

`vprint` *flag*: *expression*, ..., *expression*;

`vprint` *flag*, n: *expression*, ..., *expression*;

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.

`vprintf` *flag*: *format*, *expression*, ..., *expression*;

`vprintf` *flag*, n: *format*, *expression*, ..., *expression*;

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(*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.

`IndentPush(C)`

Increases the indentation level by C.

`IndentPop()`

Decrease (pop) the indentation level by 1. Thus the beginning of a line will have s fewer spaces than before, where s is the current number of indentation spaces. If the current level is already 0, an error occurs.

`IndentPop(C)`

Decreases the indent level by C.

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 End of File Marker

The I/O types below all need some way of indicating when a read request fails due to no more data being available. This is achieved by returning a special "end of file" (shortened to "EOF") string that is not equal to any normal string.

`Eof()`

Creates the special EOF string.

`IsEof(S)`

Given a string S, return whether S is the special EOF string.

`AtEof(I)`

Given an I/O object I, returns whether all data is known to have been read from I (and thus that further reads will return the special EOF string). Note that if this function returns `false` then it may still be the case that the next read returns EOF; typically `AtEof` only returns `true` when a previous read has already returned EOF.

3.5 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.

3.5.1 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.)

3.5.2 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.

`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
```

3.5.3 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 ]
```

3.6 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.

3.6.1 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 then 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
```

3.6.2 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(P : parameters)`

`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 fewer 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(P : parameters)`

`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 $0 \ldots 255$). If the parameter `Max` is set to a positive value then at most that many bytes will be read. Note that fewer 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.

`ReadBytes(P, n)`

Keeps reading from P, waiting for data as necessary, until either n bytes have been read or an end of file condition is encountered. The data read is returned as a sequence of bytes (integers in the range $0 \ldots 255$). Note that fewer than n bytes may be returned if the end of file condition is encountered.

`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 $0 \ldots 255$.

3.7 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 65 536. 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)*.

3.7.1 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 1 024 are usually reserved for system use, and may require special privileges to be used as the local port number.

`Socket( : parameters)`

`LocalHost`	MonStgElt	*Default : none*
`LocalPort`	RngIntElt	*Default :* 0

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 1 024 are usually reserved for system use, and may require special privileges to be used as the local port number.

`WaitForConnection(S)`

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.

3.7.2 Socket Properties

`SocketInformation(S)`

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.

3.7.3 Socket Predicates

`IsServerSocket(S)`

Returns whether S is a server socket or not.

3.7.4 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 becomes 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 fewer 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 $0 \ldots 255$). If the parameter `Max` is set to a positive value then at most that many bytes will be read. Note that fewer 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.

`ReadBytes(S, n)`

Keeps reading from S, waiting for data as necessary, until either n bytes have been read or an end of file condition is encountered. The data read is returned as a sequence of bytes (integers in the range $0 \ldots 255$). Note that fewer than n bytes may be returned if the end of file condition is encountered.

`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 $0 \ldots 255$.

`WaitForIO(S : parameters)`

`TimeLimit`	RngIntElt	*Default :* ∞

Given a sequence S of I/O objects, returns the sequence of those elements of S which are ready for I/O. If no elements of S are ready (and S is not empty) then this function will wait until one does become ready, or until the specified time limit has elapsed, whichever comes first. Note that in the case of server sockets, "ready for I/O" means that a connection attempt has been made and a call to `WaitForConnection` will return without delay.

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
```

3.8 Interactive Input

`read` *identifier*;

`read` *identifier*, *prompt*;

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.

3.9 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.

3.10 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.

3.11 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)`

Send input from external files to standard output if b is `true`. If b is `false` then input from external files will not appear in standard output.

3.12 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`).

3.13 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.

3.14 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

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, underscores, 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 internal variables which control system or global features. 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 compactions 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.

`SetGPU(b)`

`GetGPU()`

Set the NVIDIA GPU mode to b; this determines whether Magma should use NVIDIA GPUs via CUDA when present. This is only relevant to a CUDA-enabled executable (typically downloaded as `magma.cuda.exe`) and is `true` by default in that case (so the GPU is used by default); for a non-CUDA-enabled executable, the procedure has no effect. Currently, a GPU is exploited in matrix multiplication over $\mathbf{F}_2$ and small prime finite fields and consequently anything which depends on such multiplication, such as the dense F_4 Gröbner basis algorithm over such fields.

