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 V2.14 include group theory, ring theory, commutative algebra, arithmetic fields and their completions, module theory and lattice theory, finite dimensional algebras, Lie theory, representation theory, the elements of homological algebra, general schemes and curve schemes, modular forms and modular curves, 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. It also describes the facilities for semigroups and monoids. 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 and the 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 and their orders and function fields. More specialised topics include quadratic, 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, 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 is concerned with geometry and combinatorial theory. The geometry section covers finite planes and finite incidence geometry. The combinatorial theory topics include enumeration, designs, Hadamard matrices, graphs and networks.

• Volume 13 is primarily concerning 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 \textsc{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, Steve Donnelly, Willem de Graaf, Claus Fieker, Damien Fisher, Alexandra Flynn, Volker Gebhardt, Katharina Geißler, Sergei Haller, Michael Harrison, Emanuel Herrmann, Florian Heß, David Kohel, Paulette Lieby, Graham Matthews, Scott Murray, Anne O’Kane, Catherine Playoust, Richard Rannard, Colva Roney-Dougal, Andrew Solomon, Bernd Souvignier, Ben Smith, Allan Steel, Damien Stehlé, Nicole Sutherland, Bill Unger, John Voight, Alexa van der Waall, Mark Watkins and Greg White.

John Cannon
Sydney, October 2007
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. (Georgia) [2005-]: Research interests are in arithmetic geometry. Major contributions include descent methods for elliptic curves over the rational field and over function fields, the Cassels-Tate pairing, modular forms of half-integral weight, and improvements to the modular forms package. Currently working on further tools for elliptic curves, modular forms and Hilbert modular forms.

Claus Fieker, Ph.D. (TU Berlin), [2000-]: 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. He is currently working on Galois cohomology in number fields (with applications to the construction of representations of finite groups) and also on developing generic constructive techniques for Drinfeld modules.

Sergei Haller, Ph.D. (Eindhoven) [2004, 2006-]: Works in the area of linear algebraic groups. Implemented new algorithms for element operations in (split and twisted) groups of Lie type, non-reduced and extended root data, Cartan type Lie algebras, Galois cohomology, and cohomology of finite non-abelian groups.

Michael Harrison, Ph.D. (Cambridge), [2003-]: Research interests are in number theory, arithmetic and algebraic geometry. Implemented the p-adic methods for counting points on hyperelliptic curves and their Jacobians over finite fields: Kedlaya’s method and the modular parameter method of Mestre. Currently working on machinery for general surfaces and cohomology for projective varieties.
Scott Murray, Ph.D. (Chicago), [2001-2002, 2004-]: Research interests are in linear algebraic groups, finite matrix groups, Lie algebras, representation theory. Implemented algorithms for element operations in split groups of Lie type, representations of split groups of Lie type, split Cartan subalgebras of modular Lie algebra, and Lang’s Theorem in finite reductive groups. Currently working on conjugacy problems in reductive groups and finite groups of Lie type.

Allan Steel, BA (Hons, University Medal) (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).

Damien Stehlé, Ph.D. (Nancy) [2006]: Works in the areas of algorithmic number theory (in particular, the geometry of numbers) and computer arithmetic. Implemented the proveably correct floating-point LLL algorithm together with a number of fast non-rigorous variants.

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

Bill Unger, Ph.D. (Sydney), [1998-]: Works in 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 developed a new method for computing character tables of finite groups.
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.

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.

Emanuel Herrmann, [1999]: Developed code for computing integral points and S-integral points on 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 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 alge-
bric 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.

fields including a first version of the SEA algorithm.

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

**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.

**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.

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

**Mark Watkins**, [2003, 2004-2005]: Implemented a range of analytic tools for the study
of elliptic curves including analytic rank, modular degree, set of curves $Q$-isogenous to a
given curve, 4-descent, Heegner points and other point searching methods.

**Greg White**, [2000-2006]: Contributions include fast minimum weight determination,
linear codes over $\mathbb{Z}/m\mathbb{Z}$, 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.

Basic Rings

A facility for computing with arbitrary but fixed precision reals was based on Richard Brent’s (ANU) FORTRAN package MP. Richard has also made available his database of 237,578 factorizations of integers of the form $p^n \pm 1$, together with his intelligent factorization code FACTOR.

Stefi Cavallar (CWI, Amsterdam) has adapted her code for filtering relations in the CWI Number Field Sieve so as to run as part of the Magma Number Field Sieve.

The group headed by Henri Cohen (Bordeaux) made available parts of their Pari system for computational number theory for inclusion in Magma. Pascal Letard of the Pari group visited Sydney for two months in 1994 and recoded large sections of Pari for Magma. The Pari facilities installed in Magma include arithmetic for real and complex fields (the ‘free’ model), approximately 100 special functions for real and complex numbers, quadratic fields and other features.

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.

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.

Coppersmith’s method for finding small roots of univariate polynomials modulo an integer has been implemented by Damien Stehlé.

The code for Coppersmith’s index-calculus algorithm (used to compute logarithms in finite fields of characteristic 2) was developed by Emmanuel Thomé (Ecole Polytechnique).

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

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

**Global and Local Fields**

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

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

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

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

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

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

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

The facilities for general number fields and global function fields in Magma are based on the KANT V4 package developed by Michael Pohst and collaborators, originally at Düsseldorf and now 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).
Finitely-Presented Groups

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

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

The function for determining whether a given finite permutation group is a homomorphic image of a finitely presented group has been implemented in C by Volker Gebhardt (Magma) from a Magma language prototype developed by Derek Holt (Warwick).

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

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

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

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

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

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

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

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

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

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

Machinery for computing group cohomology and for producing group extensions has been provided 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.

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

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

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

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

Matrix Groups

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, Derek Holt and Colva Roney-Dougal.

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

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

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

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

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

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

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

Eamonn O’Brien (Auckland) has provided implementations of constructive recognition algorithms for matrix groups as either $(P)SL(2, q)$ or $(P)SL(3, q)$.

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 (Sydney).

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

Generators for matrix representations for groups of Lie type were constructed and implemented by Don Taylor (Sydney) with some assistance from Leanne Rylands (Western Sydney).

Permutation Groups

Greg Gamble (UWA) helped refine the concept of a $G$-set for a permutation group.

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).

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

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

Coxeter Groups

A fast algorithm for multiplying Coxeter group elements has been designed and implemented by Bob Howlett (Sydney).

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

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

Lie Groups

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

Linear Algebra and Module Theory

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

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

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

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

The functions for computing automorphism groups and isometries of lattices are based on the AUTO and ISOM programs of Bernd Souvignier (Nijmegen).

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 $F$.

Algebras

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

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.

A database of all nilpotent Lie algebras of dimension up to 6 over fields of odd characteristic has been implemented by Willem de Graaf (Trento).

Lie rings (that is, Lie algebras over defined over an euclidean ring) have been implemented. The major features of the package include a facility to construct a multiplication table and basis for a finitely presented Lie ring of finite dimension and a function to construct nilpotent quotients of finitely presented Lie ring. These features were implemented by Willem de Graaf (Trento).

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

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

Quaternion algebras over the rational field $\mathbb{Q}$ have been implemented by David Kohel (Singapore, Sydney).

Fast algorithms for computing the 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).

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

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).

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).
ACKNOWLEDGEMENTS

The vector enumeration program of Steve Linton (St. Andrews) provides MAGMA with the capability of constructing matrix representations for finitely presented associative algebras.

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.

Commutative Algebra

Gregor Kemper (Heidelberg) 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.

Algebraic Geometry

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

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

Arithmetic Geometry: Elliptic Curves

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

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

The widely-used database of all elliptic curves over \( \mathbb{Q} \) having conductor up to 130,000 constructed by John Cremona (Nottingham) is also included.

The implementation of 3-descent on elliptic curves that is available in MAGMA is mainly due to John Cremona (Nottingham) and Michael Stoll (Bremen).

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

The implementation of three descent on elliptic curves uses code developed by Tom Fisher (Cambridge) and Michael Stoll (Bremen).

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, Sydney) has provided implementations of division polynomials and isogeny structures for elliptic curves.
Reynard Lercier (Rennes) provided much advice and assistance to the MAGMA group concerning the implementation of the SEA point counting algorithm for elliptic curves.

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.

Sebastian Stamminger (Bremen) implemented his algorithm for eight descent on elliptic curves.

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

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

Tom Womack (Nottingham) contributed code for performing four-descent and for locating Heegner points on an elliptic curve.

Arithmetic Geometry: Hyperelliptic Curves

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

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

Hendrik Hubrechts (Leuven) has made available his package for point-counting and computing zeta-functions using deformation methods for parametrized families of hyperelliptic curves and their Jacobians in small, odd characteristic.

Much of the initial development of the package for computing with hyperelliptic curves is due to Michael Stoll (Düsseldorf).

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

Modular Arithmetic Geometry

Kevin Buzzard made available his code for computing modular forms of weight one, from which the MAGMA implementation was adapted.

Tim Dokchitser (Edinburgh) has implemented his techniques for computing special values of motivic $L$-functions in MAGMA, and an efficient general algorithm for root numbers of elliptic curves over number fields.

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

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

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

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

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

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

Geometry

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

Combinatorics

Michel Berkelaar (Eindhoven) gave us permission to incorporate his $\texttt{lp\_solve}$ 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.

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

The code to perform the regular expression matching in the $\texttt{regexp}$ intrinsic function comes from the V8 regexp package by Henry Spencer (Toronto).
Coding Theory

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

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

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

Graham Norton (Queensland) has provided substantial advice and help in the development of $Z_4$-codes in Magma.
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 V2.14 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.
ACKNOWLEDGEMENTS

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 Univariate Polynomial Rings ● A. Steel
23 Multivariate Polynomial Rings ● A. Steel
24 Real and Complex Fields ● W. Bosma
25 Matrices ● A. Steel
26 Sparse Matrices ● A. Steel
27 Vector Spaces ● J. Cannon, A. Steel
28 Lattices ● B. Souvignier, A. Steel ○ D. Stehlé
29 Quadratic Forms ● S. Donnelly
30 Binary Quadratic Forms ● D. Kohel
31 Orders and Algebraic Fields ● C. Fieker ○ W. Bosma, N. Sutherland
32 Quadratic Fields ● W. Bosma
33 Cyclotomic Fields ● W. Bosma, C. Fieker
34 Galois Theory of Number Fields ● C. Fieker ○ J. Klüners, K. Geißler
35 Class Field Theory ● C. Fieker
36 Algebraically Closed Fields ● A. Steel
37 Rational Function Fields ● A. Steel
38 Algebraic Function Fields ● F. Heß ○ C. Fieker, N. Sutherland
39 Class Field Theory For Global Function Fields  •  C. Fieker
40 Valuation Rings  •  W. Bosma
41 Newton Polygons  •  G. Brown, N. Sutherland
42 $p$-adic Rings and their Extensions  •  D. Fisher, B. Souvignier  ○  N. Sutherland
43 Galois Rings  •  A. Steel
44 Power, Laurent and Puiseux Series  •  A. Steel
45 Lazy Power Series Rings  •  N. Sutherland
46 Introduction to Modules  •  J. Cannon
47 Free Modules  •  J. Cannon, A. Steel
48 Modules over Dedekind Domains  •  C. Fieker, N. Sutherland
49 Chain Complexes  •  J. Carlson
50 Groups  •  J. Cannon  ○  W. Unger
51 Permutation Groups  •  J. Cannon  ○  B. Cox, W. Unger
52 Matrix Groups over General Rings  •  J. Cannon  ○  B. Cox, E.A. O’Brien, A. Steel
53 Matrix Groups over Finite Fields  •  E.A. O’Brien
54 Finite Soluble Groups  •  J. Cannon, M. Slattery  ○  E.A. O’Brien
55 Black-box Groups  •  W. Unger
57 Databases of Groups  •  W. Unger  ○  V. Gebhardt
58 Automorphism Groups  •  D. Holt  ○  W. Unger
59 Cohomology and Extensions  •  D. Holt  ○  S. Haller
60 Abelian Groups  •  J. Cannon  ○  P. Lieby
61 Finitely Presented Groups  •  J. Cannon  ○  V. Gebhardt, E.A. O’Brien, M. Vaughan-Lee
63 Polycyclic Groups  •  V. Gebhardt
64 Braid Groups  •  V. Gebhardt
65 Groups Defined by Rewrite Systems  •  D. Holt  ○  G. Matthews
66 Automatic Groups  •  D. Holt  ○  G. Matthews
67 Groups of Straight-line Programs  •  J. Cannon
68 Finitely Presented Semigroups  •  J. Cannon
69 Monoids Given by Rewrite Systems  •  D. Holt  ○  G. Matthews
70 Introduction to Lie Theory  •  S. Murray  ○  D. Taylor
71 Coxeter Systems  •  S. Murray  ○  D. Taylor
72 Root Systems  •  S. Murray  ○  S. Haller, D. Taylor
73 Root Data  •  S. Haller, S. Murray  ○  D. Taylor
74 Coxeter Groups  •  S. Murray  ○  D. Taylor
75 Reflection Groups  •  S. Murray  ○  D. Taylor
76 Groups of Lie Type  •  S. Murray  ○  S. Haller, D. Taylor
77 Algebras • J. Cannon, B. Souvignier
78 Structure Constant Algebras • J. Cannon, B. Souvignier
79 Associative Algebras ○ J. Cannon, S. Donnelly, N. Sutherland, B. Souvignier, J. Voight
80 Finitely Presented Algebras • A. Steel, S. Linton
81 Modules over An Algebra • J. Cannon, A. Steel
82 Matrix Algebras • J. Cannon, A. Steel ○ J. Carlson
83 Basic Algebras • J. Carlson
84 Quaternion Algebras • D. Kohel, J. Voight ○ S. Donnelly,
85 Lie Algebras • W. de Graaf ○ S. Haller, S. Murray
86 Quantum Groups • W. de Graaf
87 Universal Enveloping Algebras • W. de Graaf
88 Group Algebras • J. Cannon, B. Souvignier
89 K[G]-Modules and Group Representations • J. Cannon, A. Steel
90 Characters of Finite Groups • W. Bosma, J. Cannon
91 Representations of Symmetric Groups • A. Kohnert
92 Representations of Lie Groups and Algebras • D. Roozemond
93 Ideal Theory and Gröbner Bases • A. Steel ○ M. Harrison
94 Affine Algebras • A. Steel
95 Modules over Affine Algebras • A. Steel
96 Invariant Theory • A. Steel
97 Differential Rings • A. van der Waall
98 Schemes • G. Brown ○ J. Cannon, M. Harrison, N. Sutherland
99 Algebraic Curves • G. Brown ○ N. Bruin, J. Cannon, M. Harrison
100 Resolution Graphs and Splice Diagrams • G. Brown
101 Hilbert Series of Polarised Varieties • G. Brown
102 Rational Curves and Conics • D. Kohel, P. Lieby ○ M. Watkins
104 Elliptic Curves over Finite Fields • M. Harrison ○ P. Lieby
105 Elliptic Curves over Function Fields • J. Scholten ○ S. Donnelly
106 Models of Genus One Curves • T. Fisher, S. Donnelly
107 Hyperelliptic Curves ○ N. Bruin, S. Donnelly, M. Harrison, D. Kohel, P. van Wamelen
108 L-functions • T. Dokchitser
109 Modular Curves • D. Kohel
110 Congruence Subgroups of PSL₂(R) • H. Verrill
111 Arithmetic Fuchsian Groups and Shimura Curves • J. Voight
112 Modular Forms • W. Stein ○ K. Buzzard, S. Donnelly
113 Modular Symbols • W. Stein ○ K. Buzzard
114 Brandt Modules • D. Kohel
115 Supersingular Divisors on Modular Curves • D. Kohel, W. Stein
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>116</td>
<td>Modular Abelian Varieties</td>
<td>W. Stein</td>
</tr>
<tr>
<td>117</td>
<td>Finite Planes</td>
<td>J. Cannon</td>
</tr>
<tr>
<td>118</td>
<td>Incidence Geometry</td>
<td>D. Leemans</td>
</tr>
<tr>
<td>119</td>
<td>Enumerative Combinatorics</td>
<td>G. Bailey, G. White</td>
</tr>
<tr>
<td>120</td>
<td>Partitions, Words and Young Tableaux</td>
<td>J. Cannon</td>
</tr>
<tr>
<td>121</td>
<td>Symmetric Functions</td>
<td>A. Kohnert</td>
</tr>
<tr>
<td>122</td>
<td>Incidence Structures and Designs</td>
<td>J. Cannon</td>
</tr>
<tr>
<td>123</td>
<td>Hadamard Matrices</td>
<td>G. Bailey</td>
</tr>
<tr>
<td>124</td>
<td>Graphs</td>
<td>J. Cannon, P. Lieby, G. Bailey</td>
</tr>
<tr>
<td>125</td>
<td>Multigraphs</td>
<td>J. Cannon, P. Lieby</td>
</tr>
<tr>
<td>126</td>
<td>Networks</td>
<td>P. Lieby</td>
</tr>
<tr>
<td>127</td>
<td>Linear Codes over Finite Fields</td>
<td>J. Cannon, A. Steel, G. White</td>
</tr>
<tr>
<td>128</td>
<td>Algebraic-geometric Codes</td>
<td>J. Cannon, G. White</td>
</tr>
<tr>
<td>129</td>
<td>Low Density Party Check Codes</td>
<td>G. White</td>
</tr>
<tr>
<td>130</td>
<td>Linear Codes over Finite Rings</td>
<td>A. Steel, G. White</td>
</tr>
<tr>
<td>131</td>
<td>Additive Codes</td>
<td>G. White</td>
</tr>
<tr>
<td>132</td>
<td>Quantum Codes</td>
<td>G. White</td>
</tr>
<tr>
<td>133</td>
<td>Pseudo-random Bit Sequences</td>
<td>S. Contini</td>
</tr>
<tr>
<td>134</td>
<td>Linear Programming</td>
<td>B. Smith</td>
</tr>
</tbody>
</table>

**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 $d = l \times a + m \times b$.

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

```magma
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 False
```

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:

```magma
load "H28E1";
```
# VOLUME 1: OVERVIEW

## I  THE MAGMA LANGUAGE  . . . . . . . . . . . . . 1

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

## II  SETS, SEQUENCES, AND MAPPINGS  . . . . . . 145

8  INTRODUCTION TO AGGREGATES  
9  SETS  
10  SEQUENCES  
11  TUPLES AND CARTESIAN PRODUCTS  
12  LISTS  
13  ASSOCIATIVE ARRAYS  
14  COPRODUCTS  
15  RECORDS  
16  MAPPINGS  

## VOLUME 2: OVERVIEW

<table>
<thead>
<tr>
<th>III</th>
<th>BASIC RINGS AND LINEAR ALGEBRA</th>
<th>247</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>INTRODUCTION TO RINGS</td>
<td>249</td>
</tr>
<tr>
<td>18</td>
<td>RING OF INTEGERS</td>
<td>269</td>
</tr>
<tr>
<td>19</td>
<td>INTEGER RESIDUE CLASS RINGS</td>
<td>319</td>
</tr>
<tr>
<td>20</td>
<td>RATIONAL FIELD</td>
<td>339</td>
</tr>
<tr>
<td>21</td>
<td>FINITE FIELDS</td>
<td>351</td>
</tr>
<tr>
<td>22</td>
<td>UNIVARIATE POLYNOMIAL RINGS</td>
<td>379</td>
</tr>
<tr>
<td>23</td>
<td>MULTIVARIATE POLYNOMIAL RINGS</td>
<td>411</td>
</tr>
<tr>
<td>24</td>
<td>REAL AND COMPLEX FIELDS</td>
<td>439</td>
</tr>
<tr>
<td>25</td>
<td>MATRICES</td>
<td>483</td>
</tr>
<tr>
<td>26</td>
<td>SPARSE MATRICES</td>
<td>521</td>
</tr>
<tr>
<td>27</td>
<td>VECTOR SPACES</td>
<td>541</td>
</tr>
</tbody>
</table>
## VOLUME 3: OVERVIEW

<table>
<thead>
<tr>
<th>IV</th>
<th>LATTICES AND QUADRATIC FORMS</th>
<th>565</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>LATTICES</td>
<td>567</td>
</tr>
<tr>
<td>29</td>
<td>QUADRATIC FORMS</td>
<td>649</td>
</tr>
<tr>
<td>30</td>
<td>BINARY QUADRATIC FORMS</td>
<td>653</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
<th>GLOBAL ARITHMETIC FIELDS</th>
<th>667</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>ORDERS AND ALGEBRAIC FIELDS</td>
<td>669</td>
</tr>
<tr>
<td>32</td>
<td>QUADRATIC FIELDS</td>
<td>773</td>
</tr>
<tr>
<td>33</td>
<td>CYCLOTOMIC FIELDS</td>
<td>785</td>
</tr>
<tr>
<td>34</td>
<td>GALOIS THEORY OF NUMBER FIELDS</td>
<td>793</td>
</tr>
<tr>
<td>35</td>
<td>CLASS FIELD THEORY</td>
<td>827</td>
</tr>
<tr>
<td>36</td>
<td>ALGEBRAICALLY CLOSED FIELDS</td>
<td>865</td>
</tr>
<tr>
<td>37</td>
<td>RATIONAL FUNCTION FIELDS</td>
<td>887</td>
</tr>
<tr>
<td>38</td>
<td>ALGEBRAIC FUNCTION FIELDS</td>
<td>899</td>
</tr>
<tr>
<td>39</td>
<td>CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS</td>
<td>1003</td>
</tr>
</tbody>
</table>
# VOLUME 4: OVERVIEW

## VI LOCAL ARITHMETIC FIELDS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>VALUATION RINGS</td>
<td>1031</td>
</tr>
<tr>
<td>41</td>
<td>NEWTON POLYGONS</td>
<td>1037</td>
</tr>
<tr>
<td>42</td>
<td>$p$-ADIC RINGS AND THEIR EXTENSIONS</td>
<td>1065</td>
</tr>
<tr>
<td>43</td>
<td>GALOIS RINGS</td>
<td>1113</td>
</tr>
<tr>
<td>44</td>
<td>POWER, LAURENT AND PUISEUX SERIES</td>
<td>1121</td>
</tr>
<tr>
<td>45</td>
<td>LAZY POWER SERIES RINGS</td>
<td>1143</td>
</tr>
</tbody>
</table>

## VII MODULES

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>INTRODUCTION TO MODULES</td>
<td>1161</td>
</tr>
<tr>
<td>47</td>
<td>FREE MODULES</td>
<td>1165</td>
</tr>
<tr>
<td>48</td>
<td>MODULES OVER DEDEKIND DOMAINS</td>
<td>1189</td>
</tr>
<tr>
<td>49</td>
<td>CHAIN COMPLEXES</td>
<td>1211</td>
</tr>
<tr>
<td>Chapter</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>VIII</td>
<td>FINITE GROUPS</td>
<td>1227</td>
</tr>
<tr>
<td>50</td>
<td>GROUPS</td>
<td>1229</td>
</tr>
<tr>
<td>51</td>
<td>PERMUTATION GROUPS</td>
<td>1285</td>
</tr>
<tr>
<td>52</td>
<td>MATRIX GROUPS OVER GENERAL RINGS</td>
<td>1399</td>
</tr>
<tr>
<td>53</td>
<td>MATRIX GROUPS OVER FINITE FIELDS</td>
<td>1471</td>
</tr>
<tr>
<td>54</td>
<td>FINITE SOLUBLE GROUPS</td>
<td>1495</td>
</tr>
<tr>
<td>55</td>
<td>BLACK-BOX GROUPS</td>
<td>1573</td>
</tr>
<tr>
<td>56</td>
<td>ALMOST SIMPLE GROUPS</td>
<td>1579</td>
</tr>
<tr>
<td>57</td>
<td>DATABASES OF GROUPS</td>
<td>1631</td>
</tr>
<tr>
<td>58</td>
<td>AUTOMORPHISM GROUPS</td>
<td>1685</td>
</tr>
<tr>
<td>59</td>
<td>COHOMOLOGY AND EXTENSIONS</td>
<td>1703</td>
</tr>
</tbody>
</table>
**VOLUME 6: OVERVIEW**

<table>
<thead>
<tr>
<th>IX</th>
<th>FINITELY-PRESENTED GROUPS . . . . . . . . . . . . .</th>
<th>1729</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
<td>ABELIAN GROUPS</td>
<td>1731</td>
</tr>
<tr>
<td>61</td>
<td>FINITELY PRESENTED GROUPS</td>
<td>1765</td>
</tr>
<tr>
<td>62</td>
<td>FINITELY PRESENTED GROUPS: ADVANCED</td>
<td>1877</td>
</tr>
<tr>
<td>63</td>
<td>POLycyclic GROUPS</td>
<td>1953</td>
</tr>
<tr>
<td>64</td>
<td>BRAID GROUPS</td>
<td>1993</td>
</tr>
<tr>
<td>65</td>
<td>GROUPS DEFINED BY REWRITE SYSTEMS</td>
<td>2045</td>
</tr>
<tr>
<td>66</td>
<td>AUTOMATIC GROUPS</td>
<td>2063</td>
</tr>
<tr>
<td>67</td>
<td>GROUPS OF STRAIGHT-LINE PROGRAMS</td>
<td>2083</td>
</tr>
<tr>
<td>68</td>
<td>FINITELY PRESENTED SEMIGROUPS</td>
<td>2093</td>
</tr>
<tr>
<td>69</td>
<td>MONOIDS GIVEN BY REWRITE SYSTEMS</td>
<td>2105</td>
</tr>
</tbody>
</table>
**VOLUME 7: OVERVIEW**

<table>
<thead>
<tr>
<th></th>
<th>COXETER GROUPS</th>
<th>2123</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>INTRODUCTION TO LIE THEORY</td>
<td>2125</td>
</tr>
<tr>
<td>71</td>
<td>COXETER SYSTEMS</td>
<td>2131</td>
</tr>
<tr>
<td>72</td>
<td>ROOT SYSTEMS</td>
<td>2155</td>
</tr>
<tr>
<td>73</td>
<td>ROOT DATA</td>
<td>2177</td>
</tr>
<tr>
<td>74</td>
<td>COXETER GROUPS</td>
<td>2227</td>
</tr>
<tr>
<td>75</td>
<td>REFLECTION GROUPS</td>
<td>2263</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XI</th>
<th>LIE GROUPS</th>
<th>2291</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>GROUPS OF LIE TYPE</td>
<td>2293</td>
</tr>
</tbody>
</table>
# VOLUME 8: OVERVIEW

## XII ALGEBRAS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td>ALGEBRAS</td>
<td>2333</td>
</tr>
<tr>
<td>78</td>
<td>STRUCTURE CONSTANT ALGEBRAS</td>
<td>2345</td>
</tr>
<tr>
<td>79</td>
<td>ASSOCIATIVE ALGEBRAS</td>
<td>2355</td>
</tr>
<tr>
<td>80</td>
<td>FINITELY PRESENTED ALGEBRAS</td>
<td>2381</td>
</tr>
<tr>
<td>81</td>
<td>MODULES OVER AN ALGEBRA</td>
<td>2419</td>
</tr>
<tr>
<td>82</td>
<td>MATRIX ALGEBRAS</td>
<td>2455</td>
</tr>
<tr>
<td>83</td>
<td>BASIC ALGEBRAS</td>
<td>2495</td>
</tr>
<tr>
<td>84</td>
<td>QUATERNION ALGEBRAS</td>
<td>2535</td>
</tr>
<tr>
<td>85</td>
<td>LIE ALGEBRAS</td>
<td>2573</td>
</tr>
<tr>
<td>86</td>
<td>QUANTUM GROUPS</td>
<td>2627</td>
</tr>
<tr>
<td>87</td>
<td>UNIVERSAL ENVELOPING ALGEBRAS</td>
<td>2653</td>
</tr>
</tbody>
</table>

## XIII REPRESENTATION THEORY

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>GROUP ALGEBRAS</td>
<td>2663</td>
</tr>
<tr>
<td>89</td>
<td>$K[G]$-MODULES AND GROUP REPRESENTATIONS</td>
<td>2677</td>
</tr>
<tr>
<td>90</td>
<td>CHARACTERS OF FINITE GROUPS</td>
<td>2703</td>
</tr>
<tr>
<td>91</td>
<td>REPRESENTATIONS OF SYMMETRIC GROUPS</td>
<td>2725</td>
</tr>
<tr>
<td>92</td>
<td>REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS</td>
<td>2733</td>
</tr>
</tbody>
</table>
# VOLUME 9: OVERVIEW

<table>
<thead>
<tr>
<th>XIV</th>
<th>COMMUTATIVE ALGEBRA</th>
<th>2765</th>
</tr>
</thead>
<tbody>
<tr>
<td>93</td>
<td>IDEAL THEORY AND GRÖBNER BASES</td>
<td>2767</td>
</tr>
<tr>
<td>94</td>
<td>AFFINE ALGEBRAS</td>
<td>2845</td>
</tr>
<tr>
<td>95</td>
<td>MODULES OVER AFFINE ALGEBRAS</td>
<td>2861</td>
</tr>
<tr>
<td>96</td>
<td>INARIANT THEORY</td>
<td>2887</td>
</tr>
<tr>
<td>97</td>
<td>DIFFERENTIAL RINGS</td>
<td>2929</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XV</th>
<th>ALGEBRAIC GEOMETRY</th>
<th>2977</th>
</tr>
</thead>
<tbody>
<tr>
<td>98</td>
<td>SCHEMES</td>
<td>2979</td>
</tr>
<tr>
<td>99</td>
<td>ALGEBRAIC CURVES</td>
<td>3091</td>
</tr>
<tr>
<td>100</td>
<td>RESOLUTION GRAPHS AND SPLICE DIAGRAMS</td>
<td>3179</td>
</tr>
<tr>
<td>101</td>
<td>HILBERT SERIES OF POLARISED VARIETIES</td>
<td>3195</td>
</tr>
</tbody>
</table>
# VOLUME 10: OVERVIEW

<table>
<thead>
<tr>
<th>XVI</th>
<th>ARITHMETIC GEOMETRY</th>
<th>3229</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>RATIONAL CURVES AND CONICS</td>
<td>3231</td>
</tr>
<tr>
<td>103</td>
<td>ELLIPTIC CURVES</td>
<td>3255</td>
</tr>
<tr>
<td>104</td>
<td>ELLIPTIC CURVES OVER FINITE FIELDS</td>
<td>3357</td>
</tr>
<tr>
<td>105</td>
<td>ELLIPTIC CURVES OVER FUNCTION FIELDS</td>
<td>3381</td>
</tr>
<tr>
<td>106</td>
<td>MODELS OF GENUS ONE CURVES</td>
<td>3399</td>
</tr>
<tr>
<td>107</td>
<td>HYPERELLIPTIC CURVES</td>
<td>3415</td>
</tr>
<tr>
<td>108</td>
<td>L-FUNCTIONS</td>
<td>3491</td>
</tr>
</tbody>
</table>
# VOLUME 11: OVERVIEW

## XVII MODULAR ARITHMETIC GEOMETRY . . . . 3521

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>109</td>
<td>MODULAR CURVES</td>
<td>3523</td>
</tr>
<tr>
<td>110</td>
<td>CONGRUENCE SUBGROUPS OF PSL&lt;sub&gt;2&lt;/sub&gt;(R)</td>
<td>3537</td>
</tr>
<tr>
<td>111</td>
<td>ARITHMETIC FUCHSIAN GROUPS AND SHIMURA CURVES</td>
<td>3563</td>
</tr>
<tr>
<td>112</td>
<td>MODULAR FORMS</td>
<td>3587</td>
</tr>
<tr>
<td>113</td>
<td>MODULAR SYMBOLS</td>
<td>3627</td>
</tr>
<tr>
<td>114</td>
<td>BRANDT MODULES</td>
<td>3681</td>
</tr>
<tr>
<td>115</td>
<td>SUPERSINGULAR DIVISORS ON MODULAR CURVES</td>
<td>3695</td>
</tr>
<tr>
<td>116</td>
<td>MODULAR ABELIAN VARIETIES</td>
<td>3711</td>
</tr>
</tbody>
</table>
# VOLUME 12: OVERVIEW

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>XVIII GEOMETRY</td>
<td></td>
<td>3849</td>
</tr>
<tr>
<td>117</td>
<td>FINITE PLANES</td>
<td>3851</td>
</tr>
<tr>
<td>118</td>
<td>INCIDENCE GEOMETRY</td>
<td>3887</td>
</tr>
<tr>
<td>XIX COMBINATORICS</td>
<td></td>
<td>3909</td>
</tr>
<tr>
<td>119</td>
<td>ENUMERATIVE COMBINATORICS</td>
<td>3911</td>
</tr>
<tr>
<td>120</td>
<td>PARTITIONS, WORDS AND YOUNG TABLEAUX</td>
<td>3917</td>
</tr>
<tr>
<td>121</td>
<td>SYMMETRIC FUNCTIONS</td>
<td>3951</td>
</tr>
<tr>
<td>122</td>
<td>INCIDENCE STRUCTURES AND DESIGNS</td>
<td>3977</td>
</tr>
<tr>
<td>123</td>
<td>HADAMARD MATRICES</td>
<td>4011</td>
</tr>
<tr>
<td>124</td>
<td>GRAPHS</td>
<td>4021</td>
</tr>
<tr>
<td>125</td>
<td>MULTIGRAPHS</td>
<td>4101</td>
</tr>
<tr>
<td>126</td>
<td>NETWORKS</td>
<td>4149</td>
</tr>
</tbody>
</table>
### VOLUME 13: OVERVIEW

<table>
<thead>
<tr>
<th>XX</th>
<th>CODING THEORY</th>
<th>4169</th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td>LINEAR CODES OVER FINITE FIELDS</td>
<td>4171</td>
</tr>
<tr>
<td>128</td>
<td>ALGEBRAIC-GEOMETRIC CODES</td>
<td>4245</td>
</tr>
<tr>
<td>129</td>
<td>LOW DENSITY PARTY CHECK CODES</td>
<td>4253</td>
</tr>
<tr>
<td>130</td>
<td>LINEAR CODES OVER FINITE RINGS</td>
<td>4265</td>
</tr>
<tr>
<td>131</td>
<td>ADDITIVE CODES</td>
<td>4297</td>
</tr>
<tr>
<td>132</td>
<td>QUANTUM CODES</td>
<td>4321</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>XXI</th>
<th>CRYPTOGRAPHY</th>
<th>4359</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td>PSEUDO-RANDOM BIT SEQUENCES</td>
<td>4361</td>
</tr>
</tbody>
</table>

### XXII OPTIMIZATION | 4369 |
| 134| LINEAR PROGRAMMING | 4371 |
# VOLUME 1: CONTENTS

## I THE MAGMA LANGUAGE

### 1 STATEMENTS AND EXPRESSIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Introduction</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Starting, Interrupting and Terminating</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Identifiers</td>
<td>5</td>
</tr>
<tr>
<td>1.4</td>
<td>Assignment</td>
<td>6</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Simple Assignment</td>
<td>6</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Indexed Assignment</td>
<td>7</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Generator Assignment</td>
<td>8</td>
</tr>
<tr>
<td>1.4.4</td>
<td>Mutation Assignment</td>
<td>9</td>
</tr>
<tr>
<td>1.4.5</td>
<td>Deletion of Values</td>
<td>10</td>
</tr>
<tr>
<td>1.5</td>
<td>Boolean values</td>
<td>10</td>
</tr>
<tr>
<td>1.5.1</td>
<td>Creation of Booleans</td>
<td>11</td>
</tr>
<tr>
<td>1.5.2</td>
<td>Boolean Operators</td>
<td>11</td>
</tr>
<tr>
<td>1.5.3</td>
<td>Equality Operators</td>
<td>11</td>
</tr>
<tr>
<td>1.5.4</td>
<td>Iteration</td>
<td>12</td>
</tr>
<tr>
<td>1.6</td>
<td>Coercion</td>
<td>13</td>
</tr>
<tr>
<td>1.7</td>
<td>The where ... is Construction</td>
<td>14</td>
</tr>
<tr>
<td>1.8</td>
<td>Conditional Statements and Expressions</td>
<td>16</td>
</tr>
<tr>
<td>1.8.1</td>
<td>The Simple Conditional Statement</td>
<td>16</td>
</tr>
<tr>
<td>1.8.2</td>
<td>The Simple Conditional Expression</td>
<td>17</td>
</tr>
<tr>
<td>1.8.3</td>
<td>The Case Statement</td>
<td>18</td>
</tr>
<tr>
<td>1.8.4</td>
<td>The Case Expression</td>
<td>18</td>
</tr>
<tr>
<td>1.9</td>
<td>Error Handling Statements</td>
<td>19</td>
</tr>
<tr>
<td>1.9.1</td>
<td>The Error Objects</td>
<td>19</td>
</tr>
<tr>
<td>1.9.2</td>
<td>Error Checking and Assertions</td>
<td>19</td>
</tr>
<tr>
<td>1.9.3</td>
<td>Catching Errors</td>
<td>20</td>
</tr>
<tr>
<td>1.10</td>
<td>Iterative Statements</td>
<td>21</td>
</tr>
<tr>
<td>1.10.1</td>
<td>Definite Iteration</td>
<td>21</td>
</tr>
<tr>
<td>1.10.2</td>
<td>Indefinite Iteration</td>
<td>21</td>
</tr>
<tr>
<td>1.10.3</td>
<td>Early Exit from Iterative Statements</td>
<td>23</td>
</tr>
<tr>
<td>1.11</td>
<td>Runtime Evaluation: the eval Expression</td>
<td>24</td>
</tr>
<tr>
<td>1.12</td>
<td>Comments and Continuation</td>
<td>25</td>
</tr>
<tr>
<td>1.13</td>
<td>Timing</td>
<td>26</td>
</tr>
<tr>
<td>1.14</td>
<td>Types, Category Names, and Structures</td>
<td>28</td>
</tr>
<tr>
<td>1.15</td>
<td>Random Object Generation</td>
<td>30</td>
</tr>
<tr>
<td>1.16</td>
<td>Miscellaneous</td>
<td>32</td>
</tr>
<tr>
<td>1.17</td>
<td>Bibliography</td>
<td>32</td>
</tr>
</tbody>
</table>
## 2 FUNCTIONS, PROCEDURES AND PACKAGES

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>35</td>
</tr>
<tr>
<td>2.2 Functions and Procedures</td>
<td>35</td>
</tr>
<tr>
<td>2.2.1 Functions</td>
<td>35</td>
</tr>
<tr>
<td>2.2.2 Procedures</td>
<td>39</td>
</tr>
<tr>
<td>2.2.3 The forward Declaration</td>
<td>41</td>
</tr>
<tr>
<td>2.3 Packages</td>
<td>42</td>
</tr>
<tr>
<td>2.3.1 Introduction</td>
<td>42</td>
</tr>
<tr>
<td>2.3.2 Intrinsic</td>
<td>43</td>
</tr>
<tr>
<td>2.3.3 Resolving calls to intrinsics</td>
<td>45</td>
</tr>
<tr>
<td>2.3.4 Attaching and Detaching Package Files</td>
<td>46</td>
</tr>
<tr>
<td>2.3.5 Related Files</td>
<td>47</td>
</tr>
<tr>
<td>2.3.6 Importing Constants</td>
<td>47</td>
</tr>
<tr>
<td>2.3.7 Argument Checking</td>
<td>48</td>
</tr>
<tr>
<td>2.3.8 Package Specification files</td>
<td>49</td>
</tr>
<tr>
<td>2.3.9 User Startup Specification Files</td>
<td>50</td>
</tr>
<tr>
<td>2.4 Attributes</td>
<td>51</td>
</tr>
<tr>
<td>2.4.1 Predefined System Attributes</td>
<td>51</td>
</tr>
<tr>
<td>2.4.2 User-defined Attributes</td>
<td>52</td>
</tr>
<tr>
<td>2.4.3 Accessing Attributes</td>
<td>52</td>
</tr>
<tr>
<td>2.4.4 User-defined Verbose Flags</td>
<td>53</td>
</tr>
<tr>
<td>2.4.5 Examples</td>
<td>53</td>
</tr>
</tbody>
</table>

## 3 INPUT AND OUTPUT

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Introduction</td>
<td>59</td>
</tr>
<tr>
<td>3.2 Character Strings</td>
<td>59</td>
</tr>
<tr>
<td>3.2.1 Representation of Strings</td>
<td>59</td>
</tr>
<tr>
<td>3.2.2 Creation of Strings</td>
<td>60</td>
</tr>
<tr>
<td>3.2.3 Integer-Valued Functions</td>
<td>61</td>
</tr>
<tr>
<td>3.2.4 Character Conversion</td>
<td>61</td>
</tr>
<tr>
<td>3.2.5 Boolean Functions</td>
<td>62</td>
</tr>
<tr>
<td>3.2.6 Parsing Strings</td>
<td>65</td>
</tr>
<tr>
<td>3.3 Printing</td>
<td>66</td>
</tr>
<tr>
<td>3.3.1 The print-Statement</td>
<td>66</td>
</tr>
<tr>
<td>3.3.2 The printf and fprintf Statements</td>
<td>67</td>
</tr>
<tr>
<td>3.3.3 Verbose Printing (vprint, vprintf)</td>
<td>69</td>
</tr>
<tr>
<td>3.3.4 Automatic Printing</td>
<td>70</td>
</tr>
<tr>
<td>3.3.5 Indentation</td>
<td>72</td>
</tr>
<tr>
<td>3.3.6 Printing to a File</td>
<td>72</td>
</tr>
<tr>
<td>3.3.7 Printing to a String</td>
<td>73</td>
</tr>
<tr>
<td>3.3.8 Redirecting Output</td>
<td>74</td>
</tr>
<tr>
<td>3.4 External Files</td>
<td>74</td>
</tr>
<tr>
<td>3.4.1 Opening Files</td>
<td>74</td>
</tr>
<tr>
<td>3.4.2 Operations on File Objects</td>
<td>75</td>
</tr>
<tr>
<td>3.4.3 Reading a Complete File</td>
<td>76</td>
</tr>
<tr>
<td>3.5 Pipes</td>
<td>77</td>
</tr>
<tr>
<td>3.5.1 Pipe Creation</td>
<td>77</td>
</tr>
<tr>
<td>3.5.2 Operations on Pipes</td>
<td>78</td>
</tr>
<tr>
<td>3.6 Sockets</td>
<td>79</td>
</tr>
<tr>
<td>3.6.1 Socket Creation</td>
<td>79</td>
</tr>
<tr>
<td>3.6.2 Socket Properties</td>
<td>80</td>
</tr>
<tr>
<td>3.6.3 Socket Predicates</td>
<td>80</td>
</tr>
<tr>
<td>3.6.4 Socket I/O</td>
<td>81</td>
</tr>
<tr>
<td>3.7 Interactive Input</td>
<td>82</td>
</tr>
<tr>
<td>3.8 Loading a Program File</td>
<td>83</td>
</tr>
</tbody>
</table>
### VOLUME 1: CONTENTS

3.9  Saving and Restoring Workspaces 83  
3.10 Logging a Session 84  
3.11 Memory Usage 84  
3.12 System Calls 84  
3.13 Creating Names 85  

4 ENVIRONMENT AND OPTIONS ................................. 87  
4.1  Introduction 89  
4.2  Command Line Options 89  
4.3  Environment Variables 91  
4.4  Set and Get 92  
4.5  Verbose Levels 96  
4.6  Other Information Procedures 97  
4.7  History 98  
4.8  The Magma Line Editor 100  
4.8.1  Key Bindings (Emacs and VI mode) 100  
4.8.2  Key Bindings in Emacs mode only 102  
4.8.3  Key Bindings in VI mode only 103  
4.9  The Magma Help System 106  
4.9.1  Internal Help Browser 107  

5 MAGMA SEMANTICS .............................................. 109  
5.1  Introduction 111  
5.2  Terminology 111  
5.3  Assignment 112  
5.4  Uninitialized Identifiers 112  
5.5  Evaluation in Magma 113  
5.5.1  Call by Value Evaluation 113  
5.5.2  Magma's Evaluation Process 114  
5.5.3  Function Expressions 115  
5.5.4  Function Values Assigned to Identifiers 116  
5.5.5  Recursion and Mutual Recursion 116  
5.5.6  Function Application 117  
5.5.7  The Initial Context 118  
5.6  Scope 118  
5.6.1  Local Declarations 119  
5.6.2  The ‘first use’ Rule 119  
5.6.3  Identifier Classes 120  
5.6.4  The Evaluation Process Revisited 121  
5.6.5  The ‘single use’ Rule 121  
5.7  Procedure Expressions 121  
5.8  Reference Arguments 123  
5.9  Dynamic Typing 124  
5.10  Traps for Young Players 125  
5.10.1  Trap 1 125  
5.10.2  Trap 2 125  
5.11  Appendix A: Precedence 127  
5.12  Appendix B: Reserved Words 128
6 THE MAGMA PROFILER ........................................... 129
  6.1 Introduction .................................................. 131
  6.2 Profiler Basics .............................................. 131
  6.3 Exploring the Call Graph ................................... 133
      6.3.1 Internal Reports ...................................... 133
      6.3.2 HTML Reports .......................................... 135
  6.4 Recursion and the Profiler .................................. 135

7 DEBUGGING MAGMA CODE ....................................... 139
  7.1 Introduction .................................................. 141
  7.2 Using the Debugger .......................................... 141
## II SETS, SEQUENCES, AND MAPPINGS 145

### 8 INTRODUCTION TO AGGREGATES 147

8.1 Introduction 149
8.2 Restrictions on Sets and Sequences 149
8.2.1 Universe of a Set or Sequence 150
8.2.2 Modifying the Universe of a Set or Sequence 151
8.2.3 Parents of Sets and Sequences 153
8.3 Nested Aggregates 154
8.3.1 Multi-indexing 154

### 9 SETS 157

9.1 Introduction 159
9.1.1 Enumerated Sets 159
9.1.2 Formal Sets 159
9.1.3 Indexed Sets 159
9.1.4 Multisets 159
9.1.5 Compatibility 160
9.1.6 Notation 160
9.2 Creating Sets 160
9.2.1 The Formal Set Constructor 160
9.2.2 The Enumerated Set Constructor 161
9.2.3 The Indexed Set Constructor 163
9.2.4 The Multiset Constructor 164
9.2.5 The Arithmetic Progression Constructors 166
9.3 Power Sets 167
9.3.1 The Cartesian Product Constructors 167
9.4 Sets from Structures 169
9.5 Accessing and Modifying Sets 170
9.5.1 Accessing Sets and their Associated Structures 170
9.5.2 Selecting Elements of Sets 171
9.5.3 Modifying Sets 174
9.6 Operations on Sets 177
9.6.1 Boolean Functions and Operators 177
9.6.2 Binary Set Operators 178
9.6.3 Other Set Operations 179
9.7 Quantifiers 180
9.8 Reduction and Iteration over Sets 183

### 10 SEQUENCES 185

10.1 Introduction 187
10.1.1 Enumerated Sequences 187
10.1.2 Formal Sequences 187
10.1.3 Compatibility 188
10.2 Creating Sequences 188
10.2.1 The Formal Sequence Constructor 188
10.2.2 The Enumerated Sequence Constructor 189
10.2.3 The Arithmetic Progression Constructors 190
10.2.4 Literal Sequences 191
10.3 Power Sequences 191
10.4 Operators on Sequences 192
10.4.1 Access Functions 192
10.4.2 Selection Operators on Enumerated Sequences 193
III BASIC RINGS AND LINEAR ALGEBRA 247

17 INTRODUCTION TO RINGS . . . . . . . . . . . . . . . . . . . . 249
  17.1 Overview 251
  17.2 The World of Rings 252
  17.2.1 New Rings from Existing Ones 252
  17.2.2 Attributes 253
  17.3 Coercion 253
  17.3.1 Automatic Coercion 254
  17.3.2 Forced Coercion 256
  17.4 Generic Ring Functions 257
  17.4.1 Related Structures 258
  17.4.2 Numerical Invariants 258
  17.4.3 Predicates and Boolean Operations 259
  17.5 Generic Element Functions 260
  17.5.1 Parent and Category 260
  17.5.2 Creation of Elements 261
  17.5.3 Arithmetic Operations 261
  17.5.4 Equality and Membership 262
  17.5.5 Predicates on Ring Elements 263
  17.5.6 Comparison of Ring Elements 264
  17.6 Ideals and Quotient Rings 265
  17.6.1 Defining Ideals and Quotient Rings 265
  17.6.2 Arithmetic Operations on Ideals 265
  17.6.3 Boolean Operators on Ideals 266
  17.7 Other Ring Constructions 266
  17.7.1 Residue Class Fields 266
  17.7.2 Localization 266
  17.7.3 Completion 267
  17.7.4 Transcendental Extension 267

18 RING OF INTEGERS . . . . . . . . . . . . . . . . . . . . . . . . . 269
  18.1 Introduction 273
  18.1.1 Representation 273
  18.1.2 Coercion 273
  18.1.3 Homomorphisms 273
  18.2 Creation Functions 274
  18.2.1 Creation of Structures 274
  18.2.2 Creation of Elements 274
  18.2.3 Printing of Elements 275
  18.2.4 Element Conversions 276
  18.3 Structure Operations 277
  18.3.1 Related Structures 277
  18.3.2 Numerical Invariants 278
  18.3.3 Ring Predicates and Booleans 278
  18.4 Element Operations 278
  18.4.1 Arithmetic Operations 278
  18.4.2 Equality and Membership 279
18.4.3 Parent and Category 279
18.4.4 Predicates on Ring Elements 279
18.4.5 Comparison of Ring Elements 281
18.4.6 Conjugates, Norm and Trace 281
18.4.7 Other Elementary Functions 281
18.5 Random Numbers 283
18.6 Common Divisors and Common Multiples 284
18.7 Arithmetic Functions 285
18.8 Combinatorial Functions 287
18.9 Primes and Primality Testing 289
18.9.1 Primality 289
18.9.2 Other Functions Relating to Primes 291
18.10 Factorization 293
18.10.1 General Factorization 293
18.10.2 Storing Potential Factors 295
18.10.3 Specific Factorization Algorithms 296
18.10.4 Factorization Related Functions 300
18.11 Factorization Sequences 301
18.11.1 Creation and Conversion 302
18.11.2 Arithmetic 302
18.11.3 Divisors 302
18.11.4 Predicates 303
18.12 Modular Arithmetic 303
18.12.1 Arithmetic Operations 303
18.12.2 The Solution of Modular Equations 304
18.13 Infinities 305
18.13.1 Creation 305
18.13.2 Arithmetic 306
18.13.3 Comparison 306
18.13.4 Miscellaneous 306
18.14 Advanced Factorization Techniques: The Number Field Sieve 306
18.14.1 The Magma Number Field Sieve implementation 306
18.14.2 Naive NFS 307
18.14.3 Factoring with NFS Processes 308
18.14.4 Data files 312
18.14.5 Distributing NFS factorizations 313
18.14.6 Magma and CWI NFS interoperability 315
18.14.7 Tools for Finding a Suitable Polynomial 315
18.15 Bibliography 318