`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 V$x.y$–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	*Default :* `true`
`Search`	MonStgElt	*Default :* "*Both*"
`ShowSrc`	BoolElt	*Default :* `false`

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	*Default :* `true`
`Search`	MonStgElt	*Default :* "*Both*"
`ShowSrc`	BoolElt	*Default :* `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.

`%p`n

List the history line n in %p format.

`%p`n_1 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.

`%P`n

List the history line n in %P format.

`%P`n_1 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 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>-V`*char*

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

M*key* 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).

`T`*char*

Move back to just after the first occurrence of *char*.

`t`*char*

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”).

`c`*range*

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”).

`d`*range*

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").

$\boxed{\texttt{s}}$

Delete the current character and change to insert mode ("substitute").

$\boxed{\texttt{U}}$
$\boxed{\texttt{u}}$

Undo the last change.

$\boxed{\texttt{X}}$

Delete the character to the left of the cursor.

$\boxed{\texttt{x}}$

Delete the character under the cursor.

$\boxed{\texttt{Y}}$

"Yank" the whole line - i.e., copy the whole line into the yank-buffer ("Yank").

$\boxed{\texttt{y}\textit{range}}$

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 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 we 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

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 so on.

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 is 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,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,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

So far 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 $\sim$, while b and y are called *actual* and *formal reference arguments* respectively because they are prefixed by a $\sim$.

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 $\sim$). 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+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 `[ ..., ('+',A), ('-',S), ... ]` 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 `[ ..., ('+',S), ('-',S), ... ]`, 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,FUNC( n : n+1)) ]`. After the second line the current context is `[ (f,FUNC( n : n+1)), (g,FUNC(m : m + 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, FUNC(n : n+1)), (g, FUNC(m : m + FUNC(n : n+1)(m))) ]`. After typing in the second definition of f the current context is `[ (f, FUNC(n : n+2)), (g, FUNC(m : m + 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.

` ``	*left*
(	*left*
[	*left*
assigned	*right*
~	*non*
#	*non*
&* &+ &and &cat &join &meet &or	*non-associative*
$ $$	*non*
.	*left*
@ @@	*left*
! !!	*right*
^	*right*
unary-	*right*
cat	*left*
* / div mod	*left*
+ -	*left*
meet	*left*
sdiff	*left*
diff	*left*
join	*left*
adj in notadj notin notsubset subset	*non*
cmpeq cmpne eq ge gt le lt ne	*left*
not	*right*
and	*left*
or xor	*left*
^^	*non*
? else select	*right*
->	*left*
=	*left*
:= is where	*left*

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

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	*Default :* `false`
`Max`	RngIntElt	*Default :* All

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 set, then the report only displays the first `Max` entries.

`ProfilePrintByTotalTime(G)`

`Percentage`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :* All

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	*Default :* `false`
`Max`	RngIntElt	*Default :* All

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 set, then the report only displays the first `Max` entries.

`ProfilePrintChildrenByTime(G, n)`

`Percentage`	BoolElt	*Default :* `false`
`Max`	RngIntElt	*Default :* All

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(<FldReElt> T) -> FldReElt                 0.130   14880
7     +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt      0.020   14880
9     gt(<FldReElt> x, <FldReElt> 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() -> FldReElt                             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;
1.01999999999999795718
> Label(E![2,2])`Time;
0.51000000000000000888
> Label(V!2)`Time;
1.01999999999999795718
```

As can 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

Chapter 7
DEBUGGING MAGMA CODE

7.1 Introduction

In ordered 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 i this trace gives a single *frame*, 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)` *expr* Evaluate the expression *expr* 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.

`identifiers (id)` Print a list of the assigned identifiers in the current context (the context is set by the `frame` command). The `identifiers` command is equivalent to invoking the `ShowIdentifiers` intrinsic 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.

Example H7E1

We now give a sample session in the debugger. In the following, we have written a function to evaluate $f(n) = \Sigma_{i=1}^{n} 1/n$, but have in our implementation we have accidentally included the evaluation of the term at $n = 0$.