19 INTEGER RESIDUE CLASS RINGS . . . . . . . . . . . . . . . . . . 319
19.1 Introduction 321
19.2 Ideals of $\mathbb{Z}$ 321
19.3 $\mathbb{Z}$ as a Number Field Order 322
19.4 Residue Class Rings 323
19.4.1 Creation 323
19.4.2 Coercion 324
19.4.3 Elementary Invariants 325
19.4.4 Structure Operations 325
19.4.5 Ring Predicates and Booleans 326
19.4.6 Homomorphisms 326
19.5 Elements of Residue Class Rings 326
19.5.1 Creation 326
19.5.2 Arithmetic Operators 327
19.5.3 Equality and Membership 327
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.5.4</td>
<td>Parent and Category</td>
<td>327</td>
</tr>
<tr>
<td>19.5.5</td>
<td>Predicates on Ring Elements</td>
<td>327</td>
</tr>
<tr>
<td>19.5.6</td>
<td>Solving Equations over $\mathbb{Z}/m\mathbb{Z}$</td>
<td>327</td>
</tr>
<tr>
<td>19.6</td>
<td>Ideal Operations</td>
<td>329</td>
</tr>
<tr>
<td>19.7</td>
<td>The Unit Group</td>
<td>330</td>
</tr>
<tr>
<td>19.8</td>
<td>Dirichlet Characters</td>
<td>331</td>
</tr>
<tr>
<td>19.8.1</td>
<td>Creation</td>
<td>332</td>
</tr>
<tr>
<td>19.8.2</td>
<td>Element Creation</td>
<td>332</td>
</tr>
<tr>
<td>19.8.3</td>
<td>Properties of Dirichlet Groups</td>
<td>333</td>
</tr>
<tr>
<td>19.8.4</td>
<td>Properties of Elements</td>
<td>334</td>
</tr>
<tr>
<td>19.8.5</td>
<td>Evaluation</td>
<td>335</td>
</tr>
<tr>
<td>19.8.6</td>
<td>Arithmetic</td>
<td>336</td>
</tr>
<tr>
<td>19.8.7</td>
<td>Example</td>
<td>336</td>
</tr>
<tr>
<td>20</td>
<td>RATIONAL FIELD</td>
<td>339</td>
</tr>
<tr>
<td>20.1</td>
<td>Introduction</td>
<td>341</td>
</tr>
<tr>
<td>20.1.1</td>
<td>Representation</td>
<td>341</td>
</tr>
<tr>
<td>20.1.2</td>
<td>Coercion</td>
<td>341</td>
</tr>
<tr>
<td>20.1.3</td>
<td>Homomorphisms</td>
<td>342</td>
</tr>
<tr>
<td>20.2</td>
<td>Creation Functions</td>
<td>343</td>
</tr>
<tr>
<td>20.2.1</td>
<td>Creation of Structures</td>
<td>343</td>
</tr>
<tr>
<td>20.2.2</td>
<td>Creation of Elements</td>
<td>343</td>
</tr>
<tr>
<td>20.3</td>
<td>Structure Operations</td>
<td>344</td>
</tr>
<tr>
<td>20.3.1</td>
<td>Related Structures</td>
<td>344</td>
</tr>
<tr>
<td>20.3.2</td>
<td>Numerical Invariants</td>
<td>346</td>
</tr>
<tr>
<td>20.3.3</td>
<td>Ring Predicates and Booleans</td>
<td>346</td>
</tr>
<tr>
<td>20.4</td>
<td>Element Operations</td>
<td>347</td>
</tr>
<tr>
<td>20.4.1</td>
<td>Parent and Category</td>
<td>347</td>
</tr>
<tr>
<td>20.4.2</td>
<td>Arithmetic Operators</td>
<td>347</td>
</tr>
<tr>
<td>20.4.3</td>
<td>Numerator and Denominator</td>
<td>347</td>
</tr>
<tr>
<td>20.4.4</td>
<td>Equality and Membership</td>
<td>347</td>
</tr>
<tr>
<td>20.4.5</td>
<td>Predicates on Ring Elements</td>
<td>348</td>
</tr>
<tr>
<td>20.4.6</td>
<td>Comparison</td>
<td>348</td>
</tr>
<tr>
<td>20.4.7</td>
<td>Conjugates, Norm and Trace</td>
<td>348</td>
</tr>
<tr>
<td>20.4.8</td>
<td>Absolute Value and Sign</td>
<td>349</td>
</tr>
<tr>
<td>20.4.9</td>
<td>Rounding and Truncating</td>
<td>349</td>
</tr>
<tr>
<td>20.4.10</td>
<td>Rational Reconstruction</td>
<td>350</td>
</tr>
<tr>
<td>20.4.11</td>
<td>Valuation</td>
<td>350</td>
</tr>
<tr>
<td>20.4.12</td>
<td>Sequence Conversions</td>
<td>350</td>
</tr>
<tr>
<td>21</td>
<td>FINITE FIELDS</td>
<td>351</td>
</tr>
<tr>
<td>21.1</td>
<td>Introduction</td>
<td>353</td>
</tr>
<tr>
<td>21.1.1</td>
<td>Representation of Finite Fields</td>
<td>353</td>
</tr>
<tr>
<td>21.1.2</td>
<td>Conway Polynomials</td>
<td>353</td>
</tr>
<tr>
<td>21.1.3</td>
<td>Ground Field and Relationships</td>
<td>354</td>
</tr>
<tr>
<td>21.2</td>
<td>Creation Functions</td>
<td>354</td>
</tr>
<tr>
<td>21.2.1</td>
<td>Creation of Structures</td>
<td>354</td>
</tr>
<tr>
<td>21.2.2</td>
<td>Creating Relations</td>
<td>358</td>
</tr>
<tr>
<td>21.2.3</td>
<td>Special Options</td>
<td>358</td>
</tr>
<tr>
<td>21.2.4</td>
<td>Homomorphisms</td>
<td>360</td>
</tr>
<tr>
<td>21.2.5</td>
<td>Creation of Elements</td>
<td>360</td>
</tr>
<tr>
<td>21.2.6</td>
<td>Special Elements</td>
<td>361</td>
</tr>
<tr>
<td>21.2.7</td>
<td>Sequence Conversions</td>
<td>362</td>
</tr>
<tr>
<td>21.3</td>
<td>Structure Operations</td>
<td>362</td>
</tr>
<tr>
<td>21.3.1</td>
<td>Related Structures</td>
<td>363</td>
</tr>
</tbody>
</table>
VOLUME 2: CONTENTS

21.3.2 Numerical Invariants 365
21.3.3 Defining Polynomial 365
21.3.4 Ring Predicates and Booleans 365
21.3.5 Roots 366
21.4 Element Operations 367
21.4.1 Arithmetic Operators 367
21.4.2 Equality and Membership 367
21.4.3 Parent and Category 367
21.4.4 Predicates on Ring Elements 368
21.4.5 Minimal and Characteristic Polynomial 368
21.4.6 Norm, Trace and Frobenius 369
21.4.7 Order and Roots 370
21.5 Polynomials for Finite Fields 372
21.6 Discrete Logarithms 373
21.7 Permutation Polynomials 376
21.8 Bibliography 378

22 UNIVARIATE POLYNOMIAL RINGS 379

22.1 Introduction 383
22.1.1 Representation 383
22.2 Creation Functions 383
22.2.1 Creation of Structures 383
22.2.2 Print Options 384
22.2.3 Creation of Elements 385
22.3 Structure Operations 387
22.3.1 Related Structures 387
22.3.2 Changing Rings 387
22.3.3 Numerical Invariants 388
22.3.4 Ring Predicates and Booleans 388
22.3.5 Homomorphisms 388
22.4 Element Operations 389
22.4.1 Parent and Category 389
22.4.2 Arithmetic Operators 389
22.4.3 Equality and Membership 389
22.4.4 Predicates on Ring Elements 390
22.4.5 Coefficients and Terms 390
22.4.6 Degree 391
22.4.7 Roots 391
22.4.8 Derivative, Integral 393
22.4.9 Evaluation, Interpolation 394
22.4.10 Quotient and Remainder 394
22.4.11 Modular Arithmetic 395
22.4.12 Other operations 395
22.5 Common Divisors and Common Multiples 396
22.5.1 Common Divisors and Common Multiples 396
22.5.2 Content and Primitive Part 397
22.6 Polynomials over the Integers 398
22.7 Polynomials over Finite Fields 398
22.8 Factorization 399
22.8.1 Factorization and Irreducibility 399
22.8.2 Resultant and Discriminant 403
22.8.3 Hensel Lifting 404
22.9 Ideals and Quotient Rings 405
22.9.1 Creation of Ideals and Quotients 405
22.9.2 Ideal Arithmetic 405
22.9.3 Other Functions on Ideals 406
22.9.4 Other Functions on Quotients 407
22.10 Special Families of Polynomials 407
22.10.1 Orthogonal Polynomials 407
22.10.2 Permutation Polynomials 408
22.10.3 The Bernoulli Polynomial 409
22.10.4 Swinnerton-Dyer Polynomials 409
22.11 Bibliography 409

23 MULTIVARIATE POLYNOMIAL RINGS . . . . . . . . . . 411
23.1 Introduction 413
23.1.1 Representation 413
23.2 Polynomial Rings and Polynomials 414
23.2.1 Creation of Polynomial Rings 414
23.2.2 Print Names 416
23.2.3 Graded Polynomial Rings 416
23.2.4 Creation of Polynomials 417
23.3 Structure Operations 417
23.3.1 Related Structures 417
23.3.2 Numerical Invariants 418
23.3.3 Ring Predicates and Booleans 418
23.3.4 Changing Coefficient Ring 418
23.3.5 Homomorphisms 418
23.4 Element Operations 419
23.4.1 Arithmetic Operators 419
23.4.2 Equality and Membership 419
23.4.3 Predicates on Ring Elements 420
23.4.4 Coefficients, Monomials and Terms 420
23.4.5 Degrees 425
23.4.6 Univariate Polynomials 426
23.4.7 Derivative, Integral 427
23.4.8 Evaluation, Interpolation 428
23.4.9 Quotient and Reductum 429
23.4.10 Diagonalizing a Polynomial of Degree 2 429
23.5 Greatest Common Divisors 430
23.5.1 Common Divisors and Common Multiples 430
23.5.2 Content and Primitive Part 432
23.6 Factorization and Irreducibility 432
23.7 Resultants and Discriminants 436
23.8 Polynomials over the Integers 437
23.9 Bibliography 437

24 REAL AND COMPLEX FIELDS . . . . . . . . . . . . . . 439
24.1 Introduction 443
24.1.1 Overview of Real Numbers in Magma 443
24.1.2 Coercion 444
24.1.3 Homomorphisms 445
24.1.4 Special Options 445
24.2 Creation Functions 446
24.2.1 Creation of Structures 446
24.2.2 Creation of Elements 447
24.3 Structure Operations 449
24.3.1 Related Structures 449
24.3.2 Numerical Invariants 449
24.3.3 Ring Predicates and Booleans 449
24.3.4 Other Structure Functions 449
24.4 Element Operations 450
24.4.1 Generic Element Functions and Predicates 450
24.4.2 Comparison of and Membership 450
24.4.3 Other Predicates 450
24.4.4 Arithmetic 451
24.4.5 Conversions 451
24.4.6 Rounding 452
24.4.7 Precision 452
24.4.8 Constants 453
24.4.9 Simple Element Functions 453
24.4.10 Roots 454
24.4.11 Continued Fractions 459
24.4.12 Algebraic Dependencies 460
24.5 Transcendental Functions 460
24.5.1 Exponential, Logarithmic and Polylogarithmic Functions 460
24.5.2 Trigonometric Functions 462
24.5.3 Inverse Trigonometric Functions 464
24.5.4 Hyperbolic Functions 466
24.5.5 Inverse Hyperbolic Functions 467
24.6 Elliptic and Modular Functions 468
24.6.1 Eisenstein Series 468
24.6.2 Weierstrass Series 470
24.6.3 The Jacobi $\theta$ and Dedekind $\eta$-functions 471
24.6.4 The $j$-invariant and the Discriminant 472
24.6.5 Weber’s Functions 473
24.7 Theta Functions 474
24.8 Gamma, Bessel and Associated Functions 475
24.9 The Hypergeometric Function 477
24.10 Other Special Functions 478
24.11 Numerical Functions 480
24.11.1 Summation of Infinite Series 480
24.11.2 Integration 480
24.12 Bibliography 481

25 MATRICES 483

25.1 Introduction 485
25.2 Creation of Matrices 485
25.2.1 General Matrix Construction 485
25.2.2 Shortcuts 487
25.2.3 Construction of Structured Matrices 489
25.2.4 Construction of Random Matrices 492
25.2.5 Creating Vectors 493
25.3 Elementary Properties 493
25.4 Accessing or Modifying Entries 494
25.4.1 Indexing 494
25.4.2 Extracting and Inserting Blocks 495
25.4.3 Row and Column Operations 498
25.5 Building Block Matrices 500
25.6 Changing Ring 502
25.7 Elementary Arithmetic 502
25.8 Nullspaces and Solutions of Systems 503
25.9 Predicates 506
25.10 Determinant and Other Properties 507
25.11 Minimal and Characteristic Polynomials and Eigenvalues 508
25.12 Canonical Forms
25.12.1 Canonical Forms over General Rings 510
25.12.2 Canonical Forms over Fields 511
25.12.3 Canonical Forms over Euclidean Domains 514
25.13 Orders of Invertible Matrices 516
25.14 Miscellaneous Operations on Matrices 518
25.15 Bibliography 519

26 SPARSE MATRICES . . . . . . . . . . . . . . . . . . . . . 521
26.1 Introduction 523
26.2 Creation of Sparse Matrices 523
26.2.1 Construction of Initialized Sparse Matrices 523
26.2.2 Construction of Trivial Sparse Matrices 524
26.3 Accessing Sparse Matrices 526
26.3.1 Elementary Properties 526
26.3.2 Weights 526
26.3.3 Predicates 527
26.4 Accessing or Modifying Entries 527
26.5 Conversion to and from Dense Matrices 529
26.6 Changing Ring 530
26.7 Multiplying Vectors or Matrices by Sparse Matrices 530
26.8 Non-trivial Properties 530
26.8.1 Nullspace 531
26.8.2 Rank 531
26.8.3 Elementary Divisors (Smith Form) 531
26.8.4 Verbosity 532
26.9 Linear Systems (Structured Gaussian Elimination) 532
26.10 Bibliography 539

27 VECTOR SPACES . . . . . . . . . . . . . . . . . . . . . . . . 541
27.1 Introduction 543
27.1.1 Vector Space Categories 543
27.1.2 The Construction of a Vector Space 543
27.2 Creation of Vector Spaces and Arithmetic with Vectors 544
27.2.1 Construction of a Vector Space 544
27.2.2 Construction of a Vector Space with Inner Product Matrix 545
27.2.3 Construction of a Vector 545
27.2.4 Deconstruction of a Vector 547
27.2.5 Arithmetic with Vectors 547
27.2.6 Indexing Vectors and Matrices 550
27.3 Subspaces, Quotient Spaces and Homomorphisms 552
27.3.1 Construction of Subspaces 552
27.3.2 Construction of Quotient Vector Spaces 554
27.4 Changing the Coefficient Field 556
27.5 Basic Operations 557
27.5.1 Accessing Vector Space Invariants 557
27.5.2 Membership and Equality 558
27.5.3 Operations on Subspaces 559
27.6 Reducing Vectors Relative to a Subspace 559
27.7 Bases 560
27.8 Operations with Linear Transformations 562
VOLUME 3: CONTENTS

IV LATTICES AND QUADRATIC FORMS 565

28 LATTICES .......................................................... 567

28.1 Introduction ...................................................... 571
28.2 Presentation of Lattices ......................................... 572
28.3 Creation of Lattices ............................................. 573
    28.3.1 Elementary Creation of Lattices ....................... 573
    28.3.2 Lattices from Linear Codes ......................... 577
    28.3.3 Lattices from Algebraic Number Fields ............ 578
    28.3.4 Special Lattices ........................................ 580
28.4 Lattice Elements .............................................. 581
    28.4.1 Creation of Lattice Elements ......................... 581
    28.4.2 Operations on Lattice Elements ...................... 581
    28.4.3 Predicates and Boolean Operations .................. 583
    28.4.4 Access Operations ...................................... 583
28.5 Properties of Lattices ....................................... 585
    28.5.1 Associated Structures .................................. 585
    28.5.2 Attributes of Lattices ................................ 585
    28.5.3 Predicates and Booleans on Lattices ............... 587
    28.5.4 Base Ring and Base Change ............................ 587
28.6 Construction of New Lattices ............................... 588
    28.6.1 Sub- and Superlattices and Quotients ............. 588
    28.6.2 Standard Constructions of New Lattices .......... 590
28.7 Reduction of Matrices and Lattices ....................... 593
    28.7.1 LLL Reduction ......................................... 593
    28.7.2 Pair Reduction ........................................ 602
    28.7.3 Seyesen Reduction ..................................... 603
28.8 Minima and Element Enumeration .......................... 605
    28.8.1 Minimum, Density and Kissing Number .............. 605
    28.8.2 Shortest and Closest Vectors ....................... 606
    28.8.3 Short and Close Vectors ............................. 609
    28.8.4 Short and Close Vector Processes ................... 614
    28.8.5 Successive Minima and Theta Series ............... 615
28.9 Voronoi Cells, Holes and Covering Radius .............. 616
28.10 Orthogonalization ........................................... 618
28.11 Testing Matrices for Definiteness ....................... 620
28.12 Automorphism Group and Isometry Testing ............. 621
28.13 Genera and Spinor Genera ................................. 628
    28.13.1 Genus Constructions .................................. 628
    28.13.2 Invariants of Genera and Spinor Genera .......... 629
    28.13.3 Invariants of p-adic Genera ...................... 630
    28.13.4 Neighbour Relations and Graphs .................... 630
28.14 Attributes of Lattices .................................... 634
    28.14.1 Attributes of Lattices ............................... 634
    28.14.2 Lattices from Matrix Groups ....................... 634
    28.14.3 Related Operations on Matrix Groups ............. 636
    28.14.4 Invariant Forms ...................................... 636
    28.14.5 Endomorphisms ....................................... 640
V GLOBAL ARITHMETIC FIELDS

31 ORDERS AND ALGEBRAIC FIELDS

31.1 Introduction

31.2 Creation Functions

31.2.1 Creation of General Algebraic Fields

31.2.2 Creation of Orders and Fields from Orders

31.2.3 Orders and Ideals

31.2.4 Creation of Elements

31.2.5 Creation of Homomorphisms

31.3 Special Options

31.4 Structure Operations

31.4.1 General Functions

31.4.2 Related Structures

31.4.3 Representing Fields as Vector Spaces

31.4.4 Invariants

31.4.5 Basis Representation

31.4.6 Ring Predicates

31.4.7 Order Predicates

31.4.8 Field Predicates

31.4.9 Setting Properties of Orders

31.5 Element Operations

31.5.1 Parent and Category

31.5.2 Arithmetic

31.5.3 Equality and Membership

31.5.4 Predicates on Elements

31.5.5 Finding Special Elements

31.5.6 Real and Complex Valued Functions

31.5.7 Norm, Trace, and Minimal Polynomial

31.5.8 Other Functions

31.6 Ideal Class Groups

31.6.1 Setting the Class Group Bounds Globally

31.7 Unit Groups

31.8 Solving Equations

31.8.1 Norm Equations

31.8.2 Thue Equations

31.8.3 Unit Equations

31.8.4 Index Form Equations

31.9 Ideals and Quotients

31.9.1 Creation of Ideals in Orders

31.9.2 Invariants

31.9.3 Basis Representation

31.9.4 Two–Element Presentations

31.9.5 Predicates on Ideals

31.9.6 Ideal Arithmetic

31.9.7 Roots of Ideals

31.9.8 Factorization and Primes

31.9.9 Other Ideal Operations

31.9.10 Quotient Rings

31.10 Places and Divisors

31.10.1 Creation of Structures

31.10.2 Operations on Structures

31.10.3 Creation of Elements

31.10.4 Arithmetic with Places and Divisors

31.10.5 Other Functions for Places and Divisors

31.11 Bibliography
32  QUADRATIC FIELDS ........................................... 773
32.1  Introduction ................................................. 775
32.1.1  Representation .......................................... 775
32.2  Creation of Structures ................................. 776
32.3  Operations on Structures ............................... 777
32.3.1  Ideal Class Group .................................... 778
32.3.2  Norm Equations ....................................... 780
32.4  Special Element Operations ............................... 781
32.4.1  Greatest Common Divisors ............................. 781
32.4.2  Modular Arithmetic .................................... 781
32.4.3  Factorization .......................................... 782
32.4.4  Conjugates .............................................. 782
32.4.5  Other Element Functions ............................... 782
32.5  Special Functions for Ideals .............................. 784
32.6  Bibliography ............................................... 784

33  CYCLOTOmic FIELDS ........................................... 785
33.1  Introduction ................................................ 787
33.2  Creation Functions ....................................... 787
33.2.1  Creation of Cyclotomic Fields ....................... 787
33.2.2  Creation of Elements .................................. 788
33.3  Structure Operations ................................... 789
33.3.1  Invariants .............................................. 790
33.4  Element Operations ...................................... 790
33.4.1  Predicates on Elements ................................. 790
33.4.2  Conjugates .............................................. 790

34  GALOIS THEORY OF NUMBER FIELDS ................. 793
34.1  Automorphism Groups ..................................... 796
34.2  Galois Groups .............................................. 802
34.2.1  Invariants .............................................. 808
34.2.2  Subfields and Subfield towers ...................... 810
34.2.3  Solvability by Radicals ............................... 816
34.2.4  Linear Relations ....................................... 818
34.2.5  Other .................................................... 820
34.3  Subfields ..................................................... 821
34.3.1  The Subfield Lattice .................................. 821
34.4  Galois Cohomology ......................................... 824
34.5  Bibliography ............................................... 825

35  CLASS FIELD THEORY ........................................ 827
35.1  Introduction ............................................... 829
35.1.1  Overview ................................................ 829
35.1.2  MAGMA .................................................. 830
35.2  Creation ...................................................... 833
35.2.1  Ray Class Groups ..................................... 833
35.2.2  Maps ...................................................... 838
35.2.3  Abelian Extensions ..................................... 839
35.2.4  Binary Operations ..................................... 844
35.3  Galois Module Structure ................................. 844
35.3.1  Predicates .............................................. 845
35.3.2  Constructions .......................................... 845
35.4 Conversion to Number Fields 846
35.5 Invariants 847
35.6 Automorphisms 849
35.7 Norm Equations 852
35.8 Attributes 855
35.8.1 Orders 855
35.8.2 Abelian Extensions 858
35.9 Group Theoretic Functions 862
35.9.1 Generic Groups 862
35.10 Bibliography 863

36 ALGEBRAICALLY CLOSED FIELDS . . . . . . . . . . . 865
36.1 Introduction 867
36.2 Representation 867
36.3 Creation of Structures 868
36.4 Creation of Elements 869
36.4.1 Coercion 869
36.4.2 Roots 869
36.4.3 Variables 870
36.5 Related Structures 875
36.6 Properties 875
36.7 Ring Predicates and Properties 876
36.8 Element Operations 876
36.8.1 Arithmetic Operators 877
36.8.2 Equality and Membership 877
36.8.3 Parent and Category 877
36.8.4 Predicates on Ring Elements 877
36.8.5 Minimal Polynomial, Norm and Trace 878
36.9 Simplification 880
36.10 Absolute Field 881
36.11 Bibliography 885

37 RATIONAL FUNCTION FIELDS . . . . . . . . . . . . 887
37.1 Introduction 889
37.2 Creation Functions 889
37.2.1 Creation of Structures 889
37.2.2 Names 890
37.2.3 Creation of Elements 891
37.3 Structure Operations 891
37.3.1 Related Structures 891
37.3.2 Invariants 892
37.3.3 Ring Predicates and Booleans 892
37.3.4 Homomorphisms 892
37.4 Element Operations 893
37.4.1 Arithmetic 893
37.4.2 Equality and Membership 893
37.4.3 Numerator, Denominator and Degree 894
37.4.4 Predicates on Ring Elements 894
37.4.5 Evaluation 894
37.4.6 Derivative 895
37.4.7 Partial Fraction Decomposition 895
38.1 Introduction 905
38.1.1 Representations of Fields 905
38.2 Creation of Algebraic Function Fields and their Orders 906
38.2.1 Creation of Algebraic Function Fields 906
38.2.2 Creation of Orders of Algebraic Function Fields 909
38.2.3 Orders and Ideals 913
38.3 Related Structures 914
38.3.1 Parent and Category 914
38.3.2 Other Related Structures 914
38.4 General Structure Invariants 919
38.4.1 Galois Groups 923
38.4.2 Subfields 927
38.4.3 Automorphism Group 928
38.5 Global Function Fields 936
38.5.1 Functions relative to the Exact Constant Field 936
38.5.2 Functions Relative to the Constant Field 937
38.5.3 Functions related to Class Group 939
38.6 Structure Predicates 942
38.7 Homomorphisms 944
38.8 Elements 945
38.8.1 Creation of Elements 945
38.8.2 Parent and Category 947
38.8.3 Sequence Conversions 947
38.8.4 Arithmetic Operators 948
38.8.5 Equality and Membership 949
38.8.6 Predicates on Elements 949
38.8.7 Functions related to Norm and Trace 950
38.8.8 Functions related to Orders and Integrality 951
38.8.9 Functions related to Places and Divisors 952
38.8.10 Other Operations on Elements 955
38.9 Ideals 958
38.9.1 Creation of Ideals 958
38.9.2 Parent and Category 959
38.9.3 Arithmetic Operators 959
38.9.4 Roots of Ideals 959
38.9.5 Equality and Membership 961
38.9.6 Predicates on Ideals 961
38.9.7 Further Ideal Operations 963
38.10 Places 969
38.10.1 Creation of Structures 969
38.10.2 Creation of Elements 969
38.10.3 Related Structures 971
38.10.4 Structure Invariants 971
38.10.5 Structure Predicates 972
38.10.6 Element Operations 972
38.10.7 Completion at Places 974
38.11 Divisors 975
38.11.1 Creation of Structures 975
38.11.2 Creation of Elements 975
38.11.3 Related Structures 975
38.11.4 Structure Invariants 976
38.11.5 Structure Predicates 976
38.11.6 Element Operations 976
38.11.7 Functions related to Divisor Class Groups of Global Function Fields 986
38.12 Differentials 992
38.12.1 Creation of Structures 992
38.12.2 Creation of Elements 992
38.12.3 Related Structures 992
38.12.4 Subspaces 992
38.12.5 Structure Predicates 994
38.12.6 Operations on Elements 994
38.13 Weil Descent 998
38.14 Bibliography 1000

39 CLASS FIELD THEORY FOR GLOBAL FUNCTION FIELDS 1003
39.1 Ray Class Groups 1005
39.2 Creation of Class Fields 1008
39.3 Properties of Class Fields 1010
39.4 The Ring of Witt Vectors of finite length 1013
39.5 The Ring of Twisted Polynomials 1015
39.5.1 Creation of twisted polynomial rings 1015
39.5.2 Operations with the ring of twisted polynomials 1016
39.5.3 Creation of twisted polynomials 1016
39.5.4 Operations with Twisted polynomials 1018
39.6 Analytic Theory 1019
39.7 Related Functions 1025
39.8 Enumeration of Places 1027
39.9 Bibliography 1028
VI  LOCAL ARITHMETIC FIELDS  1029

40  VALUATION RINGS  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1031
40.1  Introduction  1033
40.2  Creation Functions  1033
40.2.1  Creation of Structures  1033
40.2.2  Creation of Elements  1033
40.3  Structure Operations  1034
40.3.1  Related Structures  1034
40.3.2  Numerical Invariants  1034
40.4  Element Operations  1034
40.4.1  Arithmetic Operations  1034
40.4.2  Equality and Membership  1034
40.4.3  Parent and Category  1034
40.4.4  Predicates on Ring Elements  1035
40.4.5  Other Element Functions  1035

41  NEWTON POLYGONS  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1037
41.1  Introduction  1039
41.2  Newton Polygons  1041
41.2.1  Creation of Newton Polygons  1041
41.2.2  Vertices and Faces of Polygons  1043
41.2.3  Tests for Points and Faces  1047
41.3  Polynomials Associated with Newton Polygons  1048
41.4  Finding Valuations of Roots of Polynomials from Newton Polygons  1049
41.5  Using Newton Polygons to Find Roots of Polynomials over Series Rings  1049
41.5.1  Operations not associated with Duval’s Algorithm  1049
41.5.2  Operations associated with Duval’s algorithm  1055
41.5.3  Roots of Polynomials  1061
41.6  Bibliography  1063

42  p-ADIC RINGS AND THEIR EXTENSIONS  . . . . . . . . . . . . . . . . . . . . . . . 1065
42.1  Introduction  1069
42.2  Background  1069
42.3  Overview of the p-adics in MAGMA  1070
42.3.1  p-adic Rings  1070
42.3.2  p-adic Fields  1070
42.3.3  Free Precision Rings and Fields  1071
42.3.4  Precision of Extensions  1071
42.4  Creation of Local Rings and Fields  1071
42.4.1  Creation Functions for the p-adics  1071
42.4.2  Creation Functions for Unramified Extensions  1073
42.4.3  Creation Functions for Totally Ramified Extensions  1075
42.4.4  Creation Functions for Unbounded Precision Extensions  1076
42.4.5  Miscellaneous Creation Functions  1077
42.4.6  Other Elementary Constructions  1078
42.4.7 Attributes of Local Rings and Fields
42.5 Elementary Invariants
42.6 Operations on Structures
42.7 Element Constructions and Conversions
42.7.1 Constructions
42.7.2 Element Decomposers
42.8 Operations on Elements
42.8.1 Arithmetic
42.8.2 Equality and Membership
42.8.3 Properties
42.8.4 Precision and Valuation
42.8.5 Logarithms and Exponentials
42.8.6 Norm and Trace Functions
42.8.7 Teichmüller Lifts
42.9 Linear Algebra
42.10 Roots of Elements
42.11 Polynomials
42.11.1 Operations for Polynomials
42.11.2 Roots of Polynomials
42.11.3 Factorization
42.12 Automorphisms of Local Rings and Fields
42.13 Completions
42.14 Class Field Theory
42.14.1 Unit Group
42.14.2 Norm Group
42.14.3 Class Fields
42.15 Extensions
42.16 Bibliography

43 GALOIS RINGS

43.1 Introduction
43.2 Creation Functions
43.2.1 Creation of Structures
43.2.2 Names
43.2.3 Creation of Elements
43.2.4 Sequence Conversions
43.3 Structure Operations
43.3.1 Related Structures
43.3.2 Numerical Invariants
43.3.3 Ring Predicates and Booleans
43.4 Element Operations
43.4.1 Arithmetic Operators
43.4.2 Euclidean Operations
43.4.3 Equality and Membership
43.4.4 Parent and Category
43.4.5 Predicates on Ring Elements
POWER, LAURENT AND PUISEUX SERIES . . . . . . . . 1121

44.1 Introduction 1123
44.1.1 Kinds of Series 1123
44.1.2 Puiseux Series 1123
44.1.3 Representation of Series 1124
44.1.4 Precision 1124
44.1.5 Free and Fixed Precision 1124
44.1.6 Equality 1125
44.1.7 Polynomials over Series Rings 1125
44.2 Creation Functions 1125
44.2.1 Creation of Structures 1125
44.2.2 Special Options 1127
44.2.3 Creation of Elements 1128
44.3 Structure Operations 1129
44.3.1 Related Structures 1129
44.3.2 Invariants 1130
44.3.3 Ring Predicates and Booleans 1130
44.4 Basic Element Operations 1130
44.4.1 Parent and Category 1130
44.4.2 Arithmetic Operators 1130
44.4.3 Equality and Membership 1131
44.4.4 Predicates on Ring Elements 1131
44.4.5 Precision 1131
44.4.6 Coefficients and Degree 1132
44.4.7 Evaluation and Derivative 1133
44.4.8 Square Root 1134
44.4.9 Composition and Reversion 1134
44.5 Transcendental Functions 1136
44.5.1 Exponential and Logarithmic Functions 1136
44.5.2 Trigonometric Functions and their Inverses 1138
44.5.3 Hyperbolic Functions and their Inverses 1138
44.6 The Hypergeometric Series 1139
44.7 Polynomials over Series Rings 1139
44.8 Extensions of Series Rings 1140
44.8.1 Constructions of Extensions 1140
44.8.2 Operations on Extensions 1141
44.8.3 Elements of Extensions 1142
44.8.4 Optimized Representation 1142
44.9 Bibliography 1142

LAZY POWER SERIES RINGS . . . . . . . . . . . . . . 1143

45.1 Introduction 1145
45.2 Creation of Lazy Series Rings 1146
45.3 Functions on Lazy Series Rings 1146
45.4 Elements 1147
45.4.1 Creation of Finite Lazy Series 1147
45.4.2 Arithmetic with Lazy Series 1150
45.4.3 Finding Coefficients of Lazy Series 1151
45.4.4 Predicates on Lazy Series 1154
45.4.5 Other Functions on Lazy Series 1155
# VOLUME 4: CONTENTS

## VII  MODULES

### 46  INTRODUCTION TO MODULES  
- 46.1  Overview  
- 46.2  General Modules  
- 46.3  The Presentation of Submodules  

### 47  FREE MODULES  
- 47.1  Introduction  
  - 47.1.1  Free Modules  
  - 47.1.2  Module Categories  
  - 47.1.3  Presentation of Submodules  
  - 47.1.4  Notation  
- 47.2  Definition of a Module  
  - 47.2.1  Construction of Modules of \( n \)-tuples  
  - 47.2.2  Construction of Modules of \( m \times n \) Matrices  
- 47.3  Accessing Module Information  
- 47.4  Standard Constructions  
  - 47.4.1  Changing the Coefficient Ring  
  - 47.4.2  Direct Sums  
- 47.5  Elements  
  - 47.6.1  Deconstruction of Elements  
  - 47.6.2  Operations on Module Elements  
  - 47.6.3  Properties of Vectors  
  - 47.6.4  Inner Products  
- 47.7  Bases  
- 47.8  Submodules  
  - 47.8.1  Construction of Submodules  
  - 47.8.2  Operations on Submodules  
  - 47.8.3  Membership and Equality  
  - 47.8.4  Operations on Submodules  
- 47.9  Quotient Modules  
- 47.10  Homomorphisms  
  - 47.10.1  \( \text{Hom}_R(M, N) \) for \( R \)-modules  
  - 47.10.2  \( \text{Hom}_R(M, N) \) for Matrix Modules  
  - 47.10.3  Modules \( \text{Hom}_R(M, N) \) with Given Basis  
  - 47.10.4  The Endomorphism Ring  
  - 47.10.5  The Reduced Form of a Matrix Module  
  - 47.10.6  Construction of a Matrix  
  - 47.10.7  Element Operations  

### 48  MODULES OVER DEDEKIND DOMAINS  
- 48.1  Introduction  
- 48.2  Creation of Modules  
- 48.3  Elementary Functions  
- 48.4  Predicates on Modules  
- 48.5  Arithmetic with Modules  
- 48.6  Basis of a Module  
- 48.7  Other Functions on Modules  

**Page Numbers:**
- Introduction to Modules: 1161
- Overview: 1163
- General Modules: 1163
- The Presentation of Submodules: 1164
- Free Modules: 1165
- Introduction: 1167
- Free Modules: 1167
- Module Categories: 1167
- Presentation of Submodules: 1168
- Notation: 1168
- Definition of a Module: 1168
- Construction of Modules of \( n \)-tuples: 1168
- Construction of Modules of \( m \times n \) Matrices: 1169
- Construction of a Module with Specified Basis: 1169
- Accessing Module Information: 1169
- Standard Constructions: 1170
- Changing the Coefficient Ring: 1170
- Direct Sums: 1170
- Elements: 1170
- Construction of Elements: 1171
- Deconstruction of Elements: 1172
- Operations on Module Elements: 1172
- Properties of Vectors: 1174
- Inner Products: 1174
- Bases: 1175
- Submodules: 1175
- Construction of Submodules: 1175
- Operations on Submodules: 1176
- Membership and Equality: 1176
- Operations on Submodules: 1177
- Quotient Modules: 1177
- Construction of Quotient Modules: 1177
- Homomorphisms: 1178
- \( \text{Hom}_R(M, N) \) for \( R \)-modules: 1178
- \( \text{Hom}_R(M, N) \) for Matrix Modules: 1179
- Modules \( \text{Hom}_R(M, N) \) with Given Basis: 1181
- The Endomorphism Ring: 1181
- The Reduced Form of a Matrix Module: 1182
- Construction of a Matrix: 1185
- Element Operations: 1186
- Modules over Dedekind Domains: 1189
- Introduction: 1191
- Creation of Modules: 1192
- Elementary Functions: 1198
- Predicates on Modules: 1200
- Arithmetic with Modules: 1200
- Basis of a Module: 1202
- Other Functions on Modules: 1203
### VOLUME 4: CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>48.8</td>
<td>Homomorphisms between Modules</td>
<td>1205</td>
</tr>
<tr>
<td>48.9</td>
<td>Elements of Modules</td>
<td>1207</td>
</tr>
<tr>
<td>48.9.1</td>
<td>Creation of Elements</td>
<td>1207</td>
</tr>
<tr>
<td>48.9.2</td>
<td>Arithmetic with Elements</td>
<td>1208</td>
</tr>
<tr>
<td>48.9.3</td>
<td>Other Functions on Elements</td>
<td>1209</td>
</tr>
<tr>
<td>48.10</td>
<td>Pseudo Matrices</td>
<td>1209</td>
</tr>
<tr>
<td>48.10.1</td>
<td>Construction of a Pseudo Matrix</td>
<td>1209</td>
</tr>
<tr>
<td>48.10.2</td>
<td>Elementary Functions</td>
<td>1209</td>
</tr>
<tr>
<td>48.10.3</td>
<td>Basis of a Pseudo Matrix</td>
<td>1210</td>
</tr>
<tr>
<td>48.10.4</td>
<td>Predicates</td>
<td>1210</td>
</tr>
<tr>
<td>48.10.5</td>
<td>Operations with Pseudo Matrices</td>
<td>1210</td>
</tr>
<tr>
<td>49</td>
<td>CHAIN COMPLEXES</td>
<td>1211</td>
</tr>
<tr>
<td>49.1</td>
<td>Complexes of Modules</td>
<td>1213</td>
</tr>
<tr>
<td>49.1.1</td>
<td>Creation</td>
<td>1213</td>
</tr>
<tr>
<td>49.1.2</td>
<td>Subcomplexes and Quotient Complexes</td>
<td>1214</td>
</tr>
<tr>
<td>49.1.3</td>
<td>Access Functions</td>
<td>1214</td>
</tr>
<tr>
<td>49.1.4</td>
<td>Elementary operations</td>
<td>1215</td>
</tr>
<tr>
<td>49.1.5</td>
<td>Extensions</td>
<td>1216</td>
</tr>
<tr>
<td>49.1.6</td>
<td>Predicates</td>
<td>1217</td>
</tr>
<tr>
<td>49.2</td>
<td>Chain Maps</td>
<td>1219</td>
</tr>
<tr>
<td>49.2.1</td>
<td>Creation</td>
<td>1220</td>
</tr>
<tr>
<td>49.2.2</td>
<td>Access Functions</td>
<td>1220</td>
</tr>
<tr>
<td>49.2.3</td>
<td>Elementary Operations</td>
<td>1221</td>
</tr>
<tr>
<td>49.2.4</td>
<td>Predicates</td>
<td>1221</td>
</tr>
<tr>
<td>49.2.5</td>
<td>Maps on Homology</td>
<td>1224</td>
</tr>
</tbody>
</table>
### VOLUME 5: CONTENTS

#### VIII  FINITE GROUPS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.0</td>
<td>Groups</td>
<td>1229</td>
</tr>
<tr>
<td>50.1</td>
<td>Introduction</td>
<td>1233</td>
</tr>
<tr>
<td>50.1.1</td>
<td>The Categories of Finite Groups</td>
<td>1233</td>
</tr>
<tr>
<td>50.2</td>
<td>Construction of Elements</td>
<td>1234</td>
</tr>
<tr>
<td>50.2.1</td>
<td>Construction of an Element</td>
<td>1234</td>
</tr>
<tr>
<td>50.2.2</td>
<td>Coercion</td>
<td>1234</td>
</tr>
<tr>
<td>50.2.3</td>
<td>Homomorphisms</td>
<td>1234</td>
</tr>
<tr>
<td>50.2.4</td>
<td>Arithmetic with Elements</td>
<td>1236</td>
</tr>
<tr>
<td>50.3</td>
<td>Construction of a General Group</td>
<td>1238</td>
</tr>
<tr>
<td>50.3.1</td>
<td>The General Group Constructors</td>
<td>1238</td>
</tr>
<tr>
<td>50.3.2</td>
<td>Construction of Subgroups</td>
<td>1242</td>
</tr>
<tr>
<td>50.3.3</td>
<td>Construction of Quotient Groups</td>
<td>1243</td>
</tr>
<tr>
<td>50.4</td>
<td>Standard Groups and Extensions</td>
<td>1245</td>
</tr>
<tr>
<td>50.4.1</td>
<td>Construction of a Standard Group</td>
<td>1245</td>
</tr>
<tr>
<td>50.4.2</td>
<td>Construction of Extensions</td>
<td>1247</td>
</tr>
<tr>
<td>50.5</td>
<td>Transfer Functions Between Group Categories</td>
<td>1248</td>
</tr>
<tr>
<td>50.6</td>
<td>Basic Operations</td>
<td>1251</td>
</tr>
<tr>
<td>50.6.1</td>
<td>Accessing Group Information</td>
<td>1251</td>
</tr>
<tr>
<td>50.7</td>
<td>Operations on the Set of Elements</td>
<td>1253</td>
</tr>
<tr>
<td>50.7.1</td>
<td>Order and Index Functions</td>
<td>1253</td>
</tr>
<tr>
<td>50.7.2</td>
<td>Membership and Equality</td>
<td>1254</td>
</tr>
<tr>
<td>50.7.3</td>
<td>Set Operations</td>
<td>1255</td>
</tr>
<tr>
<td>50.7.4</td>
<td>Random Elements</td>
<td>1256</td>
</tr>
<tr>
<td>50.7.5</td>
<td>Action on a Coset Space</td>
<td>1259</td>
</tr>
<tr>
<td>50.8</td>
<td>Standard Subgroup Constructions</td>
<td>1260</td>
</tr>
<tr>
<td>50.8.1</td>
<td>Abstract Group Predicates</td>
<td>1261</td>
</tr>
<tr>
<td>50.9</td>
<td>Characteristic Subgroups and Normal Structure</td>
<td>1263</td>
</tr>
<tr>
<td>50.9.1</td>
<td>Characteristic Subgroups and Subgroup Series</td>
<td>1263</td>
</tr>
<tr>
<td>50.9.2</td>
<td>The Abstract Structure of a Group</td>
<td>1265</td>
</tr>
<tr>
<td>50.10</td>
<td>Conjugacy Classes of Elements</td>
<td>1266</td>
</tr>
<tr>
<td>50.11</td>
<td>Conjugacy Classes of Subgroups</td>
<td>1270</td>
</tr>
<tr>
<td>50.11.1</td>
<td>Conjugacy Classes of Subgroups</td>
<td>1270</td>
</tr>
<tr>
<td>50.11.2</td>
<td>The Poset of Subgroup Classes</td>
<td>1274</td>
</tr>
<tr>
<td>50.12</td>
<td>Cohomology</td>
<td>1279</td>
</tr>
<tr>
<td>50.13</td>
<td>Characters and Representations</td>
<td>1280</td>
</tr>
<tr>
<td>50.13.1</td>
<td>Character Theory</td>
<td>1280</td>
</tr>
<tr>
<td>50.13.2</td>
<td>Representation Theory</td>
<td>1281</td>
</tr>
<tr>
<td>50.14</td>
<td>Databases of Groups</td>
<td>1283</td>
</tr>
<tr>
<td>50.15</td>
<td>Bibliography</td>
<td>1283</td>
</tr>
</tbody>
</table>
## 51 Permutation Groups

### 51.1 Introduction

- **51.1.1 Terminology**
- **51.1.2 The Category of Permutation Groups**
- **51.1.3 The Construction of a Permutation Group**

### 51.2 Creation of a Permutation Group

- **51.2.1 Construction of the Symmetric Group**
- **51.2.2 Construction of a Permutation**
- **51.2.3 Construction of a General Permutation Group**

### 51.3 Elementary Properties of a Group

- **51.3.1 Accessing Group Information**
- **51.3.2 Group Order**
- **51.3.3 Abstract Properties of a Group**

### 51.4 Homomorphisms

### 51.5 Building Permutation Groups

- **51.5.1 Some Standard Permutation Groups**
- **51.5.2 Direct Products and Wreath Products**

### 51.6 Permutations

- **51.6.1 Coercion**
- **51.6.2 Arithmetic with Permutations**
- **51.6.3 Properties of Permutations**
- **51.6.4 Predicates for Permutations**
- **51.6.5 Set Operations**

### 51.7 Conjugacy

### 51.8 Subgroups

- **51.8.1 Construction of a Subgroup**
- **51.8.2 Membership and Equality**
- **51.8.3 Elementary Properties of a Subgroup**
- **51.8.4 Standard Subgroups**
- **51.8.5 Maximal Subgroups**
- **51.8.6 Conjugacy Classes of Subgroups**
- **51.8.7 Classes of Subgroups Satisfying a Condition**

### 51.9 Quotient Groups

- **51.9.1 Construction of Quotient Groups**
- **51.9.2 Abelian, Nilpotent and Soluble Quotients**

### 51.10 Permutation Group Actions

- **51.10.1 G-Sets**
- **51.10.2 Creating a G-Set**
- **51.10.3 Images, Orbits and Stabilizers**
- **51.10.4 Action on a G-Space**
- **51.10.5 Action on Orbits**
- **51.10.6 Action on a G-invariant Partition**
- **51.10.7 Action on a Coset Space**
- **51.10.8 Reduced Permutation Actions**

### 51.11 Normal and Subnormal Subgroups

- **51.11.1 Characteristic Subgroups and Normal Series**
- **51.11.2 Maximal and Minimal Normal Subgroups**
- **51.11.3 Lattice of Normal Subgroups**
- **51.11.4 Composition and Chief Series**
- **51.11.5 The Socle**
- **51.11.6 The Soluble Radical and its Quotient**
- **51.11.7 Complements and Supplements**
- **51.11.8 Abelian Normal Subgroups**

### 51.12 Cosets and Transversals

- **51.12.1 Cosets**
- **51.12.2 Transversals**
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>51.13</td>
<td>Presentations</td>
<td>1369</td>
<td></td>
</tr>
<tr>
<td>51.13.1</td>
<td>Generators and Relations</td>
<td>1370</td>
<td></td>
</tr>
<tr>
<td>51.13.2</td>
<td>Permutations as Words</td>
<td>1370</td>
<td></td>
</tr>
<tr>
<td>51.14</td>
<td>Automorphism Groups</td>
<td>1371</td>
<td></td>
</tr>
<tr>
<td>51.15</td>
<td>Cohomology</td>
<td>1373</td>
<td></td>
</tr>
<tr>
<td>51.16</td>
<td>Representation Theory</td>
<td>1375</td>
<td></td>
</tr>
<tr>
<td>51.17</td>
<td>Identification</td>
<td>1377</td>
<td></td>
</tr>
<tr>
<td>51.17.1</td>
<td>Identification as an Abstract Group</td>
<td>1377</td>
<td></td>
</tr>
<tr>
<td>51.17.2</td>
<td>Identification as a Permutation Group</td>
<td>1377</td>
<td></td>
</tr>
<tr>
<td>51.18</td>
<td>Base and Strong Generating Set</td>
<td>1381</td>
<td></td>
</tr>
<tr>
<td>51.18.1</td>
<td>Construction of a Base and Strong Generating Set</td>
<td>1381</td>
<td></td>
</tr>
<tr>
<td>51.18.2</td>
<td>Defining Values for Attributes</td>
<td>1384</td>
<td></td>
</tr>
<tr>
<td>51.18.3</td>
<td>Accessing the Base and Strong Generating Set</td>
<td>1385</td>
<td></td>
</tr>
<tr>
<td>51.18.4</td>
<td>Working with a Base and Strong Generating Set</td>
<td>1386</td>
<td></td>
</tr>
<tr>
<td>51.18.5</td>
<td>Modifying a Base and Strong Generating Set</td>
<td>1388</td>
<td></td>
</tr>
<tr>
<td>51.19</td>
<td>Permutation Representations of Linear Groups</td>
<td>1388</td>
<td></td>
</tr>
<tr>
<td>51.20</td>
<td>Permutation Group Databases</td>
<td>1394</td>
<td></td>
</tr>
<tr>
<td>51.21</td>
<td>Bibliography</td>
<td>1394</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>MATRIX GROUPS OVER GENERAL RINGS</td>
<td>1399</td>
<td></td>
</tr>
<tr>
<td>52.1</td>
<td>Introduction</td>
<td>1403</td>
<td></td>
</tr>
<tr>
<td>52.1.1</td>
<td>Introduction to Matrix Groups</td>
<td>1403</td>
<td></td>
</tr>
<tr>
<td>52.1.2</td>
<td>The Support</td>
<td>1404</td>
<td></td>
</tr>
<tr>
<td>52.1.3</td>
<td>The Category of Matrix Groups</td>
<td>1404</td>
<td></td>
</tr>
<tr>
<td>52.1.4</td>
<td>The Construction of a Matrix Group</td>
<td>1404</td>
<td></td>
</tr>
<tr>
<td>52.2</td>
<td>Creation of a Matrix Group</td>
<td>1404</td>
<td></td>
</tr>
<tr>
<td>52.2.1</td>
<td>Construction of the General Linear Group</td>
<td>1404</td>
<td></td>
</tr>
<tr>
<td>52.2.2</td>
<td>Construction of a Matrix Group Element</td>
<td>1405</td>
<td></td>
</tr>
<tr>
<td>52.2.3</td>
<td>Construction of a General Matrix Group</td>
<td>1407</td>
<td></td>
</tr>
<tr>
<td>52.2.4</td>
<td>Changing Rings</td>
<td>1408</td>
<td></td>
</tr>
<tr>
<td>52.2.5</td>
<td>Coercion between Matrix Structures</td>
<td>1409</td>
<td></td>
</tr>
<tr>
<td>52.2.6</td>
<td>Accessing Associated Structures</td>
<td>1409</td>
<td></td>
</tr>
<tr>
<td>52.3</td>
<td>Homomorphisms</td>
<td>1410</td>
<td></td>
</tr>
<tr>
<td>52.3.1</td>
<td>Construction of Extensions</td>
<td>1412</td>
<td></td>
</tr>
<tr>
<td>52.4</td>
<td>Operations on Matrices</td>
<td>1413</td>
<td></td>
</tr>
<tr>
<td>52.4.1</td>
<td>Arithmetic with Matrices</td>
<td>1414</td>
<td></td>
</tr>
<tr>
<td>52.4.2</td>
<td>Predicates for Matrices</td>
<td>1416</td>
<td></td>
</tr>
<tr>
<td>52.4.3</td>
<td>Matrix Invariants</td>
<td>1416</td>
<td></td>
</tr>
<tr>
<td>52.5</td>
<td>Global Properties</td>
<td>1419</td>
<td></td>
</tr>
<tr>
<td>52.5.1</td>
<td>Group Order</td>
<td>1420</td>
<td></td>
</tr>
<tr>
<td>52.5.2</td>
<td>Membership and Equality</td>
<td>1421</td>
<td></td>
</tr>
<tr>
<td>52.5.3</td>
<td>Set Operations</td>
<td>1422</td>
<td></td>
</tr>
<tr>
<td>52.6</td>
<td>Abstract Group Predicates</td>
<td>1424</td>
<td></td>
</tr>
<tr>
<td>52.7</td>
<td>Conjugacy</td>
<td>1426</td>
<td></td>
</tr>
<tr>
<td>52.8</td>
<td>Subgroups</td>
<td>1430</td>
<td></td>
</tr>
<tr>
<td>52.8.1</td>
<td>Construction of Subgroups</td>
<td>1430</td>
<td></td>
</tr>
<tr>
<td>52.8.2</td>
<td>Elementary Properties of Subgroups</td>
<td>1431</td>
<td></td>
</tr>
<tr>
<td>52.8.3</td>
<td>Standard Subgroups</td>
<td>1431</td>
<td></td>
</tr>
<tr>
<td>52.8.4</td>
<td>Low Index Subgroups</td>
<td>1433</td>
<td></td>
</tr>
<tr>
<td>52.8.5</td>
<td>Conjugacy Classes of Subgroups</td>
<td>1434</td>
<td></td>
</tr>
<tr>
<td>52.9</td>
<td>Quotient Groups</td>
<td>1436</td>
<td></td>
</tr>
<tr>
<td>52.9.1</td>
<td>Construction of Quotient Groups</td>
<td>1437</td>
<td></td>
</tr>
<tr>
<td>52.9.2</td>
<td>Abelian, Nilpotent and Soluble Quotients</td>
<td>1438</td>
<td></td>
</tr>
<tr>
<td>52.10</td>
<td>Matrix Group Actions</td>
<td>1439</td>
<td></td>
</tr>
<tr>
<td>52.10.1</td>
<td>Orbits and Stabilizers</td>
<td>1439</td>
<td></td>
</tr>
</tbody>
</table>
52.10.2 Orbit and Stabilizer Functions for Large Groups 1442
52.10.3 Action on Orbits 1447
52.10.4 Action on a Coset Space 1449
52.10.5 Action on the Natural $G$-Module 1450
52.11 Normal and Subnormal Subgroups 1451
52.11.1 Characteristic Subgroups and Subgroup Series 1451
52.11.2 The Soluble Radical and its Quotient 1453
52.11.3 Composition and Chief Factors 1454
52.12 Coset Tables and Transversals 1456
52.13 Presentations 1456
52.13.1 Presentations 1456
52.13.2 Matrices as Words 1457
52.14 Automorphism Groups 1457
52.15 Representation Theory 1460
52.16 Base and Strong Generating Set 1463
52.16.1 Introduction 1463
52.16.2 Controlling Selection of a Base 1463
52.16.3 Construction of a Base and Strong Generating Set 1464
52.16.4 Defining Values for Attributes 1466
52.16.5 Accessing the Base and Strong Generating Set 1466
52.17 Soluble Matrix Groups 1467
52.17.1 Conversion to a PC-Group 1467
52.17.2 Soluble Group Functions 1467
52.17.3 $p$-group Functions 1468
52.17.4 Abelian Group Functions 1468
52.18 Bibliography 1468

53 MATRIX GROUPS OVER FINITE FIELDS  . . . . . . . . . . 1471
53.1 Introduction 1473
53.1.1 Overview 1473
53.2 Finding Elements with Prescribed Properties 1473
53.3 Monte-Carlo Algorithms for Subgroups 1474
53.4 Aschbacher Reduction 1476
53.4.1 Introduction 1476
53.4.2 Primitivity 1477
53.4.3 Semilinearity 1479
53.4.4 Tensor Products 1481
53.4.5 Tensor-induced Groups 1483
53.4.6 Normalisers ofExtraspecial $r$-groups and Symplectic 2-groups 1484
53.4.7 Writing Representations over Subfields 1486
53.4.8 Decompositions with Respect to a Normal Subgroup 1489
53.5 Bibliography 1492

54 FINITE SOLUBLE GROUPS  . . . . . . . . . . . . . . . . . . . . . . . . . . . 1495
54.1 Introduction 1499
54.1.1 Power-Conjugate Presentations 1499
54.2 Creation of a Group 1500
54.2.1 Construction Functions 1500
54.2.2 Definition by Presentation 1501
54.2.3 Possibly Inconsistent Presentations 1504
54.3 Basic Group Properties 1505
54.3.1 Infrastructure 1505
54.3.2 Numerical Invariants 1506
54.3.3 Predicates 1506
55.4.1 Equality and Comparison 1576
55.4.2 Attributes of Elements 1576
55.5 \textit{Set-Theoretic Operations} 1577
55.5.1 Membership and Equality 1577
55.5.2 Set Operations 1578
55.5.3 Coercions Between Related Groups 1578

56 \textbf{ALMOST SIMPLE GROUPS} \dots \dots \dots \dots \dots \dots \dots \dots \dots \dots \dots \dots 1579
56.1 \textit{Introduction} 1581
56.1.1 Overview 1581
56.2 \textit{Creating Finite Groups of Lie Type} 1582
56.2.1 Generic Creation Function 1582
56.2.2 Classical Groups 1583
56.2.3 Exceptional Groups 1587
56.3 \textit{Group Recognition} 1590
56.3.1 Non-Constructive Recognition of Alternating Groups 1590
56.3.2 Determining the Type of a Finite Group of Lie Type 1593
56.3.3 Classical forms 1596
56.3.4 Recognizing Classical Groups in their Natural Representation 1600
56.3.5 Constructive Recognition of Linear Groups 1602
56.3.6 Constructive Recognition of Suzuki Groups 1605
56.3.7 Constructive Recognition of Small Ree Groups 1611
56.3.8 Constructive Recognition of Large Ree Groups 1614
56.4 \textit{Properties of Finite Groups Of Lie Type} 1616
56.4.1 Maximal Subgroups of the Classical Groups 1616
56.4.2 Maximal Subgroups of the Exceptional Groups 1617
56.4.3 Sylow Subgroups of the Classical Groups 1618
56.4.4 Sylow Subgroups of Exceptional Groups 1619
56.4.5 Conjugacy of Subgroups of the Classical Groups 1622
56.4.6 Conjugacy of elements of the Exceptional Groups 1623
56.4.7 Irreducible Subgroups of the General Linear Group 1623
56.5 \textit{Atlas Data for the Sporadic Groups} 1624
56.6 \textit{Bibliography} 1627

57 \textbf{DATABASES OF GROUPS} \dots \dots \dots \dots \dots \dots \dots \dots \dots \dots \dots \dots 1631
57.1 \textit{Introduction} 1635
57.2 \textit{Database of Small Groups} 1636
57.2.1 Basic Small Group Functions 1637
57.2.2 Processes 1641
57.2.3 Small Group Identification 1643
57.2.4 Accessing Internal Data 1644
57.3 \textit{The \(p\)-groups of Order Dividing \(p^7\)} 1646
57.4 \textit{Metacyclic \(p\)-groups} 1647
57.5 \textit{Database of Perfect Groups} 1649
57.5.1 Specifying an Entry of the Database 1650
57.5.2 Creating the Database 1650
57.5.3 Accessing the Database 1650
57.5.4 Finding Legal Keys 1652
57.6 \textit{Database of Almost-Simple Groups} 1654
57.6.1 The Record Fields 1654
57.6.2 Creating the Database 1655
57.6.3 Accessing the Database 1656
57.7 \textit{Database of Transitive Groups} 1658
57.7.1 Accessing the Databases 1658
57.7.2 Processes 1661
57.7.3 Transitive Group Identification 1662
57.8 Database of Primitive Groups 1663
57.8.1 Accessing the Databases 1663
57.8.2 Processes 1665
57.8.3 Primitive Group Identification 1667
57.9 Database of Rational Maximal Finite Matrix Groups 1667
57.10 Database of Finite Quaternionic Matrix Groups 1669
57.11 Database of Irreducible Matrix Groups 1670
57.11.1 Accessing the Database 1671
57.12 Database of Soluble Irreducible Groups 1672
57.12.1 Basic Functions 1672
57.12.2 Searching with Predicates 1673
57.12.3 Associated Functions 1674
57.12.4 Processes 1675
57.13 Database of ATLAS Groups 1676
57.13.1 Accessing the Database 1677
57.13.2 Accessing the ATLAS Groups 1677
57.13.3 Representations of the ATLAS Groups 1678
57.14 Fundamental Groups of 3-Manifolds 1679
57.14.1 Basic Functions 1680
57.14.2 Accessing the Data 1680
57.15 Bibliography 1682

58 AUTOMORPHISM GROUPS . . . . . . . . . . . . . . . 1685
58.1 Introduction 1687
58.2 Creation of Automorphism Groups 1688
58.3 Access Functions 1690
58.4 Order Functions 1691
58.5 Representations of an Automorphism Group 1692
58.6 Automorphisms 1695
58.7 Stored Attributes of an Automorphism Group 1697
58.8 Holomorphs 1700
58.9 Bibliography 1701

59 COHOMOLOGY AND EXTENSIONS . . . . . . . . . . 1703
59.1 Introduction 1705
59.2 Creation of a Cohomology Module 1706
59.3 Accessing Properties of the Cohomology Module 1707
59.4 Calculating Cohomology 1708
59.5 Cocycles 1709
59.6 The restriction to a subgroup 1712
59.7 Other operations on cohomology modules 1713
59.8 Constructing Extensions 1714
59.9 Constructing Distinct Extensions 1717
59.10 Finite Group Cohomology 1721
59.10.1 Creation of Gamma-groups 1722
59.10.2 Accessing Information 1723
59.10.3 One Cocycles 1724
59.10.4 Group Cohomology 1725
59.11 Bibliography 1728
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>61.1</td>
<td>Introduction</td>
<td>1769</td>
</tr>
<tr>
<td>61.1.1</td>
<td>Overview of Facilities</td>
<td>1769</td>
</tr>
<tr>
<td>61.1.2</td>
<td>The Construction of Finitely Presented Groups</td>
<td>1769</td>
</tr>
<tr>
<td>61.2</td>
<td>Free Groups and Words</td>
<td>1770</td>
</tr>
<tr>
<td>61.2.1</td>
<td>Construction of a Free Group</td>
<td>1770</td>
</tr>
<tr>
<td>61.2.2</td>
<td>Construction of Words</td>
<td>1771</td>
</tr>
<tr>
<td>61.2.3</td>
<td>Access Functions for Words</td>
<td>1771</td>
</tr>
<tr>
<td>61.2.4</td>
<td>Arithmetic Operators for Words</td>
<td>1773</td>
</tr>
<tr>
<td>61.2.5</td>
<td>Comparison of Words</td>
<td>1774</td>
</tr>
<tr>
<td>61.2.6</td>
<td>Relations</td>
<td>1775</td>
</tr>
<tr>
<td>61.3</td>
<td>Construction of an FP-Group</td>
<td>1777</td>
</tr>
<tr>
<td>61.3.1</td>
<td>The Quotient Group Constructor</td>
<td>1777</td>
</tr>
<tr>
<td>61.3.2</td>
<td>The FP-Group Constructor</td>
<td>1779</td>
</tr>
<tr>
<td>61.3.3</td>
<td>Construction from a Finite Permutation or Matrix Group</td>
<td>1780</td>
</tr>
<tr>
<td>61.3.4</td>
<td>Construction of the Standard Presentation for a Coxeter Group</td>
<td>1782</td>
</tr>
<tr>
<td>61.3.5</td>
<td>Conversion from a Special Form of FP-Group</td>
<td>1783</td>
</tr>
<tr>
<td>61.3.6</td>
<td>Construction of a Standard Group</td>
<td>1784</td>
</tr>
<tr>
<td>61.3.7</td>
<td>Construction of Extensions</td>
<td>1786</td>
</tr>
<tr>
<td>61.3.8</td>
<td>Accessing the Defining Generators and Relations</td>
<td>1788</td>
</tr>
<tr>
<td>61.4</td>
<td>Homomorphisms</td>
<td>1788</td>
</tr>
<tr>
<td>61.4.1</td>
<td>General Remarks</td>
<td>1788</td>
</tr>
<tr>
<td>61.4.2</td>
<td>Construction of Homomorphisms</td>
<td>1789</td>
</tr>
<tr>
<td>61.4.3</td>
<td>Accessing Homomorphisms</td>
<td>1789</td>
</tr>
<tr>
<td>61.4.4</td>
<td>Computing Homomorphisms to Permutation Groups</td>
<td>1792</td>
</tr>
<tr>
<td>61.4.5</td>
<td>Searching for Isomorphisms</td>
<td>1799</td>
</tr>
<tr>
<td>61.5</td>
<td>Abelian, Nilpotent and Soluble Quotient</td>
<td>1801</td>
</tr>
<tr>
<td>61.5.1</td>
<td>Abelian Quotient</td>
<td>1801</td>
</tr>
<tr>
<td>61.5.2</td>
<td>p-Quotient</td>
<td>1804</td>
</tr>
<tr>
<td>61.5.3</td>
<td>The Construction of a p-Quotient</td>
<td>1805</td>
</tr>
<tr>
<td>61.5.4</td>
<td>Nilpotent Quotient</td>
<td>1807</td>
</tr>
<tr>
<td>61.5.5</td>
<td>Soluble Quotient</td>
<td>1813</td>
</tr>
<tr>
<td>61.6</td>
<td>Subgroups</td>
<td>1816</td>
</tr>
<tr>
<td>61.6.1</td>
<td>Specification of a Subgroup</td>
<td>1816</td>
</tr>
<tr>
<td>61.6.2</td>
<td>Index of a Subgroup: The Todd-Coxeter Algorithm</td>
<td>1818</td>
</tr>
<tr>
<td>61.6.3</td>
<td>Implicit Invocation of the Todd-Coxeter Algorithm</td>
<td>1823</td>
</tr>
<tr>
<td>61.6.4</td>
<td>Constructing a Presentation for a Subgroup</td>
<td>1824</td>
</tr>
<tr>
<td>61.7</td>
<td>Subgroups of Finite Index</td>
<td>1828</td>
</tr>
<tr>
<td>61.7.1</td>
<td>Low Index Subgroups</td>
<td>1828</td>
</tr>
<tr>
<td>61.7.2</td>
<td>Subgroup Constructions</td>
<td>1836</td>
</tr>
<tr>
<td>61.7.3</td>
<td>Properties of Subgroups</td>
<td>1841</td>
</tr>
<tr>
<td>61.8</td>
<td>Coset Spaces and Tables</td>
<td>1845</td>
</tr>
<tr>
<td>61.8.1</td>
<td>Coset Tables</td>
<td>1846</td>
</tr>
<tr>
<td>61.8.2</td>
<td>Coset Spaces: Construction</td>
<td>1848</td>
</tr>
<tr>
<td>61.8.3</td>
<td>Coset Spaces: Elementary Operations</td>
<td>1848</td>
</tr>
<tr>
<td>61.8.4</td>
<td>Accessing Information</td>
<td>1849</td>
</tr>
<tr>
<td>61.8.5</td>
<td>Double Coset Spaces: Construction</td>
<td>1853</td>
</tr>
<tr>
<td>61.8.6</td>
<td>Coset Spaces: Selection of Cosets</td>
<td>1854</td>
</tr>
<tr>
<td>61.8.7</td>
<td>Coset Spaces: Induced Homomorphism</td>
<td>1856</td>
</tr>
<tr>
<td>61.9</td>
<td>Simplification</td>
<td>1858</td>
</tr>
<tr>
<td>61.9.1</td>
<td>Reducing Generating Sets</td>
<td>1858</td>
</tr>
<tr>
<td>61.9.2</td>
<td>Tietze Transformations</td>
<td>1859</td>
</tr>
<tr>
<td>61.10</td>
<td>Representation Theory</td>
<td>1870</td>
</tr>
<tr>
<td>61.11</td>
<td>Small Group Identification</td>
<td>1874</td>
</tr>
<tr>
<td>61.12</td>
<td>Bibliography</td>
<td>1875</td>
</tr>
</tbody>
</table>
## VOLUME 6: CONTENTS

### 62  FINITELY PRESENTED GROUPS: ADVANCED  . . . . . . . . 1877

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>62.1</td>
<td>Introduction</td>
<td>1881</td>
</tr>
<tr>
<td>62.2</td>
<td>Low Level Operations on Presentations and Words</td>
<td>1881</td>
</tr>
<tr>
<td>62.2.1</td>
<td>Modifying Presentations</td>
<td>1882</td>
</tr>
<tr>
<td>62.2.2</td>
<td>Low Level Operations on Words</td>
<td>1884</td>
</tr>
<tr>
<td>62.3</td>
<td>Interactive Coset Enumeration</td>
<td>1886</td>
</tr>
<tr>
<td>62.3.1</td>
<td>Introduction</td>
<td>1886</td>
</tr>
<tr>
<td>62.3.2</td>
<td>Constructing and Modifying a Coset Enumeration Process</td>
<td>1887</td>
</tr>
<tr>
<td>62.3.3</td>
<td>Starting and Restarting an Enumeration</td>
<td>1892</td>
</tr>
<tr>
<td>62.3.4</td>
<td>Accessing Information</td>
<td>1894</td>
</tr>
<tr>
<td>62.3.5</td>
<td>Induced Permutation Representations</td>
<td>1903</td>
</tr>
<tr>
<td>62.3.6</td>
<td>Coset Spaces and Transversals</td>
<td>1904</td>
</tr>
<tr>
<td>62.4</td>
<td>$p$-Quotients (Process Version)</td>
<td>1907</td>
</tr>
<tr>
<td>62.4.1</td>
<td>The $p$-Quotient Process</td>
<td>1907</td>
</tr>
<tr>
<td>62.4.2</td>
<td>Using $p$-Quotient Interactively</td>
<td>1908</td>
</tr>
<tr>
<td>62.5</td>
<td>Soluble Quotients</td>
<td>1917</td>
</tr>
<tr>
<td>62.5.1</td>
<td>Introduction</td>
<td>1917</td>
</tr>
<tr>
<td>62.5.2</td>
<td>Construction</td>
<td>1917</td>
</tr>
<tr>
<td>62.5.3</td>
<td>Calculating the Relevant Primes</td>
<td>1919</td>
</tr>
<tr>
<td>62.5.4</td>
<td>The Functions</td>
<td>1919</td>
</tr>
<tr>
<td>62.5.5</td>
<td>Soluble Quotient Processes</td>
<td>1923</td>
</tr>
<tr>
<td>62.5.6</td>
<td>Initialisation</td>
<td>1924</td>
</tr>
<tr>
<td>62.5.7</td>
<td>Access Functions</td>
<td>1925</td>
</tr>
<tr>
<td>62.5.8</td>
<td>Symbolic Collector</td>
<td>1927</td>
</tr>
<tr>
<td>62.5.9</td>
<td>Relevant Primes</td>
<td>1929</td>
</tr>
<tr>
<td>62.5.10</td>
<td>Irreducible Modules</td>
<td>1930</td>
</tr>
<tr>
<td>62.5.11</td>
<td>Extension Spaces</td>
<td>1930</td>
</tr>
<tr>
<td>62.5.12</td>
<td>Lifting a Quotient</td>
<td>1931</td>
</tr>
<tr>
<td>62.5.13</td>
<td>Lifting a Quotient by Choosing an Individual Cocycle</td>
<td>1933</td>
</tr>
<tr>
<td>62.5.14</td>
<td>Soluble Quotient Process Tools</td>
<td>1934</td>
</tr>
<tr>
<td>62.5.15</td>
<td>Miscellaneous Functions</td>
<td>1937</td>
</tr>
<tr>
<td>62.5.16</td>
<td>Calculation of Standard Sections</td>
<td>1938</td>
</tr>
<tr>
<td>62.6</td>
<td>Bibliography</td>
<td>1952</td>
</tr>
</tbody>
</table>

### 63  POLYCYCLIC GROUPS  . . . . . . . . . . . . . . . . . . . . 1953

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>63.1</td>
<td>Introduction</td>
<td>1955</td>
</tr>
<tr>
<td>63.2</td>
<td>Polycyclic Groups and Polycyclic Presentations</td>
<td>1955</td>
</tr>
<tr>
<td>63.2.1</td>
<td>Introduction</td>
<td>1955</td>
</tr>
<tr>
<td>63.2.2</td>
<td>Specification of Elements</td>
<td>1956</td>
</tr>
<tr>
<td>63.2.3</td>
<td>Access Functions for Elements</td>
<td>1956</td>
</tr>
<tr>
<td>63.2.4</td>
<td>Arithmetic Operations on Elements</td>
<td>1957</td>
</tr>
<tr>
<td>63.2.5</td>
<td>Operators for Elements</td>
<td>1958</td>
</tr>
<tr>
<td>63.2.6</td>
<td>Comparison Operators for Elements</td>
<td>1958</td>
</tr>
<tr>
<td>63.2.7</td>
<td>Specification of a Polycyclic Presentation</td>
<td>1959</td>
</tr>
<tr>
<td>63.2.8</td>
<td>Properties of a Polycyclic Presentation</td>
<td>1963</td>
</tr>
<tr>
<td>63.3</td>
<td>Subgroups, Quotient Groups, Homomorphisms and Extensions</td>
<td>1963</td>
</tr>
<tr>
<td>63.3.1</td>
<td>Construction of Subgroups</td>
<td>1963</td>
</tr>
<tr>
<td>63.3.2</td>
<td>Coercions Between Groups and Subgroups</td>
<td>1964</td>
</tr>
<tr>
<td>63.3.3</td>
<td>Construction of Quotient Groups</td>
<td>1965</td>
</tr>
<tr>
<td>63.3.4</td>
<td>Homomorphisms</td>
<td>1965</td>
</tr>
<tr>
<td>63.3.5</td>
<td>Construction of Extensions</td>
<td>1966</td>
</tr>
<tr>
<td>63.3.6</td>
<td>Construction of Standard Groups</td>
<td>1966</td>
</tr>
<tr>
<td>63.4</td>
<td>Conversion between Categories</td>
<td>1969</td>
</tr>
<tr>
<td>63.5</td>
<td>Access Functions for Groups</td>
<td>1970</td>
</tr>
<tr>
<td>63.6</td>
<td>Set-Theoretic Operations in a Group</td>
<td>1971</td>
</tr>
</tbody>
</table>
VOLUME 6: CONTENTS

65.4.2 Element Operations 2057
65.5 Operations on the Set of Group Elements 2059
65.6 Homomorphisms 2061
65.6.1 General Remarks 2061
65.6.2 Construction of Homomorphisms 2061
65.7 Conversion to a Finitely Presented Group 2062
65.8 Bibliography 2062

66 AUTOMATIC GROUPS . . . . . . . . . . . . . . . . . . 2063
66.1 Introduction 2065
66.1.1 Terminology 2065
66.1.2 The Category of Automatic Groups 2065
66.1.3 The Construction of an Automatic Group 2065
66.2 Creation of Automatic Groups 2066
66.2.1 Construction of an Automatic Group 2066
66.2.2 Modifying Limits 2067
66.2.3 Accessing Group Information 2070
66.3 Properties of an Automatic Group 2072
66.4 Arithmetic with Words 2073
66.4.1 Construction of a Word 2073
66.4.2 Operations on Elements 2074
66.5 Homomorphisms 2077
66.5.1 General remarks 2077
66.5.2 Construction of Homomorphisms 2077
66.6 Set Operations 2077
66.7 The Growth Function 2079
66.8 Bibliography 2081

67 GROUPS OF STRAIGHT-LINE PROGRAMS . . . . . . . . 2083
67.1 Introduction 2085
67.2 Construction of an SLP-Group and its Elements 2085
67.2.1 Structure Constructors 2085
67.2.2 Construction of an Element 2086
67.3 Arithmetic with Elements 2086
67.3.1 Accessing the Defining Generators and Relations 2086
67.4 Addition of Extra Generators 2087
67.5 Creating Homomorphisms 2087
67.6 Operations on Elements 2089
67.6.1 Equality and Comparison 2089
67.7 Set-Theoretic Operations 2089
67.7.1 Membership and Equality 2089
67.7.2 Set Operations 2090
67.7.3 Coercions Between Related Groups 2091
67.8 Bibliography 2091
FINITELY PRESENTED SEMIGROUPS

68.1 Introduction
68.2 The Construction of Free Semigroups and their Elements
68.2.1 Structure Constructors
68.2.2 Element Constructors
68.3 Elementary Operators for Words
68.3.1 Multiplication and Exponentiation
68.3.2 The Length of a Word
68.3.3 Equality and Comparison
68.4 Specification of a Presentation
68.4.1 Relations
68.4.2 Presentations
68.4.3 Accessing the Defining Generators and Relations
68.5 Subsemigroups, Ideals and Quotients
68.5.1 Subsemigroups and Ideals
68.5.2 Quotients
68.6 Extensions
68.7 Elementary Tietze Transformations
68.8 String Operations on Words

MONOIDS GIVEN BY REWRITE SYSTEMS

69.1 Introduction
69.1.1 Terminology
69.1.2 The Category of Rewrite Monoids
69.1.3 The Construction of a Rewrite Monoid
69.2 Construction of a Rewrite Monoid
69.3 Basic Operations
69.3.1 Accessing Monoid Information
69.3.2 Properties of a Rewrite Monoid
69.3.3 Construction of a Word
69.3.4 Arithmetic with Words
69.4 Homomorphisms
69.4.1 General remarks
69.4.2 Construction of Homomorphisms
69.5 Set Operations
69.6 Conversion to a Finitely Presented Monoid
69.7 Bibliography
VOLUME 7: CONTENTS

X COXETER GROUPS 2123

70 INTRODUCTION TO LIE THEORY 2125

70.1 Descriptions of Coxeter Groups 2127
70.2 Root Systems and Root Data 2128
70.3 Coxeter and Reflection Groups 2128
70.4 Lie Algebras and Groups of Lie Type 2129
70.5 Highest Weight Representations 2129
70.6 Quantum groups and Universal Enveloping Algebras 2129
70.7 Bibliography 2129

71 COXETER SYSTEMS 2131

71.1 Introduction 2133
71.2 Coxeter Matrices 2133
71.3 Coxeter Graphs 2135
71.4 Cartan Matrices 2137
71.5 Dynkin Digraphs 2140
71.6 Finite and Affine Coxeter Groups 2142
71.7 Hyperbolic Groups 2150
71.8 Related Structures 2151
71.9 Bibliography 2153

72 ROOT SYSTEMS 2155

72.1 Introduction 2157
72.1.1 Reflections 2157
72.1.2 Definition of a Root System 2157
72.1.3 Simple and Positive Roots 2158
72.1.4 The Coxeter Group 2158
72.1.5 Nonreduced Root Systems 2159
72.2 Constructing Root Systems 2159
72.3 Operators on Root Systems 2163
72.4 Properties of Root Systems 2165
72.5 Roots and Coroots 2166
72.5.1 Accessing Roots and Coroots 2166
72.5.2 Reflections 2169
72.5.3 Operations and Properties for Roots and Coroot Indices 2171
72.6 Building Root Systems 2174
72.7 Related Structures 2176
72.8 Bibliography 2176
ROOT DATA

73.1 Introduction
73.1.1 Reflections
73.1.2 Definition of a Split Root Datum
73.1.3 Simple and Positive Roots
73.1.4 The Coxeter Group
73.1.5 Nonreduced Root Data
73.1.6 Isogeny of Split Reduced Root Data
73.1.7 Extended Root Data
73.2 Constructing Root Data
73.2.1 Constructing Sparse Root Data
73.3 Operations on Root Data
73.4 Properties of Root Data
73.5 Roots, Coroots and Weights
73.5.1 Accessing Roots and Coroots
73.5.2 Reflections
73.5.3 Operations and Properties for Root and Coroot Indices
73.5.4 Weights
73.6 Building Root Data
73.7 Morphisms of Root Data
73.8 Constants Associated with Root Data
73.9 Related Structures
73.10 Bibliography

COXETER GROUPS

74.1 Introduction
74.1.1 The Normal Form for Words
74.2 Constructing Coxeter Groups
74.3 Converting Between Types of Coxeter Group
74.4 Operations on Coxeter Groups
74.5 Properties of Coxeter Groups
74.6 Operations on Elements
74.7 Roots, Coroots and Reflections
74.7.1 Accessing Roots and Coroots
74.7.2 Operations and Properties for Root and Coroot Indices
74.7.3 Weights
74.8 Reflections
74.9 Building Coxeter Groups
74.10 Root actions
74.11 Standard action
74.12 Braid Groups
74.13 Related Structures
74.14 Bibliography

REFLECTION GROUPS

75.1 Introduction
75.2 Construction of Reflections and Pseudoreflections
75.3 Construction of Reflection Groups
75.4 Construction of Real Reflection Groups
75.5 Construction of Finite Complex Reflection Groups
75.6 Operations on Reflection Groups
75.7 Properties of Reflection Groups
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>75.8</td>
<td>Roots, Coroots and Reflections</td>
<td>2282</td>
</tr>
<tr>
<td>75.8.1</td>
<td>Accessing Roots and Coroots</td>
<td>2282</td>
</tr>
<tr>
<td>75.8.2</td>
<td>Reflections</td>
<td>2286</td>
</tr>
<tr>
<td>75.8.3</td>
<td>Weights</td>
<td>2287</td>
</tr>
<tr>
<td>75.9</td>
<td>Related Structures</td>
<td>2289</td>
</tr>
<tr>
<td>75.10</td>
<td>Bibliography</td>
<td>2289</td>
</tr>
</tbody>
</table>
## XI LIE GROUPS

### 76 GROUPS OF LIE TYPE

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>76.1</td>
<td>Introduction</td>
<td>2297</td>
</tr>
<tr>
<td>76.1.1</td>
<td>The Steinberg Presentation</td>
<td>2297</td>
</tr>
<tr>
<td>76.1.2</td>
<td>Bruhat Normalisation</td>
<td>2297</td>
</tr>
<tr>
<td>76.1.3</td>
<td>Twisted groups of Lie type</td>
<td>2298</td>
</tr>
<tr>
<td>76.2</td>
<td>Constructing Groups of Lie Type</td>
<td>2298</td>
</tr>
<tr>
<td>76.2.1</td>
<td>Split Groups</td>
<td>2299</td>
</tr>
<tr>
<td>76.2.2</td>
<td>Galois Cohomology</td>
<td>2301</td>
</tr>
<tr>
<td>76.2.3</td>
<td>Twisted Groups</td>
<td>2305</td>
</tr>
<tr>
<td>76.3</td>
<td>Operations on Groups of Lie Type</td>
<td>2306</td>
</tr>
<tr>
<td>76.4</td>
<td>Properties of Groups of Lie Type</td>
<td>2310</td>
</tr>
<tr>
<td>76.5</td>
<td>Constructing Elements</td>
<td>2311</td>
</tr>
<tr>
<td>76.6</td>
<td>Operations on Elements</td>
<td>2313</td>
</tr>
<tr>
<td>76.6.1</td>
<td>Basic Operations</td>
<td>2313</td>
</tr>
<tr>
<td>76.6.2</td>
<td>Decompositions</td>
<td>2314</td>
</tr>
<tr>
<td>76.6.3</td>
<td>Conjugacy and Cohomology</td>
<td>2315</td>
</tr>
<tr>
<td>76.7</td>
<td>Properties of Elements</td>
<td>2315</td>
</tr>
<tr>
<td>76.8</td>
<td>Roots, Coroots and Weights</td>
<td>2316</td>
</tr>
<tr>
<td>76.8.1</td>
<td>Accessing Roots and Coroots</td>
<td>2316</td>
</tr>
<tr>
<td>76.8.2</td>
<td>Reflections</td>
<td>2318</td>
</tr>
<tr>
<td>76.8.3</td>
<td>Operations and Properties for Root and Coroot Indices</td>
<td>2319</td>
</tr>
<tr>
<td>76.8.4</td>
<td>Weights</td>
<td>2320</td>
</tr>
<tr>
<td>76.9</td>
<td>Building Groups of Lie Type</td>
<td>2320</td>
</tr>
<tr>
<td>76.10</td>
<td>Automorphisms</td>
<td>2322</td>
</tr>
<tr>
<td>76.10.1</td>
<td>Basic functionality</td>
<td>2322</td>
</tr>
<tr>
<td>76.10.2</td>
<td>Constructing special automorphisms</td>
<td>2323</td>
</tr>
<tr>
<td>76.10.3</td>
<td>Operations and properties of automorphisms</td>
<td>2324</td>
</tr>
<tr>
<td>76.11</td>
<td>Algebraic homomorphisms</td>
<td>2325</td>
</tr>
<tr>
<td>76.12</td>
<td>Twisted Tori</td>
<td>2325</td>
</tr>
<tr>
<td>76.13</td>
<td>Sylow subgroups</td>
<td>2327</td>
</tr>
<tr>
<td>76.14</td>
<td>Representations</td>
<td>2328</td>
</tr>
<tr>
<td>76.15</td>
<td>Bibliography</td>
<td>2330</td>
</tr>
</tbody>
</table>
XII   ALGEBRAS .......................... 2331

77   ALGEBRAS ........................................ 2333

77.1  Introduction .................................... 2335
77.1.1 The Categories of Algebras ..................... 2335
77.2  Construction of General Algebras and their Elements .... 2335
77.2.1 Construction of a General Algebra .............. 2336
77.2.2 Construction of an Element of a General Algebra ... 2337
77.3  Construction of Subalgebras, Ideals and Quotient Algebras ... 2337
77.3.1 Subalgebras and Ideals ......................... 2337
77.3.2 Quotient Algebras ............................ 2338
77.4  Operations on Algebras and Subalgebras .......... 2338
77.4.1 Invariants of an Algebra ...................... 2338
77.4.2 Changing Rings ............................... 2339
77.4.3 Bases .................................... 2339
77.4.4 Decomposition of an Algebra ................... 2340
77.4.5 Operations on Subalgebras ..................... 2342
77.5  Operations on Elements of an Algebra .......... 2343
77.5.1 Operations on Elements ....................... 2343
77.5.2 Comparisons and Membership .................. 2344
77.5.3 Predicates on Elements ....................... 2344

78   STRUCTURE CONSTANT ALGEBRAS ................. 2345

78.1  Introduction .................................... 2347
78.2  Construction of Structure Constant Algebras and Elements ... 2347
78.2.1 Construction of a Structure Constant Algebra .... 2347
78.2.2 Construction of Elements of a Structure Constant Algebra .... 2348
78.3  Operations on Structure Constant Algebras and Elements ...... 2349
78.3.1 Operations on Structure Constant Algebras ....... 2349
78.3.2 Indexing Elements ............................ 2350
78.3.3 The Module Structure of a Structure Constant Algebra ...... 2350
78.3.4 Homomorphisms .............................. 2351

79   ASSOCIATIVE ALGEBRAS ............................. 2355

79.1  Introduction .................................... 2357
79.2  Construction of Associative Algebras .............. 2357
79.2.1 Construction of an Associative Structure Constant Algebra .... 2357
79.2.2 Associative Structure Constant Algebras from other Algebras .... 2358
79.3  Operations on Algebras and their Elements .......... 2359
79.3.1 Operations on Algebras ....................... 2359
79.3.2 Operations on Elements ....................... 2361
79.3.3 Representations ............................. 2362
79.3.4 Decomposition of an Algebra ................... 2362
79.4  Orders .................................... 2364
79.4.1 Creation of Orders ............................ 2364
79.4.2 Attributes ................................. 2367
79.4.3 Bases of Orders .............................. 2368
79.4.4 Predicates 2370
79.4.5 Operations with Orders 2370
79.5 Elements of Orders 2371
79.5.1 Creation of Elements 2371
79.5.2 Arithmetic of Elements 2372
79.5.3 Predicates on Elements 2373
79.5.4 Other Operations with Elements 2373
79.6 Ideals of Orders 2374
79.6.1 Creation of Ideals 2374
79.6.2 Attributes of Ideals 2374
79.6.3 Arithmetic for Ideals 2376
79.6.4 Predicates on Ideals 2376
79.6.5 Other Operations on Ideals 2377
79.7 Quaternionic Orders 2379
79.8 Bibliography 2380

80 FINITELY PRESENTED ALGEBRAS . . . . . . . . . . . 2381
80.1 Introduction 2383
80.2 Representation and Monomial Orders 2383
80.3 Creation of Free Algebras and Elements 2384
80.3.1 Creation of Free Algebras 2384
80.3.2 Print Names 2384
80.3.3 Creation of Polynomials 2384
80.4 Structure Operations 2385
80.4.1 Related Structures 2385
80.4.2 Numerical Invariants 2385
80.4.3 Homomorphisms 2385
80.5 Element Operations 2387
80.5.1 Arithmetic Operators 2387
80.5.2 Equality and Membership 2387
80.5.3 Predicates on Algebra Elements 2387
80.5.4 Coefficients, Monomials, Terms and Degree 2387
80.5.5 Evaluation 2390
80.6 Ideals and Gröbner Bases 2391
80.6.1 Creation of Ideals 2391
80.6.2 Gröbner Bases 2391
80.6.3 Verbosity 2393
80.6.4 Related Functions 2393
80.7 Basic Operations on Ideals 2396
80.7.1 Construction of New Ideals 2396
80.7.2 Ideal Predicates 2397
80.7.3 Operations on Elements of Ideals 2397
80.8 Changing Coefficient Ring 2398
80.9 Finitely Presented Algebras 2398
80.10 Creation of FP-Algebras 2399
80.11 Operations on FP-Algebras 2400
80.12 Finite Dimensional FP-Algebras 2402
80.13 Vector Enumeration 2406
80.13.1 Finitely Presented Modules 2406
80.13.2 S-algebras 2406
80.13.3 Finitely Presented Algebras 2406
80.13.4 Vector Enumeration 2407
80.13.5 The Isomorphism 2408
80.13.6 Sketch of the Algorithm 2408
80.13.7 Weights 2409
### VOLUME 8: CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>80.13.8</td>
<td>Setup Functions</td>
<td>2409</td>
</tr>
<tr>
<td>80.13.9</td>
<td>The Quotient Module Function</td>
<td>2409</td>
</tr>
<tr>
<td>80.13.10</td>
<td>Structuring Presentations</td>
<td>2410</td>
</tr>
<tr>
<td>80.13.11</td>
<td>Options and Controls</td>
<td>2410</td>
</tr>
<tr>
<td>80.13.12</td>
<td>Weights</td>
<td>2411</td>
</tr>
<tr>
<td>80.13.13</td>
<td>Limits</td>
<td>2411</td>
</tr>
<tr>
<td>80.13.14</td>
<td>Logging</td>
<td>2412</td>
</tr>
<tr>
<td>80.13.15</td>
<td>Miscellaneous</td>
<td>2414</td>
</tr>
<tr>
<td>80.14</td>
<td>Bibliography</td>
<td>2417</td>
</tr>
</tbody>
</table>

### 81 MODULUS OVER AN ALGEBRA

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>81.1</td>
<td>Introduction</td>
<td>2421</td>
</tr>
<tr>
<td>81.2</td>
<td>Modules over a Matrix Algebra</td>
<td>2422</td>
</tr>
<tr>
<td>81.2.1</td>
<td>Construction of an A-Module</td>
<td>2422</td>
</tr>
<tr>
<td>81.2.2</td>
<td>Accessing Module Information</td>
<td>2423</td>
</tr>
<tr>
<td>81.2.3</td>
<td>Standard Constructions</td>
<td>2425</td>
</tr>
<tr>
<td>81.2.4</td>
<td>Element Construction and Operations</td>
<td>2426</td>
</tr>
<tr>
<td>81.2.5</td>
<td>Submodules</td>
<td>2428</td>
</tr>
<tr>
<td>81.2.6</td>
<td>Quotient Modules</td>
<td>2431</td>
</tr>
<tr>
<td>81.2.7</td>
<td>Structure of a Module</td>
<td>2432</td>
</tr>
<tr>
<td>81.2.8</td>
<td>Decomposability and Complements</td>
<td>2438</td>
</tr>
<tr>
<td>81.2.9</td>
<td>Lattice of Submodules</td>
<td>2440</td>
</tr>
<tr>
<td>81.2.10</td>
<td>Homomorphisms</td>
<td>2444</td>
</tr>
<tr>
<td>81.3</td>
<td>Modules over a General Algebra</td>
<td>2450</td>
</tr>
<tr>
<td>81.3.1</td>
<td>Introduction</td>
<td>2450</td>
</tr>
<tr>
<td>81.3.2</td>
<td>Construction of Algebra Modules</td>
<td>2450</td>
</tr>
<tr>
<td>81.3.3</td>
<td>The Action of an Algebra Element</td>
<td>2451</td>
</tr>
<tr>
<td>81.3.4</td>
<td>Related Structures of an Algebra Module</td>
<td>2451</td>
</tr>
<tr>
<td>81.3.5</td>
<td>Properties of an Algebra Module</td>
<td>2452</td>
</tr>
<tr>
<td>81.3.6</td>
<td>Creation of Algebra Modules from other Algebra Modules</td>
<td>2452</td>
</tr>
</tbody>
</table>

### 82 MATRICES ALGEBRAS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>82.1</td>
<td>Introduction</td>
<td>2459</td>
</tr>
<tr>
<td>82.2</td>
<td>Construction of Matrix Algebras and their Elements</td>
<td>2459</td>
</tr>
<tr>
<td>82.2.1</td>
<td>Construction of the Complete Matrix Algebra</td>
<td>2459</td>
</tr>
<tr>
<td>82.2.2</td>
<td>Construction of a Matrix</td>
<td>2459</td>
</tr>
<tr>
<td>82.2.3</td>
<td>Constructing a General Matrix Algebra</td>
<td>2461</td>
</tr>
<tr>
<td>82.2.4</td>
<td>The Invariants of a Matrix Algebra</td>
<td>2462</td>
</tr>
<tr>
<td>82.3</td>
<td>Construction of Subalgebras, Ideals and Quotient Rings</td>
<td>2463</td>
</tr>
<tr>
<td>82.4</td>
<td>The Construction of Extensions and their Elements</td>
<td>2465</td>
</tr>
<tr>
<td>82.4.1</td>
<td>The Construction of Direct Sums and Tensor Products</td>
<td>2465</td>
</tr>
<tr>
<td>82.4.2</td>
<td>Construction of Direct Sums and Tensor Products of Elements</td>
<td>2467</td>
</tr>
<tr>
<td>82.5</td>
<td>Operations on Matrix Algebras</td>
<td>2468</td>
</tr>
<tr>
<td>82.6</td>
<td>Changing Rings</td>
<td>2468</td>
</tr>
<tr>
<td>82.7</td>
<td>Elementary Operations on Elements</td>
<td>2468</td>
</tr>
<tr>
<td>82.7.1</td>
<td>Arithmetic</td>
<td>2468</td>
</tr>
<tr>
<td>82.7.2</td>
<td>Predicates</td>
<td>2469</td>
</tr>
<tr>
<td>82.8</td>
<td>Elements of $M_n$ as Homomorphisms</td>
<td>2473</td>
</tr>
<tr>
<td>82.9</td>
<td>Elementary Operations on Subalgebras and Ideals</td>
<td>2474</td>
</tr>
<tr>
<td>82.9.1</td>
<td>Bases</td>
<td>2474</td>
</tr>
<tr>
<td>82.9.2</td>
<td>Intersection of Subalgebras</td>
<td>2474</td>
</tr>
<tr>
<td>82.9.3</td>
<td>Membership and Equality</td>
<td>2474</td>
</tr>
<tr>
<td>82.10</td>
<td>Accessing and Modifying a Matrix</td>
<td>2475</td>
</tr>
<tr>
<td>82.10.1</td>
<td>Indexing</td>
<td>2475</td>
</tr>
</tbody>
</table>
82.10.2 Extracting and Inserting Blocks 2476
82.10.3 Joining Matrices 2476
82.10.4 Row and Column Operations 2477
82.11 Canonical Forms 2477
82.11.1 Canonical Forms for Matrices over Euclidean Domains 2477
82.11.2 Canonical Forms for Matrices over a Field 2479
82.12 Diagonalising Commutative Algebras over a Field 2482
82.13 Solutions of Systems of Linear Equations 2484
82.14 Presentations for Matrix Algebras 2485
82.14.1 Quotients and Idempotents 2485
82.14.2 Generators and Presentations 2488
82.14.3 Solving the Word Problem 2492
82.15 Bibliography 2494

83 BASIC ALGEBRAS ................................................. 2495
83.1 Introduction 2497
83.2 Basic Algebras 2497
83.2.1 Creation 2497
83.2.2 Access Functions 2498
83.2.3 Elementary Operations 2500
83.3 Modules over Basic Algebras 2503
83.3.1 Indecomposable Projective Modules 2503
83.3.2 Creation 2504
83.3.3 Access Functions 2504
83.3.4 Predicates 2505
83.3.5 Elementary Operations 2505
83.4 Homomorphisms 2507
83.4.1 Creation 2508
83.4.2 Access Functions 2509
83.4.3 Projective Covers 2509
83.5 Opposite Algebras 2513
83.5.1 Creation 2514
83.5.2 Injective Modules 2514
83.6 Cohomology 2517
83.7 Group algebras of p-groups 2523
83.7.1 Access Functions 2523
83.7.2 Projective Resolutions 2523
83.7.3 Cohomology Generators 2524
83.7.4 Cohomology Rings 2525
83.7.5 Restrictions and inflations 2525
83.8 A-infinity algebra structures on group cohomology 2529
83.8.1 Homological algebra toolkit 2531
83.9 Bibliography 2533

84 QUATERNION ALGEBRAS ......................................... 2535
84.1 Introduction 2537
84.2 Creation of Quaternion Algebras 2538
84.3 Creation of Quaternion Orders 2542
84.3.1 Creation of Quaternion Orders over Number Rings 2546
84.4 Elements of Quaternion Algebras 2546
84.4.1 Creation of Elements 2547
84.4.2 Arithmetic of Elements 2547
84.5 Attributes of Quaternion Algebras 2549
84.6 Hilbert Symbols and Embeddings 2550
84.7 Predicates on Algebras 2554
84.8 Recognition Functions 2554
84.9 Attributes of Orders 2556
84.10 Operations with Orders 2557
84.11 Ideal Theory of Orders 2557
84.11.1 Creation and Access Functions 2558
84.11.2 Enumeration of Ideal Classes 2561
84.11.3 Operations on Ideals 2563
84.12 Norm Spaces and Basis Reduction 2564
84.13 Isomorphisms 2566
84.14 Units and Unit Groups 2570
84.15 Bibliography 2571

85 LIE ALGEBRAS . . . . . . . . . . . . . . . . . . . . . . 2573
85.1 Introduction 2577
85.2 Constructors for Lie Algebras 2577
85.3 Finitely Presented Lie Algebras 2581
85.3.1 Construction of the Free Lie Algebra 2581
85.3.2 Operations on Elements of the Free Lie Algebra 2582
85.3.3 Construction of a Finitely-Presented Lie Algebra 2583
85.4 Families of Lie Algebras 2586
85.4.1 Almost Reductive Lie Algebras 2586
85.4.2 Cartan-Type Lie algebras 2590
85.5 Construction of Elements 2594
85.5.1 Construction of Elements of Structure Constant Algebras 2594
85.5.2 Construction of Matrix Elements 2595
85.6 Construction of Subalgebras, Ideals and Quotients 2595
85.7 Operations on Lie Algebras 2596
85.7.1 Basic Invariants 2597
85.7.2 Changing Base Rings 2598
85.7.3 Bases 2598
85.7.4 Operations for Semisimple and Reductive Lie Algebras 2598
85.8 Operations on Subalgebras and Ideals 2602
85.8.1 Standard Ideals and Series 2603
85.9 Properties of Lie Algebras and Ideals 2607
85.10 Operations on Elements 2608
85.10.1 Indexing 2609
85.11 The Natural Module 2610
85.12 Operations on Matrix Lie Algebras and Elements 2611
85.13 Homomorphisms 2611
85.14 Automorphisms of Classical-type Reductive Algebras 2612
85.15 Representations 2613
85.16 Restrictable Lie Algebras 2616
85.17 Solvable and Nilpotent Lie Algebras Classification 2619
85.17.1 The List of Solvable Lie Algebras 2619
85.17.2 Comments on the Classification over Finite Fields 2620
85.17.3 The List of Nilpotent Lie Algebras 2621
85.17.4 Functions for Working with the Classifications 2622
85.18 Bibliography 2626
<table>
<thead>
<tr>
<th>86</th>
<th>QUANTUM GROUPS</th>
<th>2627</th>
</tr>
</thead>
<tbody>
<tr>
<td>86.1</td>
<td>Introduction</td>
<td>2629</td>
</tr>
<tr>
<td>86.2</td>
<td>Background</td>
<td>2629</td>
</tr>
<tr>
<td>86.2.1</td>
<td>Gaussian Binomials</td>
<td>2629</td>
</tr>
<tr>
<td>86.2.2</td>
<td>Quantized Enveloping Algebras</td>
<td>2630</td>
</tr>
<tr>
<td>86.2.3</td>
<td>Representations of $U_q(L)$</td>
<td>2631</td>
</tr>
<tr>
<td>86.2.4</td>
<td>PBW-type Bases</td>
<td>2631</td>
</tr>
<tr>
<td>86.2.5</td>
<td>The $\mathbb{Z}$-form of $U_q(L)$</td>
<td>2632</td>
</tr>
<tr>
<td>86.2.6</td>
<td>The Canonical Basis</td>
<td>2633</td>
</tr>
<tr>
<td>86.2.7</td>
<td>The Path Model</td>
<td>2634</td>
</tr>
<tr>
<td>86.3</td>
<td>Gauss Numbers</td>
<td>2635</td>
</tr>
<tr>
<td>86.4</td>
<td>Construction</td>
<td>2636</td>
</tr>
<tr>
<td>86.5</td>
<td>Related Structures</td>
<td>2637</td>
</tr>
<tr>
<td>86.6</td>
<td>Operations on Elements</td>
<td>2638</td>
</tr>
<tr>
<td>86.7</td>
<td>Representations</td>
<td>2640</td>
</tr>
<tr>
<td>86.8</td>
<td>Hopf Algebra Structure</td>
<td>2643</td>
</tr>
<tr>
<td>86.9</td>
<td>Automorphisms</td>
<td>2644</td>
</tr>
<tr>
<td>86.10</td>
<td>Kashiwara Operators</td>
<td>2646</td>
</tr>
<tr>
<td>86.11</td>
<td>The Path Model</td>
<td>2647</td>
</tr>
<tr>
<td>86.12</td>
<td>Elements of the Canonical Basis</td>
<td>2649</td>
</tr>
<tr>
<td>86.13</td>
<td>Homomorphisms to the Universal Enveloping Algebra</td>
<td>2651</td>
</tr>
<tr>
<td>86.14</td>
<td>Bibliography</td>
<td>2652</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>87</th>
<th>UNIVERSAL ENVELOPING ALGEBRAS</th>
<th>2653</th>
</tr>
</thead>
<tbody>
<tr>
<td>87.1</td>
<td>Introduction</td>
<td>2655</td>
</tr>
<tr>
<td>87.2</td>
<td>Background</td>
<td>2655</td>
</tr>
<tr>
<td>87.2.1</td>
<td>Universal Enveloping Algebras</td>
<td>2655</td>
</tr>
<tr>
<td>87.2.2</td>
<td>The Integral Form of a Universal Enveloping Algebra</td>
<td>2656</td>
</tr>
<tr>
<td>87.3</td>
<td>Construction of Universal Enveloping Algebras</td>
<td>2656</td>
</tr>
<tr>
<td>87.4</td>
<td>Related Structures</td>
<td>2657</td>
</tr>
<tr>
<td>87.5</td>
<td>Elements of Universal Enveloping Algebras</td>
<td>2657</td>
</tr>
<tr>
<td>87.5.1</td>
<td>Creation of Elements</td>
<td>2658</td>
</tr>
<tr>
<td>87.5.2</td>
<td>Operations on Elements</td>
<td>2659</td>
</tr>
<tr>
<td>87.6</td>
<td>Bibliography</td>
<td>2659</td>
</tr>
</tbody>
</table>
### XIII REPRESENTATION THEORY

#### 88 GROUP ALGEBRAS

- **Introduction** 2665
- **Construction of Group Algebras and their Elements**
  - Construction of a Group Algebra 2665
  - Construction of a Group Algebra Element 2667
- **Construction of Subalgebras, Ideals and Quotient Algebras**
- **Operations on Group Algebras and their Subalgebras**
  - Operations on Group Algebras 2670
  - Operations on Subalgebras of Group Algebras 2671
- **Operations on Elements** 2673

#### 89 $K[G]$-MODULES AND GROUP REPRESENTATIONS

- **Introduction** 2679
- **Construction of $K[G]$-Modules**
  - General $K[G]$-Modules 2679
  - Natural $K[G]$-Modules 2681
  - Action on an Elementary Abelian Section 2682
  - Permutation Modules 2683
  - Action on a Polynomial Ring 2685
- **The Representation Afforded by a $K[G]$-module**
- **Standard Constructions**
  - Changing the Coefficient Ring 2688
  - Writing a Module over a Smaller Field 2689
  - Direct Sum 2693
  - Tensor Products of $K[G]$-Modules 2693
  - Induction and Restriction 2694
  - The Fixed-point Space of a Module 2695
  - Changing Basis 2695
- **The Construction of all Irreducible Modules**
  - Generic Functions for Finding Irreducible Modules 2696
  - The Burnside Algorithm 2699
  - The Schur Algorithm for Soluble Groups 2700