```
> function f(n)
>   if n ge 0 then
>      return 1.0 / n + f(n - 1);
>   else
>      return 1.0 / n;
>   end if;
> end function;
>
> SetDebugOnError(true);
> f(3);
f(
    n: 3
)
f(
    n: 2
)
f(
    n: 1
)
f(
    n: 0
)
>>      return 1.0 / n + f(n - 1);
                   ^
Runtime error in '/': Division by zero
debug> p n
0
debug> p 1.0 / (n + 1)
1.00000000000000000000000000000
debug> bt
#0 *f(
    n: 0
) at <main>:1
#1  f(
    n: 1
) at <main>:1
#2  f(
    n: 2
) at <main>:1
#3  f(
    n: 3
) at <main>:1
debug> f 1
debug> p n
1
debug> p 1.0 / n
1.000000000000000000000000000000
```

PART II
SETS, SEQUENCES, AND MAPPINGS

8 INTRODUCTION TO AGGREGATES

Chapter 8
INTRODUCTION TO AGGREGATES

8.1 Introduction

This part of the Handbook comprises seven 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 | }` 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 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.

Perhaps the most common example of universes that are *not* compatible would be a prime finite field with the rationals, as not *every* rational can be coerced into the finite field, while MAGMA does not allow coercion from finite fields into the rationals in any event.

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 $U = \{1, 2, \ldots, 100\}$ of S and the universe 'ring of integers' of T. By rule (3) above, this overstructure of $U = \{1, 2, \ldots, 100\}$ 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 $\{1, 2, \ldots, 100\}$), 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 $i_1, i_2, \ldots, i_r$ 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 $i_1, i_2, \ldots, i_r$ 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 $i_j > \#S[i_1, \ldots, i_{j-1}]$, (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 $S = [\ [1,2],\ [2,3,\mathrm{undef},\mathrm{undef},7]\ ]$)

```
> S[4] := [7];
```

(and the result will be $S = [\ [1,2],\ [2,3],\ \mathrm{undef},\ [7]\ ]$). 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

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 ^^ operator to specify a multiplicity: the expression `x^^n` means the object x with multiplicity n. In the following, whenever any multiset constructor or function expects an element y, the expression x^^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, $x_1, \ldots, x_k$;

e an expression that also usually involves the variable(s) x, $x_1, \ldots, x_k$.

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):

```
{! x in F | P(x) !}
```

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 `{! x in F !}`. 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 | }`

The empty set with universe U.

`{ e1, e2, ..., en }`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the set $\{ a_1, a_2, ..., a_n \}$ 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 | e1, e2, ..., en }`

Given a list of expressions $e_1, \ldots, e_n$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the set $\{a_1, a_2, ..., a_n \}$ of elements of U.

`{ e(x) : x in E | 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 in E }`.

`{ U | e(x) : x in E | 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 |).

{ e(x$_1$,...,x$_k$) : x$_1$ in E$_1$, ..., x$_k$ in E$_k$ | P(x$_1$, ..., x$_k$) }

The set consisting of those elements $e(x_1, \ldots, x_k)$, in some common structure, for which $x_1, \ldots, x_k$ in $E_1, \ldots, E_k$ have the property that $P(x_1, \ldots, x_k)$ 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 $P(x_1, ..., x_k)$ is always true, it may be omitted (including the |).

{ U | e(x$_1$,...,x$_k$) : x$_1$ in E$_1$, ..., x$_k$ in E$_k$ | P(x$_1$, ..., x$_k$) }

As in the previous entry, the set consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true, is formed, as a set of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ 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.

`{@ @}`

The null set: an empty indexed set that does not have its universe defined.

`{@ U | @}`

The empty indexed set with universe U.

`{@ e`$_1$`, e`$_2$`, ..., 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 indexed set $Q = \{\ a_1, a_2, ..., a_n\ \}$ of elements of U.

`{@ U | e`$_1$`, e`$_2$`, ..., 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 indexed set $Q = \{a_1, a_2, ..., a_n\ \}$ of elements of U.

`{@ e(x) :  x in E | P(x) @}`

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 $|$).

`{@ U | e(x) :  x in E | P(x) @}`

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 $|$).

`{@ e(x`$_1$`,...,x`$_k$`) : x`$_1$` in E`$_1$`, ..., x`$_k$` in E`$_k$` | P(x`$_1$`, ..., x`$_k$`) @}`

The indexed set consisting of those elements $e(x_1, \ldots, x_k)$ (in some common structure), for which $x_1, \ldots, x_k$ in $E_1 \times \ldots \times E_k$ have the property that $P(x_1, \ldots, x_k)$ 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 $P(x_1, ..., x_k)$ is always true, it may be omitted.

`{@ U | e(x`$_1$`,...,x`$_k$`) : x`$_1$` in E`$_1$`, ..., x`$_k$` in E`$_k$` | P(x`$_1$`, ..., x`$_k$`)@}`

As in the previous entry, the indexed set consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as an indexed set of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ 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 ^^ (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 | *}`

The empty multiset with universe U.

`{* e_1, e_2, ..., e_n *}`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the multiset $Q = \{*\ a_1, a_2, ..., a_n\ *\}$ of elements of U.

`{* U | e_1, e_2, ..., e_m *}`

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the multiset $Q = \{*\ a_1, a_2, ..., a_n\ *\}$ of elements of U.

`{* e(x) : x in E | 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 | e(x) : x in E | 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,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) *}`

The multiset consisting of those elements $e(x_1, \ldots, x_k)$ (in some common structure), for which $x_1, \ldots, x_k$ in $E_1 \times \ldots \times E_k$ have the property that $P(x_1, \ldots, x_k)$ 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 $P(x_1, ..., x_k)$ is always true, it may be omitted.

`{* U | e(x_1,...,x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)*}`

As in the previous entry, the multiset consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as a multiset of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ 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.

`{ i..j }`

`{ U | i..j }`

The enumerated set whose elements form the arithmetic progression $i, i+1, i+2, \ldots, j$, 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 .. j by k }
```

```
{ U | i .. j by k }
```