#### 90 CHARACTERS OF FINITE GROUPS

- **Creation Functions** 2705
- **Structure Creation** 2705
- **Element Creation** 2705
- **The Table of Irreducible Characters** 2706
- **Character Ring Operations**
  - Related Structures 2710
- **Element Operations**
  - Arithmetic 2711
  - Predicates and Booleans 2711
  - Accessing Class Functions 2712
  - Conjugation of Class Functions 2713
  - Functions Returning a Scalar 2713
  - The Schur Index 2714
  - Attribute 2717
  - Induction, Restriction and Lifting 2717
  - Symmetrization 2717
  - Permutation Character 2718
90.3.11 Composition and Decomposition 2718
90.3.12 Finding Irreducibles 2719
90.3.13 Brauer Characters 2722
90.4 Bibliography 2723

91 REPRESENTATIONS OF SYMMETRIC GROUPS . . . . . . . . 2725
91.1 Introduction 2727
91.2 Representations of the Symmetric Group 2727
91.2.1 Integral Representations 2727
91.2.2 The Seminormal and Orthogonal Representations 2728
91.3 Characters of the Symmetric Group 2729
91.3.1 Single Values 2729
91.3.2 Irreducible Characters 2729
91.3.3 Character Table 2729
91.4 Representations of the Alternating Group 2729
91.5 Characters of the Alternating Group 2730
91.5.1 Single Values 2730
91.5.2 Irreducible Characters 2730
91.5.3 Character Table 2730
91.6 Bibliography 2731

92 REPRESENTATIONS OF LIE GROUPS AND ALGEBRAS . . 2733
92.1 Introduction 2735
92.1.1 Highest weight modules 2735
92.1.2 Toral Elements 2736
92.1.3 Other Highest Weight Representations 2736
92.2 Constructing Weight Multisets 2737
92.3 Constructing Representations 2738
92.3.1 Lie Algebras 2738
92.3.2 Groups of Lie Type 2740
92.4 Operations on Weight Multisets 2741
92.4.1 Basic Operations 2741
92.4.2 Conversion functions 2744
92.4.3 Calculating with Representations 2745
92.5 Operations on Representations 2754
92.5.1 Lie Algebras 2754
92.5.2 Groups of Lie Type 2758
92.6 Other Functions for Representation Decompositions 2759
92.6.1 Operations Related to the Symmetric Group 2763
92.7 Subgroups of Small Rank 2763
92.8 Bibliography 2764
XIV  COMMUTATIVE ALGEBRA  2765

93  IDEAL THEORY AND GRÖBNER BASES  . . . . . . . . . 2767

93.1  Introduction  2771
93.2  Representation and Monomial Orders  2771
93.2.1  Lexicographical: lex  2772
93.2.2  Graded Lexicographical: glex  2772
93.2.3  Graded Reverse Lexicographical: grevlex  2772
93.2.4  Elimination (k): elim  2773
93.2.5  Elimination List: elim  2773
93.2.6  Inverse Block: invblock  2773
93.2.7  Univariate: univ  2774
93.2.8  Weight: weight  2774
93.2.9  Graded Reverse Lexicographical with Weights: grevlexw  2774
93.3  Creation of Polynomial Rings and Ideals  2774
93.3.1  Creation of Polynomial Rings  2774
93.3.2  Creation of Ideals and Accessing their Bases  2776
93.4  Gröbner Bases  2777
93.4.1  Gröbner Bases over Fields  2777
93.4.2  Gröbner Bases over Euclidean Rings  2777
93.4.3  Construction of Gröbner Bases  2779
93.4.4  Verbosity  2783
93.4.5  Related Functions  2784
93.5  Basic Operations on Ideals  2794
93.5.1  Construction of New Ideals  2794
93.5.2  Ideal Predicates  2796
93.5.3  Operations on Elements of Ideals  2798
93.6  Computation of Varieties  2800
93.7  Elimination  2802
93.7.1  Construction of Elimination Ideals  2802
93.7.2  Univariate Elimination Ideal Generators  2804
93.7.3  Relation Ideals  2807
93.8  Changing Coefficient Ring  2808
93.9  Changing Monomial Order  2808
93.10  Variable Extension of Ideals  2810
93.11  Homogenization of Ideals  2811
93.12  Extension and Contraction of Ideals  2811
93.13  Dimension of Ideals  2812
93.14  Radical and Decomposition of Ideals  2812
93.14.1  Radical  2813
93.14.2  Primary Decomposition  2814
93.14.3  Triangular Decomposition  2820
93.14.4  Equidimensional Decomposition  2822
93.15  Normalisation and Noether Normalisation  2823
93.15.1  Noether Normalisation  2823
93.15.2  Normalisation  2824
93.16  Graded Polynomial Rings  2827
93.16.1  Creation of Graded Polynomial Rings  2828
93.16.2  Elements of Graded Polynomial Rings  2828
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>93.16.3</td>
<td>Degree-$d$ Gröbner Bases</td>
<td>2829</td>
</tr>
<tr>
<td>93.17</td>
<td>Hilbert Series and Hilbert Polynomial</td>
<td>2832</td>
</tr>
<tr>
<td>93.18</td>
<td>Hilbert-driven Gröbner Basis Construction</td>
<td>2834</td>
</tr>
<tr>
<td>93.19</td>
<td>Syzygy Modules</td>
<td>2836</td>
</tr>
<tr>
<td>93.20</td>
<td>Maps between Rings</td>
<td>2838</td>
</tr>
<tr>
<td>93.21</td>
<td>Symmetric Polynomials</td>
<td>2839</td>
</tr>
<tr>
<td>93.22</td>
<td>Functions for Polynomial Algebra and Module Generators</td>
<td>2840</td>
</tr>
<tr>
<td>93.23</td>
<td>Bibliography</td>
<td>2843</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>94</th>
<th>AFFINE ALGEBRAS</th>
<th>2845</th>
</tr>
</thead>
<tbody>
<tr>
<td>94.1</td>
<td>Introduction</td>
<td>2847</td>
</tr>
<tr>
<td>94.2</td>
<td>Creation of Affine Algebras</td>
<td>2847</td>
</tr>
<tr>
<td>94.3</td>
<td>Operations on Affine Algebras</td>
<td>2849</td>
</tr>
<tr>
<td>94.4</td>
<td>Maps between Affine Algebras</td>
<td>2852</td>
</tr>
<tr>
<td>94.5</td>
<td>Finite Dimensional Affine Algebras</td>
<td>2852</td>
</tr>
<tr>
<td>94.6</td>
<td>Affine Algebras which are Fields</td>
<td>2854</td>
</tr>
<tr>
<td>94.7</td>
<td>Rings and Fields of Fractions of Affine Algebras</td>
<td>2856</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>95</th>
<th>MODULES OVER AFFINE ALGEBRAS</th>
<th>2861</th>
</tr>
</thead>
<tbody>
<tr>
<td>95.1</td>
<td>Introduction</td>
<td>2863</td>
</tr>
<tr>
<td>95.2</td>
<td>Reduced Modules</td>
<td>2863</td>
</tr>
<tr>
<td>95.2.1</td>
<td>Creation of Free Modules</td>
<td>2863</td>
</tr>
<tr>
<td>95.2.2</td>
<td>Creation of Module Elements</td>
<td>2864</td>
</tr>
<tr>
<td>95.2.3</td>
<td>Module Element Access and Operations</td>
<td>2864</td>
</tr>
<tr>
<td>95.2.4</td>
<td>Creation of Submodules and Quotient Modules</td>
<td>2865</td>
</tr>
<tr>
<td>95.2.5</td>
<td>Module Access</td>
<td>2866</td>
</tr>
<tr>
<td>95.2.6</td>
<td>Predicates</td>
<td>2867</td>
</tr>
<tr>
<td>95.2.7</td>
<td>Module Operations</td>
<td>2867</td>
</tr>
<tr>
<td>95.2.8</td>
<td>Free Resolutions</td>
<td>2868</td>
</tr>
<tr>
<td>95.2.9</td>
<td>Homomorphisms between Modules and Matrix Modules</td>
<td>2869</td>
</tr>
<tr>
<td>95.3</td>
<td>Embedded Modules</td>
<td>2871</td>
</tr>
<tr>
<td>95.3.1</td>
<td>Introduction</td>
<td>2871</td>
</tr>
<tr>
<td>95.3.2</td>
<td>Creation of Generic Free Modules</td>
<td>2873</td>
</tr>
<tr>
<td>95.3.3</td>
<td>Print Options</td>
<td>2875</td>
</tr>
<tr>
<td>95.3.4</td>
<td>Creation of Module Elements</td>
<td>2875</td>
</tr>
<tr>
<td>95.3.5</td>
<td>Module Element Access and Operations</td>
<td>2876</td>
</tr>
<tr>
<td>95.3.6</td>
<td>Creation of Submodules and Quotient Modules</td>
<td>2878</td>
</tr>
<tr>
<td>95.3.7</td>
<td>Module Access</td>
<td>2879</td>
</tr>
<tr>
<td>95.3.8</td>
<td>Predicates</td>
<td>2879</td>
</tr>
<tr>
<td>95.3.9</td>
<td>Module Bases</td>
<td>2880</td>
</tr>
<tr>
<td>95.3.10</td>
<td>Module Operations</td>
<td>2880</td>
</tr>
<tr>
<td>95.3.11</td>
<td>Homogeneous Modules</td>
<td>2882</td>
</tr>
<tr>
<td>95.3.12</td>
<td>Syzygy Modules</td>
<td>2883</td>
</tr>
<tr>
<td>95.3.13</td>
<td>Free Resolutions</td>
<td>2884</td>
</tr>
<tr>
<td>95.4</td>
<td>Bibliography</td>
<td>2885</td>
</tr>
</tbody>
</table>
IN Variant Theory

96.1 Introduction
96.2 Invariant Rings of Finite Groups
96.2.1 Creation
96.2.2 Access
96.3 Group Actions on Polynomials
96.4 Permutation Group Actions on Polynomials
96.5 Matrix Group Actions on Polynomials
96.6 Algebraic Group Actions on Polynomials
96.7 Verbosity
96.8 Construction of Invariants of Specified Degree
96.9 Construction of $G$-modules
96.10 Molien Series
96.11 Primary Invariants
96.12 Secondary Invariants
96.13 Fundamental Invariants
96.14 The Module of an Invariant Ring
96.15 The Algebra of an Invariant Ring and Algebraic Relations
96.16 Properties of Invariant Rings
96.17 Steenrod Operations
96.18 Minimalization and Homogeneous Module Testing
96.19 Attributes of Invariant Rings and Fields
96.20 Invariant Rings of Linear Algebraic Groups
96.20.1 Creation
96.20.2 Access
96.20.3 Functions
96.21 Invariant Fields
96.21.1 Creation
96.21.2 Access
96.21.3 Functions for Invariant Fields
96.22 Invariants of the Symmetric Group
96.23 Bibliography

Differential Rings

97.1 Introduction
97.2 Differential Rings and Fields
97.2.1 Creation
97.2.2 Creation of Differential Ring Elements
97.3 Structure Operations on Differential Rings
97.3.1 Category and Parent
97.3.2 Related Structures
97.3.3 Derivation and Differential
97.3.4 Numerical Invariants
97.3.5 Predicates and Booleans
97.4 Element Operations on Differential Ring Elements
97.4.1 Category and Parent
97.4.2 Arithmetic
97.4.3 Predicates and Booleans
97.4.4 Coefficients and Terms
97.4.5 Conjugates, Norm and Trace
97.4.6 Functions on Elements
97.5 Changing Related Structures
97.6 Ring and Field Extensions
97.7 Ideals and Quotient Rings 2950
  97.7.1 Defining Ideals and Quotient Rings 2950
  97.7.2 Boolean Operations on Ideals 2951
97.8 Wronskian Matrix 2951
97.9 Differential Operator Rings 2952
  97.9.1 Creation 2952
  97.9.2 Creation of Differential Operators 2953
97.10 Structure Operations on Differential Operator Rings 2954
  97.10.1 Category and Parent 2954
  97.10.2 Related Structures 2954
  97.10.3 Derivation and Differential 2954
  97.10.4 Predicates and Booleans 2955
97.11 Element Operations on Differential Operators 2955
  97.11.1 Category and Parent 2955
  97.11.2 Arithmetic 2956
  97.11.3 Predicates and Booleans 2957
  97.11.4 Coefficients and Terms 2957
  97.11.5 Order and Degree 2958
  97.11.6 Related Differential Operators 2959
  97.11.7 Application of Operators 2960
97.12 Related Maps 2961
97.13 Changing Related Structures 2962
97.14 Euclidean Algorithms, GCDs and LCMs 2964
  97.14.1 Euclidean Right and Left Division 2964
  97.14.2 Greatest Common Right and Left Divisors 2965
  97.14.3 Least Common Left Multiples 2966
97.15 Related Matrices 2968
97.16 Singular Places and Indicial Polynomials 2969
  97.16.1 Singular Places 2969
  97.16.2 Indicial Polynomials 2971
97.17 Rational Solutions 2972
97.18 Newton Polygons 2973
97.19 Symmetric Powers 2975
97.20 Differential Operators of Algebraic Functions 2976
97.21 Bibliography 2976
VOLUME 9: CONTENTS

XV ALGEBRAIC GEOMETRY

98 SCHEMES ................................................................. 2979

98.1 Introduction and First Examples 2983
98.1.1 Ambient Spaces 2984
98.1.2 Schemes 2985
98.1.3 Rational Points 2986
98.1.4 Projective Closure 2988
98.1.5 Maps 2989
98.1.6 Linear Systems 2991
98.1.7 Aside: Types of Schemes 2992

98.2 Ambients 2993
98.2.1 Affine and Projective Spaces 2993
98.2.2 Scrolls and Products 2995
98.2.3 Functions and Homogeneity on Ambient Spaces 2997
98.2.4 Prelude to Points 2998

98.3 Constructing Schemes 3000

98.4 Different Types of Scheme 3005

98.5 Basic Attributes of Schemes 3006
98.5.1 Functions of the Ambient Space 3006
98.5.2 Functions of the Equations 3007

98.6 Function Fields and their Elements 3009

98.7 Rational Points and Point Sets 3012

98.8 Zero-dimensional Schemes 3016

98.9 Local Geometry of Schemes 3018
98.9.1 Point conditions 3019
98.9.2 Point computations 3019

98.10 Global Geometry of Schemes 3020

98.11 Base Change for Schemes 3022

98.12 Affine Patches and Projective Closure 3025

98.13 Arithmetic Properties of Schemes and Points 3027
98.13.1 Height 3027
98.13.2 Restriction of Scalars 3028
98.13.3 Local Solubility 3028
98.13.4 Searching for Points 3032

98.14 Maps between Schemes 3033
98.14.1 Creation of Maps 3034
98.14.2 Basic Attributes 3043
98.14.3 Maps and Points 3045
98.14.4 Maps and Schemes 3046
98.14.5 Maps and Closure 3050
98.14.6 Automorphisms 3051

98.15 Tangent and Secant Varieties and Isomorphic Projections 3062
98.15.1 Tangent Varieties 3062
98.15.2 Secant Varieties 3063
98.15.3 Isomorphic Projection to Subspaces 3064

98.16 Linear Systems 3066
98.16.1 Creation of Linear Systems 3067
98.16.2 Basic Algebra of Linear Systems 3073
98.16.3 Linear Systems and Maps 3078

98.17 Parametrization of Del Pezzo Surfaces 3078

98.18 Advanced Examples 3085
98.18.1 A Pair of Twisted Cubics 3085
98.18.2 Curves in Space 3088

98.19 Bibliography 3089
ALGEBRAIC CURVES

99.1 First Examples
99.1.1 Ambients
99.1.2 Curves
99.1.3 Projective Closure
99.1.4 Points
99.1.5 Choosing Coordinates
99.1.6 Function Fields and Divisors

99.2 Ambient Spaces

99.3 Algebraic Curves
99.3.1 Creation
99.3.2 Base Change
99.3.3 Basic Attributes
99.3.4 Basic Invariants
99.3.5 Random Curves
99.3.6 Ordinary Plane Curves

99.4 Local Geometry
99.4.1 Creation of Points on Curves
99.4.2 Operations at a Point
99.4.3 Singularity Analysis
99.4.4 Resolution of Singularities
99.4.5 Local Intersection Theory

99.5 Global Geometry
99.5.1 Genus and Singularities
99.5.2 Projective Closure and Affine Patches
99.5.3 Special Forms of Curves

99.6 Maps and Curves
99.6.1 Elementary Maps
99.6.2 Maps Induced by Morphisms

99.7 Automorphism Groups of Curves
99.7.1 Group Creation Functions
99.7.2 Automorphisms
99.7.3 Automorphism Group Operations
99.7.4 Pullbacks and Pushforwards
99.7.5 Quotients of Curves

99.8 Function Fields
99.8.1 Function Fields
99.8.2 Representations of the Function Field
99.8.3 Differentials

99.9 Divisors
99.9.1 Places
99.9.2 Divisor Group
99.9.3 Creation of Divisors
99.9.4 Arithmetic of Divisors
99.9.5 Other Operations on Divisors

99.10 Linear Equivalence of Divisors
99.10.1 Linear Equivalence and Class Group
99.10.2 Riemann–Roch Spaces
99.10.3 Index Calculus

99.11 Advanced Examples
99.11.1 Trigonal Curves
99.11.2 Algebraic Geometric Codes

99.12 Bibliography
100  RESOLUTION GRAPHS AND SPLICE DIAGRAMS . . . . . . 3179

100.1 Introduction 3181
100.2 Resolution Graphs 3181
  100.2.1 Graphs, Vertices and Printing 3182
  100.2.2 Creation from Curve Singularities 3184
  100.2.3 Creation from Pencils 3186
  100.2.4 Creation by Hand 3187
  100.2.5 Modifying Resolution Graphs 3188
  100.2.6 Numerical Data Associated to a Graph 3189
100.3 Splice Diagrams 3190
  100.3.1 Creation of Splice Diagrams 3190
  100.3.2 Numerical Functions of Splice Diagrams 3192
100.4 Translation Between Graphs 3193
  100.4.1 Splice Diagrams from Resolution Graphs 3193
100.5 Bibliography 3194

101  HILBERT SERIES OF POLARISED VARIETIES . . . . . . 3195

101.1 Introduction 3197
  101.1.1 Key Warning and Disclaimer 3197
  101.1.2 Overview of the Chapter 3199
101.2 Hilbert Series and Graded Rings 3200
  101.2.1 Hilbert Series and Hilbert Polynomials 3200
  101.2.2 Interpreting the Hilbert Numerator 3202
101.3 Baskets of Singularities 3205
  101.3.1 Point Singularities 3206
  101.3.2 Curve Singularities 3208
  101.3.3 Baskets of Singularities 3210
  101.3.4 Curves and Dissident Points 3212
101.4 Generic Polarised Varieties 3212
  101.4.1 Accessing the Data 3213
  101.4.2 Generic Creation, Checking, Changing 3214
101.5 Subcanonical Curves 3215
  101.5.1 Creation of Subcanonical Curves 3215
  101.5.2 Catalogue of Subcanonical Curves 3216
101.6 K3 Surfaces 3216
  101.6.1 Creating and Comparing K3 Surfaces 3216
  101.6.2 Accessing the Key Data 3217
  101.6.3 Modifying K3 Surfaces 3217
101.7 The K3 Database 3218
  101.7.1 Searching the K3 Database 3218
  101.7.2 Working with the K3 Database 3221
101.8 Fano 3-folds 3222
  101.8.1 Creation: $f = 1, 2$ or $\geq 3$ 3223
  101.8.2 A Preliminary Fano Database 3224
101.9 Calabi–Yau 3-folds 3224
101.10 Building Databases 3225
  101.10.1 The K3 Database 3225
  101.10.2 Making New Databases 3226
101.11 Bibliography 3227
# VOLUME 10: CONTENTS

## XVI ARITHMETIC GEOMETRY

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>102.1</td>
<td>Introduction</td>
<td>3233</td>
</tr>
<tr>
<td>102.2</td>
<td>Rational Curves and Conics</td>
<td>3234</td>
</tr>
<tr>
<td>102.2.1</td>
<td>Rational Curve and Conic Creation</td>
<td>3234</td>
</tr>
<tr>
<td>102.2.2</td>
<td>Access Functions</td>
<td>3235</td>
</tr>
<tr>
<td>102.2.3</td>
<td>Rational Curve and Conic Examples</td>
<td>3236</td>
</tr>
<tr>
<td>102.3</td>
<td>Conics</td>
<td>3239</td>
</tr>
<tr>
<td>102.3.1</td>
<td>Elementary Invariants</td>
<td>3239</td>
</tr>
<tr>
<td>102.3.2</td>
<td>Alternative Defining Polynomials</td>
<td>3239</td>
</tr>
<tr>
<td>102.3.3</td>
<td>Alternative Models</td>
<td>3240</td>
</tr>
<tr>
<td>102.3.4</td>
<td>Other Functions on Conics</td>
<td>3240</td>
</tr>
<tr>
<td>102.4</td>
<td>Local–Global Correspondence</td>
<td>3241</td>
</tr>
<tr>
<td>102.4.1</td>
<td>Local Conditions for Conics</td>
<td>3241</td>
</tr>
<tr>
<td>102.4.2</td>
<td>Norm Residue Symbol</td>
<td>3241</td>
</tr>
<tr>
<td>102.5</td>
<td>Rational Points on Conics</td>
<td>3243</td>
</tr>
<tr>
<td>102.5.1</td>
<td>Finding Points</td>
<td>3243</td>
</tr>
<tr>
<td>102.5.2</td>
<td>Point Reduction</td>
<td>3245</td>
</tr>
<tr>
<td>102.6</td>
<td>Isomorphisms</td>
<td>3247</td>
</tr>
<tr>
<td>102.6.1</td>
<td>Isomorphisms with Standard Models</td>
<td>3247</td>
</tr>
<tr>
<td>102.7</td>
<td>Automorphisms</td>
<td>3251</td>
</tr>
<tr>
<td>102.7.1</td>
<td>Automorphisms of Rational Curves</td>
<td>3251</td>
</tr>
<tr>
<td>102.7.2</td>
<td>Automorphisms of Conics</td>
<td>3252</td>
</tr>
<tr>
<td>102.8</td>
<td>Bibliography</td>
<td>3254</td>
</tr>
</tbody>
</table>

## 103 ELLIPTIC CURVES

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>103.1</td>
<td>Introduction</td>
<td>3261</td>
</tr>
<tr>
<td>103.2</td>
<td>Creation Functions</td>
<td>3262</td>
</tr>
<tr>
<td>103.2.1</td>
<td>Creation of an Elliptic Curve</td>
<td>3262</td>
</tr>
<tr>
<td>103.2.2</td>
<td>Creation Predicates</td>
<td>3266</td>
</tr>
<tr>
<td>103.2.3</td>
<td>Changing the Base Ring</td>
<td>3266</td>
</tr>
<tr>
<td>103.2.4</td>
<td>Alternative Models</td>
<td>3267</td>
</tr>
<tr>
<td>103.2.5</td>
<td>Predicates on Curve Models</td>
<td>3268</td>
</tr>
<tr>
<td>103.2.6</td>
<td>Twists of Elliptic Curves</td>
<td>3269</td>
</tr>
<tr>
<td>103.3</td>
<td>Operations on Curves</td>
<td>3272</td>
</tr>
<tr>
<td>103.3.1</td>
<td>Elementary Invariants</td>
<td>3272</td>
</tr>
<tr>
<td>103.3.2</td>
<td>Associated Structures</td>
<td>3275</td>
</tr>
<tr>
<td>103.3.3</td>
<td>Predicates on Elliptic Curves</td>
<td>3275</td>
</tr>
<tr>
<td>103.4</td>
<td>Subgroup Schemes</td>
<td>3276</td>
</tr>
<tr>
<td>103.4.1</td>
<td>Creation of Subgroup Schemes</td>
<td>3276</td>
</tr>
<tr>
<td>103.4.2</td>
<td>Associated Structures</td>
<td>3277</td>
</tr>
<tr>
<td>103.4.3</td>
<td>Predicates on Subgroup Schemes</td>
<td>3277</td>
</tr>
<tr>
<td>103.4.4</td>
<td>Points of Subgroup Schemes</td>
<td>3277</td>
</tr>
<tr>
<td>103.5</td>
<td>Operations on Point Sets</td>
<td>3278</td>
</tr>
<tr>
<td>103.5.1</td>
<td>Creation of Point Sets</td>
<td>3279</td>
</tr>
<tr>
<td>103.5.2</td>
<td>Associated Structures</td>
<td>3279</td>
</tr>
<tr>
<td>103.5.3</td>
<td>Predicates on Point Sets</td>
<td>3280</td>
</tr>
</tbody>
</table>
**VOLUME 10: CONTENTS**

103.6  *Operations on Points*  
103.6.1  Creation of Points  
103.6.2  Creation Predicates  
103.6.3  Access Operations  
103.6.4  Associated Structures  
103.6.5  Arithmetic  
103.6.6  Division Points  
103.6.7  Point Order  
103.6.8  Predicates on Points  
103.6.9  Weil Pairing  
103.7  *Polynomials*  
103.8  *Curves over the Rationals*  
103.8.1  Local Invariants  
103.8.2  Kodaira Symbols  
103.8.3  Complex Multiplication  
103.8.4  Isogenous Curves  
103.8.5  Mordell–Weil Group  
103.8.6  Heights and Height Pairing  
103.8.7  Two-Descent and Two-Coverings  
103.8.8  The Cassels-Tate Pairing  
103.8.9  Four-Descent  
103.8.10  Eight-Descent  
103.8.11  Three-Descent  
103.8.12  Heegner Points  
103.8.13  Analytic Information  
103.9  *Integral and S-integral Points*  
103.9.1  Integral Points  
103.9.2  S-integral Points  
103.10  *Elliptic Curve Database*  
103.11  *Curves over Number Fields*  
103.11.1  Local Invariants  
103.11.2  Complex Multiplication  
103.11.3  Torsion Information  
103.11.4  Heights  
103.11.5  Selmer Groups  
103.11.6  Mordell–Weil Group  
103.11.7  Elliptic Curve Chabauty  
103.11.8  Auxiliary functions for etale algebras  
103.12  *Morphisms*  
103.12.1  Creation Functions  
103.12.2  Structure Operations  
103.12.3  The Endomorphism Ring  
103.12.4  The Automorphism Group  
103.12.5  Predicates on Isogenies  
103.13  *The formal group law*  
103.14  *Curves over p-adic Fields*  
103.14.1  Local Invariants  
103.15  *Bibliography*  

<table>
<thead>
<tr>
<th>103.6</th>
<th>Operations on Points</th>
<th>3281</th>
</tr>
</thead>
<tbody>
<tr>
<td>103.6.1</td>
<td>Creation of Points</td>
<td>3281</td>
</tr>
<tr>
<td>103.6.2</td>
<td>Creation Predicates</td>
<td>3282</td>
</tr>
<tr>
<td>103.6.3</td>
<td>Access Operations</td>
<td>3282</td>
</tr>
<tr>
<td>103.6.4</td>
<td>Associated Structures</td>
<td>3282</td>
</tr>
<tr>
<td>103.6.5</td>
<td>Arithmetic</td>
<td>3283</td>
</tr>
<tr>
<td>103.6.6</td>
<td>Division Points</td>
<td>3283</td>
</tr>
<tr>
<td>103.6.7</td>
<td>Point Order</td>
<td>3286</td>
</tr>
<tr>
<td>103.6.8</td>
<td>Predicates on Points</td>
<td>3286</td>
</tr>
<tr>
<td>103.6.9</td>
<td>Weil Pairing</td>
<td>3288</td>
</tr>
<tr>
<td>103.7</td>
<td>Polynomials</td>
<td>3289</td>
</tr>
<tr>
<td>103.8</td>
<td>Curves over the Rationals</td>
<td>3290</td>
</tr>
<tr>
<td>103.8.1</td>
<td>Local Invariants</td>
<td>3290</td>
</tr>
<tr>
<td>103.8.2</td>
<td>Kodaira Symbols</td>
<td>3291</td>
</tr>
<tr>
<td>103.8.3</td>
<td>Complex Multiplication</td>
<td>3292</td>
</tr>
<tr>
<td>103.8.4</td>
<td>Isogenous Curves</td>
<td>3293</td>
</tr>
<tr>
<td>103.8.5</td>
<td>Mordell–Weil Group</td>
<td>3294</td>
</tr>
<tr>
<td>103.8.6</td>
<td>Heights and Height Pairing</td>
<td>3298</td>
</tr>
<tr>
<td>103.8.7</td>
<td>Two-Descent and Two-Coverings</td>
<td>3302</td>
</tr>
<tr>
<td>103.8.8</td>
<td>The Cassels-Tate Pairing</td>
<td>3305</td>
</tr>
<tr>
<td>103.8.9</td>
<td>Four-Descent</td>
<td>3306</td>
</tr>
<tr>
<td>103.8.10</td>
<td>Eight-Descent</td>
<td>3309</td>
</tr>
<tr>
<td>103.8.11</td>
<td>Three-Descent</td>
<td>3310</td>
</tr>
<tr>
<td>103.8.12</td>
<td>Heegner Points</td>
<td>3316</td>
</tr>
<tr>
<td>103.8.13</td>
<td>Analytic Information</td>
<td>3323</td>
</tr>
<tr>
<td>103.9</td>
<td>Integral and S-integral Points</td>
<td>3327</td>
</tr>
<tr>
<td>103.9.1</td>
<td>Integral Points</td>
<td>3328</td>
</tr>
<tr>
<td>103.9.2</td>
<td>S-integral Points</td>
<td>3329</td>
</tr>
<tr>
<td>103.10</td>
<td>Elliptic Curve Database</td>
<td>3331</td>
</tr>
<tr>
<td>103.11</td>
<td>Curves over Number Fields</td>
<td>3334</td>
</tr>
<tr>
<td>103.11.1</td>
<td>Local Invariants</td>
<td>3335</td>
</tr>
<tr>
<td>103.11.2</td>
<td>Complex Multiplication</td>
<td>3336</td>
</tr>
<tr>
<td>103.11.3</td>
<td>Torsion Information</td>
<td>3336</td>
</tr>
<tr>
<td>103.11.4</td>
<td>Heights</td>
<td>3336</td>
</tr>
<tr>
<td>103.11.5</td>
<td>Selmer Groups</td>
<td>3337</td>
</tr>
<tr>
<td>103.11.6</td>
<td>Mordell–Weil Group</td>
<td>3342</td>
</tr>
<tr>
<td>103.11.7</td>
<td>Elliptic Curve Chabauty</td>
<td>3343</td>
</tr>
<tr>
<td>103.11.8</td>
<td>Auxiliary functions for etale algebras</td>
<td>3345</td>
</tr>
<tr>
<td>103.12</td>
<td>Morphisms</td>
<td>3347</td>
</tr>
<tr>
<td>103.12.1</td>
<td>Creation Functions</td>
<td>3347</td>
</tr>
<tr>
<td>103.12.2</td>
<td>Structure Operations</td>
<td>3351</td>
</tr>
<tr>
<td>103.12.3</td>
<td>The Endomorphism Ring</td>
<td>3352</td>
</tr>
<tr>
<td>103.12.4</td>
<td>The Automorphism Group</td>
<td>3353</td>
</tr>
<tr>
<td>103.12.5</td>
<td>Predicates on Isogenies</td>
<td>3353</td>
</tr>
<tr>
<td>103.13</td>
<td>The formal group law</td>
<td>3354</td>
</tr>
<tr>
<td>103.14</td>
<td>Curves over p-adic Fields</td>
<td>3354</td>
</tr>
<tr>
<td>103.14.1</td>
<td>Local Invariants</td>
<td>3354</td>
</tr>
<tr>
<td>103.15</td>
<td>Bibliography</td>
<td>3355</td>
</tr>
</tbody>
</table>
# ELLIPTIC CURVES OVER FINITE FIELDS

- **104.1 Supersingular Curves**
- **104.2 The Order of the Group of Points**
  - **104.2.1 Point Counting**
  - **104.2.2 Zeta Functions**
  - **104.2.3 Cryptographic Elliptic Curve Domains**
- **104.3 Enumeration of Points**
- **104.4 Abelian Group Structure**
- **104.5 Pairings on Elliptic Curves**
  - **104.5.1 Weil pairing**
  - **104.5.2 Tate pairing**
  - **104.5.3 Eta pairing**
  - **104.5.4 Ate pairing**
- **104.6 Weil Descent in Characteristic Two**
- **104.7 Discrete Logarithms**
- **104.8 Bibliography**

# ELLIPTIC CURVES OVER FUNCTION FIELDS

- **105.1 An Overview of Relevant Theory**
- **105.2 Local computations**
- **105.3 Heights**
- **105.4 The Torsion Subgroup**
- **105.5 The Mordell-Weil Group**
- **105.6 Two-descent**
- **105.7 The \(L\)-function and counting points**
- **105.8 Action of Frobenius**
- **105.9 Extended Examples**
- **105.10 Bibliography**

# MODELS OF GENUS ONE CURVES

- **106.1 Introduction**
- **106.2 Related functionality**
- **106.3 Creation of Genus One Models**
- **106.4 Predicates on Genus One Models**
- **106.5 Access Functions**
- **106.6 Minimisation and Reduction**
- **106.7 Genus One Models as Coverings**
- **106.8 Families of Elliptic Curves with Prescribed \(n\)-Torsion**
- **106.9 Transformations between Genus One Models**
- **106.10 Invariants for Genus One Models**
- **106.11 Covariants and Contravariants for Genus One Models**
- **106.12 Extended Example**
- **106.13 Bibliography**
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>107.1</td>
<td>Introduction</td>
<td>3419</td>
</tr>
<tr>
<td>107.2</td>
<td>Creation Functions</td>
<td>3419</td>
</tr>
<tr>
<td>107.2.1</td>
<td>Creation of a Hyperelliptic Curve</td>
<td>3419</td>
</tr>
<tr>
<td>107.2.2</td>
<td>Creation Predicates</td>
<td>3420</td>
</tr>
<tr>
<td>107.2.3</td>
<td>Changing the Base Ring</td>
<td>3421</td>
</tr>
<tr>
<td>107.2.4</td>
<td>Models</td>
<td>3422</td>
</tr>
<tr>
<td>107.2.5</td>
<td>Predicates on Models</td>
<td>3424</td>
</tr>
<tr>
<td>107.2.6</td>
<td>Twisting Hyperelliptic Curves</td>
<td>3425</td>
</tr>
<tr>
<td>107.2.7</td>
<td>Type Change Predicates</td>
<td>3426</td>
</tr>
<tr>
<td>107.3</td>
<td>Operations on Curves</td>
<td>3426</td>
</tr>
<tr>
<td>107.3.1</td>
<td>Elementary Invariants</td>
<td>3426</td>
</tr>
<tr>
<td>107.3.2</td>
<td>Igusa Invariants</td>
<td>3427</td>
</tr>
<tr>
<td>107.3.3</td>
<td>Base Ring</td>
<td>3430</td>
</tr>
<tr>
<td>107.4</td>
<td>Creation from Invariants</td>
<td>3430</td>
</tr>
<tr>
<td>107.5</td>
<td>Function Field</td>
<td>3431</td>
</tr>
<tr>
<td>107.5.1</td>
<td>Function Field and Polynomial Ring</td>
<td>3431</td>
</tr>
<tr>
<td>107.6</td>
<td>Points</td>
<td>3432</td>
</tr>
<tr>
<td>107.6.1</td>
<td>Creation of Points</td>
<td>3432</td>
</tr>
<tr>
<td>107.6.2</td>
<td>Random Points</td>
<td>3433</td>
</tr>
<tr>
<td>107.6.3</td>
<td>Predicates on Points</td>
<td>3433</td>
</tr>
<tr>
<td>107.6.4</td>
<td>Access Operations</td>
<td>3434</td>
</tr>
<tr>
<td>107.6.5</td>
<td>Arithmetic of Points</td>
<td>3434</td>
</tr>
<tr>
<td>107.6.6</td>
<td>Enumeration and Counting Points</td>
<td>3434</td>
</tr>
<tr>
<td>107.6.7</td>
<td>Frobenius</td>
<td>3436</td>
</tr>
<tr>
<td>107.7</td>
<td>Isomorphisms and Transformations</td>
<td>3436</td>
</tr>
<tr>
<td>107.7.1</td>
<td>Creation of Isomorphisms</td>
<td>3436</td>
</tr>
<tr>
<td>107.7.2</td>
<td>Arithmetic with Isomorphisms</td>
<td>3437</td>
</tr>
<tr>
<td>107.7.3</td>
<td>Invariants of Isomorphisms</td>
<td>3438</td>
</tr>
<tr>
<td>107.7.4</td>
<td>Automorphism Group and Isomorphism Testing</td>
<td>3438</td>
</tr>
<tr>
<td>107.8</td>
<td>Jacobians</td>
<td>3442</td>
</tr>
<tr>
<td>107.8.1</td>
<td>Creation of a Jacobian</td>
<td>3442</td>
</tr>
<tr>
<td>107.8.2</td>
<td>Access Operations</td>
<td>3442</td>
</tr>
<tr>
<td>107.8.3</td>
<td>Base Ring</td>
<td>3442</td>
</tr>
<tr>
<td>107.8.4</td>
<td>Changing the Base Ring</td>
<td>3443</td>
</tr>
<tr>
<td>107.9</td>
<td>Points on the Jacobian</td>
<td>3443</td>
</tr>
<tr>
<td>107.9.1</td>
<td>Creation of Points</td>
<td>3444</td>
</tr>
<tr>
<td>107.9.2</td>
<td>Random Points</td>
<td>3447</td>
</tr>
<tr>
<td>107.9.3</td>
<td>Booleans and Predicates for Points</td>
<td>3448</td>
</tr>
<tr>
<td>107.9.4</td>
<td>Access Operations</td>
<td>3448</td>
</tr>
<tr>
<td>107.9.5</td>
<td>Arithmetic of Points</td>
<td>3448</td>
</tr>
<tr>
<td>107.9.6</td>
<td>Order of Points on the Jacobian</td>
<td>3449</td>
</tr>
<tr>
<td>107.9.7</td>
<td>Frobenius</td>
<td>3450</td>
</tr>
<tr>
<td>107.9.8</td>
<td>Weil Pairing</td>
<td>3450</td>
</tr>
<tr>
<td>107.10</td>
<td>Rational Points and Group Structure over finite fields</td>
<td>3451</td>
</tr>
<tr>
<td>107.10.1</td>
<td>Enumeration of Points</td>
<td>3451</td>
</tr>
<tr>
<td>107.10.2</td>
<td>Counting Points on the Jacobian</td>
<td>3451</td>
</tr>
<tr>
<td>107.10.3</td>
<td>Deformation Point Counting</td>
<td>3456</td>
</tr>
<tr>
<td>107.10.4</td>
<td>Abelian Group Structure</td>
<td>3457</td>
</tr>
<tr>
<td>107.11</td>
<td>Jacobians over Number Fields or Q</td>
<td>3458</td>
</tr>
<tr>
<td>107.11.1</td>
<td>Searching For Points</td>
<td>3458</td>
</tr>
<tr>
<td>107.11.2</td>
<td>Torsion</td>
<td>3458</td>
</tr>
<tr>
<td>107.11.3</td>
<td>Heights and Regulator</td>
<td>3460</td>
</tr>
<tr>
<td>107.11.4</td>
<td>The 2-Selmer Group</td>
<td>3464</td>
</tr>
<tr>
<td>107.12</td>
<td>Chabauty's Method</td>
<td>3469</td>
</tr>
<tr>
<td>107.13</td>
<td>Kummer Surfaces</td>
<td>3474</td>
</tr>
</tbody>
</table>
107.13.1 Creation of a Kummer Surface 3474
107.13.2 Structure Operations 3474
107.13.3 Base Ring 3474
107.13.4 Changing the Base Ring 3474
107.14 Points on the Kummer Surface 3475
107.14.1 Creation of Points 3475
107.14.2 Access Operations 3475
107.14.3 Predicates on Points 3475
107.14.4 Arithmetic of Points 3476
107.14.5 Rational Points on the Kummer Surface 3476
107.14.6 Pullback to the Jacobian 3477
107.15 Analytic Jacobians of Hyperelliptic Curves 3477
107.15.1 Creation and Access Functions 3478
107.15.2 Maps between Jacobians 3480
107.15.3 From Period Matrix to Curve 3486
107.15.4 Voronoi Cells 3488
107.16 Bibliography 3489

108 L-FUNCTIONS 3491
108.1 Overview 3493
108.2 Built-in L-series 3494
108.3 Computing L-values 3499
108.4 Arithmetic with L-series 3501
108.5 General L-series 3503
108.5.1 Terminology 3503
108.5.2 Constructing a General L-Series 3504
108.5.3 Setting the Coefficients 3508
108.5.4 Specifying the Coefficients Later 3508
108.5.5 Generating the Coefficients from Local Factors 3510
108.6 Accessing the Invariants 3510
108.7 Precision 3512
108.7.1 L-series with Unusual Coefficient Growth 3512
108.7.2 Computing L(s) when Im(s) is Large (ImS Parameter) 3513
108.7.3 Implementation of L-series Computations (Asymptotics Parameter) 3513
108.8Verbose Printing 3513
108.9 Advanced Examples 3514
108.9.1 Self-made L-series of an Elliptic Curve 3514
108.9.2 Self-made Dedekind Zeta Function 3515
108.9.3 L-series of a Genus 2 Hyperelliptic Curve 3515
108.9.4 Experimental Mathematics for Small Conductor 3517
108.9.5 Tensor Product of L-series Coming from l-adic Representations 3518
108.9.6 Non-abelian Twist of an Elliptic Curve 3519
108.10 Bibliography 3520
## XVII MODULAR ARITHMETIC GEOMETRY

### 109 MODULAR CURVES

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>109.1</td>
<td>Introduction</td>
<td>3525</td>
</tr>
<tr>
<td>109.2</td>
<td>Creation Functions</td>
<td>3525</td>
</tr>
<tr>
<td>109.2.1</td>
<td>Creation of a Modular Curve</td>
<td>3525</td>
</tr>
<tr>
<td>109.2.2</td>
<td>Creation of Points</td>
<td>3525</td>
</tr>
<tr>
<td>109.3</td>
<td>Invariants</td>
<td>3526</td>
</tr>
<tr>
<td>109.4</td>
<td>Modular Polynomial Databases</td>
<td>3527</td>
</tr>
<tr>
<td>109.5</td>
<td>Parametrized Structures</td>
<td>3530</td>
</tr>
<tr>
<td>109.6</td>
<td>Associated Structures</td>
<td>3532</td>
</tr>
<tr>
<td>109.7</td>
<td>Automorphisms</td>
<td>3533</td>
</tr>
<tr>
<td>109.8</td>
<td>Class Polynomials</td>
<td>3533</td>
</tr>
<tr>
<td>109.9</td>
<td>Bibliography</td>
<td>3536</td>
</tr>
</tbody>
</table>

### 110 CONGRUENCE SUBGROUPS OF $\text{PSL}_2(\mathbb{R})$

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>110.1</td>
<td>Introduction</td>
<td>3539</td>
</tr>
<tr>
<td>110.2</td>
<td>Congruence Subgroups</td>
<td>3540</td>
</tr>
<tr>
<td>110.2.1</td>
<td>Creation of Subgroups of $\text{PSL}_2(\mathbb{R})$</td>
<td>3541</td>
</tr>
<tr>
<td>110.2.2</td>
<td>Relations</td>
<td>3542</td>
</tr>
<tr>
<td>110.2.3</td>
<td>Basic Attributes</td>
<td>3542</td>
</tr>
<tr>
<td>110.3</td>
<td>Structure of Congruence Subgroups</td>
<td>3543</td>
</tr>
<tr>
<td>110.3.1</td>
<td>Cusps and Elliptic Points of Congruence Subgroups</td>
<td>3544</td>
</tr>
<tr>
<td>110.4</td>
<td>Elements of $\text{PSL}_2(\mathbb{R})$</td>
<td>3546</td>
</tr>
<tr>
<td>110.4.1</td>
<td>Creation</td>
<td>3546</td>
</tr>
<tr>
<td>110.4.2</td>
<td>Membership and Equality Testing</td>
<td>3546</td>
</tr>
<tr>
<td>110.4.3</td>
<td>Basic Functions</td>
<td>3546</td>
</tr>
<tr>
<td>110.5</td>
<td>The Upper Half Plane</td>
<td>3547</td>
</tr>
<tr>
<td>110.5.1</td>
<td>Creation</td>
<td>3547</td>
</tr>
<tr>
<td>110.5.2</td>
<td>Basic Attributes</td>
<td>3548</td>
</tr>
<tr>
<td>110.6</td>
<td>Action of $\text{PSL}_2(\mathbb{R})$ on the Upper Half Plane</td>
<td>3549</td>
</tr>
<tr>
<td>110.6.1</td>
<td>Arithmetic</td>
<td>3550</td>
</tr>
<tr>
<td>110.6.2</td>
<td>Distances, Angles and Geodesics</td>
<td>3550</td>
</tr>
<tr>
<td>110.7</td>
<td>Farey Symbols and Fundamental Domains</td>
<td>3551</td>
</tr>
<tr>
<td>110.8</td>
<td>Points and Geodesics</td>
<td>3553</td>
</tr>
<tr>
<td>110.9</td>
<td>Graphical Output</td>
<td>3553</td>
</tr>
<tr>
<td>110.10</td>
<td>Bibliography</td>
<td>3561</td>
</tr>
</tbody>
</table>
111 ARITHMETIC FUCHSIAN GROUPS AND SHIMURA CURVES 3563

111.1 Arithmetic Fuchsian Groups 3565
111.1.1 Creation 3565
111.1.2 Quaternionic Functions 3567
111.1.3 Basic Invariants 3570
111.1.4 Group Structure 3571
111.2 Unit Disc 3573
111.2.1 Creation 3573
111.2.2 Basic Operations 3574
111.2.3 Access Operations 3574
111.2.4 Distance and Angles 3576
111.2.5 Structural Operations 3577
111.3 Fundamental Domains 3579
111.4 Triangle Groups 3581
111.4.1 Creation of Triangle Groups 3582
111.4.2 Fundamental Domain 3582
111.4.3 CM Points 3582
111.5 Bibliography 3585

112 MODULAR FORMS 3587

112.1 Introduction 3589
112.1.1 Modular Forms 3589
112.1.2 Status and Future Directions 3590
112.1.3 Categories 3591
112.1.4 Verbose Output 3591
112.1.5 An Illustrative Overview 3592
112.2 Creation Functions 3595
112.2.1 Ambient Spaces 3595
112.2.2 Base Extension 3598
112.2.3 Elements 3599
112.2.4 Memory Management 3600
112.3 Bases 3600
112.4 q-Expansions 3602
112.5 Arithmetic 3604
112.6 Predicates 3605
112.7 Properties 3607
112.8 Subspaces 3609
112.9 Operators 3611
112.10 Eisenstein Series 3614
112.11 Weight Half Forms 3615
112.12 Weight One Forms 3615
112.13 Newforms 3616
112.13.1 Labels 3618
112.14 Reductions and Embeddings 3620
112.15 Congruences 3622
112.16 Algebraic Relations 3623
112.17 Elliptic Curves 3624
112.18 Modular Symbols 3625
112.19 Bibliography 3626
## VOLUME 11: CONTENTS

### 113 MODULAR SYMBOLS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>113.1</td>
<td>Introduction</td>
<td>3629</td>
</tr>
<tr>
<td>113.1.1</td>
<td>Modular Symbols</td>
<td>3629</td>
</tr>
<tr>
<td>113.2</td>
<td>Basics</td>
<td>3630</td>
</tr>
<tr>
<td>113.2.1</td>
<td>Verbose Output</td>
<td>3630</td>
</tr>
<tr>
<td>113.2.2</td>
<td>Categories</td>
<td>3630</td>
</tr>
<tr>
<td>113.3</td>
<td>Creation Functions</td>
<td>3631</td>
</tr>
<tr>
<td>113.3.1</td>
<td>Ambient Spaces</td>
<td>3631</td>
</tr>
<tr>
<td>113.3.2</td>
<td>Memory Management</td>
<td>3632</td>
</tr>
<tr>
<td>113.3.3</td>
<td>Labels</td>
<td>3635</td>
</tr>
<tr>
<td>113.3.4</td>
<td>Creation of Elements</td>
<td>3636</td>
</tr>
<tr>
<td>113.4</td>
<td>Bases</td>
<td>3640</td>
</tr>
<tr>
<td>113.5</td>
<td>Associated Vector Space</td>
<td>3642</td>
</tr>
<tr>
<td>113.6</td>
<td>Degeneracy Maps</td>
<td>3644</td>
</tr>
<tr>
<td>113.7</td>
<td>Decomposition</td>
<td>3646</td>
</tr>
<tr>
<td>113.8</td>
<td>Subspaces</td>
<td>3649</td>
</tr>
<tr>
<td>113.9</td>
<td>Operators</td>
<td>3652</td>
</tr>
<tr>
<td>113.10</td>
<td>The Hecke Algebra</td>
<td>3656</td>
</tr>
<tr>
<td>113.11</td>
<td>The Intersection Pairing</td>
<td>3657</td>
</tr>
<tr>
<td>113.12</td>
<td>(q)-Expansions</td>
<td>3659</td>
</tr>
<tr>
<td>113.13</td>
<td>Special Values of (L)-functions</td>
<td>3661</td>
</tr>
<tr>
<td>113.13.1</td>
<td>Winding Elements</td>
<td>3663</td>
</tr>
<tr>
<td>113.14</td>
<td>The Associated Complex Torus</td>
<td>3664</td>
</tr>
<tr>
<td>113.14.1</td>
<td>The Period Map</td>
<td>3669</td>
</tr>
<tr>
<td>113.14.2</td>
<td>Projection Mappings</td>
<td>3670</td>
</tr>
<tr>
<td>113.15</td>
<td>Modular Abelian Varieties</td>
<td>3671</td>
</tr>
<tr>
<td>113.15.1</td>
<td>Modular Degree and Torsion</td>
<td>3672</td>
</tr>
<tr>
<td>113.15.2</td>
<td>Tamagawa Numbers and Orders of Component Groups</td>
<td>3674</td>
</tr>
<tr>
<td>113.16</td>
<td>Elliptic Curves</td>
<td>3676</td>
</tr>
<tr>
<td>113.17</td>
<td>Dimension Formulas</td>
<td>3678</td>
</tr>
<tr>
<td>113.18</td>
<td>Bibliography</td>
<td>3679</td>
</tr>
</tbody>
</table>

### 114 BRANDT MODULES

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>114.1</td>
<td>Introduction</td>
<td>3683</td>
</tr>
<tr>
<td>114.2</td>
<td>Brandt Module Creation</td>
<td>3683</td>
</tr>
<tr>
<td>114.2.1</td>
<td>Creation of Elements</td>
<td>3685</td>
</tr>
<tr>
<td>114.2.2</td>
<td>Operations on Elements</td>
<td>3685</td>
</tr>
<tr>
<td>114.2.3</td>
<td>Categories and Parent</td>
<td>3686</td>
</tr>
<tr>
<td>114.2.4</td>
<td>Elementary Invariants</td>
<td>3686</td>
</tr>
<tr>
<td>114.2.5</td>
<td>Associated Structures</td>
<td>3687</td>
</tr>
<tr>
<td>114.2.6</td>
<td>Verbose Output</td>
<td>3688</td>
</tr>
<tr>
<td>114.3</td>
<td>Subspaces and Decomposition</td>
<td>3689</td>
</tr>
<tr>
<td>114.3.1</td>
<td>Boolean Tests on Subspaces</td>
<td>3690</td>
</tr>
<tr>
<td>114.4</td>
<td>Hecke Operators</td>
<td>3691</td>
</tr>
<tr>
<td>114.5</td>
<td>(q)-Expansions</td>
<td>3692</td>
</tr>
<tr>
<td>114.6</td>
<td>Dimensions of Spaces</td>
<td>3692</td>
</tr>
<tr>
<td>114.7</td>
<td>Bibliography</td>
<td>3693</td>
</tr>
</tbody>
</table>
## 115 SUPERSINGULAR DIVISORS ON MODULAR CURVES

115.1 Introduction 3697  
115.1.1 Categories 3698  
115.1.2 Verbose Output 3698  
115.2 Creation Functions 3698  
115.2.1 Ambient Spaces 3698  
115.2.2 Elements 3699  
115.2.3 Subspaces 3700  
115.3 Basis 3701  
115.4 Properties 3702  
115.5 Associated Spaces 3703  
115.6 Predicates 3704  
115.7 Arithmetic 3705  
115.8 Operators 3707  
115.9 The Monodromy Pairing 3708  
115.10 Bibliography 3709

## 116 MODULAR ABELIAN VARIETIES

116.1 Introduction 3717  
116.1.1 Categories 3718  
116.1.2 Verbose Output 3718  
116.2 Creation and Basic Functions 3719  
116.2.1 Creating the Modular Jacobian \(J_0(N)\) 3719  
116.2.2 Creating the Modular Jacobians \(J_1(N)\) and \(J_H(N)\) 3720  
116.2.3 Abelian Varieties Attached to Modular Forms 3722  
116.2.4 Abelian Varieties Attached to Modular Symbols 3724  
116.2.5 Creation of Abelian Subvarieties 3725  
116.2.6 Creation Using a Label 3726  
116.2.7 Invariants 3727  
116.2.8 Conductor 3730  
116.2.9 Number of Points 3730  
116.2.10 Inner Twists and Complex Multiplication 3731  
116.2.11 Predicates 3734  
116.2.12 Equality and Inclusion Testing 3739  
116.2.13 Modular Embedding and Parameterization 3740  
116.2.14 Coercion 3741  
116.2.15 Modular Symbols to Homology 3744  
116.2.16 Embeddings 3745  
116.2.17 Base Change 3747  
116.2.18 Additional Examples 3748  
116.3 Homology 3751  
116.3.1 Creation 3751  
116.3.2 Invariants 3752  
116.3.3 Functors to Categories of Lattices and Vector Spaces 3752  
116.3.4 Modular Structure 3754  
116.3.5 Additional Examples 3755  
116.4 Homomorphisms 3756  
116.4.1 Creation 3757  
116.4.2 Restriction, Evaluation, and Other Manipulations 3758  
116.4.3 Kernels 3762  
116.4.4 Images 3763  
116.4.5 Cokernels 3765  
116.4.6 Matrix Structure 3766  
116.4.7 Arithmetic 3768  
116.4.8 Polynomials 3771
116.4.9 Invariants 3772
116.4.10 Predicates 3773

116.5 Endomorphism Algebras and Hom Spaces 3776
116.5.1 Creation 3776
116.5.2 Subgroups and Subrings 3777
116.5.3 Pullback and Pushforward of Hom Spaces 3780
116.5.4 Arithmetic 3780
116.5.5 Quotients 3781
116.5.6 Invariants 3782
116.5.7 Structural Invariants 3784
116.5.8 Matrix and Module Structure 3785
116.5.9 Predicates 3787
116.5.10 Elements 3789

116.6 Arithmetic of Abelian Varieties 3790
116.6.1 Direct Sum 3790
116.6.2 Sum in an Ambient Variety 3792
116.6.3 Intersections 3793
116.6.4 Quotients 3795

116.7 Decomposing and Factoring Abelian Varieties 3796
116.7.1 Decomposition 3796
116.7.2 Factorization 3797
116.7.3 Decomposition with respect to an Endomorphism or a Commutative Ring 3798
116.7.4 Additional Examples 3798

116.8 Building blocks 3800
116.8.1 Background and Notation 3800

116.9 Orthogonal Complements 3804
116.9.1 Complements 3804
116.9.2 Dual Abelian Variety 3805
116.9.3 Intersection Pairing 3807
116.9.4 Projections 3808
116.9.5 Left and Right Inverses 3809
116.9.6 Congruence Computations 3811

116.10 New and Old Subvarieties and Natural Maps 3812
116.10.1 Natural Maps 3812
116.10.2 New Subvarieties and Quotients 3814
116.10.3 Old Subvarieties and Quotients 3815

116.11 Elements of Modular Abelian Varieties 3816
116.11.1 Arithmetic 3817
116.11.2 Invariants 3818
116.11.3 Predicates 3819
116.11.4 Homomorphisms 3821
116.11.5 Representation of Torsion Points 3822

116.12 Subgroups of Modular Abelian Varieties 3823
116.12.1 Creation 3823
116.12.2 Elements 3825
116.12.3 Arithmetic 3826
116.12.4 Underlying Abelian Group and Lattice 3828
116.12.5 Invariants 3829
116.12.6 Predicates and Comparisons 3830

116.13 Rational Torsion Subgroups 3832
116.13.1 Cuspidal Subgroup 3832
116.13.2 Upper and Lower Bounds 3834
116.13.3 Torsion Subgroup 3835

116.14 Hecke and Atkin-Lehner Operators 3835
116.14.1 Creation 3835
116.14.2 Invariants 3837

116.15 $L$-series 3838
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>116.15.1</td>
<td>Creation</td>
<td>3838</td>
</tr>
<tr>
<td>116.15.2</td>
<td>Invariants</td>
<td>3839</td>
</tr>
<tr>
<td>116.15.3</td>
<td>Characteristic Polynomials of Frobenius Elements</td>
<td>3840</td>
</tr>
<tr>
<td>116.15.4</td>
<td>Values at Integers in the Critical Strip</td>
<td>3841</td>
</tr>
<tr>
<td>116.15.5</td>
<td>Leading Coefficient</td>
<td>3843</td>
</tr>
<tr>
<td>116.16</td>
<td>Complex Period Lattice</td>
<td>3844</td>
</tr>
<tr>
<td>116.16.1</td>
<td>Period Map</td>
<td>3844</td>
</tr>
<tr>
<td>116.16.2</td>
<td>Period Lattice</td>
<td>3844</td>
</tr>
<tr>
<td>116.17</td>
<td>Tamagawa Numbers and Component Groups of Neron Models</td>
<td>3844</td>
</tr>
<tr>
<td>116.17.1</td>
<td>Component Groups</td>
<td>3844</td>
</tr>
<tr>
<td>116.17.2</td>
<td>Tamagawa Numbers</td>
<td>3845</td>
</tr>
<tr>
<td>116.18</td>
<td>Elliptic Curves</td>
<td>3846</td>
</tr>
<tr>
<td>116.18.1</td>
<td>Creation</td>
<td>3846</td>
</tr>
<tr>
<td>116.18.2</td>
<td>Invariants</td>
<td>3847</td>
</tr>
<tr>
<td>116.19</td>
<td>Bibliography</td>
<td>3848</td>
</tr>
</tbody>
</table>
VOLUME 12: CONTENTS

XVIII GEOMETRY

117  FINITE PLANES  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3851

117.1  Introduction  3853
117.1.1  Planes in Magma  3853
117.2  Construction of a Plane  3853
117.3  The Point-Set and Line-Set of a Plane  3856
117.3.1  Introduction  3856
117.3.2  Creating Point-Sets and Line-Sets  3856
117.3.3  Using the Point-Set and Line-Set to Create Points and Lines  3856
117.3.4  Retrieving the Plane from Points, Lines, Point-Sets and Line-Sets  3860
117.4  The Set of Points and Set of Lines  3860
117.5  The Defining Points of a Plane  3861
117.6  Subplanes  3862
117.7  Structures Associated with a Plane  3863
117.8  Numerical Invariants of a Plane  3864
117.9  Properties of Planes  3865
117.10  Identity and Isomorphism  3865
117.11  The Connection between Projective and Affine Planes  3866
117.12  Operations on Points and Lines  3867
117.12.1  Elementary Operations  3867
117.12.2  Deconstruction Functions  3868
117.12.3  Other Point and Line Functions  3871
117.13  Arcs  3872
117.14  Unital  3875
117.15  The Collineation Group of a Plane  3876
117.15.1  The Collineation Group Function  3877
117.15.2  General Action of Collineations  3878
117.15.3  Central Collineations  3882
117.15.4  Transitivity Properties  3883
117.16  Translation Planes  3884
117.17  Planes and Designs  3884
117.18  Planes, Graphs and Codes  3885

118  INCIDENCE GEOMETRY  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3887

118.1  Introduction  3889
118.2  Construction of Incidence and Coset Geometries  3890
118.2.1  Construction of an Incidence Geometry  3890
118.2.2  Construction of a Coset Geometry  3894
118.3  Elementary Invariants  3897
118.4  Conversion Functions  3899
118.5  Residues  3900
118.6  Truncations  3901
118.7  Shadows  3901
118.8  Shadow Spaces  3901
118.9  Automorphism Group and Correlation Group  3902
118.10  Properties of Incidence Geometries and Coset Geometries  3902
118.11  Intersection Properties of Coset Geometries  3903
118.12  Primitivity Properties on Coset Geometries  3904
118.13  Diagram of an Incidence Geometry  3905
118.14  Bibliography  3908
VOLUME 12: CONTENTS

XIX  COMBINATORICS  3909

119  ENUMERATIVE COMBINATORICS  . . . . . . . . . . . . . . . . 3911
119.1  Introduction  3913
119.2  Combinatorial Functions  3913
119.3  Subsets of a Finite Set  3915

120  PARTITIONS, WORDS AND YOUNG TABLEAUX  . . . . . . 3917
120.1  Introduction  3919
120.2  Partitions  3919
120.3  Words  3922
120.3.1  Ordered Monoids  3922
120.3.2  Plactic Monoids  3925
120.4  Tableaux  3928
120.4.1  Tableau Monoids  3928
120.4.2  Creation of Tableaux  3930
120.4.3  Enumeration of Tableaux  3933
120.4.4  Random Tableaux  3935
120.4.5  Basic Access Functions  3936
120.4.6  Properties  3939
120.4.7  Operations  3941
120.4.8  The Robinson-Schensted-Knuth Correspondence  3944
120.4.9  Counting Tableaux  3948
120.5  Bibliography  3950

121  SYMMETRIC FUNCTIONS  . . . . . . . . . . . . . . . . . . . . . 3951
121.1  Introduction  3953
121.2  Creation  3955
121.2.1  Creation of Symmetric Function Algebras  3955
121.2.2  Creation of Symmetric Functions  3957
121.3  Structure Operations  3960
121.3.1  Related Structures  3960
121.3.2  Ring Predicates and Booleans  3961
121.3.3  Predicates on Basis Types  3961
121.4  Element Operations  3961
121.4.1  Parent and Category  3961
121.4.2  Print Styles  3962
121.4.3  Additive Arithmetic Operators  3962
121.4.4  Multiplication  3963
121.4.5  Plethysm  3964
121.4.6  Boolean Operators  3964
121.4.7  Accessing Elements  3965
121.4.8  Multivariate Polynomials  3966
121.4.9  Frobenius Homomorphism  3967
121.4.10 Inner Product  3968
121.4.11 Combinatorial Objects  3968
121.4.12 Symmetric Group Character  3968
121.4.13 Restrictions  3969
121.5  Transition Matrices  3970
121.5.1  Transition Matrices from Schur Basis  3970
121.5.2  Transition Matrices from Monomial Basis  3972
121.5.3  Transition Matrices from Homogeneous Basis  3973
121.5.4  Transition Matrices from Power Sum Basis  3974
122 INCIDENCE STRUCTURES AND DESIGNS . . . . . . . . 3977

122.1 Introduction 3979
122.2 Construction of Incidence Structures and Designs 3980
122.3 The Point-Set and Block-Set of an Incidence Structure 3984
122.3.1 Introduction 3984
122.3.2 Creating Point-Sets and Block-Sets 3985
122.3.3 Creating Points and Blocks 3985
122.4 General Design Constructions 3987
122.4.1 The Construction of Related Structures 3987
122.4.2 The Witt Designs 3990
122.4.3 Difference Sets and their Development 3990
122.5 Elementary Invariants of an Incidence Structure 3992
122.6 Elementary Invariants of a Design 3993
122.7 Operations on Points and Blocks 3995
122.8 Elementary Properties of Incidence Structures and Designs 3997
122.9 Resolutions, Parallelisms and Parallel Classes 3999
122.10 Conversion Functions 4002
122.11 Identity and Isomorphism 4003
122.12 The Automorphism Group of an Incidence Structure 4004
122.12.1 Construction of Automorphism Groups 4004
122.12.2 Action of Automorphisms 4007
122.13 Incidence Structures, Graphs and Codes 4009
122.14 Bibliography 4010

123 HADAMARD MATRICES . . . . . . . . . . . . . . . . . 4011

123.1 Introduction 4013
123.2 Equivalence Testing 4013
123.3 Associated 3–Designs 4015
123.4 Automorphism Group 4016
123.5 Databases 4016
123.5.1 Updating the Databases 4017

124 GRAPHS . . . . . . . . . . . . . . . . . . . . . . . . 4021

124.1 Introduction 4025
124.2 Construction of Graphs and Digraphs 4026
124.2.1 Bounds on the Graph Order 4026
124.2.2 Construction of a General Graph 4027
124.2.3 Construction of a General Digraph 4030
124.2.4 Operations on the Support 4032
124.2.5 Construction of a Standard Graph 4033
124.2.6 Construction of a Standard Digraph 4035
124.3 Graphs with a Sparse Representation 4036
124.4 The Vertex–Set and Edge–Set of a Graph 4038
124.4.1 Introduction 4038
124.4.2 Creating Edges and Vertices 4038
124.4.3 Operations on Vertex-Sets and Edge-Sets 4040
124.4.4 Operations on Edges and Vertices 4041
124.5 Labelled, Capacitated and Weighted Graphs 4042
124.6 Standard Constructions for Graphs 4042
124.6.1 Subgraphs and Quotient Graphs 4042
124.6.2 Incremental Construction of Graphs 4044
124.6.3 Constructing Complements, Line Graphs; Contraction, Switching 4047
124.7 Unions and Products of Graphs 4049
124.8 Converting between Graphs and Digraphs 4051
124.9 Construction from Groups, Codes and Designs 4051
124.9.1 Graphs Constructed from Groups 4051
124.9.2 Graphs Constructed from Designs 4052
124.9.3 Miscellaneous Graph Constructions 4053
124.10 Elementary Invariants of a Graph 4054
124.11 Elementary Graph Predicates 4055
124.12 Adjacency and Degree 4057
124.12.1 Adjacency and Degree Functions for a Graph 4057
124.12.2 Adjacency and Degree Functions for a Digraph 4058
124.13 Connectedness 4060
124.13.1 Connectedness in a Graph 4060
124.13.2 Connectedness in a Digraph 4061
124.13.3 Graph Triconnectivity 4061
124.13.4 Maximum Matching in Bipartite Graphs 4063
124.13.5 General Vertex and Edge Connectivity in Graphs and Digraphs 4064
124.14 Distances, Paths and Circuits in a Graph 4067
124.14.1 Distances, Paths and Circuits in a Possibly Weighted Graph 4067
124.14.2 Distances, Paths and Circuits in a Non-Weighted Graph 4067
124.15 Maximum Flow, Minimum Cut, and Shortest Paths 4068
124.16 Matrices and Vector Spaces Associated with a Graph or Digraph 4069
124.17 Spanning Trees of a Graph or Digraph 4069
124.18 Directed Trees 4070
124.19 Colourings 4071
124.20 Cliques, Independent Sets 4072
124.21 Planar Graphs 4077
124.22 Automorphism Group of a Graph or Digraph 4080
124.22.1 The Automorphism Group Function 4080
124.22.2 nauty Invariants 4081
124.22.3 Graph Colouring and Automorphism Group 4083
124.22.4 Variants of Automorphism Group 4084
124.22.5 Action of Automorphisms 4088
124.23 Symmetry and Regularity Properties of Graphs 4091
124.24 Graph Database and Graph Generation 4093
124.24.1 Strongly Regular Graphs 4093
124.24.2 Generating Graphs 4095
124.24.3 A General Facility 4098
124.25 Bibliography 4100

125 MULTIGRAPHS .................. 4101
125.1 Introduction 4105
125.2 Construction of Multigraphs 4106
125.2.1 Construction of a General Multigraph 4106
125.2.2 Construction of a General Multidigraph 4107
125.2.3 Printing of a Multi(di)graph 4108
125.2.4 Operations on the Support 4109
125.3 The Vertex–Set and Edge–Set of Multigraphs 4110
125.4 Vertex and Edge Decorations 4113
125.4.1 Vertex Decorations: Labels 4113
125.4.2 Edge Decorations 4114
125.4.3 Unlabelled, or Uncapacitated, or Unweighted Graphs 4117
125.5 Standard Construction for Multigraphs 4120
125.5.1 Subgraphs 4120
125.5.2 Incremental Construction of Multigraphs 4122
125.5.3 Vertex Insertion, Contraction 4126
125.5.4 Unions of Multigraphs 4127
125.6 Conversion Functions 4128
125.6.1 Orientated Graphs 4129
125.6.2 Converse 4129
125.6.3 Converting between Simple Graphs and Multigraphs 4129
125.7 Elementary Invariants and Predicates for Multigraphs 4130
125.8 Adjacency and Degree 4132
125.8.1 Adjacency and Degree Functions for Multigraphs 4132
125.8.2 Adjacency and Degree Functions for Multidigraphs 4133
125.9 Connectedness 4135
125.9.1 Connectedness in a Multigraph 4135
125.9.2 Connectedness in a Multidigraph 4136
125.9.3 Triconnectivity for Multigraphs 4136
125.9.4 Maximum Matching in Bipartite Multigraphs 4137
125.9.5 General Vertex and Edge Connectivity in Multigraphs and Multidigraphs 4137
125.10 Spanning Trees 4139
125.11 Planar Graphs 4139
125.12 Distances, Shortest Paths and Minimum Weight Trees 4143
125.13 Bibliography 4148

126 NETWORKS . . . . . . . . . . . . . . . . . . . . . . 4149
126.1 Introduction 4151
126.2 Construction of Networks 4151
126.2.1 Magma Output: Printing of a Network 4153
126.3 Standard Construction for Networks 4155
126.3.1 Subgraphs 4155
126.3.2 Incremental Construction: Adding Edges 4159
126.3.3 Union of Networks 4160
126.4 Maximum Flow and Minimum Cut 4161
126.5 Bibliography 4167
XX CODING THEORY 4169

127 LINEAR CODES OVER FINITE FIELDS . . . . . . . . . . 4171

127.1 Introduction 4175
127.2 Construction of Codes 4176
127.2.1 Construction of General Linear Codes 4176
127.2.2 Some Trivial Linear Codes 4178
127.2.3 Some Basic Families of Codes 4179
127.3 Invariants of a Code 4181
127.3.1 Basic Numerical Invariants 4181
127.3.2 The Ambient Space and Alphabet 4181
127.3.3 The Code Space 4182
127.3.4 The Dual Space 4182
127.3.5 The Information Space and Information Sets 4183
127.3.6 The Syndrome Space 4184
127.3.7 The Generator Polynomial 4185
127.4 Operations on Codewords 4186
127.4.1 Construction of a Codeword 4186
127.4.2 Arithmetic Operations on Codewords 4186
127.4.3 Distance and Weight 4187
127.4.4 Vector Space and Related Operations 4187
127.4.5 Predicates for Codewords 4188
127.4.6 Accessing Components of a Codeword 4189
127.5 Coset Leaders 4189
127.6 Subcodes 4191
127.6.1 The Subcode Constructor 4191
127.6.2 Sum, Intersection and Dual 4192
127.6.3 Membership and Equality 4194
127.7 Properties of Codes 4194
127.8 The Weight Distribution 4196
127.8.1 The Minimum Weight 4196
127.8.2 The Weight Distribution 4201
127.8.3 The Weight Enumerator 4202
127.8.4 The MacWilliams Transform 4204
127.8.5 Words 4205
127.8.6 Covering Radius and Diameter 4207
127.9 Families of Linear Codes 4208
127.9.1 Cyclic and Quasicyclic Codes 4208
127.9.2 BCH Codes and their Generalizations 4210
127.9.3 Quadratic Residue Codes and their Generalizations 4213
127.9.4 Reed–Solomon and Justesen Codes 4214
127.9.5 Maximum Distance Separable Codes 4215
127.10 New Codes from Existing 4215
127.10.1 Standard Constructions 4215
127.10.2 Changing the Alphabet of a Code 4218
127.10.3 Combining Codes 4219
127.11 Coding Theory and Cryptography 4224
127.11.1 Standard Attacks 4224
127.11.2 Generalized Attacks 4226
127.12 Bounds

127.12.1 Best Known Bounds for Linear Codes

127.12.2 Bounds on the Cardinality of a Largest Code

127.12.3 Bounds on the Minimum Distance

127.12.4 Asymptotic Bounds on the Information Rate

127.12.5 Other Bounds

127.13 Best Known Linear Codes

127.14 Decoding

127.15 Transforms

127.15.1 Mattson–Solomon Transforms

127.15.2 Krawchouk Polynomials

127.16 Automorphism Groups

127.16.1 Introduction

127.16.2 Group Actions

127.16.3 Automorphism Group

127.16.4 Equivalence and Isomorphism of Codes

127.17 Bibliography

128 ALGEBRAIC-GEOMETRIC CODES

128.1 Introduction

128.2 Creation of an Algebraic Geometric Code

128.3 Properties of AG–Codes

128.4 Access Functions

128.5 Decoding AG Codes

128.6 Bibliography

129 LOW DENSITY PARTY CHECK CODES

129.1 Introduction

129.1.1 Constructing LDPC Codes

129.1.2 Access Functions

129.1.3 LDPC Decoding and Simulation

129.1.4 Density Evolution

130 LINEAR CODES OVER FINITE RINGS

130.1 Introduction

130.2 Construction of Codes

130.2.1 Construction of General Linear Codes

130.2.2 Construction of Simple Linear Codes

130.2.3 Construction of General Cyclic Codes

130.3 Invariants of Codes

130.4 Codes over $\mathbb{Z}_4$

130.4.1 The Gray Map

130.4.2 Families of Codes over $\mathbb{Z}_4$

130.4.3 Derived Binary Codes

130.4.4 The Standard Form

130.4.5 Other $\mathbb{Z}_4$ functions

130.5 Construction of Subcodes of Linear Codes

130.5.1 The Subcode Constructor

130.6 Weight Distributions

130.6.1 Hamming Weight

130.6.2 Lee Weight

130.6.3 Euclidean Weight

130.7 Weight Enumerators
## VOLUME 13: CONTENTS

**130.8** Constructing New Codes from Old

- **130.8.1** Sum, Intersection and Dual
- **130.8.2** Standard Constructions

**130.9** Operations on Codewords

- **130.9.1** Construction of a Codeword
- **130.9.2** Operations on Codewords and Vectors
- **130.9.3** Accessing Components of a Codeword

**130.10** Boolean Predicates

**130.11** Bibliography

---

**131** ADDITIVE CODES

- **131.1** Introduction
- **131.2** Construction of Additive Codes
  - **131.2.1** Construction of General Additive Codes
  - **131.2.2** Some Trivial Additive Codes
- **131.3** Invariants of an Additive Code
  - **131.3.1** The Ambient Space and Alphabet
  - **131.3.2** Basic Numerical Invariants
  - **131.3.3** The Code Space
  - **131.3.4** The Dual Space
- **131.4** Operations on Codewords
  - **131.4.1** Construction of a Codeword
  - **131.4.2** Arithmetic Operations on Codewords
  - **131.4.3** Distance and Weight
  - **131.4.4** Vector Space and Related Operations
  - **131.4.5** Predicates for Codewords
  - **131.4.6** Accessing Components of a Codeword
- **131.5** Subcodes
  - **131.5.1** The Subcode Constructor
  - **131.5.2** Sum, Intersection and Dual
  - **131.5.3** Membership and Equality
- **131.6** Properties of Codes
- **131.7** The Weight Distribution
  - **131.7.1** The Minimum Weight
  - **131.7.2** The Weight Distribution
  - **131.7.3** The Weight Enumerator
  - **131.7.4** The MacWilliams Transform
  - **131.7.5** Words
- **131.8** Families of Linear Codes
  - **131.8.1** Cyclic Codes
  - **131.8.2** Quasicyclic Codes
- **131.9** New Codes from Old
  - **131.9.1** Standard Constructions
  - **131.9.2** Combining Codes
  - **131.9.3** Sum, Intersection and Dual
  - **131.9.4** Membership and Equality

---

**132** QUANTUM CODES

- **132.1** Introduction
- **132.2** Constructing Quantum Codes
  - **132.2.1** Construction of General Quantum Codes
  - **132.2.2** Construction of Special Quantum Codes
  - **132.2.3** CSS Codes
  - **132.2.4** Cyclic Quantum Codes
  - **132.2.5** Quasi-Cyclic Quantum Codes
| 132.3  | Access Functions                           | 4335  |
| 132.3.1 | Quantum Error Group                       | 4336  |
| 132.4   | Inner Products and Duals                  | 4338  |
| 132.5   | Weight Distribution and Minimum Weight    | 4340  |
| 132.6   | New Codes From Old                        | 4343  |
| 132.7   | Best Known Quantum Codes                  | 4344  |
| 132.8   | Best Known Bounds                         | 4347  |
| 132.9   | Automorphism Group                        | 4348  |
| 132.10  | Hilbert Spaces                            | 4350  |
| 132.10.1| Creation of Quantum States                | 4351  |
| 132.10.2| Manipulation of Quantum States            | 4353  |
| 132.10.3| Inner Product and Probabilities of Quantum States | 4354 |
| 132.10.4| Unitary Transformations on Quantum States  | 4357  |
| 132.11  | Bibliography                              | 4358  |
## XXI CRYPTOGRAPHY

### 133 PSEUDO-RANDOM BIT SEQUENCES

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>133.1</td>
<td>Introduction</td>
<td>4363</td>
</tr>
<tr>
<td>133.2</td>
<td>Linear Feedback Shift Registers</td>
<td>4363</td>
</tr>
<tr>
<td>133.3</td>
<td>Number Theoretic Bit Generators</td>
<td>4364</td>
</tr>
<tr>
<td>133.4</td>
<td>Correlation Functions</td>
<td>4366</td>
</tr>
<tr>
<td>133.5</td>
<td>Decimation</td>
<td>4367</td>
</tr>
</tbody>
</table>
XXII  OPTIMIZATION  4369

134  LINEAR PROGRAMMING  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4371

134.1  Introduction  4373
134.2  Explicit LP Solving Functions  4374
134.3  Creation of LP objects  4376
134.4  Operations on LP objects  4376
134.5  Bibliography  4379
PART I
THE MAGMA LANGUAGE

1 STATEMENTS AND EXPRESSIONS 3
2 FUNCTIONS, PROCEDURES AND PACKAGES 33
3 INPUT AND OUTPUT 57
4 ENVIRONMENT AND OPTIONS 87
5 MAGMA SEMANTICS 109
6 THE MAGMA PROFILER 129
7 DEBUGGING MAGMA CODE 139
1 STATEMENTS AND EXPRESSIONS

1.1 Introduction . . . . . . . . . 5
1.2 Starting, Interrupting and Terminating . . . . . . . . . . . . 5

<Ctrl>-C 5
quit; 5
<Ctrl>-D 5
<Ctrl>\ 5

1.3 Identifiers . . . . . . . . . . 5
1.4 Assignment . . . . . . . . . . 6

1.4.1 Simple Assignment . . . . . . . . 6
x := e; 6
x, x, ..., x := e; 6
 := e; 6
assigned 6

1.4.2 Indexed Assignment . . . . . . . . 7
x[e1][e2][e3] := e; 7
x[e1,e2,...,en] := e; 7

1.4.3 Generator Assignment . . . . . . . 8
E<x, x, ...x> := e; 8
E<x> := e; 8
AssignNames(~S, [s1, ... sn]) 9

1.4.4 Mutation Assignment . . . . . . . 9
x o:= e; 9

1.4.5 Deletion of Values . . . . . . . 10
delete 10

1.5 Boolean values . . . . . . . . 10

1.5.1 Creation of Booleans . . . . . . . . 11
Booleans() 11
# 11
true 11
false 11
Random(B) 11

1.5.2 Boolean Operators . . . . . . . . 11
and 11
or 11
xor 11
not 11

1.5.3 Equality Operators . . . . . . . . 11
eq 11
ne 12
cmpeq 12
cmpne 12

1.5.4 Iteration . . . . . . . . . . . . 12

1.6 Coercion . . . . . . . . . . . 13

IsCoercible(S, x) 13

1.7 The where ... is Construction . . 14
e1 where id is e2 14
e1 where id := e2 14

1.8 Conditional Statements and Expressions . . . . . . . . . . . . 16

1.8.1 The Simple Conditional Statement . . . . . . . . . . . . 16

1.8.2 The Simple Conditional Expression . . . . . . . . . . . . 17
bool select e1 else e2 17

1.8.3 The Case Statement . . . . . . . . 18

1.8.4 The Case Expression . . . . . . . . 18

1.9 Error Handling Statements . . . 19

1.9.1 The Error Objects . . . . . . . . . 19
Error(x) 19
e'Position 19
e'Traceback 19
e'Object 19
e'Type 19

1.9.2 Error Checking and Assertions . . . . 19
error e, ..., e; 19
error if bool, e, ..., e; 19
assert bool; 20

1.9.3 Catching Errors . . . . . . . . . . 20

1.10 Iterative Statements . . . . . 21

1.10.1 Definite Iteration . . . . . . . . . . 21

1.10.2 Indefinite Iteration . . . . . . . . . 21

1.10.3 Early Exit from Iterative Statements . . . . . . . . . . 23
continue; 23
continue id; 23
break; 23
break id; 23

1.11 Runtime Evaluation: the eval Expression . . . . . . . . . . . . 24
eval expression 24

1.12 Comments and Continuation . . 25
// 25
/* */ 25
\ 25

1.13 Timing . . . . . . . . . . . . 26
Cputime() 26
Cputime(t) 26
Realtime() 26
Realtime(t) 26
THE MAGMA LANGUAGE

Part I

1.14 Types, Category Names, and Structures . . . . . . . . . . 28
  Type(x) 28
  Category(x) 28
  ExtendedType(x) 28
  ExtendedCategory(x) 28
  ISA(T, U) 28
  MakeType(S) 28
  ElementType(S) 28
  CoveringStructure(S, T) 28

1.15 Random Object Generation . . . 30
  SetSeed(s, c) 30
  SetSeed(s) 30
  GetSeed() 30
  Random(S) 30
  Random(a, b) 30
  Random(b) 30

1.16 Miscellaneous . . . . . . . . . . . . . . . 32
  IsIntrinsic(S) 32

1.17 Bibliography . . . . . . . . . . . . . . . 32
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’.

\[\text{<Ctrl>-C}\]

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 \[\text{<Ctrl>-C}\] is typed twice within half a second, MAGMA will exit completely immediately.

\[\text{quit;}\]
\[\text{<Ctrl>-D}\]

Terminate the current MAGMA-session.

\[\text{<Ctrl>-\}\]

Immediately quit MAGMA (send the signal SIGQUIT to the MAGMA process on Unix machines). This is occasionally useful when \[\text{<Ctrl>-C}\] 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 \text{{pCore}}, \text{{pQuotient}} and the like, where the \(p\) indicates a prime). Note that these identifiers are \textit{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 \texttt{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 \textit{not} change the corresponding mutation assignment operator (in this case \( += \)).

\[
x_1, x_2, \ldots, x_n := expression; \\
\]

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

\[
_ := expression; \\
\]

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

\[
\text{assigned } x \\
\]

An expression which yields the value \texttt{true} if the ‘local’ identifier \( x \) has a value currently assigned to it and \texttt{false} otherwise. Note that the \texttt{assigned}-expression will return \texttt{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 \texttt{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:

\begin{verbatim}
> d := Xgcd(12, 15);
\end{verbatim}

To obtain the multipliers as well, type

\begin{verbatim}
> d, x, y := Xgcd(12, 15);
\end{verbatim}

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

\begin{verbatim}
> d, x := Xgcd(12, 15);
> d, _, y := Xgcd(12, 15);
> _, x, y := Xgcd(12, 15);
\end{verbatim}

1.4.2 Indexed Assignment

\begin{verbatim}
x[expression_1][expression_2]...[expression_n] := expression;
x[expression_1,expression_2,...,expression_n] := expression;
\end{verbatim}

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 \texttt{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.

\begin{verbatim}
> 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 ]
]
\end{verbatim}
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.

\[
E< x_1, x_2, \ldots, x_n > := \text{expression};
\]

If the right hand side expression returns a structure that allows naming of ‘generators’, such as finitely generated groups or algebras, polynomial rings, this assigns the first \( n \) names to the variables \( x_1, x_2, \ldots, x_n \). Naming of generators usually has two aspects; firstly, the strings \( x_1, x_2, \ldots, x_n \) are used for printing of the generators, and secondly, to the identifiers \( x_1, x_2, \ldots, x_n \) 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:

\[
E := \text{expression};
\]

\[
x_1 := E.1; x_2 := E.2; \ldots x_n := E.n;
\]

\[
E<[x]> := \text{expression};
\]

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

Example H1E4

We demonstrate the sequence method of generator naming.

\[
> \text{P}<[X]> := \text{PolynomialRing(RationalField(), 5)};
> \text{P};
\]

Polynomial ring of rank 5 over Rational Field
Lexicographical Order
Variables: X[1], X[2], X[3], X[4], X[5]
\[
> \text{X} ;
[
X[1], \\
X[2], \\
X[3], \\
X[4], \\
X[5]
]
\]

\[
> \&*\text{X} ;
\]

\[
> (\&*\text{X})^2;
\]
AssignNames(~S, [s₁, ... sₙ])

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⁻¹b*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.
The following simple program to produce a set consisting of the first 10 powers of 2 involves the use of two different mutation assignments.

```plaintext
> 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:

```plaintext
> 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

\begin{itemize}
\item \texttt{Booleans()}
  \begin{itemize}
  \item The Boolean structure.
  \end{itemize}
\item \texttt{#B}
  \begin{itemize}
  \item Cardinality of Boolean structure (2).
  \end{itemize}
\item \texttt{true}
  \begin{itemize}
  \item The Boolean elements.
  \end{itemize}
\item \texttt{false}
  \begin{itemize}
  \item The Boolean elements.
  \end{itemize}
\item \texttt{Random(B)}
  \begin{itemize}
  \item Return a random Boolean.
  \end{itemize}
\end{itemize}

1.5.2 Boolean Operators

\begin{itemize}
\item \texttt{x and y}
  \begin{itemize}
  \item Returns \texttt{true} if both \texttt{x} and \texttt{y} are \texttt{true}, \texttt{false} otherwise. If \texttt{x} is \texttt{false}, the expression for \texttt{y} is not evaluated.
  \end{itemize}
\item \texttt{x or y}
  \begin{itemize}
  \item Returns \texttt{true} if \texttt{x} or \texttt{y} is \texttt{true} (or both are \texttt{true}), \texttt{false} otherwise. If \texttt{x} is \texttt{true}, the expression for \texttt{y} is not evaluated.
  \end{itemize}
\item \texttt{x xor y}
  \begin{itemize}
  \item Returns \texttt{true} if either \texttt{x} or \texttt{y} is \texttt{true} (but not both), \texttt{false} otherwise.
  \end{itemize}
\item \texttt{not x}
  \begin{itemize}
  \item Negate the truth value of \texttt{x}.
  \end{itemize}
\end{itemize}

1.5.3 Equality Operators

\begin{itemize}
\item \texttt{Magma} provides two equality operators: \texttt{eq} for strong (comparable) equality testing, and \texttt{cmpeq} for weak equality testing. The operators depend on the concept of \textit{comparability}. Objects \texttt{x} and \texttt{y} in \texttt{Magma} are said to be \textit{comparable} if both of the following points hold:
\begin{enumerate}
\item \texttt{x} and \texttt{y} are both elements of a structure \texttt{S} or there is a structure \texttt{S} such \texttt{x} and \texttt{y} will be coerced into \texttt{S} by automatic coercion;
\item There is an equality test for elements of \texttt{S} defined within \texttt{Magma}.
\end{enumerate}

The possible automatic coercions are listed in the descriptions of the various \texttt{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.

\begin{itemize}
\item \texttt{x eq y}
  \begin{itemize}
  \item If \texttt{x} and \texttt{y} are comparable, return \texttt{true} if \texttt{x} equals \texttt{y} (which will always work by the second rule above). If \texttt{x} and \texttt{y} are not comparable, an error results.
  \end{itemize}
\end{itemize}
\[ x \text{ ne } y \]

If \( x \) and \( y \) are comparable, return \textbf{true} if \( x \) does not equal \( y \). If \( x \) and \( y \) are not comparable, an error results.

\[ x \text{ cmpeq } y \]

If \( x \) and \( y \) are comparable, return whether \( x \) equals \( y \). Otherwise, return \textbf{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 \texttt{eq} is usually used to allow MAGMA to pick up common unintentional type errors.

\[ x \text{ cmpne } y \]

If \( x \) and \( y \) are comparable, return whether \( x \) does not equal \( y \). Otherwise, return \textbf{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 \texttt{ne} is usually used to allow MAGMA to pick up common unintentional type errors.

\[ \text{Example H1E7} \]

We illustrate the different semantics of \texttt{eq} and \texttt{cmpeq}.

\>
> 1 \texttt{eq} 2/2;
true
> 1 \texttt{cmpeq} 2/2;
true
> 1 \texttt{eq} "x";
Runtime error in \texttt{eq}: Bad argument types
> 1 \texttt{cmpeq} "x";
false
> [1] \texttt{eq} ["x"];
Runtime error in \texttt{eq}: Incompatible sequences
> [1] \texttt{cmpeq} ["x"];
false

1.5.4 Iteration

A Boolean structure \( B \) may be used for enumeration: \texttt{for x in B do}, and \texttt{x in B in set} and sequence constructors.
Example H1E8

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

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

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

\[ x = y \iff (x \land y) \lor (\neg x \land \neg y). \]

```

```
> 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.

\[ \text{S} ! \text{x} \]

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.

\[ \text{IsCoercible}(S, x) \]

Given a structure \(S\) and an object \(x\), attempt to coerce \(x\) into \(S\); if successful, return \text{true} and the result of the coercion, otherwise return \text{false}. 
1.7 The where ... is Construction

By the use of the \texttt{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.

\begin{align*}
\text{expression}_1 \text{ where } \text{identifier is } \text{expression}_2 \\
\text{expression}_1 \text{ where } \text{identifier := expression}_2
\end{align*}

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 \texttt{where ... is} construction alone except for when the construction is part of an expression list — see below.

The \texttt{where} operator is left-associative. This means that there can be multiple uses of \texttt{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 \texttt{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 \texttt{where} operator also extends left in expression lists. That is, if there is an expression $E$ in a expression list which is a \texttt{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 \texttt{where ... is}.

\begin{verbatim}
> x := 1;
> x where x is 10;
10
> x;
1
> Order(G) + Degree(G) where G is Sym(3);
9
\end{verbatim}

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

\begin{verbatim}
> x := 1;
\end{verbatim}
> 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
else
    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 then
    statements
elif Boolean expression then
    statements
else
    statements
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
elif Boolean expression then
    statements
else
    statements
end if;
```

is equivalent to

```
if Boolean expression then
    statements
else
    if Boolean expression then
```

...
Example H1E10

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

1.8.2 The Simple Conditional Expression

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

Example H1E11

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

```plaintext
> 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:

```plaintext
> 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:

```plaintext
> 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

```plaintext
> 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_{left,i}` is evaluated until one is found whose value equals $v$; if this happens the value of the corresponding right-hand expression `expression_{right,i}` is returned. If no left-hand expression with value $v$ is found the value of the default expression `expression_{def}` is returned.

The default case cannot be omitted, and must come last.
1.9 Error Handling Statements

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

1.9.1 The Error Objects

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

Error(x)

Constructs an error object with user information given by x, which can be of any type. The object x is stored in the Object 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

eroerror 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.

eroerror 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;

If the given boolean expression evaluates to `false`, raise an error, with the error information being an appropriate message. If the `Assertions` flag is set to `false` (see `SetAssertions`), the check is not made and the statement has no effect.

1.9.3 Catching Errors

```plaintext
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.

```plaintext
> 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 the 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 $b + k \cdot s < e$ (for $e > 0$) or $b + k \cdot s > e$ (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}, \ldots, x_{1n_1} in S_1, \ldots, x_{m1}, \ldots, x_{mm_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.

```magma
> 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
```

```magma
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.

```magma
> x := Random(1, 1000);
> x;
172
> i := 0;
> repeat
>   while IsEven(x) do
>     i +:= 1;
>     x div:= 2;
```
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

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

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

46 89
89 46
1.11 Runtime Evaluation: the eval Expression

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

```
1+1
```

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.

```magma
> 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\n> 34;
> x;
1234

1.13 Timing

\textbf{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).

\textbf{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.

\textbf{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.

\textbf{Realtime(t)}

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

\textbf{time statement;}

Execute the statement and print the time taken when the statement is completed.
If the verbose flag \( \text{flag} \) (see the function \textbf{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 \textsc{magma} code found in packages where one wants to print the timing of some sub-algorithm if and only if an appropriate verbose flag is turned on.

**Example H1E20**

The \texttt{time} command can be used to time a single statement.

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

Alternatively, we can extract the current time \( t \) and use \texttt{Cputime}. This method can be used to time the execution of several statements.

```maple
> 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 \texttt{vtime} with \texttt{vprint} within a function.

```maple
> 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 \texttt{Cat}) are used. Examples of these kinds of types are “polynomial rings” (\texttt{RngUPol}) and “finite fields” (\texttt{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 \texttt{ECat}) can be thought of as a type taking a parameter. Using extended types, we can talk about “polynomial rings over the integers” (\texttt{RngUPol[RngInt]}), or “maps from the integers to the rationals” (\texttt{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.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{Type(x)} \\
\hline
\textbf{Category(x)} \\
\hline
\end{tabular}
\end{center}

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

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{ExtendedType(x)} \\
\hline
\textbf{ExtendedCategory(x)} \\
\hline
\end{tabular}
\end{center}

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

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{ISA(T, U)} \\
\hline
\end{tabular}
\end{center}

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, \(\text{ISA(RngInt, Rng)}\) is true, because the ring of integers \(\mathbb{Z}\) is a ring.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{MakeType(S)} \\
\hline
\end{tabular}
\end{center}

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.

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{ElementType(S)} \\
\hline
\end{tabular}
\end{center}

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

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{CoveringStructure(S, T)} \\
\hline
\end{tabular}
\end{center}

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.
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.

```plaintext
> 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.

```plaintext
> 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:

```plaintext
> 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)**

(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 $[1..4]$.

```plaintext
> 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 $[1..100]$.

```plaintext
> 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.

```plaintext
> 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.

```plaintext
> 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`.

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

1.17 Bibliography


2 FUNCTIONS, PROCEDURES
AND PACKAGES

2.1 Introduction . . . . . . . . . . 35

2.2 Functions and Procedures . . . 35
2.2.1 Functions . . . . . . . . . . . . 35
f := func< x₁, ..., xₙ : - | e>; 36
f := func< x₁, ..., xₙ, ... : - | e>; 36
2.2.2 Procedures . . . . . . . . . . . . 39
p := proc< x₁, ..., xₙ : - | e>; 40
p := proc< x₁, ..., xₙ, ... : - | e>; 40
2.2.3 The forward Declaration . . . . . . 41
forward 41

2.3 Packages . . . . . . . . . . . 42
2.3.1 Introduction . . . . . . . . . . . . 42
2.3.2 Intrinsics . . . . . . . . . . . . . 43
intrinsic 43
2.3.3 Resolving calls to intrinsics . . . . . 45
2.3.4 Attaching and Detaching Package Files46
Attach(F) 47
Detach(F) 47
freeze; 47
2.3.5 Related Files . . . . . . . . . . . . 47
2.3.6 Importing Constants . . . . . . . . 47
import "filename": ident_list; 47
2.3.7 Argument Checking . . . . . . . . . . . . 48
require condition: print_args; 48
requirerange v, L, U; 48
requirege v, L; 48
2.3.8 Package Specification files . . . . . . 49
AttachSpec(S) 49
DetachSpec(S) 49
2.3.9 User Startup Specification Files . . . 50

2.4 Attributes . . . . . . . . . . . 51
2.4.1 Predefined System Attributes . . . . . 51
2.4.2 User-defined Attributes . . . . . . 52
AddAttribute(C, F) 52
declare attributes C: F₁, ..., Fₙ; 52
2.4.3 Accessing Attributes . . . . . . . . 52
S.fieldname 52
assigned 52
S.fieldname := e; 52
delete S.fieldname; 53
GetAttributes(C) 53
ListAttributes(C) 53
2.4.4 User-defined Verbose Flags . . . . . . 53
declare verbose F, m; 53
2.4.5 Examples . . . . . . . . . . . . 53
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

\[
f := \text{function}(x_1, \ldots, x_n:\text{parameters})
  \begin{align*}
  &\text{statements} \\
  &\text{end function;}
  \end{align*}
\]

\[
\text{function } f(x_1, \ldots, x_n:\text{parameters})
  \begin{align*}
  &\text{statements} \\
  &\text{end function;}
  \end{align*}
\]

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 \text{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 \text{if} \ldots \text{then} \ldots \text{else} \ldots \text{construction}.)

The function may return multiple values. Usually one uses the form \text{return expression, \ldots, 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, ..., m_n are the actual arguments.

```
f := function(x_1, ..., x_n, ...: 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, ..., y_m are the arguments to the function, and m ≥ 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 ≥ n, the arguments y_i are bound to the last parameter x_n as a list [y_n, ..., y_m*].

```
f := func< x_1, ..., x_n: 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, ..., x_n). Optional parameters are permitted as in the standard function constructor.

```
f := func< x_1, ..., x_n, ...: 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.

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

Example H2E2

This example illustrates the use of parameters.

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

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);
```
2.2.2 Procedures

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 \( (~y_i) \). Inside the procedure only referenced variables (and local variables) may be (re-)assigned to. The procedure \( p \) is invoked by typing \( p(x_1, \ldots, x_n) \), where the same succession of variables and referenced variables is used (see the example below). Procedures cannot return values.

If there are parameters given, they must consist of a comma-separated list of clauses each of the form \( \text{identifier} := \text{value} \). 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 \( \text{parameters} \), may be omitted.

As in the case of \textit{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).

Creates and assigns a new \textit{variadic} procedure to \( p \). The use of a variadic procedure is identical to that of a variadic function, described previously.
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.

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.

``` magma
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):

``` magma
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 \( a^{n-1} \equiv 1 \) mod \( n \), 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.

```plaintext
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

\[
\text{name::type} \\
\text{~name::type} \\
\text{~name}
\]

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:

- \[ \text{.} \] Any type
- \[ \text{[]} \] Sequence type
- \[ \text{()} \] Set type
- \[ \text{[[ ]]} \] Set or Sequence type
- \[ \text{[@@]} \] Iset type
- \[ \text{[* *]} \] Multiset type
- \[ < > \] Tuple type

or a composite type:

\[ \text{[type]} \] Sequences over *type*
\[ \text{тип} \] Sets over *type*
\[ \text{[[тип]]} \] Sets or sequences over *type*
\[ \text{@тип@тип} \] Indexed sets over *type*
\[ \text{*тип*тип} \] Multisets over *type*

where *type* is either a simple or extended type. The reference form *type* \(~\text{name}\) requires that the input argument must be initialized to an object of that type. The reference form \(~\text{name}\) 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:

```magma
intrinsic myGCD(x::RngIntElt, y::RngIntElt) -> RngIntElt
{ Return the GCD of x and y }
  return ...;
```
A procedural intrinsic for Append taking a reference to a sequence $Q$ and any object then modifying $Q$:

\[
\text{intrinsic Append}(\sim Q::\text{SeqEnum}, . \ x) \\
\{ \text{Append x to Q } \} \\
\ldots; \\
\text{end intrinsic;}
\]

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

\[
\text{intrinsic IsConjugate}(G::\text{GrpPerm}, R::[ \{ \} ], S::[ \{ \} ]) \rightarrow \text{BoolElt} \\
\{ \text{True iff partitions R and S of the support of G are conjugate in G } \} \\
\text{return } \ldots; \\
\text{end intrinsic;}
\]

### 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 an 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:

\[
\text{intrinsic overloaded}(x::\text{RngUPolElt}, y::\text{RngUPolElt}) \rightarrow \text{RngIntElt} \\
\{ \text{Overload 1 } \} \\
\text{return } 1; \\
\text{end intrinsic;}
\]

\[
\text{intrinsic overloaded}(x::\text{RngUPolElt}[\text{RngInt}], y::\text{RngUPolElt}) \rightarrow \text{RngIntElt}
\]
THE MAGMA LANGUAGE

\begin{verbatim}
{ 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:

\texttt{> R1<x> := PolynomialRing(Integers());}
\texttt{> R2<y> := PolynomialRing(Rationals());}
\texttt{> f1 := x + 1;}
\texttt{> f2 := y + 1;}
\texttt{> overloaded(f2, f2); 1}
\texttt{> overloaded(f1, f2); 2}
\texttt{> overloaded(f2, f1); 3}
\texttt{> overloaded(f1, f1); 4}
\end{verbatim}

\section*{2.3.4 Attaching and Detaching Package Files}

The procedures \texttt{Attach} and \texttt{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 \texttt{freeze}; at the top then the package file becomes \texttt{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.

\[\text{Attach}(F)\]
Procedure to attach the package file \(F\).

\[\text{Detach}(F)\]
Procedure to detach the package file \(F\).

\[\text{freeze;}\]
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 \(\text{file.m}\):

- \(\text{file.sig}\) — sig file containing signature information;
- \(\text{file.lck}\) — 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) \(\text{.lck}\) 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 \(\text{.lck}\) file to disappear. In this case the user should search for \(\text{.lck}\) files that should be removed.

### 2.3.6 Importing Constants

\[\text{import } "\text{filename}" : \ident\_list;\]
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 \text{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:

```m
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

```m
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.

```m
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).

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

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

```m
requirege v, L;
```

The argument variable `v` must be the name of one of the argument variables (including parameters) and must be of integer type. `L` must yield an integer value. If `v` is not greater than or equal to `L`, then an appropriate error message is printed and execution aborts with the error pointer pointing to the caller.
Example H2E10

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

\[
\begin{align*}
\text{intrinsic } & \text{ Binomial}(n::\text{RngIntElt}, k::\text{RngIntElt}) \rightarrow \text{RngIntElt} \\
\{ & \text{ Return } n \text{ choose } k \\
& \text{ require } n, 0; \\
& \text{ require } k, 0, n; \\
& \text{ return } \text{Factorial}(n) \div \text{Factorial}(n - k) \div \text{Factorial}(k); \\
\end{align*}
\]

end intrinsic;

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

\[
\begin{align*}
\text{intrinsic } & \text{ pElement}(G::\text{Grp}, p::\text{RngIntElt}) \rightarrow \text{GrpElt} \\
\{ & \text{ Return } p\text{-element of group } G \\
& \text{ require } \text{IsPrime}(p): "\text{Argument 2 is not prime}"; \\
& \text{ x := random\{x: x in G | Order(x) mod p eq 0\}; \\
& \text{ return } x^{\text{Order(x) div p}}; \\
\end{align*}
\]

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:

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

is completely equivalent to the code:

```magma
> 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:

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

is completely equivalent to the code:

```magma
> 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, ..., 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.

```
S'fieldname
```

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 $S$.

```
assigned S'fieldname
```

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 $S$.

```
S'fieldname := 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 $S$. 
delete S'fieldname;

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.

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 $C$.

### 2.4.4 User-defined Verbose Flags

Since version V2.7, 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.

### 2.4.5 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.