The enumerated set consisting of the integers forming the arithmetic progression $i, i + k, i + 2 * k, \ldots, j$, 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 $\{S[i_1], \ldots, S[i_r]\}$ 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 in E | 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 in E | 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.

The expression P may be omitted if it is always true.

```
random{e(x1, ..., xk) :  x1 in E1, ..., xk in Ek | P(x1, ..., xk)}
```

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, return the value of the expression $e(y_1, \cdots, y_k)$ for a randomly chosen element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, for which $P(y_1, \ldots, y_k)$ is true.

The expression 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(~R, ~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 in E | 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$_1$, ..., x$_k$) : x$_1$ in E$_1$, ..., x$_k$ in E$_k$ | P(x$_1$, ..., x$_k$) }

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, return the value of the expression $e(y_1, \cdots, y_k)$ for the first element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, for which $P(y_1, \ldots, y_k)$ is true. An error occurs if no element of $E_1 \times \cdots \times E_k$ satisfies P.

The expression 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 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(S)`

`Min(S)`

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(S)`

`Max(S)`

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(x)`

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(~S, x)`

`Include(S, x)`

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(~S, x)`

`Exclude(S, x)`

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.

`ChangeUniverse(~S, V)`

`ChangeUniverse(S, V)`

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.

`CanChangeUniverse(S, V)`

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 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 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 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 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 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 $t_1, \ldots, t_r$ 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.

The expression P may be omitted if it is always true.

```
exists(t){e(x1, ..., xk): x1 in E1, ..., xk in Ek | P(x1, ..., xk)}
```

```
exists(t1, ..., tr){ e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P }
```

Given enumerated structures $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, the Boolean value `true` is returned if there is an element $< y_1, \ldots, y_k >$ in the Cartesian product $E_1 \times \cdots \times E_k$, such that $P(y_1, \ldots, y_k)$ is true. If $P(x_1, \ldots, x_k)$ is not true for any element $(y_1, \ldots, y_k)$ of $E_1 \times \cdots \times E_k$, then the Boolean value `false` is returned.

Moreover, if $P(x_1, \ldots, x_k)$ is found to be true for the element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \cdots, y_k)$. If $P(x_1, \ldots, x_k)$ is never true for an element of $E_1 \times \cdots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y_1, \cdots, y_k)$ 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 $P(x_1, \ldots, x_k)$ is never true. The clause `(t)` may be omitted entirely.