```plaintext
> // 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
> // 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|>);
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).

```magma
> 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:

```magma
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.
3 INPUT AND OUTPUT

3.1 Introduction .................................. 59

3.2 Character Strings ............................... 59
  3.2.1 Representation of Strings .................... 59
  3.2.2 Creation of Strings .......................... 60
       "abc" .................................. 60
       BinaryString(s) ......................... 60
       BString(s) .............................. 60
       cat .................................. 60
       * .................................... 60
       cat:= .................................. 60
       *:= .................................... 60
       &cat s .................................. 60
       &* s .................................... 60
       ^ ...................................... 60
       s[i] .................................. 60
       s[i] .................................. 61
       ElementToSequence(s) ...................... 61
       Eltseq(s) ................................ 61
       ElementToSequence(s) ...................... 61
       Eltseq(s) ................................ 61
       Substring(s, n, k) .......................... 61

3.2.3 Integer-Valued Functions ..................... 61
       # ..................................... 61
       Index(s, t) ................................ 61
       Position(s, t) ............................ 61

3.2.4 Character Conversion .......................... 61
       StringToCode(s) ........................... 61
       CodeToString(n) ........................... 61
       StringToInteger(s) ....................... 62
       StringToInteger(s, b) ..................... 62
       StringToIntegerSequence(s) ............... 62
       IntegerToString(n) ....................... 62
       IntegerToString(n, b) ..................... 62

3.2.5 Boolean Functions ............................. 62
       eq ..................................... 62
       ne ..................................... 62
       in ..................................... 62
       notin .................................... 63
       lt ...................................... 63
       le ...................................... 63
       gt ...................................... 63
       ge ...................................... 63

3.2.6 Parsing Strings ............................... 65
       Split(S, D) ................................ 65
       Split(S) .................................. 65
       Regexp(R, S) ............................... 65

3.3 Printing .................................. 66
  3.3.1 The print-Statement ......................... 66
       print e; .................................. 66
       print e, ..., e; ............................ 66
       print e: -; .................................. 66
  3.3.2 The printf and fprintf Statements ............. 67
       printf format, e, ..., e; ..................... 67
       fprintf file, format, e, ..., e; .............. 68
  3.3.3 Verbose Printing (vprint, vprintf) ............ 69
       vprint flag: e, ..., e; ....................... 69
       vprint flag, n: e, ..., e; ................. 69
       vprintf flag: format, e, ..., e; ............... 69
       vprintf flag, n: format, e, ..., e; .......... 69

3.3.4 Automatic Printing ........................... 70
       ShowPrevious() ............................ 70
       ShowPrevious(i) ............................ 70
       ClearPrevious() ........................... 70
       SetPreviousSize(n) ......................... 71
       GetPreviousSize() .......................... 71

3.3.5 Indentation .................................. 72
       IndentPush() ............................... 72
       IndentPop() ................................. 72

3.3.6 Printing to a File ............................ 72
       PrintFile(F, x) ............................. 72
       Write(F, x) ................................ 72
       WriteBinary(F, s) ......................... 73
       PrintFile(F, x, L) ........................... 73
       Write(F, x, L) .............................. 73
       PrintFileMagma(F, x) ....................... 73

3.3.7 Printing to a String ......................... 73
       Sprint(x) .................................. 73
       Sprint(x, L) ................................ 73
       Sprintf(F, ...) .............................. 73

3.3.8 Redirecting Output ............................ 74
       SetOutputFile(F) ............................ 74
       UnsetOutputFile() ........................... 74
       HasOutputFile() ............................. 74

3.4 External Files ................................ 74
  3.4.1 Opening Files ................................ 74
       Open(S, T) .................................. 74

3.4.2 Operations on File Objects ..................... 75
       Flush(F) .................................... 75
       Tell(F) ..................................... 75
      .Seek(F, o, p) ................................ 75
       Rewind(F) ................................... 75
       Put(F, S) .................................. 75
       Puts(F, S) ................................... 75
       Getc(F) ..................................... 75
       Gets(F) ..................................... 75
       IsEof(S) .................................... 75
       Ungetc(F, c) .................................. 75
3.4.3 Reading a Complete File . . . . . . 76
Read(F)  76
ReadBinary(F)  76

3.5 Pipes . . . . . . . . . . . . 77
3.5.1 Pipe Creation . . . . . . . . . . . 77
POpen(C, T)  77
Pipe(C, S)  77

3.5.2 Operations on Pipes . . . . . . . . 78
Read(P : -)  78
ReadBytes(P : -)  78
Write(P, s)  79
WriteBytes(P, Q)  79

3.6 Sockets . . . . . . . . . . . . 79
3.6.1 Socket Creation . . . . . . . . . . . 79
Socket(H, P : -)  79
Socket( : -)  80
WaitForConnection(S)  80

3.6.2 Socket Properties . . . . . . . . . 80
SocketInformation(S)  80

3.6.3 Socket Predicates . . . . . . . . . 80
IsServerSocket(S)  80

3.6.4 Socket I/O . . . . . . . . . . . . . 81
Read(S : -)  81
ReadBytes(S : -)  81
Write(S, s)  81
WriteBytes(S, Q)  81

3.7 Interactive Input . . . . . . . . . 82
read id;  82
read id, prompt;  82
readi id;  83
readi id, prompt;  83

3.8 Loading a Program File . . . . . . 83
load "filename";  83
iload "filename";  83

3.9 Saving and Restoring Workspaces 83
save "filename";  83
restore "filename";  83

3.10 Logging a Session . . . . . . . . 84
SetLogFile(F)  84
UnsetLogFile()  84
SetEchoInput(b)  84

3.11 Memory Usage . . . . . . . . . . 84
GetMemoryUsage()  84
GetMaximumMemoryUsage()  84
ResetMaximumMemoryUsage()  84

3.12 System Calls . . . . . . . . . . . 84
Alarm(s)  84
ChangeDirectory(s)  84
GetCurrentDirectory()  84
Getpid()  85
Getuid()  85
System(C)  85
%! shell-command  85

3.13 Creating Names . . . . . . . . . . 85
Tempname(P)  85
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, \n the second line, and then \r \t \r in tended 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 ≥ 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 \(s\).

\(\text{ElementToSequence}(s)\)
\(\text{Eltseq}(s)\)

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

\(\text{ElementToSequence}(s)\)
\(\text{Eltseq}(s)\)

Returns the sequence of numeric values representing the characters of \(s\).

\(\text{Substring}(s, n, k)\)

Return the substring of \(s\) of length \(k\) starting at position \(n\).

3.2.3 Integer-Valued Functions

\(#s\)

The length of the string \(s\).

\(\text{Index}(s, t)\)
\(\text{Position}(s, t)\)

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.

\(\text{StringToCode}(s)\)

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.

\(\text{CodeToString}(n)\)

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**

```plaintext
> "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?

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] ];
```
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.

\[ \text{\#}[i : i \in [1..\#I] \mid I[i] \text{ eq } \text{IntegerToString}(j)] : j \in [0..9] \];

We would like to test if the ASCII-encoding of the string ‘Magma’ appears. This could be done as follows, using \text{StringToCode} and \text{in}, or alternatively, \text{Position}. To reduce the typing, we first abbreviate \text{IntegerToString} to \text{is} and \text{StringToCode} to \text{sc}.

\[ \text{sc} := \text{StringToCode}; \]
\[ \text{its} := \text{IntegerToString}; \]
\[ M := \text{its}(\text{sc}("M")) \ast \text{its}(\text{sc}("a")) \ast \text{its}(\text{sc}("g")) \ast \text{its}(\text{sc}("m")) \ast \text{its}(\text{sc}("a")); \]
\[ M; \]
\[ 779710310997 \]
\[ M \text{ in } I; \]
\[ \text{false} \]
\[ \text{Position}(I, M); \]
\[ 0 \]

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.

\[ \text{Iseq} := [ I[i] : i \in [1..\#I] ]; \]
\[ \text{Mseq} := [ M[i] : i \in [1..\#M] ]; \]
\[ \text{IsSubsequence}(Mseq, Iseq); \]
\[ \text{false} \]
\[ \text{IsSubsequence}(Mseq, Iseq: \text{Kind} := \text{"Sequential"}); \]
\[ \text{true} \]

Finally, we find that the string ‘magma’ lies in between ‘Pi’ and ‘pi’:

\[ \text{"Pi" le } \text{"magma"}; \]
\[ \text{true} \]
\[ \text{"magma" lt } \text{"pi"}; \]
\[ \text{true} \]
### 3.2.6 Parsing Strings

**Split(S, D)**

- **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**.

```plaintext
> 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("abxcdyezab", "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 they should not have an initial \ before them as the UNIX commands **grep** and **ed** require so).
Example H3E3

We demonstrate some elementary uses of `Regexp`.

```magma
> 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: `%0` and `%0`, 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 `%0` 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 `%0` 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 `%0` (and `%0` 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 `printf` 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`.

```plaintext
> 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 ]
```
Example H3E5

Some further uses of the printf statement are illustrated below.

```magma
> 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 "%0\n", x, "Magma";
Polynomial([0, 1])
```

Example H3E6

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

```magma
> 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 \(2^{109} \text{ mod } p\)
70602400912917605986812821219 \(2^{102} \text{ mod } p\)
74214633706907132624082231038 \(2^{110} \text{ mod } p\)
129638414606681695789005139447 \(2^{106} \text{ mod } p\)
141204801825835211973625642438 \(2^{103} \text{ mod } p\)
259276829213363391578010278894 \(2^{107} \text{ mod } p\)
267650600228229401496703205319 \(2^{100} \text{ mod } p\)
282409603651670423947251284876 \(2^{104} \text{ mod } p\)
51853658426726783156020557788 \(2^{108} \text{ mod } p\)
535301200456458802993406410638 \(2^{101} \text{ mod } p\)
564819207303340847894502569752 \(2^{105} \text{ mod } p\)

### 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 $f(\text{arguments})$; 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;
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 \texttt{SetIndent} and \texttt{GetIndent} are used to control and examine the number of spaces used for each indentation level (default 4).

\begin{verbatim}
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.
\end{verbatim}

\begin{verbatim}
IndentPop()

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

3.3.6 Printing to a File

\begin{verbatim}
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 \texttt{Overwrite} is set to true, in which case the file is overwritten.
\end{verbatim}
Ch. 3

INPUT AND OUTPUT

73

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 \texttt{Sprintf}.  
\begin{verbatim}
> Q := [Sprintf("{%4o<->%-4o}", x, x): x in [1,10,100,1000]];  
> Q;  
[ { 1<->1 }, { 10<->10 }, { 100<->100 }, {1000<->1000} ]
\end{verbatim}

3.3.8 Redirecting Output

\begin{verbatim}
SetOutputFile(F)
\end{verbatim}

\texttt{Overwrite} \hspace{1cm} \texttt{BOOLELT} \hspace{1cm} \texttt{Default : false}

Redirect all \texttt{Magma} output to the file specified by the string \texttt{F}. By using \texttt{SetOutputFile(F: Overwrite := true)} the file \texttt{F} is emptied before output is written onto it.

\begin{verbatim}
UnsetOutputFile()
\end{verbatim}

Close the output file, so that output will be directed to standard output again.

\begin{verbatim}
HasOutputFile()
\end{verbatim}

If \texttt{Magma} currently has an output or log file \texttt{F}, return \texttt{true} and \texttt{F}; otherwise return \texttt{false}.

3.4 External Files

\texttt{Magma} provides a special \textit{file} type for the reading and writing of external files. Most of the standard \texttt{C} library functions can be applied to such files to manipulate them.

3.4.1 Opening Files

\begin{verbatim}
Open(S, T)
\end{verbatim}

Given a filename (string) \texttt{S}, together with a type indicator \texttt{T}, open the file named by \texttt{S} and return a \texttt{Magma} file object associated with it. Tilde expansion is performed on \texttt{S}. The standard \texttt{C} library function \texttt{fopen()} is used, so the possible characters allowed in \texttt{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 \texttt{T} to open the file for reading, and give the value "w" for \texttt{T} to open the file for writing, etc. (Note that in the PC version of \texttt{Magma}, the character "b" should also be included in \texttt{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 \texttt{delete} statement or by reassigning the variable associated with the file); there is no \texttt{Fclose} function. This ensures that the file is not closed while there are still multiple references to it. (The function is called \texttt{Open} instead of \texttt{Fopen} to follow Perl-style conventions. The following functions also follow such conventions where possible.)
3.4.2 Operations on File Objects

**Flush(F)**
Given a file $F$, flush the buffer of $F$.

**Tell(F)**
Given a file $F$, return the offset in bytes of the file pointer within $F$.

**Seek(F, o, p)**
Perform fseek($F$, o, p); i.e. move the file pointer of $F$ to offset $o$ (relative to $p$: 0 means beginning, 1 means current, 2 means end).

**Rewind(F)**
Perform rewind($F$); i.e. move the file pointer of $F$ to the beginning.

**Put(F, S)**
Put (write) the characters of the string $S$ to the file $F$.

**Puts(F, S)**
Put (write) the characters of the string $S$, followed by a newline character, to the file $F$.

**Getc(F)**
Given a file $F$, get and return one more character from file $F$ as a string. If $F$ is at end of file, a special EOF marker string is returned; the function IsEof should be applied to the character to test for end of file. (Thus the only way to loop over a file character by character is to get each character and test whether it is the EOF marker before processing it.)

**Gets(F)**
Given a file $F$, get and return one more line from file $F$ as a string. The newline character is removed before the string is returned. If $F$ is at end of file, a special EOF marker string is returned; the function IsEof should be applied to the string to test for end of file.

**IsEof(S)**
Given a string $S$, return whether $S$ is the special EOF marker.

**Ungetc(F, c)**
Given a character (length one string) $C$, together with a file $F$, perform ungetc($C$, $F$); i.e. push the character $C$ back into the input buffer of $F$. 
Example H3E9

We write a function to count the number of lines in a file. Note the method of looping over the characters of the file: we must get the line and then test whether it is the special EOF marker.

```magma
> function LineCount(F)
>     FP := Open(F, "r");
>     c := 0;
>     while true do
>         s := Gets(FP);
>         if IsEof(s) then
>             break;
>         end if;
>         c +:= 1;
>     end while;
>     return c;
> end function;
> LineCount("/etc/passwd");
59
```

3.4.3 Reading a Complete File

**Read(F)**

Function that returns the contents of the text-file with name indicated by the string $F$. Here $F$ may be an expression returning a string.

**ReadBinary(F)**

Function that returns the contents of the text-file with name indicated by the string $F$ as a binary string.

Example H3E10

In this example we show how **Read** can be used to import the complete output from a separate C program into a Magma session. We assume that a file **mystery.c** (of which the contents are shown below) is present in the current directory. We first compile it, from within Magma, and then use it to produce output for the Magma version of our **mystery** function.

```c
#include <stdio.h>
main(argc, argv)
int argc;
char **argv;
{
    int n, i;
    n = atoi(argv[1]);
    for (i = 1; i <= n; i++)
        printf("%d\n", i * i);
```
return 0;
}
> System("cc mystery.c -o mystery");
> mysteryMagma := function(n)
>   System("./mystery " cat IntegerToString(n) cat " >outfile");
>   output := Read("outfile");
>   return StringToIntegerSequence(output);
> end function;
> mysteryMagma(5);
[ 1, 4, 9, 16, 25 ]

3.5 Pipes

Pipes are used to communicate with newly-created processes. Currently pipes are only available on UNIX systems.

The Magma I/O module is currently undergoing revision, and the current pipe facilities are a mix of the old and new methods. A more uniform model will be available in future releases.

3.5.1 Pipe Creation

POpen(C, T)

Given a shell command line C, together with a type indicator T, open a pipe between the Magma process and the command to be executed. The standard C library function `popen()` is used, so the possible characters allowed in T are the same as those allowed for that function in the current operating system, and have the same interpretation. Thus one should give the value "r" for T so that Magma can read the output from the command, and give the value "w" for T so that Magma can write into the input of the command. See the Pipe intrinsic for a method for sending input to, and receiving output from, a single command.

Important: this function returns a File object, and the I/O functions for files described previously must be used rather than those described in the following.

Pipe(C, S)

Given a shell command C and an input string S, create a pipe to the command C, send S into the standard input of C, and return the output of C as a string. Note that for many commands, S should finish with a new line character if it consists of only one line.
Example H3E11

We write a function which returns the current time as 3 values: hour, minutes, seconds. The function opens a pipe to the UNIX command “date” and applies regular expression matching to the output to extract the relevant fields.

```magma
function GetTime()
  D := POpen("date", "r");
  date := Gets(D);
  _, _, f := Regexp("([0-9][0-9]):([0-9][0-9]):([0-9][0-9])", date);
  h, m, s := Explode(f);
  return h, m, s;
end function;

h, m, s := GetTime();
h, m, s;
14 30 01
h, m, s := GetTime();
h, m, s;
14 30 04
```

3.5.2 Operations on Pipes

When a read request is made on a pipe, the available data is returned. If no data is currently available, then the process waits until some does becomes available, and returns that. (It will also return if the pipe has been closed and hence no more data can be transmitted.) It does not continue trying to read more data, as it cannot tell whether or not there is some “on the way”.

The upshot of all this is that care must be exercised as reads may return less data than is expected.

```magma
Read(P : parameters) RngIntElt Default : 0
Waits for data to become available for reading from P and then returns it as a string. If the parameter Max is set to a positive value then at most that many characters will be read. Note that less than Max characters may be returned, depending on the amount of currently available data.

If the pipe has been closed then the special EOF marker string is returned.

ReadBytes(P : parameters) RngIntElt Default : 0
Waits for data to become available for reading from P and then returns it as a sequence of bytes (integers in the range 0..255). If the parameter Max is set to a positive value then at most that many bytes will be read. Note that less than Max bytes may be returned, depending on the amount of currently available data.

If the pipe has been closed then the empty sequence is returned.
```
Write(P, s)

Writes the characters of the string s to the pipe P.

WriteBytes(P, Q)

Writes the bytes in the byte sequence Q to the pipe P. Each byte must be an integer in the range 0..255.

3.6 Sockets

Sockets may be used to establish communication channels between machines on the same network. Once established, they can be read from or written to in much the same ways as more familiar I/O constructs like files. One major difference is that the data is not instantly available, so the I/O operations take much longer than with files. Currently sockets are only available on UNIX systems.

Strictly speaking, a socket is a communication endpoint whose defining information consists of a network address and a port number. (Even more strictly speaking, the communication protocol is also part of the socket. MAGMA only uses TCP sockets, however, so we ignore this point from now on.)

The network address selects on which of the available network interfaces communication will take place; it is a string identifying the machine on that network, in either domain name or dotted-decimal format. For example, both "localhost" and "127.0.0.1" identify the machine on the loopback interface (which is only accessible from the machine itself), whereas "foo.bar.com" or "10.0.0.3" might identify the machine in a local network, accessible from other machines on that network.

The port number is just an integer that identifies the socket on a particular network interface. It must be less than 65536. A value of 0 will indicate that the port number should be chosen by the operating system.

There are two types of sockets, which we will call client sockets and server sockets. The purpose of a client socket is to initiate a connection to a server socket, and the purpose of a server socket is to wait for clients to initiate connections to it. (Thus the server socket needs to be created before the client can connect to it.) Once a server socket accepts a connection from a client socket, a communication channel is established and the distinction between the two becomes irrelevant, as they are merely each side of a communication channel.

In the following descriptions, the network address will often be referred to as the host. So a socket is identified by a (host, port) pair, and an established communication channel consists of two of these pairs: (local-host, local-port), (remote-host, remote-port).

3.6.1 Socket Creation

Socket(H, P : parameters)

| LocalHost | MonStgElT | Default : none |
| LocalPort | RNGIntElT | Default : 0 |

Attempts to create a (client) socket connected to port P of host H. Note: these are the remote values; usually it does not matter which local values are used for client
sockets, but for those rare occasions where it does they may be specified using the parameters `LocalHost` and `LocalPort`. If these parameters are not set then suitable values will be chosen by the operating system. Also note that port numbers below 1024 are usually reserved for system use, and may require special privileges to be used as the local port number.

```
Socket(: parameters)
```

| LocalHost  | MontageLT | 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 1024 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.

### 3.6.2 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.

### 3.6.3 Socket Predicates

```
IsServerSocket(S)
```

Returns whether `S` is a server socket or not.
3.6.4 Socket I/O

Due to the nature of the network, it takes significant time to transmit data from one machine to another. Thus when a read request is begun it may take some time to complete, usually because the data to be read has not yet arrived. Also, data written to a socket may be broken up into smaller pieces for transmission, each of which may take different amounts of time to arrive. Thus, unlike files, there is no easy way to tell if there is still more data to be read; the current lack of data is no indicator as to whether more might arrive.

When a read request is made on a socket, the available data is returned. If no data is currently available, then the process waits until some does 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 less than Max characters may be returned, depending on the amount of currently available data.

If the socket has been closed then the special EOF marker string is returned.

ReadBytes(S : parameters)  
Max RngIntElt Default : 0
Waits for data to become available for reading from S and then returns it as a sequence of bytes (integers in the range 0..255). If the parameter Max is set to a positive value then at most that many bytes will be read. Note that less than Max bytes may be returned, depending on the amount of currently available data.

If the socket has been closed then the empty sequence is returned.

Write(S, s)  
Writes the characters of the string s to the socket S.

WriteBytes(S, Q)  
Writes the bytes in the byte sequence Q to the socket S. Each byte must be an integer in the range 0..255.
Example H3E12

Here is a trivial use of sockets to send a message from one Magma process to another running on the same machine. The first Magma process sets up a server socket and waits for another Magma to contact it.

> // First Magma process
> server := Socket(: LocalHost := "localhost");
> SocketInformation(server);
<localhost, 32794>
> S1 := WaitForConnection(server);

The second Magma process establishes a client socket connection to the first, writes a greeting message to it, and closes the socket.

> // Second Magma process
> S2 := Socket("localhost", 32794);
> SocketInformation(S2);
<localhost, 32795> <localhost, 32794>
> Write(S2, "Hello, other world!");
> delete S2;

The first Magma process is now able to continue; it reads and displays all data sent to it until the socket is closed.

> // First Magma process
> SocketInformation(S1);
<localhost, 32794> <localhost, 32795>
> repeat
>   msg := Read(S1);
>   msg;
> until IsEof(msg);
Hello, other world!
EOF

3.7 Interactive Input

\begin{itemize}
\item \textbf{read identifier;}
\item \textbf{read identifier, prompt;}
\end{itemize}

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.
This statement will cause Magma to assign to the given identifier the literal integer appearing (at run-time) on the following line. This allows the user to specify integer input at run-time. If the optional prompt is given (a string), that is printed first.

### 3.8 Loading a Program File

**load "filename";**

Input the file with the name specified by the string. The file will be read in, and the text will be treated as Magma input. Tilde expansion of file names is allowed.

**iload "filename";**

(Interactive load.) Input the file with the name specified by the string. The file will be read in, and the text will be treated as Magma input. Tilde expansion of file names is allowed. In contrast to load, the user has the chance to interact as each line is read in:

As the line is read in, it is displayed and the system waits for user response. At this point, the user can skip the line (by moving “down”), edit the line (using the normal editing keys) or execute it (by pressing “enter”). If the line is edited, the new line is executed and the original line is presented again.

### 3.9 Saving and Restoring Workspaces

**save "filename";**

Copy all information present in the current Magma workspace onto a file specified by the string "filename". The workspace is left intact, so executing this command does not interfere with the current computation.

**restore "filename";**

Copy a previously stored Magma workspace from the file specified by the string "filename" into central memory. Information present in the current workspace prior to the execution of this command will be lost. The computation can now proceed from the point it was at when the corresponding save-command was executed.
3.10 Logging a Session

\[ \text{SetLogFile}(F) \quad \text{Overwrite : bool} \quad \text{Default} : \text{false} \]
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 \( \text{SetLogFile}(F : \text{Overwrite} := \text{true}) \) the file \( F \) is emptied before input and output are written onto it. See also \( \text{HasOutputFile} \).

\[ \text{UnsetLogFile()} \]
Stop logging Magma’s output.

\[ \text{SetEchoInput}(b) \]
Set to true or false according to whether or not input from external files should also be sent to standard output.

3.11 Memory Usage

\[ \text{GetMemoryUsage()} \]
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.

\[ \text{GetMaximumMemoryUsage()} \]
Return the maximum memory usage of Magma (in bytes as an integer) which has been attained since last reset (see \( \text{ResetMaximumMemoryUsage} \)). This is the maximum process data size, which does not include the executable code.

\[ \text{ResetMaximumMemoryUsage()} \]
Reset the value of the maximum memory usage of Magma to be the current memory usage of Magma (see \( \text{GetMaximumMemoryUsage} \)).

3.12 System Calls

\[ \text{Alarm}(s) \]
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.

\[ \text{ChangeDirectory}(s) \]
Change to the directory specified by the string \( s \). Tilde expansion is allowed.

\[ \text{GetCurrentDirectory}() \]
Returns the current directory as a string.
Getpid()  
Returns Magma’s process ID (value of the Unix C system call `getpid()`).

Getuid()  
Returns the user ID (value of the Unix C system call `getuid()`).

System(C)  
Execute the system command specified by the string C. This is done by calling the C function `system()`.

This also returns the system command’s return value as an integer. On most Unix systems, the lower 8 bits of this value give the process status while the next 8 bits give the value given by the command to the C function `exit()` (see the Unix manual entries for `system(3)` or `wait(2)`, for example). Thus one should normally divide the result by 256 to get the exit value of the program on success.

See also the Pipe intrinsic function.

`shell-command`  
Execute the given command in the Unix shell then return to Magma. Note that this type of shell escape (contrary to the one using a System call) takes place entirely outside Magma and does not show up in Magma’s history.

3.13 Creating Names  
Sometimes it is necessary to create names for files from within Magma that will not clash with the names of existing files.

Tempname(P)  
Given a prefix string P, return a unique temporary name derived from P (by use of the C library function `mktemp()`).
4 ENVIRONMENT AND OPTIONS

4.1 Introduction . . . . . . . . . 89
4.2 Command Line Options . . . . 89
magma -b 89
magma -c filename 89
magma -d 90
magma -n 90
magma -q name 90
magma -r workspace 90
magma -s filename 90
magma -S integer 90
GetMemoryLimit() 94
SetOutputFile(F) 94
UnsetOutputFile() 94
SetPath(s) 95
GetPath() 95
SetPrintLevel(1) 95
GetPrintLevel() 95
SetPrompt(s) 95
GetPrompt() 95
SetQuitOnError(b) 95
SetRows(n) 95
GetRows() 95
SetTraceback(n) 95
GetTraceback() 95
SetSeed(s, c) 96
GetSeed() 96
GetVersion() 96
SetViMode(b) 96
GetViMode() 96
4.3 Environment Variables . . . . . 91
MAGMA_STARTUP_FILE 91
MAGMA_PATH 91
MAGMA_MEMORY_LIMIT 91
MAGMA_LIBRARY_ROOT 91
MAGMA_LIBRARIES 91
MAGMA_SYSTEM_SPEC 91
MAGMA_USER_SPEC 91
MAGMA_HELP_DIR 91
4.4 Set and Get . . . . . . . . . 92
SetAssertions(b) 92
GetAssertions() 92
SetAutoColumns(b) 92
GetAutoColumns() 92
SetAutoCompact(b) 92
GetAutoCompact() 92
SetBeep(b) 92
GetBeep() 92
SetColumns(n) 92
GetColumns() 92
GetCurrentDirectory() 92
SetEchoInput(b) 93
GetEchoInput() 93
GetEnvironmentValue(s) 93
GetEnv(s) 93
SetHistorySize(n) 93
GetHistorySize() 93
SetIgnorePrompt(b) 93
GetIgnorePrompt() 93
SetIgnoreSpaces(b) 93
GetIgnoreSpaces() 93
SetIndent(n) 93
GetIndent() 93
SetLibraries(s) 94
GetLibraries() 94
SetLibraryRoot(s) 94
GetLibraryRoot() 94
SetLineEditor(b) 94
GetLineEditor() 94
SetLogFile(F) 94
UnsetLogFile() 94
SetMemoryLimit(n) 94
4.5 Verbose Levels . . . . . . . . . 96
SetVerbose(s, i) 96
SetVerbose(s, b) 96
GetVerbose(s) 96
IsVerbose(s) 96
IsVerbose(s, l) 96
ListVerbose() 97
ClearVerbose() 97
4.6 Other Information Procedures . . 97
ShowMemoryUsage() 97
ShowIdentifiers() 97
ShowValues() 97
Traceback() 97
ListSignatures(C) 97
ListSignatures(F, C) 97
ListCategories() 98
ListTypes() 98
4.7 History . . . . . . . . . . . 98
%p 98
%p
%p
n1 n2 98
%P 98
%P
%p
n1 n2 98
%S 98
%S
%sn 98
%sn
%n1 n2 98
%sn
%n
%n1 n2 99
4.8 The Magma Line Editor . . . 100
SetViMode 100
SetEmacsMode 100

4.8.1 Key Bindings (Emacs and VI mode) 100
<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Return&gt;</td>
<td>$</td>
</tr>
<tr>
<td>&lt;Backspace&gt;</td>
<td>%</td>
</tr>
<tr>
<td>&lt;Delete&gt;</td>
<td>;</td>
</tr>
<tr>
<td>&lt;Tab&gt;</td>
<td>,</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-A</td>
<td>Mb</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-B</td>
<td>MB</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-C</td>
<td>MF</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-D</td>
<td>$char</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-E</td>
<td>fchar</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-F</td>
<td>l</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-G</td>
<td>L</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-H</td>
<td>A</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-I</td>
<td>a</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-J</td>
<td>C</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-K</td>
<td>range</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-L</td>
<td>D</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-M</td>
<td>drange</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-N</td>
<td>I</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-O</td>
<td>i</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-P</td>
<td>j</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-Q</td>
<td>k</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-R</td>
<td>P</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-S</td>
<td>char</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-T</td>
<td>Range</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-U</td>
<td>S</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-V</td>
<td>s</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-W</td>
<td>U</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-X</td>
<td>u</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-Y</td>
<td>X</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-Z</td>
<td>Y</td>
</tr>
</tbody>
</table>

4.8.2 Key Bindings in Emacs mode only . 102
<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Ctrl&gt;-A</td>
<td>Mb</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-B</td>
<td>MB</td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-C</td>
<td>MF</td>
</tr>
</tbody>
</table>

4.8.3 Key Bindings in VI mode only . 103
<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td></td>
</tr>
<tr>
<td>&lt;Ctrl&gt;-space</td>
<td></td>
</tr>
</tbody>
</table>

4.9 The Magma Help System . . . 106
SetHelpExternalBrowser(S, T) 107
SetHelpExternalBrowser(S) 107
SetHelpUseExternalBrowser(b) 107
SetHelpExternalSystem(s) 107
SetHelpUseExternalSystem(b) 107
GetHelpExternalBrowser() 107
GetHelpExternalSystem() 107
GetHelpUseExternal() 107

4.9.1 Internal Help Browser . . . . . . 107
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.
4.4 Set and Get

The Set- procedures allow the user to attach values to certain environment variables. The Get- functions enable one to obtain the current values of these variables.

**SetAssertions(b)**

Controls the checking of assertions (see the `assert` statement in the chapter on the language). Default is `SetAssertions(true)`.

**GetAssertions()**

**SetAutoColumns(b)**

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)`.

**GetAutoColumns()**

**SetAutoCompact(b)**

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)`.

**GetAutoCompact()**

**SetBeep(b)**

Controls ‘beeps’. Default is `SetBeep(true)`.

**GetBeep()**

**SetColumns(n)**

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)`).

**GetColumns()**

**GetCurrentDirectory()**

Returns the current directory as a string. (Use `ChangeDirectory(s)` to change the working directory.)
**SetEchoInput(b)**
**GetEchoInput()**

Set to **true** or **false** according to whether or not input from external files should also be sent to standard output.

**GetEnvironmentValue(s)**
**GetEnv(s)**

Returns the value of the external environment variable *s* as a string.

**SetHistorySize(n)**
**GetHistorySize()**

Controls the number of lines saved in the history. If the number is set to 0, no history is preserved.

**SetIgnorePrompt(b)**
**GetIgnorePrompt()**

Controls the option to ignore the prompt to allow the pasting of input lines back in. If enabled, any leading ‘>’ characters (possibly separated by white space) are ignored by the history system when the input file is a terminal, unless the line consists of the ‘>’ character alone (without a following space), which could not come from a prompt since in a prompt a space or another character follows a ‘>’. Default is **SetIgnorePrompt(false)**.

**SetIgnoreSpaces(b)**
**GetIgnoreSpaces()**

Controls the option to ignore spaces when searching in the line editor. If the user moves up or down in the line editor using \(<\text{Ctrl}\>-P \) or \(<\text{Ctrl}\>-N\) (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 | BOOL
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).

SetOutputFile(F)
Overwrite | BOOL
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).

SetTraceback(n)
GetTraceback()

Controls whether MAGMA should produce a traceback of user function calls before each error message. The default value is SetTraceback(true).
SetSeed(s, c)  
GetSeed()  

Controls the initialization seed and step number for pseudo-random number generation. For details, see the section on random object generation in the chapter on statements and expressions.

GetVersion()  

Return integers $x$, $y$ and $z$ such the current version of Magma is $Vx.y-z$.

SetViMode(b)  
GetViMode()  

Controls the type of line editor used: Emacs (false) or VI style. Default is SetViMode(false).

4.5 Verbose Levels

By turning verbose printing on for certain modules within Magma, some information on computations that are performed can be obtained. For each option, the verbosity may have different levels. The default is level 0 for each option.

There are also 5 slots available for user-defined verbose flags. The flags can be set in user programs by SetVerbose("Usern", true) where $n$ should be one of 1, 2, 3, 4, 5, and the current setting is returned by GetVerbose("Usern").

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)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isa</td>
<td>BOOLELT</td>
<td>true</td>
</tr>
<tr>
<td>Search</td>
<td>MONSTGELT</td>
<td>“Both”</td>
</tr>
<tr>
<td>ShowSrc</td>
<td>BOOLELT</td>
<td>false</td>
</tr>
</tbody>
</table>

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)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isa</td>
<td>BOOLELT</td>
<td>true</td>
</tr>
<tr>
<td>Search</td>
<td>MONSTGELT</td>
<td>“Both”</td>
</tr>
<tr>
<td>ShowSrc</td>
<td>BOOLELT</td>
<td>false</td>
</tr>
</tbody>
</table>

Given an intrinsic F and category C, list all signatures of F which match the category C among their arguments or return values. The parameters are as for the previous procedure.
ListCategories()
ListTypes()

Procedure to list the (abbreviated) names for all available categories in MAGMA.

4.7 History
Magma provides a history system which allows the recall and editing of previous lines. The history system is invoked by typing commands which begin with the history character ‘%’. Currently, the following commands are available.

%p
List the contents of the history buffer. Each line is preceded by its history line number.

%pn
List the history line \(n\) in %p format.

%pn\_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.

%Pn
List the history line \(n\) in %P format.

%Pn\_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.

%sn
Print the history line \(n\) in %s format.

%sn\_1 \(n\_2\)
Print the history lines in the range \(n_1\) to \(n_2\) in %s format.
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).

%Sn
Print the history line n in %S format.

%n1 n2
Print the history lines in the range n1 to n2 in %S format.

% Reenter the last line into the input stream.

%n
Reenter the line specified by line number n into the input stream.

%n1 n2
Reenter the history lines in the range n1 to n2 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.

%en
Edit the line specified by line number n.

%en1 n2
Edit the history lines in the range n1 to n2.

%! 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.
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”).

\texttt{<Ctrl>-E}

Move to the end of the line (“end”).

\texttt{<Ctrl>-F}

Move forward a character (“forward”).

\texttt{<Ctrl>-H}

Same as Backspace.

\texttt{<Ctrl>-I}

Same as Tab.

\texttt{<Ctrl>-J}

Same as Return.

\texttt{<Ctrl>-K}

Delete all characters from the cursor to the end of the line (“kill”).

\texttt{<Ctrl>-L}

Redraw the line on a new line (helpful if the screen gets wrecked by programs like “write”, etc.).

\texttt{<Ctrl>-M}

Same as \texttt{<Return>}.  

\texttt{<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 \texttt{SetIgnoreSpaces}) as the string consisting of all the characters before the cursor. Also, if \texttt{<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 \texttt{<Ctrl>-P} (see below), press \texttt{<Return>}, then successively press \texttt{<Ctrl>-N} followed by \texttt{<Return>} for each line.

\texttt{<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 \texttt{SetIgnoreSpaces}) as the string consisting of all the characters before the cursor. For example, typing at a
new line \texttt{x:=} and then <\texttt{Ctrl}>-P will go back to the last line which assigned \texttt{x} (if a line begins with, say, \texttt{x :=}, it will also be taken).

\begin{itemize}
  \item \texttt{<Ctrl>-U}
    \begin{itemize}
      \item Clear the whole of the current line.
    \end{itemize}
  \item \texttt{<Ctrl>-V}
    \begin{itemize}
      \item Insert the following character literally.
    \end{itemize}
  \item \texttt{<Ctrl>-W}
    \begin{itemize}
      \item Delete the previous word.
    \end{itemize}
  \item \texttt{<Ctrl>-X}
    \begin{itemize}
      \item Same as \texttt{<Ctrl>-U}.
    \end{itemize}
  \item \texttt{<Ctrl>-Y}
    \begin{itemize}
      \item Insert the contents of the yank-buffer before the character under the cursor.
    \end{itemize}
  \item \texttt{<Ctrl>-Z}
    \begin{itemize}
      \item Stop MAGMA.
    \end{itemize}
  \item \texttt{<Ctrl>-_}
    \begin{itemize}
      \item Undo the last change.
    \end{itemize}
  \item \texttt{<Ctrl>-\}
    \begin{itemize}
      \item Immediately quit MAGMA.
    \end{itemize}
\end{itemize}

On most systems the arrow keys also have the obvious meaning.

\section{4.8.2 Key Bindings in Emacs mode only}

M\texttt{key} means press the Meta key and then \texttt{key}. (At the moment, the Meta key is only the \texttt{Esc} key.)

\begin{itemize}
  \item \texttt{Mb}
    \begin{itemize}
      \item Move back a word ("Back").
    \end{itemize}
  \item \texttt{MB}
    \begin{itemize}
      \item Move forward a word ("Forward").
    \end{itemize}
\end{itemize}
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”).

- **Fchar**
  Move back to the first occurrence of `char`.

- **fchar**
  Move forward to the first occurrence of `char`.

- **h**
  Move back a character (<Ctrl>-H = Backspace).
Move back a character (<Ctrl>-L = forward on some keyboards).

Move back to just after the first occurrence of char.

Move forward to just before the first occurrence of char.

Move forward a space-separated word (“Word”).

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:

Move to the end of the line and change to insert mode (“Append”).

Move forward a character (if not already at the end of the line) and change to insert mode (“append”).

Delete all the characters to the end of line and change to insert mode (“Change”).

Delete all the characters to the specified range and change to insert mode (“change”).

Delete all the characters to the end of line (“Delete”).

Delete all the characters to the specified range (“delete”).

Move to the first non-space character in the line and change to insert mode (“Insert”).
Change to insert mode ("insert").

\textbf{j}

Go forward a line in the history buffer (same as $<\text{Ctrl}>$-N).

\textbf{k}

Go back a line in the history buffer (same as $<\text{Ctrl}>$-P).

\textbf{P}

Insert the contents of the yank-buffer before the character under the cursor.

\textbf{p}

Insert the contents of the yank-buffer before the character after the cursor.

\textbf{R}

Enter over-type mode: typed characters replace the old characters under the cursor without insertion. Pressing Esc returns to the command mode.

\textbf{rchar}

Replace the character the cursor is over with \textit{char}.

\textbf{S}

Delete the whole line and change to insert mode ("Substitute").

\textbf{s}

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

\textbf{U}
\textbf{u}

Undo the last change.

\textbf{X}

Delete the character to the left of the cursor.

\textbf{x}

Delete the character under the cursor.

\textbf{Y}

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

\textbf{yrange}

Copy all characters from the cursor to the specified range into the yank-buffer ("yank").
4.9 The Magma Help System

Magma provides extensive online help facilities that can be accessed in different ways. The easiest way to access the documentation is by typing:

```
magmahelp
```

Which should start some browser (usually netscape) on the main page of the Magma documentation.

The easiest way to get some information about any Magma intrinsic is by typing:

```
> 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

5.1 Introduction . . . . . . . . 111
5.2 Terminology . . . . . . . . 111
5.3 Assignment . . . . . . . . 112
5.4 Uninitialized Identifiers . . . 112
5.5 Evaluation in Magma . . . . 113
  5.5.1 Call by Value Evaluation . . . . 113
  5.5.2 Magma’s Evaluation Process . . . 114
  5.5.3 Function Expressions . . . . . 115
  5.5.4 Function Values Assigned to Identifiers116
  5.5.5 Recursion and Mutual Recursion . 116
  5.5.6 Function Application . . . . . . 117
  5.5.7 The Initial Context . . . . . . . 118
5.6 Scope . . . . . . . . . . . 118
  5.6.1 Local Declarations . . . . . . . 119
  5.6.2 The ‘first use’ Rule . . . . . . . . . 119
  5.6.3 Identifier Classes . . . . . . . . . 120
  5.6.4 The Evaluation Process Revisited . 121
  5.6.5 The ‘single use’ Rule . . . . . . . . 121
5.7 Procedure Expressions . . . . 121
5.8 Reference Arguments . . . . 123
5.9 Dynamic Typing . . . . . . 124
5.10 Traps for Young Players . . . 125
  5.10.1 Trap 1 . . . . . . . . . . . . . 125
  5.10.2 Trap 2 . . . . . . . . . . . . . 125
5.11 Appendix A: Precedence . . . 127
5.12 Appendix B: Reserved Words . 128
Chapter 5
MAGMA SEMANTICS

5.1 Introduction
This chapter describes the semantics of MAGMA (how expressions are evaluated, how identifiers are treated, etc.) in a fairly informal way. Although some technical language is used (particularly in the opening few sections) the chapter should be easy and essential reading for the non-specialist. The chapter is descriptive in nature, describing how MAGMA works, with little attempt to justify why it works the way it does. As the chapter proceeds, it becomes more and more precise, so while early sections may gloss over or omit things for the sake of simplicity and learnability, full explanations are provided later.

It is assumed that the reader is familiar with basic notions like a function, an operator, an identifier, a type ...

And now for some buzzwords: MAGMA is an imperative, call by value, statically scoped, dynamically typed programming language, with an essentially functional subset. The remainder of the chapter explains what these terms mean, and why a user might want to know about such things.

5.2 Terminology
Some terminology will be useful. It is perhaps best to read this section only briefly, and to refer back to it when necessary.

The term expression will be used to refer to a textual entity. The term value will be used to refer to a run-time value denoted by an expression. To understand the difference between an expression and a value consider the expressions 1+2 and 3. The expressions are textually different but they denote the same value, namely the integer 3.

A function expression is any expression of the form function ... end function or of the form func< ... | ... >. The former type of function expression will be said to be in the statement form, the latter in the expression form. A function value is the run-time value denoted by a function expression. As with integers, two function expressions can be textually different while denoting the same (i.e., extensionally equal) function value. To clearly distinguish function values from function expressions, the notation FUNC( ... : ... ) will be used to describe function values.

The formal arguments of a function in the statement form are the identifiers that appear between the brackets just after the function keyword, while for a function in the expression form they are the identifiers that appear before the |. The arguments to a function are the expressions between the brackets when a function is applied.

The body of a function in the statement form is the statements after the formal arguments. The body of a function in the expression form is the expression after the | symbol.
An identifier is said to occur \textit{inside} a function expression when it occurs textually anywhere in the body of a function.

\section{5.3 Assignment}

An assignment is an association of an identifier to a \textit{value}. The statement,

\begin{verbatim}
> a := 6;
\end{verbatim}

establishes an association between the identifier \texttt{a} and the value \texttt{6} (6 is said to be the value of \texttt{a}, or to be assigned to \texttt{a}). A collection of such assignments is called a context.

When a value \texttt{V} is assigned to an identifier \texttt{I} one of two things happens:

(1) if \texttt{I} has not been previously assigned to, it is added to the current context and associated with \texttt{V}. \texttt{I} is said to be \textit{declared} when it is assigned to for the first time.

(2) if \texttt{I} has been previously assigned to, the value associated with \texttt{I} is changed to \texttt{V}. \texttt{I} is said to be \textit{re-assigned}.

The ability to assign and re-assign to identifiers is why \textsc{Magma} is called an \textit{imperative} language.

One very important point about assignment is illustrated by the following example. Say we type,

\begin{verbatim}
> a := 6;
> b := a+7;
\end{verbatim}

After executing these two lines the context is \([ (a,6), (b,13) ]\). Now say we type,

\begin{verbatim}
> a := 0;
\end{verbatim}

The context is now \([ (a,0), (b,13) ]\). Note that changing the value of \texttt{a} does \textit{not} change the value of \texttt{b} because \texttt{b}'s value is statically determined at the point where it is assigned. Changing \texttt{a} does \textit{not} produce the context \([ (a,0), (b,7) ]\).

\section{5.4 Uninitialized Identifiers}

Before executing a piece of code \textsc{Magma} attempts to check that it is semantically well formed (i.e., that it will execute without crashing). One of the checks \textsc{Magma} makes is to check that an identifier is declared (and thus initialized) before it is used in an expression. So, for example assuming \texttt{a} had not been previously declared, then before executing either of the following lines \textsc{Magma} will raise an error:

\begin{verbatim}
> a;
> b := a;
\end{verbatim}

\textsc{Magma} can determine that execution of either line will cause an error since \texttt{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,

```plaintext
> 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,

```plaintext
> 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,

```plaintext
> f := function( n, b )
>     if b then return n else return 1;
> end function;
```

and we apply \( f \) as follows

```plaintext
> 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,
\[ \text{false and (4/0 eq 6);} \]
Magma will reply with the answer false since Magma knows that \text{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,

```plaintext
> a := 6;
> b := 7;
> c := a+b;
```

giving the context `[ (a,6),(b,7),(c,13) ]`. Now say I type,

```plaintext
> 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,

```plaintext
> 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 \(func< n \mid b(6) \rangle\) 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 \(func< n \mid 6+7+13+n \rangle\) 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,

```plaintext
> 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 \texttt{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,

\[
> f := \text{function}(...) \ldots a := g(...) \ldots \text{end function};
> g := \text{function}(...) \ldots b := f(...) \ldots \text{end function};
\]

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,

\[
> \text{forward } f, g;
> f := \text{function}(...) \ldots a := g(...) \ldots \text{end function};
> g := \text{function}(...) \ldots b := f(...) \ldots \text{end function};
\]

(Strictly speaking it is only necessary to declare \( g \) forward as the value of \( f \) will be known by the time the function expression \texttt{function(...) \ldots b := f(...) \ldots \text{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,

\[
> f := \text{func< a, b | a+b >};
\]

producing the context \([ (f,\text{FUNC(a,b : a+b)}) ] \).

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,

\[
\text{“simplify } (\text{FUNC(a, b : A(a,b)))}}(3,13) \text{ to its canonical form”}
\]

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;
  end if;
> return 1;
```
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) the formal arguments to a function expression.

(b) all loop identifiers.

(c) the `$\$$` pseudo-identifier.

(d) all identifiers whose first use in a function expression is as a value (i.e., not on the left hand side of an `:=`, nor as an actual reference argument to a procedure).

Because value identifiers stand as placeholders for values to be substituted during the evaluation process, they are effectively constants, and hence they cannot be assigned to. Assigning to a value identifier would be akin to writing something like `7 := 8;`!

The class of variable identifiers includes all those identifiers which are declared as local, either implicitly by the first use rule, or explicitly through a local declaration. Identifiers in this class may be assigned to.

The class of reference identifiers will be discussed later.
5.6.4 The Evaluation Process Revisited

The reason it is important to know the class of an identifier is that the class of an identifier effects how it is treated during the evaluation process. Previously it was stated that the evaluation process was,

(1) replace each identifier in the expression by its value in the current context.

(2) simplify the resultant value to its canonical form.

Strictly speaking the first step of this process should read,

(1') replace each free identifier in the expression by its value in the current context, where
an identifier is said to be free if it is a value identifier which is not a formal argument,
a loop identifier, or the $\$$ identifier.

This definition of the replacement step ensures for example that while computing the value of a function expression $F$, MAGMA does not attempt to replace $F$’s formal arguments with values from the current context!

5.6.5 The ‘single use’ Rule

As a final point on identifier classes it should be noted that an identifier may belong to only one class within an expression. Specifically therefore an identifier can only be used in one way inside a function body. Consider the following function,

> a := 7;
> f := function(n) a := a; return a; end function;

It is not the case that $a$ is considered as a variable identifier on the left hand side of the :=, and as a value identifier on the right hand side of the :=. Rather $a$ is considered to be a value identifier as its first use is as a value on the right hand side of the := (remember that MAGMA inspects the right hand side of an assignment, and hence sees $a$ first as a value identifier, before it inspects the left hand side where it sees $a$ being used as a variable identifier).

5.7 Procedure Expressions

To date we have only discussed function expressions, these being a mechanism for computing new values from the values of identifiers in the current context. Together with assignment this provides us with a means of changing the current context – to compute a new value for an identifier in the current context, we call a function and then re-assign the identifier with the result of this function. That is we do

> X := f(Y);

where $Y$ is a list of arguments possibly including the current value of $X$.

At times however using re-assignment to change the value associated with an identifier can be both un-natural and inefficient. Take the problem of computing some reduced form of a matrix. We could write a function that looked something like this,
reduce :=
  function( m )
    local lm;
    ...
    lm := m;
    while not reduced do
      ...
      lm := some_reduction(m);
      ...
    end while;
    ...
  end function;

Note that the local \( lm \) is necessary since we cannot assign to the function’s formal argument \( m \) since it stands for a value (and values cannot be assigned to). Note also that the function is inefficient in its space usage since at any given point in the program there are at least two different copies of the matrix (if the function was recursive then there would be more than two copies!).

Finally the function is also un-natural. It is perhaps more natural to think of writing a program that takes a given matrix and changes that matrix into its reduced form (i.e., the original matrix is lost). To accommodate for this style of programming, Magma includes a mechanism, the procedure expression with its reference arguments, for changing an association of an identifier and a value in place.

Before examining procedure expressions further, it is useful to look at a simple example of a procedure expression. Say we type,

\[
\begin{align*}
& a := 5; b := 6; \\
& p := \text{procedure}( x, \sim y ) \ y := x; \text{ end procedure;}
\end{align*}
\]

This gives us a context that looks like \( [ (a,5), (b,6), (p, \text{PROC}(x,\sim y : y := x;)) ] \), using a notation analogous to the \text{FUNC} notation.

Say we now type the following statement,

\[
\begin{align*}
& p(a, \sim b);
\end{align*}
\]

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 \( [ (a,5), (b,5), (p, \text{PROC}(a,\sim b : b := a;)) ] \). \( a \) 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 \( \text{only} \) \( b \) 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,

\[
> \ a := 5; \ b := 6; \\
> \ p := \text{procedure(} \ x, \ \sim y \ \text{)} \ y := x; \ \text{end procedure;}
\]
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,

```magma
> 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,

```magma
> 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 $g$,

```magma
> g := function(a,b)
>     if false then
>         return a+true;
>     else
>         return a+b;
>     end if;
```
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,

```plaintext
> f := func< n | n + 2 >;
```

If we again type,

```plaintext
> g(6);
```

Magma will again reply with the answer 13!

To see why this is so consider how the current context changes. After typing in the initial definitions of $f$ and $g$ the current context is $[(f, \text{FUNC}(n : n+1)), (g, \text{FUNC}(m : m + \text{FUNC}(n : n+1)(m)))]$. After typing in the second definition of $f$ the current context is $[(f, \text{FUNC}(n : n+2)), (g, \text{FUNC}(m : m + \text{FUNC}(n : n+1)(m)))]$. Remember that changing the value of one identifier, in this case $f$, does not change the value of any other identifiers, in this case $g$! In order to change the value of $g$ to reflect the new value of $f$, $g$ would have to be re-assigned.
### 5.11 Appendix A: Precedence

The table below defines the relative precedence of operators in Magma, with decreasing strength (so operators higher in the table bind more strongly). The column on the right indicates whether the operator is left-, right-, or non-associative.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>''</code></td>
<td>left</td>
</tr>
<tr>
<td>(</td>
<td>left</td>
</tr>
<tr>
<td>[</td>
<td>left</td>
</tr>
<tr>
<td>assigned</td>
<td>right</td>
</tr>
<tr>
<td>-</td>
<td>non</td>
</tr>
<tr>
<td>#</td>
<td>non-associative</td>
</tr>
<tr>
<td>&amp;* &amp;+ &amp;∧ &amp;cat &amp;join &amp;meet &amp;∨</td>
<td>non-associative</td>
</tr>
<tr>
<td>$ $$$</td>
<td>non</td>
</tr>
<tr>
<td>.</td>
<td>left</td>
</tr>
<tr>
<td>@ @@</td>
<td>left</td>
</tr>
<tr>
<td>! !!</td>
<td>right</td>
</tr>
<tr>
<td>^</td>
<td>right</td>
</tr>
<tr>
<td>unary-</td>
<td>right</td>
</tr>
<tr>
<td>cat</td>
<td>left</td>
</tr>
<tr>
<td>* / div mod</td>
<td>left</td>
</tr>
<tr>
<td>+ -</td>
<td>left</td>
</tr>
<tr>
<td>meet</td>
<td>left</td>
</tr>
<tr>
<td>sdiff</td>
<td>left</td>
</tr>
<tr>
<td>diff</td>
<td>left</td>
</tr>
<tr>
<td>join</td>
<td>left</td>
</tr>
<tr>
<td>adj in notadj notin notsubset subset</td>
<td>non</td>
</tr>
<tr>
<td>cmpeq cmpne eq ge gt le lt ne</td>
<td>left</td>
</tr>
<tr>
<td>not</td>
<td>right</td>
</tr>
<tr>
<td>and</td>
<td>left</td>
</tr>
<tr>
<td>or xor</td>
<td>left</td>
</tr>
<tr>
<td>^^</td>
<td>non</td>
</tr>
<tr>
<td>? else select</td>
<td>right</td>
</tr>
<tr>
<td>-&gt;</td>
<td>left</td>
</tr>
<tr>
<td>=</td>
<td>left</td>
</tr>
<tr>
<td>:= is where</td>
<td>left</td>
</tr>
</tbody>
</table>
### Appendix B: Reserved Words

The list below contains all reserved words in the Magma language; these cannot be used as identifier names.

<table>
<thead>
<tr>
<th></th>
<th>end</th>
<th>le</th>
<th>requirerange</th>
</tr>
</thead>
<tbody>
<tr>
<td>adj</td>
<td>eq</td>
<td>load</td>
<td>restore</td>
</tr>
<tr>
<td>and</td>
<td>error</td>
<td>local</td>
<td>return</td>
</tr>
<tr>
<td>assert</td>
<td>eval</td>
<td>lt</td>
<td>save</td>
</tr>
<tr>
<td>assigned</td>
<td>exists</td>
<td>meet</td>
<td>sdiff</td>
</tr>
<tr>
<td>break</td>
<td>exit</td>
<td>mod</td>
<td>select</td>
</tr>
<tr>
<td>by</td>
<td>false</td>
<td>ne</td>
<td>subset</td>
</tr>
<tr>
<td>case</td>
<td>for</td>
<td>not</td>
<td>then</td>
</tr>
<tr>
<td>cat</td>
<td>forall</td>
<td>notadj</td>
<td>time</td>
</tr>
<tr>
<td>catch</td>
<td>forward</td>
<td>notin</td>
<td>to</td>
</tr>
<tr>
<td>clear</td>
<td>fprintf</td>
<td>notsubset</td>
<td>true</td>
</tr>
<tr>
<td>cmpeq</td>
<td>freeze</td>
<td>or</td>
<td>try</td>
</tr>
<tr>
<td>cmpne</td>
<td>function</td>
<td>print</td>
<td>until</td>
</tr>
<tr>
<td>continue</td>
<td>ge</td>
<td>printf</td>
<td>vprintf</td>
</tr>
<tr>
<td>declare</td>
<td>gt</td>
<td>procedure</td>
<td>vprintff</td>
</tr>
<tr>
<td>default</td>
<td>if</td>
<td>quit</td>
<td>vtime</td>
</tr>
<tr>
<td>delete</td>
<td>iload</td>
<td>random</td>
<td>when</td>
</tr>
<tr>
<td>diff</td>
<td>import</td>
<td>read</td>
<td>where</td>
</tr>
<tr>
<td>div</td>
<td>in</td>
<td>readi</td>
<td>while</td>
</tr>
<tr>
<td>do</td>
<td>intrinsic</td>
<td>repeat</td>
<td>xor</td>
</tr>
<tr>
<td>elif</td>
<td>is</td>
<td>require</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>join</td>
<td>requirege</td>
<td></td>
</tr>
</tbody>
</table>
6 THE MAGMA PROFILER

6.1 Introduction . . . . . . . . . 131
6.2 Profiler Basics . . . . . . . . . 131
   SetProfile(b) 131
   ProfileReset() 131
   ProfileGraph() 132
6.3 Exploring the Call Graph . . 133
   6.3.1 Internal Reports . . . . . . . . . . . . . . . 133
   ProfilePrintByTotalCount(G) 134
   ProfilePrintByTotalTime(G) 134
   ProfilePrintChildrenByCount(G, n) 134
   ProfilePrintChildrenByTime(G, n) 134
6.3.2 HTML Reports . . . . . . . . . . . 135
   ProfileHTMLOutput(G, prefix) 135
6.4 Recursion and the Profiler . . 135
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.

<table>
<thead>
<tr>
<th>SetProfile(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turns profiling on (if ( b ) is \texttt{true} ) or off (if ( b ) is \texttt{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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ProfileReset()</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>
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 \( ⟨x, y⟩ \) 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.

```magma
> 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
```
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 : −1

Print the list of functions in the call graph, sorted in descending order by the total number of times they were called. The Time and Count fields of the report give the total time and total number of times the function was called. If Percentage is true, then the Time and Count fields represent their values as percentages of the total value. If Max is non-negative, then the report only displays the first Max entries.

ProfilePrintByTotalTime(G)

Percentage  BoolElt  Default : false
Max          RNGIntElt Default : −1

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 : −1

Given a vertex n in the call graph G, print the list of functions called by the function n, sorted in descending order by the number of times they were called by n. The Time and Count fields of the report give the time spent during calls by the function n and the number of times the function was called by the function n. If Percentage is true, then the Time and Count fields represent their values as percentages of the total value. If Max is non-negative, then the report only displays the first Max entries.

ProfilePrintChildrenByTime(G, n)

Percentage  BoolElt  Default : false
Max          RNGIntElt Default : −1

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);

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Time</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;main&gt;</td>
<td>10.940</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>fibonacci</td>
<td>10.940</td>
<td>392835</td>
</tr>
<tr>
<td>3</td>
<td>eq(RngIntElt x, RngIntElt y) → BoolElt</td>
<td>1.210</td>
<td>710646</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Time</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>fibonacci (*)</td>
<td>182.430</td>
<td>392834</td>
</tr>
<tr>
<td>3</td>
<td>eq(&lt;RngIntElt&gt; x, &lt;RngIntElt&gt; y) -&gt; BoolElt</td>
<td>1.210</td>
<td>710645</td>
</tr>
<tr>
<td>4</td>
<td>- (&lt;RngIntElt&gt; x, &lt;RngIntElt&gt; y) -&gt; RngIntElt</td>
<td>0.630</td>
<td>392834</td>
</tr>
<tr>
<td>5</td>
<td>+ (&lt;RngIntElt&gt; x, &lt;RngIntElt&gt; y) -&gt; RngIntElt</td>
<td>0.250</td>
<td>196417</td>
</tr>
</tbody>
</table>

* 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.

```magma
gt := 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:

```magma
ProfilePrintByTotalTime(G);
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Time</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>&lt;main&gt;</td>
<td>1.020</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>recursive</td>
<td>1.020</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>delay</td>
<td>1.020</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Cputime(T) -&gt; FldReElt</td>
<td>0.130</td>
<td>14880</td>
</tr>
<tr>
<td>7</td>
<td>+(x, y) -&gt; RngIntElt</td>
<td>0.020</td>
<td>14880</td>
</tr>
<tr>
<td>9</td>
<td>gt(x, y) -&gt; BoolElt</td>
<td>0.020</td>
<td>14880</td>
</tr>
<tr>
<td>3</td>
<td>ne(x, y) -&gt; BoolElt</td>
<td>0.000</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>-(x, y) -&gt; RngIntElt</td>
<td>0.000</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Cputime() -&gt; FldReElt</td>
<td>0.000</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>SetProfile(v)</td>
<td>0.000</td>
<td>1</td>
</tr>
</tbody>
</table>

However, printing the children of `recursive`, and displaying the results in percentages, does yield a surprise:

```magma
ProfilePrintChildrenByTime(G, 2 : Percentage);
```

Function: `recursive`
Function Time: 1.020
Function Count: 2

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Time</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>delay</td>
<td>100.00</td>
<td>33.33</td>
</tr>
<tr>
<td>2</td>
<td>recursive</td>
<td>50.00</td>
<td>16.67</td>
</tr>
</tbody>
</table>
3 \text{ne(<RngIntElt> x, <RngIntElt> y) -> BoolElt} \quad 0.00 \quad 33.33
4 \text{-(<RngIntElt> x, <RngIntElt> y) -> RngIntElt} \quad 0.00 \quad 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 \textit{recursive} includes the time for the recursive call, which is also counted separately. In more detail:

\begin{verbatim}
> V := Vertices(G);
> E := Edges(G);
> Label(V!1)'Name;
<main>
> Label(V!2)'Name;
recursive
> Label(E![1,2])'Time;
1.019999999999999795718
> Label(E![2,2])'Time;
0.51000000000000000888
> Label(V!2)'Time;
1.019999999999999795718
\end{verbatim}

As can seen in the above, the total time for \textit{recursive} is approximately one second, as expected. The double-counting of the recursive call can be seen in the values of \textit{Time} for the edges [1,2] and [2,2].
7 DEBUGGING MAGMA CODE

7.1 Introduction . . . . . . . . 141
SetDebugOnError(f) 141

7.2 Using the Debugger . . . . . 141
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 in 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.
- **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 \( f(n) = \sum_{i=1}^{n} 1/n \), but have in our implementation we have accidentally included the evaluation of the term at \( n = 0 \).

```magma
> 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 '1/': Division by zero

debug> p n
0
debug> p 1.0 / (n + 1)
1.00000000000000000000000000000000

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
1.00000000000000000000000000000000
PART II
SETS, SEQUENCES, AND MAPPINGS

8 INTRODUCTION TO AGGREGATES 147
9 SETS 157
10 SEQUENCES 185
11 TUPLES AND CARTESIAN PRODUCTS 207
12 LISTS 215
13 ASSOCIATIVE ARRAYS 221
14 COPRODUCTS 225
15 RECORDS 231
16 MAPPINGS 237
8 INTRODUCTION TO AGGREGATES

8.1 Introduction ........................................... 149

8.2 Restrictions on Sets and Sequences .............. 149
  8.2.1 Universe of a Set or Sequence .............. 150
  8.2.2 Modifying the Universe of a Set or Sequence .......... 151

8.3 Nested Aggregates .................................. 154
  8.3.1 Multi-indexing ................................ 154

8.2.3 Parents of Sets and Sequences ............ 153
Chapter 8
INTRODUCTION TO AGGREGATES

8.1 Introduction
This part of the Handbook comprises four chapters on aggregate objects in MAGMA as well as a chapter on maps.

Sets, sequences, tuples and lists are the four main types of aggregates, and each has its own specific purpose. Sets are used to collect objects that are elements of some common structure, and the most important operation is to test element membership. Sequences also contain objects of a common structure, but here the emphasis is on the ordering of the objects, and the most important operation is that of accessing (or modifying) elements at given positions. Sets will contain at most one copy of any element, whereas sequences may contain arbitrarily many copies of the same object. Enumerated sets and sequences are of arbitrary but finite length and will store all elements explicitly (with the exception of arithmetic progressions), while formal sets and sequences may be infinite, and use a Boolean function to test element membership. Indexed sets are a hybrid form of sets allowing indexing like sequences. Elements of Cartesian products of structures in MAGMA will be called tuples; they are of fixed length, and each coefficient must be in the corresponding structure of the defining Cartesian product. Lists are arbitrary finite ordered collections of objects of any type, and are mainly provided to the user to store assorted data to which access is not critical.

8.2 Restrictions on Sets and Sequences
Here we will explain the subtleties behind the mechanism dealing with sets and sequences and their universes and parents. Although the same principles apply to their formal counterparts, we will only talk about enumerated sets and sequences here, for two reasons: the enumerated versions are much more useful and common, and the very restricted number of operations on formal sets/sequences make issues of universe and overstructure of less importance for them.

In principle, every object $e$ in MAGMA has some parent structure $S$ such that $e \in S$; this structure can be used for type checking (are we allowed to apply function $f$ to $e$?), algorithm look-up etc. To avoid storing the structure with every element of a set or sequence and having to look up the structure of every element separately, only elements of a common structure are allowed in sets or sequences, and that common parent will only be stored once.
8.2.1 Universe of a Set or Sequence

This common structure is called the **universe** of the set or sequence. In the general constructors it may be specified up front to make clear what the universe for the set or sequence will be; the difference between the sets $i$ and $s$ in

```plaintext
> 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:

```plaintext
> i := { 1, 2, 3 }
```

Only empty sets and sequences that have been obtained directly from the constructions

```plaintext
> 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

```plaintext
> 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

```plaintext
> 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

```plaintext
> 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 \( x \in \mathbb{Z} \) 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 ];

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 \textit{parent} for $S$ with the desired universe and using the \texttt{!} operator for the coercion; as we will see in the next subsection, such a parent can be created using the \texttt{PowerSet} and \texttt{PowerSequence} commands. Thus, for example, the set $\{1,2\}$ can be made into a sequence of rationals as follows:

```magma
> I := \{1, 2\};
> P := \texttt{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 \textit{not} necessary that the old universe could be coerced completely into the new one: the set $\{3/3\}$ of rationals can be coerced into $\texttt{PowerSet(Integers())}$. 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 \texttt{join} or \texttt{cat}) can only applied to sets and sequences that are \textit{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:

```magma
> T := \{1, 2\} \texttt{join} \{1/3\};
```

for the same reason that we can do

```magma
> 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 \textit{over-structure} for the parents $A$ and $B$ (see below);
5. nested sets and sequences are compatible only when they are of the same ‘depth’ and ‘type’ (that is, sets and sequences appear in exactly the same recursive order in both) and the universes are compatible.

The possibility of finding an overstructure $C$ for the universe $A$ and $B$ of sets or sequences $S$ and $T$ (such that $A \subset C \supset B$), is again module dependent. We refer the reader for
details to the Introductions of Parts III–VI, and we give some examples here; the next subsection contains the rules for parents of sets and sequences.

### 8.2.3 Parents of Sets and Sequences

The universe of a set or sequence $S$ is the common parent for all its elements; but $S$ itself is a Magma object as well, so it should have a parent too.

The parent of a set is a power set: the set of all subsets of the universe of $S$. It can be created using the `PowerSet` function. Similarly, `PowerSequence(A)` creates the parent structure for a sequence of elements from the structure $A$ – that is, the elements of `PowerSequence(A)` are all sequences of elements of $A$.

The rules for finding a common overstructure for structures $A$ and $B$, where either $A$ or $B$ is a set/sequence or the parent of a set/sequence, are as follows. (If neither $A$ nor $B$ is a set, sequence, or its parent we refer to the Part of this manual describing the operations on $A$ and $B$.)

1. The overstructure of $A$ and $B$ is the same as that of $B$ and $A$.
2. If $A$ is the null set or sequence (empty, and no universe specified) the overstructure of $A$ and $B$ is $B$.
3. If $A$ is a set or sequence with universe $U$, the overstructure of $A$ and $B$ is the overstructure of $U$ and $B$; in particular, the overstructure of $A$ and $A$ will be the universe $U$ of $A$.
4. If $A$ is the parent of a set (a power set), then $A$ and $B$ can only have a common overstructure if $B$ is also the parent of a set, in which case the overstructure is the power set of the overstructure of the universes $U$ and $V$ of $A$ and $B$ respectively. Likewise for sequences instead of sets.

We give two examples to illustrate rules (3) and (4). It is possible to create a set with a set as its universe:

```plaintext
> 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

```plaintext
> 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:

```plaintext
> a := [ [ 1 ], [ 1, 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 \( b \),

\[
> \ c := a \ \text{cat} \ b;
\]

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

\[
\]

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 \( S = [ [1, 2], [2, 3, \text{undef}, \text{undef}, 7] ] \))

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

(and the result will be \( S = [ [1, 2], [2, 3], \text{undef}, [7] ] \)). But the following results in an error:

\[
> \ S[4,1] := 7;
\]

Finally we point out that multi-indexing should not be confused with the use of sequences as
indexes to create subsequences. For example, to create a subsequence of \( S = [5, 13, 17, 29] \) consisting of the second and third terms, one may use

\[
> \ S := [5, 13, 17, 29];
> \ T := S[ [2, 3] ];
\]

To obtain the second term of this subsequence one could have done:

\[
> \ x := S[ [2, 3] ][2];
\]

(so \( x \) now has the value \( S[3] = 17 \)), but it would have been more efficient to index the indexing sequence, since it is rather expensive to build the subsequence \( [ S[2], S[3] ] \) first, so:

\[
> \ x := S[ [2, 3][2] ];
\]

has the same effect but is better (of course \( x := S[3] \) would be even better in this simple example.) To add to the confusion, it is possible to mix the above constructions for indexing, since one can use lists of sequences and indices for indexing; continuing our example, there is now a third way to do the same as above, using an indexing list that first takes out the subsequence consisting of the second and third terms and then extracts the second term of that:

\[
> \ x := S[ [2, 3], 2 ];
\]

Similarly, the construction

\[
> \ x := S[ [2, 3], [2] ];
\]

pulls out the subsequence consisting of the second term of the subsequence of terms two and three of \( S \), in other words, this assigns the sequence consisting of the element 17, not just the element itself!
9 SETS

9.1 Introduction ........................................ 159
9.1.1 Enumerated Sets ................................. 159
9.1.2 Formal Sets ......................................... 159
9.1.3 Indexed Sets ........................................ 159
9.1.4 Multisets ........................................... 159
9.1.5 Compatibility ....................................... 160
9.1.6 Notation ............................................ 160

9.2 Creating Sets .......................................... 160
9.2.1 The Formal Set Constructor ..................... 160
{ | x in F | P(x) }
9.2.2 The Enumerated Set Constructor ................. 161
{ e(x) : x in E | P(x) }
9.2.3 The Indexed Set Constructor ..................... 163
{ e(x) : x in E | P(x) }
9.2.4 The Multiset Constructor ......................... 164
{ | i .. j by k }
9.2.5 The Arithmetic Progression Constructors ....... 166
{ | i .. j by k }

9.3 Power Sets ............................................. 167
PowerSet(R)
PowerFormalSet(R)
PowerMultiset(R)

9.4 Sets from Structures ................................. 169
Set(M)
FormalSet(M)

9.5 Accessing and Modifying Sets ....................... 170
9.5.1 Accessing Sets and their Associated Structures ............................................. 170
#
Category(S)
Type(S)
Parent(R)
Universe(R)
Index(S, x)
Position(S, x)
S[i]
S[I]

9.5.2 Selecting Elements of Sets ....................... 171
Random(R)
random{ e(x) : x in E | P(x) }
random{ e(x_1, ..., x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) }
Representative(R)
Rep(R)
ExtractRep(~R, ~r)
rep{ e(x) : x in E | P(x) }
rep{ e(x_1, ..., x_k) : x_1 in E_1, ..., x_k in E_k | P(x_1, ..., x_k) }
Minimum(S)
Min(S)
Maximum(S)
Max(S)
Hash(x)

9.5.3 Modifying Sets .................................... 174
Include(~S, x)
Include(S, x)
Exclude(~S, x)
Exclude(S, x)
ChangeUniverse(~S, V)
9.6 Operations on Sets . . . . . . . 177
9.6.1 Boolean Functions and Operators . 177
IsNull(R) 177
IsEmpty(R) 177
eq 177
in 177
notin 177
subset 178
notsubset 178
eq 178
ne 178
IsDisjoint(R, S) 178
9.6.2 Binary Set Operators . . . . . . 178
join 178
meet 179
diff 179
sdiff 179
9.6.3 Other Set Operations . . . . . . . 179
Multiplicity(S, x) 179
Multiplicities(S) 179
Subsets(S) 179
Subsets(S, k) 179
Multisets(S, k) 180
Subsequences(S, k) 180
Permutations(S) 180
Permutations(S, k) 180
9.7 Quantifiers . . . . . . . . . 180
exists(t){ e(x): x in E | P(x) } 180
exists(t, ..., t_r){ e(x): x in E | P(x) } 180
forall(t){ e(x): x in E | P(x) } 181
forall(t, ..., t_r){ e(x): x in E | P(x) } 181

9.8 Reduction and Iteration over Sets 183
x in S 183
& 183
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 \( \text{in} \) 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):

\[
\{! x \in F \mid P(x) !\}
\]

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.

\[
\{ U \mid \}
\]

The empty set with universe \(U\).

\[
\{ e_1, e_2, \ldots, e_n \}
\]

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 set \(\{a_1, a_2, \ldots, a_n\}\) 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):

\[
> T := \{ \text{Integers()} \mid (7^2+1)/5, (8^2+1)/5, (9^2-1)/5 \};
> T;
{ 10, 13, 16 }
> Parent(T);
Set of subsets of Integer Ring
\]

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 \(\{a_1, a_2, \ldots, a_n\}\) 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, \ldots, x_k) : x_1 in E_1, \ldots, x_k in E_k | P(x_1, \ldots, 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 $P(x_1, \ldots, x_k)$ is always true, it may be omitted (including the $|$).

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.

```ansi
> 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;
```
CH. 9

SETS

\{< 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, \ldots, 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 indexed set \( Q = \{ a_1, a_2, \ldots, a_n \} \) of elements of \( U \).

\{U | e_1, e_2, \ldots, 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 indexed set \( Q = \{ a_1, a_2, \ldots, a_n \} \) of elements of \( U \).

\{e(x) : x \in E \mid 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 \mid e(x) : x \in E \mid 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 \(|\)
The indexed set consisting of those elements \( e(x_1,\ldots,x_k) \) (in some common structure), for which \( x_1,\ldots,x_k \in E_1 \times \cdots \times E_k \) 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 \( P(x_1,\ldots,x_k) \) is always true, it may be omitted.

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.

```plaintext
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 \(^\wedge\wedge\) (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 \( U \).
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 \( Q = \{ \ast a_1, a_2, \ldots, a_n \ast \} \) of elements of \( U \).

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 \( Q = \{ \ast a_1, a_2, \ldots, a_n \ast \} \) of elements of \( U \).

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 \(|\)).

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 \(|\)).

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

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 an 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.

```plaintext
> 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.

```plaintext
{ 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.
The enumerated set consisting of the integers forming the arithmetic progression \( i, i + k, i + 2k, \ldots, j \), 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 \( i + nk \) 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 \( i + nk \) 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.

```latex
\begin{verbatim}
> 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 }
\end{verbatim}
```

### 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).

The structure comprising all enumerated subsets of structure \( R \).

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 \texttt{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 $R$; \texttt{false} otherwise.

PowerFormalSet(R)

The structure comprising all formal subsets of structure $R$.

$S \in P$

Returns \texttt{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 $R$; \texttt{false} otherwise.

$S \in P$

Returns \texttt{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 $R$; \texttt{false} otherwise.

$P \setminus S$

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 $R$.

$P \setminus S$

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 $R$.

$P \setminus S$

Return an 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.

\textbf{Category}(S)

The category of the object \( S \). For a set this will be one of \texttt{SetEnum}, \texttt{SetIndx}, \texttt{SetMulti}, or \texttt{SetFormal}. For a power set the type is one of \texttt{PowSetEnum}, \texttt{PowSetIndx}, \texttt{PowSetMulti}.

\textbf{Parent}(R)

Returns the parent structure of \( R \), that is, the structure consisting of all (enumerated) sequences over the universe of \( R \).

\textbf{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.

\textbf{Index}(S, x)

\textbf{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.

\textbf{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.

\textbf{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.

```magma
> 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.

\[
\text{random}\{ e(x) : x \in E \mid P(x) \}
\]

Given an enumerated structure \( E \) and a Boolean expression \( P \), return the value of the expression \( e(y) \) for a randomly chosen element \( y \) of \( E \) for which \( P(y) \) is true.

\( P \) may be omitted if it is always true.

\[
\text{random}\{e(x_1, \ldots, x_k) : x_1 \in E_1, \ldots, x_k \in E_k \mid P(x_1, \ldots, x_k)\}
\]

Given enumerated structures \( E_1, \ldots, E_k \), and a Boolean expression \( P(x_1, \ldots, x_k) \), return the value of the expression \( e(y_1, \ldots, y_k) \) 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.

\( P \) 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.

\[
> p := 10007;
> F := \text{FiniteField}(p);
> \text{proots} := \{ z : z \in F \mid \text{IsPrimitive}(z) \};
> \#\text{proots};
5002
> \text{Random(}\text{proots});
5279
\]

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 \text{random}.

\[
> \text{random}\{ x : x \in F \mid \text{IsPrimitive}(x) \};
4263
\]

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)**

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, \ldots, x_k) : x_1 in E_1, \ldots, x_k in E_k | P(x_1, \ldots, 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 $e(y_1, \ldots, y_k)$ 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 $P$.

$P$ 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 $x, y, z \leq 1000$).

```plaintext
> 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 \texttt{lt} and \texttt{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 \texttt{lt} and \texttt{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(\sim 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(\sim 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.

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.

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.

```plaintext
> 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.

```plaintext
> R := { 218, 271, 511 }; // it has been emptied !
> { { x, y } : x, y in cubes | x+y in R };
```
\{ [-729, 1000],
    [-125, 343],
    [-1, 512],
\}

\begin{itemize}
    \item \textbf{SetToIndexedSet(E)}
        \begin{itemize}
            \item Given an enumerated set \( E \), this function returns an indexed set with the same elements (and universe) as \( E \).
        \end{itemize}
    \item \textbf{IndexedSetToSet(S)}
        \begin{itemize}
            \item Given an indexed set \( S \), this function returns an enumerated set with the same elements (and universe) as \( E \).
        \end{itemize}
    \item \textbf{IndexedSetToSequence(S)}
        \begin{itemize}
            \item Given an indexed set \( S \), this function returns a sequence with the same elements (and universe) as \( E \).
        \end{itemize}
    \item \textbf{MultisetToSet(S)}
        \begin{itemize}
            \item Given a multiset \( S \), this function returns an enumerated set with the same elements (and universe) as \( S \).
        \end{itemize}
    \item \textbf{SetToMultiset(E)}
        \begin{itemize}
            \item Given an enumerated set \( E \), this function returns a multiset with the same elements (and universe) as \( E \).
        \end{itemize}
    \item \textbf{SequenceToMultiset(Q)}
        \begin{itemize}
            \item Given an enumerated sequence \( E \), this function returns a multiset with the same elements (and universe) as \( E \).
        \end{itemize}
\end{itemize}
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.
\textbf{R subset S}

Returns \texttt{true} if the enumerated, indexed or multi-set $R$ is a subset of the set $S$, \texttt{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.

\textbf{R notsubset S}

Returns \texttt{true} if the enumerated, indexed, or multi-set $R$ is a not a subset of the set $S$, \texttt{false} otherwise. Coercion of the elements of $R$ into $S$ is attempted if necessary, and an error occurs if this fails.

\textbf{R eq S}

Returns \texttt{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.

\textbf{R ne S}

Returns \texttt{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.

\textbf{IsDisjoint(R, S)}

Returns \texttt{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.

\subsection{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

\[ Q := \{ \neg x \text{ in } R \} \]

converts an enumerated set $R$ into a formal set $Q$.

\textbf{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.
**Sets**

**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 $S$).

**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 $S$).

**Example H9E11**

```plaintext
> 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.
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 \mid 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.

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 \( E \), then the Boolean value false is returned.

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.

\( P \) may be omitted if it is always true.
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 $e(y_1, \ldots, y_k)$. 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 $e(y_1, \ldots, y_k)$ 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.

$P$ 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):

```plaintext
> 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
```

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
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.

$P$ may be omitted if it is always true.

```latex
forall(t){e(x_1, \ldots, x_k): x_1 \text{ in } E_1, \ldots, x_k \text{ in } E_k | P(x_1, \ldots, x_k)}$
```

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 $e(y_1, \ldots, y_k)$. 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 $e(y_1, \ldots, y_k)$ 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.

$P$ 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.

```plaintext
> 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 \&.

\( x \text{ in } S \)

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.

\&\circ\ 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 \( \circ : U \times U \rightarrow U \), 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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Value</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;+</td>
<td>empty</td>
<td>( U ) ! 0</td>
</tr>
<tr>
<td>&amp;*</td>
<td>empty</td>
<td>( U ) ! 1</td>
</tr>
<tr>
<td>&amp;and</td>
<td>empty</td>
<td>true</td>
</tr>
<tr>
<td>&amp;or</td>
<td>empty</td>
<td>false</td>
</tr>
<tr>
<td>&amp;join</td>
<td>empty</td>
<td>null</td>
</tr>
<tr>
<td>&amp;meet</td>
<td>empty</td>
<td>error</td>
</tr>
</tbody>
</table>

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 \texttt{choose} defined below takes a set \( S \) and an integer \( k \) as input, and produces a set of all subsets of \( S \) with cardinality \( k \).

```plaintext
> 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;```

```plaintext```
end if;
end function;

So, for example:

S := { 1, 2, 3, 4 };
choose(S, 2);
{
    { 1, 3 },
    { 1, 4 },
    { 2, 4 },
    { 2, 3 },
    { 1, 2 },
    { 3, 4 }
}

Try to guess what happens if \( k < 0 \).
10 SEQUENCES

10.1 Introduction . . . . . . . . 187
10.1.1 Enumerated Sequences . . . . . . 187
10.1.2 Formal Sequences . . . . . . . . 187
10.1.3 Compatibility . . . . . . . . . . 188
10.2 Creating Sequences . . . . 188
10.2.1 The Formal Sequence Constructor . 188
[! x in F | P(x) !] 188
10.2.2 The Enumerated Sequence Constructor . . . . . . . . . . . . . . 189
[ U | e ] 189
[ e1, e2, ..., en ] 189
[ U | e1, e2, ..., em ] 189
[ e(x) : x in E | P(x) ] 189
[ U | e(x) : x in E | P(x) ] 189
[ e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) ] 189
[ U | e(x1, ..., xk) : x1 in E1, ..., xk in Ek | P(x1, ..., xk) ] 190
10.2.3 The Arithmetic Progression Constructors . . . . . . . . . . . . . . 190
[ i .. j ] 190
[ U | i .. j ] 190
[ i .. j by k ] 190
[ U | i .. j by k ] 190
10.2.4 Literal Sequences . . . . . . 191
\[ m1, ..., mn \] 191
10.3 Power Sequences . . . . . 191
PowerSequence(R) 191
in 191
!
10.4 Operators on Sequences . . . . 192
10.4.1 Access Functions . . . . . . . . 192
# 192
Parent(S) 192
Universe(S) 192
S[i] 192
10.4.2 Selection Operators on Enumerated Sequences . . . . . . . . . . . . . . 193
S[i] 193
Minimum(S) 193
Min(S) 193
Maximum(S) 193
Max(S) 193
Index(S, x) 193
Index(S, x, f) 193
Position(S, x) 193
Position(S, x, f) 193
Representative(R) 193
Rep(R) 193
Random(R) 194
Explode(R) 194
Eltsseq(R) 194
10.4.3 Modifying Enumerated Sequences . . . . . . . . . . . . . . 194
Append(~S, x) 194
Append(S, x) 194
Exclude(~S, x) 194
Exclude(S, x) 194
Include(~S, x) 195
Include(S, x) 195
Insert(~S, i, x) 195
Insert(S, i, x) 195
Insert(~S, k, m, T) 195
Insert(S, k, m, T) 195
Prune(~S) 196
Prune(S) 196
Remove(~S, i) 196
Remove(S, i) 196
Reverse(~S) 196
Reverse(S) 196
Rotate(~S, p) 196
Rotate(S, p) 196
Sort(~S) 197
Sort(S) 197
Sort(~S, C) 197
Sort(S, C) 197
Sort(~S, C, ~p) 197
Sort(S, C, ~p) 197
Undefine(~S, i) 197
Undefine(S, i) 197
ChangeUniverse(S, V) 197
ChangeUniverse(S, V) 197
CanChangeUniverse(S, V) 198
10.4.4 Creating New Enumerated Sequences from Existing Ones . . . . . . . . . . . . . . 199
cat 199
Partition(S, p) 199
Partition(S, P) 199
Setseq(S) 199
SetSequence(S) 199
Seqset(S) 200
SequenceToSet(S) 200
And(S, T) 201
And(~S, T) 201
Or(S, T) 201
Or(~S, T) 201
Xor(S, T) 201
Xor(~S, T) 201
Not(S) 201
Not(~S) 201
10.5 Predicates on Sequences . . . . 201
Chapter 10
SEQUENCES

10.1 Introduction
A sequence in Magma is a linearly ordered collection of objects belonging to some common structure (called the universe of the sequence).

There are two types of sequence: enumerated sequences, of which the elements are all stored explicitly (with one exception, see below); and formal sequences, of which elements are stored implicitly by means of a predicate that allows for testing membership. In particular, enumerated sequences are always finite, and formal sequences are allowed to be infinite. In this chapter a sequence will be either a formal or an enumerated sequence.

10.1.1 Enumerated Sequences
An enumerated sequence of length \( l \) is an array of indefinite length of which only finitely many terms — including the \( l \)-th term, but no term of bigger index — have been defined to be elements of some common structure. Such sequence is called complete if all of the terms (from index 1 up to the length \( l \)) are defined.

In practice the length of any sequence is bounded by the constant integer \( \beta \) (usually \( 2^{29} \)).

Incomplete enumerated sequences are allowed as a convenience for the programmer in building complete enumerated sequences. Some sequence functions require their arguments to be complete; if that is the case, it is mentioned explicitly in the description below. However, all functions using sequences in other Magma modules always assume that a sequence that is passed in as an argument is complete. Note that the following line converts a possibly incomplete sequence \( S \) into a complete sequence \( T \):

\[
T := \{ s : s \in S \};
\]

because the enumeration using the \( \in \) operator simply ignores undefined terms.

Enumerated sequences of Booleans are highly optimized (stored as bit-vectors).

10.1.2 Formal Sequences
A formal sequence consists of elements of some range set on which a certain predicate assumes the value ‘true’.

There is only a very limited number of operations that can be performed on them.
10.1.3 Compatibility
The binary operators for sequences do not allow mixing of the formal and enumerated sequence types (so one cannot take the concatenation of an enumerated sequence and a formal sequence, for example); but it is easy to convert an enumerated sequence into a formal sequence – see the section on binary operators below.

By the limitation on their construction formal sequences can only contain elements from one structure in Magma. The elements of enumerated sequences are also restricted, in the sense that either some common structure must be specified upon creation, or Magma must be able to find such universe automatically. The rules for compatibility of elements and the way Magma deals with these parents is the same for sequences and sets, and is outlined in the Introduction to this Part of the Handbook.

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 \( \text{in} \) 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):

\[ [! x \text{ in } F \mid P(x) !] \]

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 \([! x \text{ in } F !]\).
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.

\[
\text{[ ]}
\]

The null sequence (empty, and no universe specified).

\[
\text{[ U | ]}
\]

The empty sequence with universe $U$.

\[
\text{[ e}_1, e_2, \ldots, e_n \text{ ]}
\]

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 $Q = [a_1, a_2, \ldots, a_n]$ of elements of $U$.

\[
\text{[ U | e}_1, e_2, \ldots, e_m \text{ ]}
\]

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 $Q = [a_1, a_2, \ldots, a_n]$ of elements of $U$.

\[
\text{[ e(x) : x in E | P(x) ]}
\]

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 $[ e(x) : x \text{ in E } ]$.

\[
\text{[ U | e(x) : x in E | P(x) ]}
\]

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.

\[
\text{[ e(x}_1, \ldots, x_k \text{ ) : x}_1 \text{ in E}_1, \ldots, x_k \text{ in E}_k | P(x}_1, \ldots, x_k \text{ ) ]}
\]

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 $P(x_1, \ldots, x_k)$ is always true, it may be omitted.
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.

\[
[ U \mid e(x_1, \ldots, x_k) : x_1 \in E_1, \ldots, x_k \in E_k \mid P(x_1, \ldots, x_k) ]
\]

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.

\[
> s := [ \text{IntegerRing}(200) \mid x : x \in [25..125] ];
\]
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.

\[ [m_1, \ldots, m_n] \]

Given a succession of literal integers \(m_1, \ldots, m_n\), build the enumerated sequence \([m_1, \ldots, m_n]\), in a time and space efficient way.

10.3 Power Sequences

The \texttt{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).

\texttt{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.

\texttt{S in P}

Returns \texttt{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 \(R\); \texttt{false} otherwise.

\texttt{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

\begin{verbatim}
> 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;
\end{verbatim}
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 $\mathsf{lt}$ and $\mathsf{eq}$ 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 $\mathsf{gt}$ and $\mathsf{eq}$ 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($S$, $x$), Position($S$, $x$)

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)
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)
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.

\[
\text{Include}(\sim S, x)
\]

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.

\[
\text{Insert}(\sim S, i, x)
\]

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 \([s_1, \ldots, s_{i-1}, x, s_i, \ldots, s_n]\). 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.

\[
\text{Insert}(\sim S, k, m, T)
\]

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(\sim 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(\sim S, i)
Remove(S, i)

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

Create the enumerated sequence formed by reversing the order of the terms in the complete enumerated sequence \( S \), i.e., the sequence \([s_n, \ldots, s_1]\).

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(\sim S, p)
Rotate(S, p)

Given a complete sequence \( S \) and an integer \( p \), create the enumerated sequence formed by cyclically rotating the terms of the sequence \( p \) terms: if \( p \) is positive, rotation will be to the right; if \( p \) is negative, \( S \) is cyclically rotated \(-p\) terms to the left; if \( p \) is zero nothing happens.

There are two versions of this: a procedure, where \( S \) is replaced by the new sequence, and a function, which returns the new sequence. The procedural version takes a reference \( \sim S \) to \( S \) as an argument.

Note that the procedural version is much more efficient since the sequence \( S \) will not be copied.
Sort(~S)

Given a complete enumerated sequence S whose terms belong to a structure on which \texttt{lt} and \texttt{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.: \texttt{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.

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 \text{true} and \( T \), otherwise return \text{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 \( n \):

\[
p'' = \left\lfloor \frac{q' + n q}{q''} \right\rfloor p' - p, \quad q'' = \left\lfloor \frac{q' + n q}{q''} \right\rfloor q' - q.
\]

```
> farey := function(n)
>     f := [ RationalField() | 0, 1/n ];
>     p := 0;
>     q := 1;
>     while p/q < 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 \( \frac{p}{q}, \frac{p'}{q'} \) 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);
>     end if;
```
The third method is very straightforward, and uses Sort and Setseq (defined above).

```plaintext
> 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).

**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.
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].

```plaintext
> 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:

```plaintext
> 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)**
  - The sequence whose \(i\)th entry is the logical and of the \(i\)th entries of \(S\) and \(T\). The result is placed in \(S\) if it is given by reference (\(\sim\)).

- **Or(S, T)**
  - The sequence whose \(i\)th entry is the logical or of the \(i\)th entries of \(S\) and \(T\). The result is placed in \(S\) if it is given by reference.

- **Xor(S, T)**
  - The sequence whose \(i\)th entry is the logical xor of the \(i\)th entries of \(S\) and \(T\). The result is placed in \(S\) if it is given by reference.

- **Not(S)**
  - The sequence whose \(i\)th entry is the logical not of the \(i\)th entry of \(S\). The result is placed in \(S\) if it is given by reference.

10.5 Predicates on Sequences

Boolean valued operators and functions on enumerated sequences exist to test whether entries are defined (see previous section), to test for membership and containment, and to compare sequences with respect to an ordering on its entries. On formal sequences, only element membership can be tested.

- **IsComplete(S)**
  - Boolean valued function, returning true if and only if each of the terms \(S[i]\) for \(1 \leq i \leq \#S\) is defined, for an enumerated sequence \(S\).

- **IsDefined(S, i)**
  - Given an enumerated sequence \(S\) and an index \(i\), this returns true if and only if \(S[i]\) is defined. (Hence the result is false if \(i > \#S\), but an error results if \(i < 1\).) Note that the index \(i\) is allowed to be a multi-index; if \(i = [i_1, \ldots, i_r]\) 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 \(i_j \leq \#S[i_1, \ldots, i_{j-1}]\) for \(1 \leq j \leq s\), then an error occurs.
IsEmpty(S)

Boolean valued function, returning true if and only if the enumerated sequence S is empty.

IsNull(S)

Boolean valued function, returning true if and only if the enumerated sequence S is empty and its universe is undefined, false otherwise.

10.5.1 Membership Testing

Here, S and T denote sequences. The element x is always assumed to be compatible with S.

x in S

Returns true if the object x occurs as a term of the enumerated or formal sequence S, false otherwise. If x is not in the universe of S, coercion is attempted. If that fails, an error results.

x notin S

Returns true if the object x does not occur as a term of the enumerated or formal sequence S, false otherwise. If x is not in the universe of S, coercion is attempted. If that fails, an error results.

IsSubsequence(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 \( \text{eq} (=) \), \( \text{le} (\leq) \), \( \text{lt} (<) \), \( \text{gt} (>\) ), and \( \text{ge} (\geq) \) to be invoked on its elements.

With these comparison operators the \textit{lexicographical} ordering is used to order complete enumerated sequences. Sequences \( S \) and \( T \) are equal (\( S \ \text{eq} \ T \)) if and only if they have the same length and all terms are the same. A sequence \( S \) precedes \( T \) (\( S \ \text{lt} \ T \)) 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 \( S \ \text{lt} \ T \) is true also. In all other cases where \( S \neq T \) one has \( S \ \text{gt} \ T \).

\[ S \ \text{lt} \ T \]

Returns \textbf{true} if the sequence \( S \) precedes the sequence \( T \) under the ordering induced from \( S \), \textbf{false} otherwise. Thus, \textbf{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 \( \#S < \#T \).

\[ S \ \text{le} \ T \]

Returns \textbf{true} if the sequence \( S \) either precedes the sequence \( T \), under the ordering induced from \( S \), or is equal to \( T \), \textbf{false} otherwise. Thus, \textbf{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 \( \#S \leq \#T \).

\[ S \ \text{ge} \ T \]

Returns \textbf{true} if the sequence \( S \) either comes after the sequence \( T \), under the ordering induced from \( S \), or is equal to \( T \), \textbf{false} otherwise. Thus, \textbf{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 \( \#S \geq \#T \).

\[ S \ \text{gt} \ T \]

Returns \textbf{true} if the sequence \( S \) comes after the sequence \( T \) under the ordering induced from \( S \), \textbf{false} otherwise. Thus, \textbf{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.

\[
\text{Self}(n) \\
\text{Self}()
\]

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.

\[ s := [ \text{if } i > 2 \text{ select } \text{Self}(i-2)+\text{Self}(i-1) \text{ else } 1 : i \in [1..100] ]; \]
\[ &+s; \]
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 \&.

\[ \& \circ S \]

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: \( + \), \( * \), \text{and}, \text{or}, \text{join}, \text{meet}, \text{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:
### 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.

```plaintext
> [<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.

```plaintext
> // 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

11 TUPLES AND CARTESIAN PRODUCTS

11.1 Introduction . . . . . . . . 209

11.2 Cartesian Product Constructor and Functions . . . . . . . . 209
  car< > 209
  CartesianProduct(R, S) 209
  CartesianProduct(L) 209
  CartesianPower(R, k) 209
  Flat(C) 209
  NumberOfComponents(C) 210
  Component(C, i) 210
  C[i] 210
  # 210
  Rep(C) 210
  Random(C) 210

11.3 Creating and Modifying Tuples 210
  elt< > 210
  ! 210
  < a₁, a₂, ..., aₖ > 210

11.4 Tuple Access Functions . . . 212
  Append(T, x) 210
  Append(~T, x) 211
  Prune(T) 211
  Prune(~T) 211
  Flat(T) 211

11.5 Equality . . . . . . . . . . 212
  eq 212
  ne 212

11.6 Other operations . . . . . . 213
  &* 213
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 $\text{car}< \ldots >$ is used for the creation of cartesian products of structures.

$\text{car}< R_1, \ldots, R_k >$
Given a list of sets or algebraic structures $R_1, \ldots, R_k$, construct the cartesian product set $R_1 \times \cdots \times R_k$.

$\text{CartesianProduct}(R, S)$
Given structures $R$ and $S$, construct the cartesian product set $R \times S$. This is the same as calling the $\text{car}$ constructor with the two arguments $R$ and $S$.

$\text{CartesianProduct}(L)$
Given a sequence or tuple $L$ of structures, construct the cartesian product of the elements of $L$.

$\text{CartesianPower}(R, k)$
Given a structure $R$ and an integer $k$, construct the cartesian power set $R^k$

$\text{Flat}(C)$
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 $\text{Flat}$ 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 $\mathbb{Q}$ and $\mathbb{Z}$.

```maple
> 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 $T = < a_1, a_2, \ldots, a_k >$ 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 $T = < a_1, a_2, \ldots, a_k >$ 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

\textbf{Parent(T)}

The cartesian product to which the tuple \( T \) belongs.

\textbf{#T}

Number of components of the tuple \( T \).

\textbf{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 \( T \).

\textbf{Explode(T)}

Given a tuple \( T \) of length \( n \), this function returns the \( n \) entries of \( T \) (in order).

\textbf{TupleToList(T) / Tuplist(T)}

Given a tuple \( T \) return a list containing the entries of \( T \).

\textbf{Example H11E3}

\begin{verbatim}
> 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)>
(1/6) * (59 + 6*zeta_3)
> f[3] := 7;
> f;
<11/2, 13/3, 7>
\end{verbatim}

11.5 Equality

\textbf{T eq U}

Return \texttt{true} if and only if the tuples \( T \) and \( U \) are equal.

\textbf{T ne U}

Return \texttt{true} if and only if the tuples \( T \) and \( U \) are distinct.
11.6 Other operations

\texttt{\&*T}

For a tuple $T$ where each component lies in a structure that supports multiplication and such there exists a common over structure, return the product of the entries.
12 LISTS

12.1 Introduction . . . . . . . . 217
12.2 Construction of Lists . . . . 217
[* *]
[* e₁, e₂, ..., eₙ *]
12.3 Creation of New Lists . . . . 217
cat:= 217
Append(S, x) 217
Append(~S, x) 217
Insert(~S, i, x) 218
Insert(S, i, x) 218
Prune(S) 218
Prune(~S) 218

12.4 Access Functions . . . . . . 218
IsEmpty(S) 218
S[i] 218
IsDefined(L, i) 219

12.5 Assignment Operator . . . . 219
S[i] := x 219
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 e1, ..., en, defining elements a1, a2, ..., an, create the list containing a1, a2, ..., an.

12.3 Creation of New Lists
Here, S denotes the list [* s1, ..., sn *], while T denotes the list [* t1, ..., tm *].

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 [* s1, ..., sn, t1, ..., tm *].

S cat:= T
(Procedure.) Destructively concatenate the terms of the list T to S; i.e. so S becomes the list [* s1, ..., sn, t1, ..., tm *].

Append(S, x)
The list formed by adding the object x to the end of the list S, i.e. the list [* s1, ..., sn, x *].

Append(~S, x)
(Procedure.) Destructively add the object x to the end of the list S; i.e. so S becomes the list [* s1, ..., sn, x *].
\begin{itemize}
\item \textbf{Insert(}~\textit{S, i, x}~\textbf{)}
\end{itemize}
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 \( \ast s_1, \ldots, s_{i-1}, x, s_i, \ldots, s_n \ast \). 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.

\begin{itemize}
\item \textbf{Prune(}~\textit{S}~\textbf{)}
\end{itemize}
The list formed by removing the last term of the list \( S \), i.e. the list \( \ast s_1, \ldots, s_{n-1} \ast \).

\begin{itemize}
\item \textbf{Prune(}~\textit{\sim S}~\textbf{)}
\end{itemize}
(Procedure.) Destructively remove the last term of the list \( S \); i.e. so \( S \) becomes the list \( \ast s_1, \ldots, s_{n-1} \ast \).

\begin{itemize}
\item \textbf{SequenceToList(}~\textit{Q}\textbf{)}
\end{itemize}
\begin{itemize}
\item \textbf{Seqlist(}~\textit{Q}\textbf{)}
\end{itemize}
Given a sequence \( Q \), construct a list whose terms are the elements of \( Q \) taken in the same order.

\begin{itemize}
\item \textbf{TupleToList(}~\textit{T}\textbf{)}
\end{itemize}
\begin{itemize}
\item \textbf{Tuplist(}~\textit{T}\textbf{)}
\end{itemize}
Given a tuple \( T \), construct a list whose terms are the elements of \( T \) taken in the same order.

\begin{itemize}
\item \textbf{Reverse(}~\textit{L}\textbf{)}
\end{itemize}
Given a list \( L \) return the same list, but in reverse order.

\section{12.4 Access Functions}
\begin{itemize}
\item \textbf{\#S}
\end{itemize}
The length of the list \( S \).

\begin{itemize}
\item \textbf{IsEmpty(}~\textit{S}\textbf{)}
\end{itemize}
Return whether \( S \) is empty (has zero length).

\begin{itemize}
\item \textbf{S[i]}
\end{itemize}
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 Introduction for the interpretation).
IsDefined(L, i)
Checks whether the $i$th item in $L$ is defined or not, that is it returns $\text{true}$ if $i$ is at most the length of $L$ and $\text{false}$ otherwise.

12.5 Assignment Operator

$S[i] := x$
Redefine the $i$-th term of the list $S$ to be $x$. If $i \leq 0$, then an error results. If $i = \#S + 1$, then $x$ is appended to $S$. Otherwise, if $i > \#S + 1$, an error results. Here $i$ is allowed to be a multi-index.
13 ASSOCIATIVE ARRAYS

13.1 Introduction . . . . . . . . 223
13.2 Operations . . . . . . . . . 223

AssociativeArray() 223
AssociativeArray(I) 223
A[x] := y 223
A[x] 223
IsDefined(A, x) 223
Remove(~ A, x) 223
Universe(A) 223
Keys(A) 223
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

\begin{verbatim}
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.
\end{verbatim}
Example H13E1

This example shows simple use of associative arrays. First we create an array indexed by rationals.

```plaintext
> A := AssociativeArray();
> A[1/2] := 7;
> 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 $S_3$.