The expression 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 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(t){ e(x) :  x in E | P(x) }
```

```
forall(t_1, ..., t_r){ 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 $P(x)$ 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 $t_1, \ldots, t_r$ 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.

The expression P may be omitted if it is always true.

```
forall(t){e(x_1, ..., x_k):  x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k)}
```

```
forall(t_1, ..., t_r){ e(x_1, ..., x_k) :  x_1 in E_1, ..., x_k in E_k | P }
```

Given sets $E_1, \ldots, E_k$, and a Boolean expression $P(x_1, \ldots, x_k)$, the Boolean value `true` is returned if $P(x_1, \ldots, x_k)$ is true for every element $(x_1, \ldots, x_k)$ in the Cartesian product $E_1 \times \cdots \times E_k$.

If $P(x_1, \ldots, x_k)$ fails to be true for some element $(y_1, \ldots, y_k)$ of $E_1 \times \cdots \times E_k$, then the Boolean value `false` is returned.

Moreover, if $P(x_1, \ldots, x_k)$ is false for the element $< y_1, \ldots, y_k >$ of $E_1 \times \cdots \times E_k$, then in the first form of the exists expression, the variable t will be assigned the value of the expression $e(y_1, \cdots, y_k)$. If $P(x_1, \ldots, x_k)$ is true for every element of $E_1 \times \cdots \times E_k$, then the variable t will be left unassigned. In the second form, where r variables $t_1, \ldots, t_r$ are given, the result $e(y_1, \cdots, y_k)$ 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 $P(x_1, \ldots, x_k)$ is never true. The clause `(t)` may be omitted entirely.

The expression 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 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 $S = \{ a_1, a_2, \ldots, a_n\}$ of elements belonging to an algebraic structure U, and an (associative) operator $\circ : U \times U \to U$, form the element $a_{i_1} \circ a_{i_2} \circ a_{i_3} \circ \ldots \circ a_{i_n}$, for some permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.

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:

	empty	*null*
&+	U ! 0	error
&*	U ! 1	error
&and	`true`	`true`
&or	`false`	`false`
&join	*empty*	*null*
&meet	error	error

Warning: since the reduction may take place in an arbitrary order on the arguments $a_1, \ldots, a_n$, 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

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 an enumerated sequence must be less than 2^{30}.

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, $x_1, \ldots, x_k$;

e an expression that also usually involves the variable(s) x, $x_1, \ldots, x_k$.

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`$_2$`, ..., e`$_n$` ]`

Given a list of expressions $e_1, \ldots, e_n$, defining elements $a_1, a_2, \ldots, a_n$ all belonging to (or automatically coercible into) a single algebraic structure U, create the sequence $Q = [a_1, a_2, ..., a_n\ ]$ 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`$_2$`, ..., e`$_m$` ]`

Given a list of expressions $e_1, \ldots, e_m$, which define elements $a_1, a_2, \ldots, a_n$ that are all coercible into U, create the sequence $Q = [a_1, a_2, ..., a_n\ ]$ of elements of U.

```
[ e(x) : x in E | 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 in E ]` .

```
[ U | e(x) : x in E | 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`$_1$`,...,x`$_k$`) : x`$_1$` in E`$_1$`, ..., x`$_k$` in E`$_k$` | P(x`$_1$`, ..., x`$_k$`) ]`

The sequence consisting of those elements $e(x_1, \ldots, x_k)$, in some common structure, for which $x_1, \ldots, x_k$ in $E_1, \ldots, E_k$ have the property that $P(x_1, \ldots, x_k)$ 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 `x`$_i$`, x`$_{i+1}$` in E`$_i$.

Also, if $P(x_1, ..., x_k)$ is always true, it may be omitted.

```
[ U | e(x1,...,xk) :  x1 in E1, ..., xk in Ek | P(x1, ..., xk) ]
```

As in the previous entry, the sequence consisting of those elements $e(x_1, \ldots, x_k)$ for which $P(x_1, \ldots, x_k)$ is true is formed, as a sequence of elements of U (an error occurs if not all $e(x_1, \ldots, x_k)$ 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 | i..j ]
```

The enumerated sequence of integers whose elements form the arithmetic progression $i, i+1, i+2, \ldots, j$, 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 ..  j by k ]
```

```
[ U | i ..  j by k ]
```

The enumerated sequence consisting of the integers forming the arithmetic progression $i, i+k, i+2*k, \ldots, j$, 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.

`\[ m_1, ..., 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 $[s_1, \ldots, s_n, s_{n+1}, \ldots, s_{i-1}, x]$ replaces S, where $s_{n+1}, \ldots, s_{i-1}$ 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 $[s_1, \ldots, s_n]$. Further, i and j are integers or multi-indices (see Introduction).

`S[I]`

The sequence $[s_{i_1}, \ldots, s_{i_r}]$ 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 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(R)`

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(R)`

Given an enumerated sequence R of length r this function returns the r entries of the sequence (in order).

`Eltseq(R)`

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(~S, x)`

`Append(S, x)`

Create an enumerated sequence by adding the object x to the end of S, i.e., the enumerated sequence $[s_1, \ldots s_n, x]$.

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(~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 $[s_1, \ldots s_{i-1}, s_{i+1}, \ldots, s_n]$, 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.

```
Include(~S, x)
```
```
Include(S, x)
```

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 $[s_1, \ldots, s_n, x]$ 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.

```
Insert(~S, i, x)
```
```
Insert(S, i, x)
```

Create the sequence formed by inserting the object x at position i in S and moving the terms $S[i], \ldots, S[n]$ down one place, i.e., the enumerated sequence $[s_1, \ldots\ s_{i-1}$, x, s_i, …, $s_n]$. 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 $S[n+1], \ldots, S[i-1]$ 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.

```
Insert(~S, k, m, T)
```
```
Insert(S, k, m, T)
```

Create the sequence $[s_1,\ \ldots,\ s_{k-1},\ t_1,\ \ldots,\ t_l,\ s_{m+1},\ \ldots,\ s_n]$. 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 $[s_1, \ldots, s_n, s_{n+1}, \ldots, s_{k-1}, t_1, \ldots, t_l]$ is created, where $s_{n+1}, \ldots, s_{k-1}$ are all undefined. In the case where T is the empty sequence, terms $s_k, \ldots, s_m$ 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(~S)`

`Prune(S)`

Create the enumerated sequence formed by removing the last term of the sequence S, i.e., the sequence $[s_1, \ldots, s_{n-1}]$. 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(~S, i)`

`Remove(S, i)`

Create the enumerated sequence formed by removing the i-th term from S, i.e., the sequence $[s_1, \ldots s_{i-1}, s_{i+1}, \ldots, s_n]$. 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(~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 $[s_n, \ldots, s_1]$.

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(~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(~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(~S, C)`

`Sort(~S, C, ~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(~S, ~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(~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 $[s_1, \ldots, s_n, t_1, \ldots, t_m]$.

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 $i = 1, \ldots, \#P$.

`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 operations work pointwise on sequences of booleans of equal length.

`And(S, T)`

`And(~S, T)`

The sequence whose ith entry is the logical and of the ith entries of S and T. The result is placed in S if it is given by reference (~).

`Or(S, T)`

`Or(~S, T)`

The sequence whose ith entry is the logical or of the ith 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 ith entry is the logical xor of the ith entries of S and T. The result is placed in S if it is given by reference.

`Not(S)`

`Not(~S)`

The sequence whose ith entry is the logical not of the ith 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 $1 \le i \le \#S$ 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 $i = [i_1, \ldots, i_r]$ is a multi-index and $i_j > \#S[i_1, \ldots, i_{j-1}]$ the function returns false, but if S is s levels deep and $r > s$ while $i_j \le \#S[i_1, \ldots, i_{j-1}]$ for $1 \le j \le s$, 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(S, T)
```

```
IsSubsequence(S, T: Kind := option)
```

`Kind` MonStgElt *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.

```
S eq T
```

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.

```
S ne T
```

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` ($\leq$), `lt` ($<$), `gt` ($>$), and `ge` ($\geq$) 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 $S \neq T$ one has `S gt T`.

```
S lt T
```

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 $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#S$ and $\#S < \#T$.

```
S le T
```

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 $1 \leq i < k$) for some k, or $S[i] = T[i]$ for $1 \leq i \leq \#S$ and $\#S \leq \#T$.

`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 $S = [a_1, a_2, \ldots, a_n]$ of elements belonging to an algebraic structure U, and an (associative) operator $\circ : U \times U \to U$, form the element $a_1 \circ a_2 \circ a_3 \circ \ldots \circ a_n$.

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:

	empty	*null*
&+	U ! 0	error
&∗	U ! 1	error
&and	`true`	`true`
&or	`false`	`false`
&join	*empty*	*null*
&meet	error	error
&cat	*empty*	*null*

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

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< R_1, ..., R_k >`

Given a list of sets or algebraic structures $R_1, \ldots, R_k$, construct the cartesian product set $R_1 \times \cdots \times R_k$.

`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 **Q** and **Z**.

```
> C := car< RationalField(), Integers() >;
> C;
Cartesian Product<Rational Field, Ring of Integers>
```

11.3 Creating and Modifying Tuples

`elt< C | a_1, a_2, ..., a_k >`

`C ! < a_1, a_2, ..., a_k >`

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a sequence of elements $a_1, a_2, \ldots, a_k$, such that a_i belongs to the set R_i $(i = 1, \ldots, k)$, create the tuple $T = < a_1, a_2, ..., a_k >$ of C.

`< a_1, a_2, ..., a_k >`

Given a cartesian product $C = R_1 \times \cdots \times R_k$ and a list of elements $a_1, a_2, \ldots, a_k$, such that a_i belongs to the set R_i, $(i = 1, \ldots, k)$, create the tuple $T = < a_1, a_2, ..., a_k >$ 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(~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(~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 eq U`

Return `true` if and only if the tuples T and U are equal.

`T 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

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 `[*` and `*]`.

`[* *]`

The empty list.

`[* e1, e2, ..., en *]`

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 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 cat:= T`

(Procedure.) Destructively concatenate the terms of the list T to S; i.e. so S becomes the list $[* \; s_1, \dots, s_n, t_1, \dots, t_m \; *]$.

`Append(S, x)`

The list formed by adding the object x to the end of the list S, i.e. the list $[* \; s_1, \dots s_n, x \; *]$.

`Append(~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(~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 $S[i], \ldots, S[n]$ down one place, i.e., the list $[* \, s_1, \ldots, s_{i-1}, x, s_i, \ldots, s_n \, *]$. 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 $[* \, s_1, \ldots, s_{n-1} \, *]$.

`Prune(~S)`

(Procedure.) Destructively remove the last term of the list S; i.e. so S becomes the list $[* \, s_1, \ldots, s_{n-1} \, *]$.

`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 ith 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

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(~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

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

```
cop< S_1, S_2, ..., S_k >
```

```
cop< [ S_1, S_2, ..., S_k ] >
```

Given a list or a sequence of two or more structures S_1, S_2, ..., S_k, this function creates and returns their coproduct C as well as a sequence of maps $[m_1, m_2, \ldots, m_k]$ that provide the injections $m_i : S_i \to C$.

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.

```
m(e)
```

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.

```
C ! e
```

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

`Injections(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 $n_1, \ldots, n_m$ from structures $S_1, \ldots, S_m$ that compose the coproduct C, to some structure S, this function returns the universal map $C \to S$.

15 RECORDS

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.

`recformat< L >`

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

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 $< a, b >$ of G. A subgraph having the additional property that every element of A appears as the first component of some pair $< a, b >$ 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* `<a, b>`, or as an *arrow pair* `a -> 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 $f : A \to B$, 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 $f : A \to B$, 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 $f : A \to B$, as defined by $x \mapsto e(x)$, with corresponding inverse $f^{-1} : B \to A$, 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 $f : A \to B$, 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 $f : A \to B$, 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 $f : A \to B$ 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 $f : A \to B$, 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 $f : A \to B$, and a mapping $g : B \to C$, construct the composition h of the mappings f and g as the mapping $h = g \circ f : A \to C$.

```
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.

`HasPreimage(x, f)`

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.

`Parent(m)`

The parent of m.

`Domain(P)`

`Codomain(P)`

The domain and codomain of the maps for which P is the parent.

`Maps(D, C)`

`Iso(D, C)`

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.

`Aut(S)`

The parent of automorphisms of S.

INDEX OF INTRINSICS