```plaintext
> G := Sym(3);
> A := AssociativeArray(G);
> v := 1; for x in G do A[x] := v; v := v + 1; end for;
> A;
Associative Array with index universe GrpPerm: G, Degree 3, Order 2 * 3
> Keys(A);
{ (1, 3, 2),
  (2, 3),
  (1, 3),
  (1, 2, 3),
  (1, 2),
  Id(G) }
> A[G!(1,3,2)];
3
```
14 COPRODUCTS

14.1 Introduction  . . . . . . . . 227
14.2 Creation Functions  . . . . . 227
  14.2.1 Creation of Coproducts  . . . . . 227
cop< > 227
cop< > 227
  14.2.2 Creation of Coproduct Elements . 227
m(e) 227
! 227
14.3 Accessing Functions  . . . . 228
  Injections(C) 228

# 228
Constituent(C, i) 228
Index(x) 228
14.4 Retrieve  . . . . . . . . . . 228
  Retrieve(x) 228
14.5 Flattening  . . . . . . . . . 229
  Flat(C) 229
14.6 Universal Map  . . . . . . . 229
  UniversalMap(C, S, [ n1, ..., nm ]) 229
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 con-
structed 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
\[ \text{cop}< S_1, S_2, \ldots, S_k > \]

Given a list or a sequence of two or more structures \( S_1, S_2, \ldots, S_k \), 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 \( m_i : S_i \rightarrow C \).

14.2.2 Creation of Coproduct Elements
Coproduct elements are usually created by the injections returned as the second return
value from the \( \text{cop}< > \) constructor. The bang (!) operator may also be used but only if the
type of the relevant constituent is unique for the particular coproduct.

\[ \text{m(e)} \]

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.

```plaintext
> 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.

\[ \text{Flat}(C) \]

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

\[ \text{UniversalMap}(C, S, [ n_1, \ldots, n_m ]) \]

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

15.1 Introduction . . . . . . . . 233
15.2 The Record Format Constructor 233
   recformat< > 233
15.3 Creating a Record . . . . . . 234
   rec< > 234
15.4 Access and Modification
   Functions . . . . . . . . . . 235

Format(r) 235
Names(F) 235
Names(r) 235
r'fieldname 235
r'fieldname:= e; 235
delete 235
assigned 235
r''s 235
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 \texttt{recformat< ... >} is used for the creation of record formats. A record format must be created before records in that format are created.

\texttt{recformat< L >}

(and optional parents or categories) in \texttt{L}

Construct the record format corresponding to the non-empty fieldname list \texttt{L}. Each term of \texttt{L} must be one of the following:

(a) \texttt{fieldname} in which case there is no restriction on values that may be stored in this field of records having this format;

(b) \texttt{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) \texttt{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 \texttt{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; \( \text{misc} \), which has no restrictions; and \( \text{seq} \), a sequence (with any universe possible).

\[
\begin{align*}
&\text{RF} := \text{recformat}< n : \text{Integers()}, \text{misc}, \text{seq} : \text{SeqEnum} >; \\
&\text{RF}; \\
&\text{Names(RF)}; \\
&[ n, \text{misc}, \text{seq} ]
\end{align*}
\]

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.

\[
\text{rec} < F \mid L >
\]

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 \( \text{fieldname} := \text{expression} \) where \( \text{fieldname} \) 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.

\[
\begin{align*}
&\text{RF} := \text{recformat}< n : \text{Integers()}, \text{misc}, \text{seq} : \text{SeqEnum} >; \\
&r := \text{rec}< \text{RF} \mid >; \\
&r; \\
&\text{recRF} \mid > \\
&s := \text{rec}< \text{RF} \mid \text{misc} := \text{"adsifaj"}, n := 42, \text{seq} := [ \text{GF(13)} \mid 4, 8, 1 ]>; \\
&s; \\
&\text{recRF} \mid n := 42, \text{misc} := \text{adsifaj}, \text{seq} := [ 4, 8, 1 ]> \\
&t := \text{rec}< \text{RF} \mid \text{seq} := [ 4.7, 1.9 ], n := 51/3 >; \\
&t; \\
&\text{recRF} \mid n := 17, \text{seq} := [ 4.7, 1.9 ]> \\
&u := \text{rec}< \text{RF} \mid \text{misc} := \text{RModule(PolynomialRing(Integers(7)), 4)}>; \\
&u; \\
&\text{recRF} \mid \text{misc} := \text{RModule of dimension 4 with base ring Univariate Polynomial Algebra over Integers(7)}>
\end{align*}
\]
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.

$r^{\text{fieldname}}$

Return the field of record $r$ with this fieldname. The format of $r$ must include this fieldname, and the field must be assigned in $r$.

$r^{\text{fieldname}} := \text{expression};$

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^{\text{fieldname}}**

(Statement.) Delete the current value of the given field of record $r$.

**assigned r^{\text{fieldname}}**

Returns true if and only if the given field of record $r$ currently contains a value.

$r^{\text{string}}$

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 $r^{\text{fieldname}}$ may be used, including in left hand side assignment, assigned and delete.
Example H15E3

```plaintext
RF := recformat< n : Integers(), misc, seq : SeqEnum >;
r := rec< RF | >;
s := rec< RF | misc := "adsifaj", n := 42, seq := [ GF(13) | 4, 8, 1 ]>;
t := rec< RF | seq := [ 4.7, 1.9 ], n := 51/3 >;
u := rec< RF | misc := RModule(PolynomialRing(Integers(7)), 4) >;
V4 := u'misc;
assigned r'seq;
false
r'seq := Append(t'seq, t'n); assigned r'seq;
true
r;
rec<RF | seq := [ 4.7, 1.9, 17 ]>
// The following produces an error:
t''(s'misc);
>> t''(s'misc);
^  
Runtime error in ': Field 'adsifaj' does not exist in this record
delete u''("m" cat "isc"); u;
rec<RF | >
```
16 MAPPINGS

16.1 Introduction . . . . . . . . 239
16.1.1 The Map Constructors . . . . . . 239
16.1.2 The Graph of a Map . . . . . . 240
16.1.3 Rules for Maps . . . . . . . . 240
16.1.4 Homomorphisms . . . . . . . . 240
16.1.5 Checking of Maps . . . . . . . . 240

16.2 Creation Functions . . . . 241
16.2.1 Creation of Maps . . . . . . . . 241
map< > 241
map< > 241
map< > 241
16.2.2 Creation of Partial Maps . . . . 242
pmap< > 242
pmap< > 242
pmap< > 242
16.2.3 Creation of Homomorphisms . . . 242
hom< > 242
hom< > 242
hom< > 242
hom< > 243
pmap< > 243
16.2.4 Coercion Maps . . . . . . . . 243
Coercion(D, C) 243
Bang(D, C) 243

16.3 Operations on Mappings . . . 243
16.3.1 Composition . . . . . . . . 243

* 243
Components(f) 243
16.3.2 (Co)Domain and (Co)Kernel . . . 244
Domain(f) 244
Codomain(f) 244
Image(f) 244
Kernel(f) 244
16.3.3 Inverse . . . . . . . . . . 244
Inverse(m) 244
16.3.4 Function . . . . . . . . . . 244
Function(f) 244

16.4 Images and Preimages . . . 245
f(a) 245
f(S) 245
f(C) 245
HasPreimage(x, f) 245

16.5 Parents of Maps . . . . . . 246
Parent(m) 246
Domain(P) 246
Codomain(P) 246
Maps(D, C) 246
Iso(D, C) 246
Aut(S) 246
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 \textit{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 \textit{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 \textit{Magma} category \texttt{Map}.

16.1.1 The Map Constructors

There are three main mapping constructors: the general map constructor \texttt{map< >}, the homomorphism constructor \texttt{hom< >}, and the partial map constructor \texttt{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 \textit{domain} \( A \) and a \textit{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 \textit{graph} of the map, or a \textit{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 \texttt{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 \textit{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 \textit{images} for \textit{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 \textit{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 $a \rightarrow b$. The specification of a (sub)graph in a map constructor should then consist of either a (comma separated) list, a sequence, or a set of such tuples or arrow pairs (a mixture is permitted).

16.1.3 Rules for Maps

The specification of a rule in the map constructor involves a free variable and an expression, usually involving the free variable, separated by $:->$, for example $x :-> 3*x - 1$. The scope of the free variable is restricted to the map constructor (so the use of $x$ does not interfere with values of $x$ outside the constructor). A general expression is allowed in the rule, which may involve intrinsic or user functions, and even in-line definitions of such functions.

16.1.4 Homomorphisms

Probably the most useful form of the map-constructor is the version for homomorphisms. Most interesting mappings in algebra are homomorphisms, and if an algebraic structure $A$ belongs to a family of algebraic structures which form a variety we have the fundamental result that a homomorphism is uniquely determined by the images of any generating set. This provides us with a particularly compact way of defining and representing homomorphisms. While the syntax of the homomorphism constructor is similar to that of the general mapping constructor, the semantics are sometimes different.

The kind of homomorphism built by the hom-constructor is determined entirely by the domain: thus, a group homomorphism results from applying $\text{hom}$ 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.

\[
\text{map} \triangleleft A \rightarrow B \mid G \triangleright
\]

Given a finite structure $A$, a structure $B$ and a graph $G$ of $A \times B$, construct the mapping $f : A \rightarrow B$, as defined by $G$. The graph $G$ may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter. Note that $G$ must be a full graph, i.e., every element of $A$ must occur exactly once as a first component.

\[
\text{map} \triangleleft A \rightarrow B \mid x \rightarrow e(x) \triangleright
\]

Given a set or structure $A$, a set or structure $B$, a variable $x$ and an expression $e(x)$, usually involving $x$, construct the mapping $f : A \rightarrow B$, as defined by $e(x)$. It is the user’s responsibility to ensure that a value is defined for every $x \in A$. The scope of the variable $x$ is restricted to the map-constructor.

\[
\text{map} \triangleleft A \rightarrow B \mid x \rightarrow e(x), y \rightarrow i(y) \triangleright
\]

Given a set or structure $A$, a set or structure $B$, a variable $x$, an expression $e(x)$, usually involving $x$, a variable $y$, and an expression $i(y)$, usually involving $y$, construct the mapping $f : A \rightarrow B$, as defined by $x \mapsto e(x)$, with corresponding inverse $f^{-1} : B \rightarrow A$, as defined by $y \mapsto i(y)$. It is the user’s responsibility to ensure that a value $e(x)$ is defined for every $x \in A$, a value $i(y)$ is defined for every $y \in B$, and that $i(y)$ is the true inverse of $e(x)$. The scope of the variables $x$ and $y$ is restricted to the map-constructor.
16.2.2 Creation of Partial Maps

Partial mappings are quite different to both general mappings and homomorphisms, in that images need not be defined for every element of the domain.

\[ \text{pmap< } A \rightarrow B \mid G \] 

Given a finite structure \( A \) of cardinality \( n \), a structure \( B \) and a subgraph \( G \) of \( A \times B \), construct the partial map \( f : A \rightarrow B \), as defined by \( G \). The subgraph \( G \) may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

\[ \text{pmap< } A \rightarrow B \mid x :\rightarrow e(x) \] 

Given a set \( A \), a set \( B \), a variable \( x \) and an expression \( e(x) \), construct the partial map \( f : A \rightarrow B \), as defined by \( e(x) \). This form of the map constructor is a special case of the previous one whereby the image of \( x \) can be defined using a single expression. Again the scope of \( x \) is restricted to the map-constructor.

\[ \text{pmap< } A \rightarrow B \mid x :\rightarrow e(x), y :\rightarrow i(y) \] 

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.

\[ \text{hom< } A \rightarrow B \mid G \] 

Given a finitely generated algebraic structure \( A \) and a structure \( B \), as well as a graph \( G \) of \( A \times B \), construct the homomorphism \( f : A \rightarrow B \) defined by extending the map of the generators of \( A \) to all of \( A \). The graph \( G \) may be given by either a set, sequence, or list of tuples or arrow-pairs as described in the Introduction to this Chapter.

The detailed requirements on the specification are module dependent, and can be found in the chapter describing the domain \( A \).

\[ \text{hom< } A \rightarrow B \mid y_1, \ldots, y_n \] 
\[ \text{hom< } A \rightarrow B \mid x_1 \rightarrow y_1, \ldots, x_n \rightarrow y_n \] 

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.
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.

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 \texttt{Coercion} function can be used.

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.

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 \( h = g \circ f : A \to C \).

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.

\[ \text{Domain}(f) \]

The domain of the mapping \( f \).

\[ \text{Codomain}(f) \]

The codomain of the mapping \( f \).

\[ \text{Image}(f) \]

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.

\[ \text{Kernel}(f) \]

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

\[ \text{Inverse}(m) \]

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.

\[ \text{Function}(f) \]

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 \( K \ast y @@ f \), 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 \( A \).

\[ \text{HasPreimage}(x, f) \]

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 \texttt{PowMap}. The type \texttt{PowMapAut} which inherits from \texttt{PowMap} is type which the parents of automorphisms inherit from.

There is also a power structure of maps (of type \texttt{PowStr}, similar to that of other structures) which is used as a common overstructure of the different parents.

\begin{itemize}
\item \texttt{Parent(m)}
  
  The parent of \texttt{m}.
\item \texttt{Domain(P)}\texttt{Codomain(P)}
  
  The domain and codomain of the maps for which \texttt{P} is the parent.
\item \texttt{Maps(D, C)}\texttt{Iso(D, C)}
  
  The parent of maps (or isomorphisms) from \texttt{D} to \texttt{C}. \texttt{Iso} will only return a different structure to \texttt{Maps} if it has been specifically implemented for such maps.
\item \texttt{Aut(S)}
  
  The parent of automorphisms of \texttt{S}.
\end{itemize}
INDEX OF INTRINSICS
!, 1-13, 1-168, 1-191, 1-210, 1-227,
2-261, 2-275, 2-326, 2-332, 2-343,
344, 2-360, 2-385, 2-417, 2-448,
2-546, 3-581, 3-656, 3-662, 3-694,
695, 3-766, 3-822, 3-869, 3-891,
3-945–947, 3-969, 3-975, 3-1013,
4-1033, 4-1084, 4-1117, 4-1128,
4-1147, 4-1171, 4-1185, 4-1207,
5-1234, 5-1276, 5-1278, 5-1293,
1294, 5-1306, 5-1405, 1406, 5-1409,
5-1514, 5-1525, 5-1575, 5-1578,
5-1695, 6-1742, 6-1755, 6-1771,
6-1956, 6-1964, 6-2003–2005, 6-2056,
6-2073, 2074, 6-2086, 6-2091, 6-2096,
6-2116, 7-2311, 8-2337, 8-2348,
8-2371, 2372, 8-2384, 8-2427, 8-2441,
2442, 8-2445, 8-2459, 2460, 8-2547,
8-2582, 8-2594, 2595, 8-2638, 8-2658,
8-2667, 8-2671, 8-2705, 2706, 9-2864,
9-2875, 2876, 9-2936, 9-2953, 9-2998,
9-3010, 9-3013, 9-3104, 9-3117,
9-3133, 9-3143, 9-3158, 9-3183,
10-3281, 10-3432, 10-3444, 3445,
10-3475, 11-3546, 3547, 11-3574,
11-3599, 11-3638, 11-3685, 11-3699,
11-3741, 11-3789, 12-3856, 3857,
12-3922, 3923, 12-3925, 3926, 12-3930,
3931, 12-3958, 3959, 12-3966, 12-3985,
3986, 12-4038, 4039, 12-4110, 4111,
13-4186, 13-4292, 13-4306, 13-4351
!!, 3-747, 3-958, 5-1306, 11-3645
∼, 12-3964
( , ), 2-548, 4-1174, 5-1236, 5-1307,
5-1414, 5-1575, 6-1773, 6-2058,
6-2075, 8-2361, 13-4187, 13-4293,
13-4307
( , , ), 5-1236, 5-1307, 5-1414, 5-1516,
5-1695, 6-1773, 6-1958, 6-2058,
6-2075
(,), 7-2313
( ), 1-227, 1-245, 2-562, 4-1186,
6-1776, 6-1789, 6-1849, 6-2037,
8-2712, 10-3279
*, 1-60, 1-243, 2-261, 2-265, 2-279,
2-302, 2-306, 2-327, 2-329, 2-336,
2-347, 2-367, 2-389, 2-405, 2-419,
2-451, 2-502, 2-530, 2-547, 2-562,
3-582, 3-588, 3-657, 3-665, 3-721,
3-747, 3-756, 3-766, 3-769, 3-822,
3-844, 3-877, 3-893, 3-948, 3-958,
959, 3-972, 3-975, 976, 3-994,
3-1012, 1013, 3-1018, 4-1034, 4-1088,

4-1119, 4-1130, 4-1142, 4-1150,
4-1172, 4-1186, 4-1200, 4-1208, 1209,
4-1221, 5-1236, 5-1306, 5-1309,
5-1367, 5-1414, 5-1440, 5-1516,
5-1575, 5-1695, 6-1744, 6-1756,
6-1773, 6-1848, 1849, 6-1957, 6-2016,
6-2057, 6-2075, 6-2086, 6-2096,
6-2117, 7-2313, 7-2322, 8-2342, 2343,
8-2372, 8-2374, 8-2376, 8-2387,
8-2396, 8-2401, 8-2427, 8-2469,
8-2500, 8-2505, 8-2547, 8-2563,
8-2582, 8-2597, 8-2608, 8-2638,
8-2659, 8-2669, 8-2673, 8-2711,
8-2742, 2743, 9-2794, 9-2850, 9-2865,
9-2876, 2877, 9-2941, 9-2956, 9-3010,
9-3039, 9-3133, 9-3150, 9-3156,
9-3161, 10-3283, 10-3353, 10-3410,
10-3437, 10-3449, 10-3476, 10-3501,
11-3546, 11-3549, 3550, 11-3574,
11-3577, 11-3604, 11-3685, 11-3705,
11-3768, 11-3790, 11-3817, 12-3923,
12-3926, 12-3941, 12-3963, 13-4186,
13-4293, 13-4306, 13-4353
*:=, 1-60, 2-262, 2-279, 2-327, 2-347,
2-367, 2-389, 2-419, 2-451, 3-582,
3-877, 4-1034, 4-1119, 5-1516,
6-1957, 6-2016, 8-2387, 8-2742,
10-3283, 10-3449, 12-3963
+, 2-261, 2-265, 2-279, 2-302, 2-306,
2-327, 2-329, 2-347, 2-367, 2-389,
2-405, 2-419, 2-451, 2-502, 2-547,
2-559, 3-581, 3-591, 3-691, 3-721,
3-756, 3-766, 3-769, 3-877, 3-893,
3-912, 3-948, 3-959, 3-972, 3-976,
3-994, 3-1013, 3-1018, 4-1034,
4-1088, 4-1119, 4-1130, 4-1142,
4-1150, 4-1172, 4-1177, 4-1200,
4-1208, 4-1221, 6-1744, 6-1755,
7-2174, 7-2216, 8-2343, 8-2370,
8-2372, 8-2376, 8-2387, 8-2396,
8-2401, 8-2427, 8-2430, 8-2442,
8-2468, 8-2500, 8-2547, 8-2582,
8-2608, 8-2638, 8-2659, 8-2673,
8-2711, 8-2742, 9-2794, 9-2850,
9-2865, 9-2867, 9-2876, 9-2880,
9-2941, 9-2956, 9-3010, 9-3150,
9-3156, 9-3161, 10-3283, 10-3408,
10-3448, 11-3550, 11-3574, 11-3604,
11-3685, 11-3705, 11-3768, 3769,
11-3780, 11-3792, 11-3817, 11-3826,
12-3963, 12-4044–4046, 12-4122–4124,
12-4159, 4160, 13-4186, 13-4192,


INDEX OF INTRINSICS

AbsoluteDiscriminant, AbsoluteDegree, Abs, AbelianSubfield, 5
AbelianQuotient, 5
AbelianGroup, {, }
=, 2-262, 2-279, 2-327, 2-347, 2-451, 5-1516, 6-1957, 1958, 6-2016
AddAttribute, 10-3429
AddCubics, 8
AddEdge, AddEdges, 8
AbsoluteField, 3-702
AbsoluteFunctionField, 3-916
AbsoluteGaloisGroup, 3-852
AbsoluteInvariants, 10-3429
AbsoluteLogarithmicHeight, 3-724
AbsolutelyIrreducibleConstituents, 8-2699
AbsolutelyIrreducibleModule, 8-2432
AbsolutelyIrreducibleModules, 8-2696
AbsolutelyIrreducibleModulesBurnside, 8-2699
AbsolutelyIrreducibleModulesInit, 8-2702
AbsolutelyIrreducibleModulesSchur, 5-1557, 8-2700
AbsolutelyIrreducibleRepresentationProcessDelete, 8-2702
AbsolutelyIrreducibleRepresentationsInit, 8-2702
AbsolutelyIrreducibleRepresentationsSchur, 5-1557
AbsoluteMinimalPolynomial, 3-726, 3-950
AbsoluteModuleOverMinimalField, 8-2689
AbsoluteModulesOverMinimalField, 8-2690
AbsoluteNorm, 2-369, 3-725, 3-749
AbsoluteOrder, 3-702, 3-916
AbsolutePolynomial, 3-881
AbsolutePrecision, 4-1091, 4-1131, 4-1142
AbsoluteQuotientRing, 3-881
AbsoluteRank, 7-2195
AbsoluteRationalScroll, 9-2997
AbsoluteRepresentation, 5-1451
AbsoluteRepresentationMatrix, 3-727
AbsoluteTotallyramifiedExtension, 4-1077
AbsoluteTrace, 2-369, 3-726
AbsoluteValue, 2-281, 2-349, 2-398, 2-437, 2-453, 11-3575
AbsoluteValues, 3-723
Absolutize, 3-881
ActingGroup, 5-1723, 7-2302
ActingWord, 5-1371
Action, 5-1336, 5-1343, 6-1849, 8-2424, 8-2504, 12-3879, 12-4007, 12-4088
ActionGenerator, 3-636, 8-2424, 8-2503, 8-2687
ActionGenerators, 8-2687
ActionGroup, 8-2687
ActionImage, 5-1343, 12-3879, 12-4007, 12-4089
ActionKernel, 5-1343, 12-3879, 12-4008, 12-4089
ActionMatrix, 8-2451, 8-2531
AdamsOperator, 8-2748
AddAttribute, 1-52
AddColumn, 2-498, 8-2477
AddConstraints, 13-4376
AddCubics, 10-3315, 10-3408
AddEdge, 12-4046, 12-4124, 4125, 12-4160
AddEdges, 12-4046, 4047, 12-4125, 12-4160
INDEX OF INTRINSICS

AlternatingPower, 8–2749
AlternatingSum, 2–480
AlternatingWeylSum, 8–2754
Ambient, 9–3006, 9–3073
AmbientModule, 11–3687
AmbientVariety, 11–3829
AmbiguousForms, 3–660
AModule, 8–2504, 8–2523
AnalyticDrinfeldModule, 3–1021
AnalyticHomomorphisms, 10–3482
AnalyticInformation, 10–3391
AnalyticJacobian, 10–3478
AnalyticModule, 3–1024
AnalyticRank, 10–3325, 10–3391
And, 1–201
and, 1–11
Angle, 11–3551, 11–3576
AnisotropicSubdatum, 7–2195
AntiAutomorphismTau, 8–2644
Antipode, 8–2643
AntisymmetricForms, 3–637
AntisymmetricMatrix, 2–490, 491
ApparentCodimension, 9–3205, 9–3214
ApparentEquationDegrees, 9–3205, 9–3214
ApparentSyzygyDegrees, 9–3205, 9–3214
Append, 1–194, 1–210, 211, 1–217
Apply, 9–2960
ApplyTransformation, 10–3410
ApproximateByTorsionGroup, 11–3823
ApproximateByTorsionPoint, 11–3822
ApproximateOrder, 11–3818
ApproximateStabiliser, 5–1444
AQLInvariants, 5–1537, 6–1801, 1802, 6–1984
Arcos, 2–464, 4–1138
Arcsec, 2–465
Arcsec, 2–465
Arcsin, 2–464, 4–1138
Arctan, 2–465, 4–1138
Arctan2, 2–465
AreCohomologous, 5–1724
AreIdentical, 6–2020
AreInvolutionsConjugate, 5–1475
Arg, 2–451
Argcosech, 2–467
Arccosh, 2–467, 4–1139
Arccoth, 2–468
Argsch, 2–467
Argsinh, 2–467, 4–1138
Argtanh, 2–467, 4–1139
Argument, 2–451, 11–3575
ArithmeticGenus, 9–3020, 9–3123
ArithmeticGeometricMean, 2–478
ArithmeticTriangleGroup, 11–3582
ArithmeticVolume, 11–3570, 11–3576
Arrows, 9–3193
ArtinMap, 3–850
ArtinSchreierExtension, 3–999
ArtinSchreierImage, 3–1014
ArtinSchreierMap, 3–1014
ASigmaL, 5–1389
ASL, 5–1389
AssertAttribute, 2–296, 2–359, 4–1127, 5–1315, 5–1384, 5–1429, 5–1464, 5–1466, 8–2717, 9–2875
AssertEmbedding, 11–3746
AssignCapacities, 12–4114, 4115
AssignCapacity, 12–4114
assigned, 1–6, 1–52, 1–235
AssignEdgeLabels, 12–4115
AssignLabel, 12–4113, 4114
AssignLabels, 12–4113, 4114
AssignLPCMatrix, 13–4256
AssignNamePrefix, 3–868
AssignVertexLabels, 12–4113
AssignWeight, 12–4114
AssignWeights, 12–4114, 4115
AssociatedEllipticCurve, 10–3303, 10–3307
AssociatedHyperellipticCurve, 10–3307
AssociatedNewSpace, 11–3647
AssociatedPrimitiveCharacter, 2–334
AssociativeAlgebra, 8–2336, 8–2357, 2358
AssociativeArray, 1–223
AtEqPairing, 10–3371
AteTPairing, 10–3371
AtkinLehner, 11–3653
AtkinLehnerInvolution, 11–3533
AtkinLehnerOperator, 11–3612, 11–3691, 11–3707, 11–3835, 3836
AtkinModularPolynomial, 11–3527
ATLASGroup, 5–1677
ATLASGroupNames, 5–1677
Attach, 1–47
AttachSpec, 1–49
Augmentation, 8–2674
AugmentationIdeal, 8–2671
AugmentationMap, 8–2670
AugmentCode, 13–4215, 13–4319
Aut, 1–246, 9–3057, 10–3436, 12–4006, 13–4241
AutoCorrelation, 13–4366
AutomaticGroup, 6–2066, 2067
Automorphism, 7–2322, 9–3051, 9–3054, 3055, 9–3057, 9–3128, 10–3251, 3252, 10–3348
INDEX OF INTRINSICS

AutomorphismGroup, 2-345, 2-365, 3-621, 3-623, 3-796, 797, 3-850, 3-928, 3-932, 4-1106, 5-1371, 5-1457, 5-1544, 5-1547, 5-1688, 5-1690, 7-2322, 8-2448, 9-3057, 9-3132, 10-3439, 12-3877, 12-3902, 12-4004, 12-4080, 13-4240, 13-4320, 13-4348
AutomorphismGroupStabilizer, 12-4005, 13-4241
AutomorphismOmega, 8-2644
Automorphisms, 3-796, 3-928, 3-932, 4-1106, 9-3132
AutomorphismSubgroup, 12-4005, 13-4241
AutomorphismTalpah, 8-2644
AutomorphousClasses, 3-630
AuxiliaryLevel, 11-3702
BachBound, 3-731
BadPlaces, 10-3335, 10-3385
BadPrimes, 10-3241, 10-3290, 10-3464
BaerDerivation, 12-3884
BaerSubplane, 12-3884
Ball, 12-4068
Bang, 1-243
BarAutomorphism, 8-2644
Base, 5-1385, 5-1466
BaseChange, 3-587, 9-3023, 9-3108, 10-3266, 3267, 10-3421, 10-3443, 10-3474
BaseChangeMatrix, 8-2514
BaseComponent, 9-3074
BaseCurve, 11-3532
BaseElement, 6-2031
BaseExtend, 2-332, 3-587, 7-2307, 9-3023, 10-3266, 3267, 10-3421, 10-3443, 10-3474, 11-3598, 11-3684, 11-3747, 11-3776
BaseField, 2-345, 2-357, 2-557, 3-701, 3-848, 3-875, 3-915, 3-1012, 1013, 4-1079, 7-2164, 8-2549, 9-2937, 9-3006, 9-3109, 10-3235, 10-3430, 10-3442, 10-3474, 10-3479
BaseImage, 5-1386
BaseImageWordStrip, 5-1387
BaseModule, 8-2462, 8-2611
BaseMPolynomial, 2-315
BasePoint, 5-1385, 5-1466
BasePoints, 9-3049, 9-3076
BaseRing, 2-333, 334, 2-387, 2-417, 2-494, 2-526, 3-587, 3-702, 3-848, 3-891, 3-915, 3-1013, 3-1016, 3-1018, 4-1079, 4-1129, 4-1141, 4-1146, 4-1169, 4-1198, 5-1409, 7-2164, 7-2195, 7-2305, 7-2307, 8-2338, 8-2367, 8-2385, 8-2423, 8-2462, 8-2498, 8-2549, 8-2597, 9-2866, 9-2879, 9-2937, 9-2954, 9-3006, 9-3109, 10-3235, 10-3275, 10-3277, 10-3406, 10-3430, 10-3442, 10-3474, 10-3479, 11-3542, 11-3568, 11-3607, 11-3686, 11-3702, 11-3727, 12-3960
BaseScheme, 9-3048, 9-3074
BasicAlgebra, 8-2497, 2498
BasicDegrees, 7-2239, 7-2279
BasicOrbit, 5-1385, 5-1466
BasicOrbitLength, 5-1385, 5-1466
BasicOrbitLengths, 5-1385, 5-1466
BasicOrbits, 5-1385
BasicStabiliser, 5-1385, 5-1467
BasicStabiliserChain, 5-1385, 5-1467
BasicStabilizer, 5-1385, 5-1467
BasicStabilizerChain, 5-1385, 5-1467
Basis, 2-345, 2-560, 3-586, 3-713, 3-752, 3-919, 3-965, 3-981, 4-1175, 4-1202, 4-1210, 8-2339, 8-2368, 8-2375, 8-2391, 8-2451, 8-2474, 8-2499, 8-2549, 8-2598, 8-2708, 9-2777, 9-2880, 9-3166, 10-3387, 11-3600, 11-3640, 11-3686, 11-3701, 11-3784, 13-4182, 13-4273, 13-4305
BasisChange, 7-2204
BasisDenominator, 3-586
BasisElement, 2-560, 8-2339, 8-2391, 8-2474, 8-2598, 8-2777, 9-2880
BasisMatrix, 2-560, 3-586, 3-714, 3-752, 3-920, 3-965, 8-2375, 8-2556, 8-2671, 9-2880, 13-4182, 13-4305
BasisOfDifferentialsFirstKind, 3-992, 9-3149
BasisOfHolomorphicDifferentials, 3-992, 9-3149
BasisProduct, 8-2348, 8-2594
BasisProducts, 8-2348, 8-2594
BasisReduction, 3-599
Basket, 9-3211, 9-3213
BBSModulus, 13-4366
BCHBound, 13-4229
BCHCode, 13-4210
BDLC, 13-4232
BDLCLowerBound, 13-4227
BDLCUpperBound, 13-4227
Bell, 2-288, 12-3914
BerlekampMassey, 13-4363
BernoulliApproximation, 2-478, 12-3914
BernoulliNumber, 2-478, 12-3914
BernoulliPolynomial, 2-409, 12-3914
BesselFunction, 2-476
BestApproximation, 2-459
BestDimensionLinearCode, 13-4232
BestKnownLinearCode, 13-4232
BestKnownQuantumCode, 13-4345
BestLengthLinearCode, 13-4232
BestTranslation, 2-317
BettiNumber, 10-3392
BettiNumbers, 9-3214
<table>
<thead>
<tr>
<th>INDEX OF INTRINSICS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bfstree, 12-4070, 12-4139</td>
</tr>
<tr>
<td>Bicomponents, 12-4061, 12-4135</td>
</tr>
<tr>
<td>Big0, 4-1084, 4-1129</td>
</tr>
<tr>
<td>Bigperiodmatrix, 10-3479</td>
</tr>
<tr>
<td>Binomial, 2-287, 12-3913</td>
</tr>
<tr>
<td>Bitflip, 13-4357</td>
</tr>
<tr>
<td>BklC, 13-4232</td>
</tr>
<tr>
<td>Bklcupperbound, 13-4227</td>
</tr>
<tr>
<td>BkqC, 13-4345</td>
</tr>
<tr>
<td>Block, 12-3985, 12-3996</td>
</tr>
<tr>
<td>Blocks, 5-1477, 8-2722, 12-3992</td>
</tr>
<tr>
<td>BlocksAction, 5-1347</td>
</tr>
<tr>
<td>Blocksset, 12-3985</td>
</tr>
<tr>
<td>Blocksimage, 5-1477, 5-1477</td>
</tr>
<tr>
<td>Blocksizes, 12-3993, 12-3995</td>
</tr>
<tr>
<td>BlocksKernel, 5-1347</td>
</tr>
<tr>
<td>Blowup, 9-3120</td>
</tr>
<tr>
<td>BllC, 13-4232</td>
</tr>
<tr>
<td>Bllcupperbound, 13-4227</td>
</tr>
<tr>
<td>Boolean, 1-11</td>
</tr>
<tr>
<td>Borelsubgroup, 12-3898</td>
</tr>
<tr>
<td>Branch, 8-2746</td>
</tr>
<tr>
<td>Branchvertexpath, 12-4071</td>
</tr>
<tr>
<td>Brauercharacter, 8-2722</td>
</tr>
<tr>
<td>Bravaisgroup, 3-636</td>
</tr>
<tr>
<td>Bruhatdescendants, 7-2240</td>
</tr>
<tr>
<td>CalabiYau, 9-3224</td>
</tr>
<tr>
<td>CanDetermineIsomorphism, 11-3734</td>
</tr>
<tr>
<td>CanIdentifyGroup, 5-1643</td>
</tr>
<tr>
<td>CanContinueEnumeration, 6-2009</td>
</tr>
<tr>
<td>CanContinueEnumeration, 6-1893</td>
</tr>
<tr>
<td>Canonb, 8-2640</td>
</tr>
<tr>
<td>Canonicalbasis, 8-2641</td>
</tr>
<tr>
<td>Canonicalclass, 9-3188</td>
</tr>
<tr>
<td>CanonicalDivisor, 9-3161</td>
</tr>
<tr>
<td>CanonicalElements, 8-2649</td>
</tr>
<tr>
<td>CanonicalFactorRepresentation, 6-2009</td>
</tr>
<tr>
<td>CanonicalHeight, 10-3298, 10-3460</td>
</tr>
<tr>
<td>CanonicalImage, 9-3168</td>
</tr>
<tr>
<td>Canonicalinvolution, 11-3533</td>
</tr>
<tr>
<td>CanonicalLinearSystem, 9-3115</td>
</tr>
<tr>
<td>CanonicalLinearSystemFromIdeal, 9-3114</td>
</tr>
<tr>
<td>Canonicalmap, 9-3168</td>
</tr>
<tr>
<td>CanonicalModularPolynomial, 11-3528</td>
</tr>
<tr>
<td>CanRedoEnumeration, 6-1893</td>
</tr>
<tr>
<td>CanSignNormalize, 3-1025</td>
</tr>
<tr>
<td>CanteautChabaudsAttack, 13-4225</td>
</tr>
<tr>
<td>Capacities, 12-4116</td>
</tr>
<tr>
<td>Capacity, 12-4116</td>
</tr>
<tr>
<td>CarlitzModule, 3-1019</td>
</tr>
<tr>
<td>Cartaninteger, 7-2222</td>
</tr>
<tr>
<td>CartanMatrix, 7-2137, 2138, 7-2145, 7-2163, 7-2193, 7-2238, 7-2278, 7-2309, 8-2490, 9-3190</td>
</tr>
<tr>
<td>CartanName, 7-2148, 7-2163, 7-2193, 7-2237, 7-2277, 7-2308, 8-2598</td>
</tr>
<tr>
<td>CartanSubalgebra, 8-2604</td>
</tr>
<tr>
<td>CartesiansetPower, 1-209</td>
</tr>
<tr>
<td>CartesiansetProduct, 1-209, 12-4050</td>
</tr>
<tr>
<td>Cartier, 3-997, 9-3151</td>
</tr>
<tr>
<td>CartierRepresentation, 3-997, 9-3151</td>
</tr>
<tr>
<td>CasimirValue, 8-2745</td>
</tr>
<tr>
<td>CasselsMap, 10-3338</td>
</tr>
<tr>
<td>CasselsTatePairing, 10-3305</td>
</tr>
<tr>
<td>Category, 1-60, 1-199, 1-217, 13-4219, 13-4290, 13-4320</td>
</tr>
</tbody>
</table>
INDEX OF INTRINSICS

ClassGroupAbelianInvariants, 3–940, 3–988, 9–3165
ClassGroupCyclicFactorGenerators, 3–732
ClassGroupExactSequence, 3–940, 3–989
ClassGroupGenerationBound, 3–987
ClassGroupGetUseMemory, 3–735
ClassGroupPRank, 3–941, 3–991, 9–3166
ClassGroupSetUseMemory, 3–735
ClassGroupStructure, 3–660
ClassicalForms, 5–1596
ClassicalMaximals, 5–1616
ClassicalModularPolynomial, 11–3528
ClassicalPeriod, 11–3670
ClassicalSylow, 5–1618
ClassicalSylowNormaliser, 5–1618
ClassicalSylowTopC, 5–1618
ClassicalType, 5–1601
ClassInvariants, 5–1428
ClassMap, 5–1266, 5–1314, 5–1426, 5–1521
ClassNumber, 3–659, 3–731, 3–779, 3–940, 3–988, 9–3164
ClassNumberApproximation, 3–987
ClassNumberApproximationBound, 3–987
ClassPowerCharacter, 8–2713
ClassRepresentative, 2–322, 3–761, 5–1268, 5–1313, 5–1427, 5–1521
ClassRepresentativeFromInvariants, 5–1428
ClassTwo, 5–1555
ClearDenominator, 11–3772
ClearDenominators, 2–431
ClearPrevious, 1–70
ClearVerbose, 1–97
ClebschGraph, 12–4054
ClebschInvariants, 10–3427, 3428
ClebschToIgusaClebsch, 10–3429
CliqueNumber, 12–4074
ClosestVectors, 3–607
ClosestVectorsMatrix, 3–607
ClosestVectors, 3–609
CloseVectorsMatrix, 3–610
CloseVectorsProcess, 3–614
Closure, 8–2759
ClosureGraph, 12–4052
Cluster, 9–3002, 9–3017
cmpeq, 1–12
cmpne, 1–12
CMPoints, 11–3584
CMTwists, 11–3731
CoboundaryMapImage, 5–1713
CocycleMap, 5–1725
CodeComplement, 13–4215, 13–4319
CodeToString, 1–61
Codimension, 9–3020, 9–3213
Codomain, 1–244, 1–246, 2–562, 4–1186, 5–1300, 5–1411, 6–1790, 6–2037,
7–2323, 8–2509, 9–3043, 10–3438, 11–3772, 11–3783
CoefficientHeight, 3–724, 3–749
CoefficientIdeals, 4–1209
CoefficientLength, 3–725, 3–750
CoefficientMap, 9–3076
Coefficients, 2–390, 2–420, 4–1086, 4–1132, 4–1142, 4–1151, 8–2387, 8–2639, 8–2659, 8–2674, 9–2957, 10–3272
CoefficientsNonSpiral, 4–1153
CoefficientSpace, 9–3076
CoefficientField, 3–848
CoefficientRing, 3–848
Coercion, 1–243
CohomologicalDimension, 5–1279, 5–1373, 5–1708
Cohomology, 5–1725
CohomologyClass, 5–1724
CohomologyElementToChainMap, 8–2525
CohomologyElementToCompactChainMap, 8–2525
CohomologyGeneratorToChainMap, 8–2518, 2519
CohomologyGroup, 5–1708
CohomologyLeftModuleGenerators, 8–2518
CohomologyModule, 8–484, 5–1706, 1707
CohomologyRightModuleGenerators, 8–2518
CohomologyRing, 8–2525
CohomologyRingGenerators, 8–2517
CohomologyRingQuotient, 8–2531
CohomologyToChainMap, 8–2531
CoisogenyGroup, 7–2198, 7–2239, 7–2279, 7–2310
Cokernel, 2–563, 4–1186, 4–1220, 8–2509, 11–3765, 11–3795
Collect, 6–1910, 8–2746
CollectRelations, 6–1908
CollineationGroup, 12–3877
CollineationGroupStabilizer, 12–3877
CollineationSubgroup, 12–3877
Colon, 8–2376
INDEX OF INTRINSICS

ColonIdeal, 3–758, 3–959, 9–2795
Column, 12–3937
ColumnLength, 12–3938
Columns, 12–3937
ColumnSkewLength, 12–3937
ColumnSubmatrix, 2–496, 497
ColumnSubmatrixRange, 2–497
ColumnWeight, 2–527
ColumnWeights, 2–527
ColumnWord, 12–3939
CombineInvariants, 3–809
CommonEigenspaces, 8–2482
CommonModularStructure, 11–3740
CommonOverfield, 2–357
CommonZeros, 3–953, 9–3155
Commutator, 7–2313
CommutatorIdeal, 8–2359, 8–2558
CommutatorModule, 8–2359
comp, 2–267, 3–707
CompactInjectiveResolution, 8–2515
CompactPresentation, 5–1570
CompactProjectiveResolution, 8–2510, 8–2524
CompactProjectiveResolutionGroup, 8–2524
CompanionMatrix, 2–404, 8–2460, 9–2968
Complement, 2–559, 9–3076, 11–3650, 11–3804, 12–3987, 12–4047
ComplementaryDivisor, 3–986, 9–3164
ComplementaryErrorFunction, 2–479
ComplementBasis, 5–1531
ComplementOfImage, 11–3804
Complements, 5–1364, 5–1541, 8–2438
Complete, 6–1794, 6–2032
CompleteClassGroup, 3–735
CompleteDigraph, 12–4035
CompleteGraph, 12–4034
CompleteKArc, 12–3872
CompleteTheSquare, 10–3403
CompleteUnion, 12–4050
CompleteWeightEnumerator, 13–4203, 13–4285, 13–4315, 4316
Complex, 4–1213
ComplexEmbeddings, 11–3620
ComplexField, 2–446, 447
ComplexReflectionGroup, 7–2275
ComplexToPolar, 2–451
ComplexValue, 11–3549, 11–3574
Component, 1–210, 9–3188, 12–4060, 4061, 12–4135, 4136
ComponentGroupOfIntersection, 11–3793
ComponentGroupOfKernel, 11–3762
ComponentGroupOrder, 11–3674, 11–3844
Components, 1–243, 3–847, 9–3039, 12–4060, 12–4135
ComposeQuotients, 6–1937
ComposeTransformations, 10–3410
Composite, 4–1078
CompositeFields, 3–682
Composition, 3–657, 3–933, 4–1134, 8–2718
CompositionFactors, 5–1265, 5–1357, 5–1454, 5–1539, 8–2433
CompositionSeries, 5–1352, 5–1539, 6–1757, 8–2340, 8–2433, 8–2605
Composite, 3–682
Composition, 3–657, 3–933, 4–1134, 8–2718
CompositionFactors, 5–1265, 5–1357, 5–1454, 5–1539, 8–2433
CompositionSeries, 5–1352, 5–1539, 6–1757, 8–2340, 8–2433, 8–2605
ConductorRange, 10–3332
ConformalHamiltonianLieAlgebra, 8–2593
ConformalSpecialLieAlgebra, 8–2592
ConformalSymplecticGroup, 5–1585
ConformalUnitaryGroup, 5–1584
CongruenceGroup, 11–3622, 11–3667
CongruenceGroupAnemic, 11–3622
CongruenceModulus, 11–3672, 11–3811
CongruenceSubgroup, 11–3541
Conic, 9–3107, 10–3234, 10–3248, 12–3872
ConjecturalRegulator, 10–3325
ConjugacyClasses, 5–1267, 5–1311, 5–1426, 5–1521, 7–2239, 8–2562
ConjugateIntoBorel, 7–2315
ConjugateIntoTorus, 7–2315
ConjugatePartition, 12–3936
ConjugatesToPowerSums, 3–820
ConjugationClassLength, 8–2763
Connect, 9–3188
ConnectedKernel, 11–3762
ConnectingHomomorphism, 4–1224
ConnectionNumber, 12–3996
ConnectionPolynomial, 13–4363
Consistency, 6–1908
ConstaCyclicCode, 13–4208
ConstantCoefficient, 2–390, 3–1018
ConstantField, 3–914, 9–2937
ConstantFieldExtension, 3–918, 9–2945, 9–2962

INDEX OF INTRINSICS

 xi
INDEX OF INTRINSICS
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>DifferentialRing</td>
<td>9-2934</td>
</tr>
<tr>
<td>DifferentialRingExtension</td>
<td>9-2947</td>
</tr>
<tr>
<td>DifferentialSpace</td>
<td>3-916, 3-986, 3-992, 993, 9-3149, 9-3167</td>
</tr>
<tr>
<td>Differentiation</td>
<td>3-956</td>
</tr>
<tr>
<td>DifferentiationSequence</td>
<td>3-956</td>
</tr>
<tr>
<td>Digraph</td>
<td>12-4030</td>
</tr>
<tr>
<td>DihedralForms</td>
<td>11-3616</td>
</tr>
<tr>
<td>DihedralGroup</td>
<td>5-1245, 5-1302, 5-1500, 6-1785, 6-1967</td>
</tr>
<tr>
<td>DihedralSubspace</td>
<td>11-3610</td>
</tr>
<tr>
<td>Dilog</td>
<td>2-461</td>
</tr>
<tr>
<td>Dimension</td>
<td>2-557, 2-560, 3-585, 3-630, 3-980, 4-1198, 4-1210, 5-1707, 7-2164, 7-2195, 7-2238, 7-2308, 8-2338, 8-2367, 8-2402, 8-2442, 8-2452, 8-2474, 8-2499, 8-2504, 8-2597, 9-2812, 9-2852, 9-3020, 9-3074, 9-3167, 9-3207, 9-3210, 9-3213, 10-3442, 10-3479, 11-3607, 11-3687, 11-3702, 11-3727, 11-3752, 11-3784, 13-4181, 13-4304, 13-4350</td>
</tr>
<tr>
<td>DimensionByFormula</td>
<td>11-3607, 3608</td>
</tr>
<tr>
<td>DimensionCuspForms</td>
<td>11-3678</td>
</tr>
<tr>
<td>DimensionCuspFormsGamma0</td>
<td>11-3678</td>
</tr>
<tr>
<td>DimensionCuspFormsGamma1</td>
<td>11-3678</td>
</tr>
<tr>
<td>DimensionNewCuspFormsGamma0</td>
<td>11-3678</td>
</tr>
<tr>
<td>DimensionNewCuspFormsGamma1</td>
<td>11-3678</td>
</tr>
<tr>
<td>DimensionOfCentreOfEndomorphismRing</td>
<td>3-641</td>
</tr>
<tr>
<td>DimensionOfEndomorphismRing</td>
<td>3-640</td>
</tr>
<tr>
<td>DimensionOfExactConstantField</td>
<td>3-920</td>
</tr>
<tr>
<td>DimensionOfFieldOfGeometricIrreducibility</td>
<td>9-3145</td>
</tr>
<tr>
<td>DimensionOfHomology</td>
<td>4-1215</td>
</tr>
<tr>
<td>DimensionsOfHomology</td>
<td>4-1215</td>
</tr>
<tr>
<td>DimensionsOfInjectiveModules</td>
<td>8-2499</td>
</tr>
<tr>
<td>DimensionsOfProjectiveModules</td>
<td>8-2499</td>
</tr>
<tr>
<td>DimensionsOfTerms</td>
<td>4-1215</td>
</tr>
<tr>
<td>DirectProduct</td>
<td>5-1247, 5-1304, 5-1412, 5-1509, 1510, 6-1786, 6-1966, 6-2101, 7-2256, 7-2321, 9-2996, 9-3103, 11-3790, 13-4216, 13-4290, 13-4319</td>
</tr>
<tr>
<td>DirectSum</td>
<td>3-591, 4-1170, 4-1215, 6-1748, 7-2174, 7-2216, 8-2349, 8-2420, 8-2452, 8-2465, 8-2467, 8-2602, 8-2693, 11-3790, 13-4216, 13-4289, 13-4319, 13-4343</td>
</tr>
<tr>
<td>DirectSumDecomposition</td>
<td>7-2175, 7-2217, 8-2602, 8-2755</td>
</tr>
<tr>
<td>DirichletCharacter</td>
<td>11-3608, 11-3727</td>
</tr>
<tr>
<td>DirichletCharacters</td>
<td>11-3608, 11-3728</td>
</tr>
<tr>
<td>DirichletGroup</td>
<td>2-332</td>
</tr>
<tr>
<td>Disconnect</td>
<td>9-3188</td>
</tr>
<tr>
<td>Discriminant</td>
<td>2-346, 2-403, 2-436, 3-656, 3-710, 3-778, 3-784, 3-847, 3-920, 4-1080, 8-2367, 8-2549, 8-2557, 10-3239, 10-3273, 10-3411, 10-3426, 11-3686, 11-3783</td>
</tr>
<tr>
<td>DiscriminantDivisor</td>
<td>3-1011</td>
</tr>
<tr>
<td>DiscriminantOfHeckeAlgebra</td>
<td>11-3656</td>
</tr>
<tr>
<td>DiscToPlane</td>
<td>11-3577</td>
</tr>
<tr>
<td>Display</td>
<td>6-1909</td>
</tr>
<tr>
<td>DisplayBurnsideMatrix</td>
<td>5-1533</td>
</tr>
<tr>
<td>DisplayFareySymbolDomain</td>
<td>11-3555</td>
</tr>
<tr>
<td>DisplayPolygons</td>
<td>11-3553</td>
</tr>
<tr>
<td>Distance</td>
<td>2-454, 4-1104, 11-3550, 11-3576, 12-4067, 12-4144, 13-4187, 13-4293, 13-4307</td>
</tr>
<tr>
<td>DistanceMatrix</td>
<td>12-4069</td>
</tr>
<tr>
<td>DistancePartition</td>
<td>12-4068</td>
</tr>
<tr>
<td>Distances</td>
<td>12-4144</td>
</tr>
<tr>
<td>DistinctDegreeFactorization</td>
<td>2-403</td>
</tr>
<tr>
<td>DistinctExtensions</td>
<td>5-1717</td>
</tr>
<tr>
<td>DistinguishedOrbitsOnSimples</td>
<td>7-2195</td>
</tr>
<tr>
<td>div</td>
<td>2-279, 2-419, 8-2387, 9-2877</td>
</tr>
<tr>
<td>DivideIntegers</td>
<td>11-3758</td>
</tr>
<tr>
<td>DivisionPoints</td>
<td>10-3283</td>
</tr>
<tr>
<td>DivisionPolynomial</td>
<td>10-3289</td>
</tr>
<tr>
<td>Divisor</td>
<td>3-768, 3-952, 3-965, 3-975, 3-995, 9-3150, 9-3158-3160, 13-4251</td>
</tr>
<tr>
<td>DivisorGroup</td>
<td>3-767, 3-916, 3-971, 3-975, 9-3158</td>
</tr>
<tr>
<td>DivisorIdeal</td>
<td>8-2400, 9-2849</td>
</tr>
<tr>
<td>DivisorMap</td>
<td>9-3168</td>
</tr>
<tr>
<td>DivisorOfDegreeOne</td>
<td>3-976, 9-3147</td>
</tr>
<tr>
<td>Divisors</td>
<td>2-300, 2-302, 3-729, 3-759</td>
</tr>
<tr>
<td>DivisorSigma</td>
<td>2-285</td>
</tr>
<tr>
<td>Dodecocode</td>
<td>13-4330</td>
</tr>
<tr>
<td>Domain</td>
<td>1-244, 1-246, 2-562, 4-1186, 5-1300, 5-1410, 6-1790, 6-2037, 7-2323, 8-2509, 9-3043, 10-3438, 11-3772, 11-3783</td>
</tr>
<tr>
<td>DominantCharacter</td>
<td>8-2745</td>
</tr>
<tr>
<td>DominantLSPath</td>
<td>8-2647</td>
</tr>
<tr>
<td>DominantWeight</td>
<td>7-2214, 7-2251, 7-2288, 7-2320</td>
</tr>
<tr>
<td>Double</td>
<td>10-3476</td>
</tr>
<tr>
<td>DoubleCoset</td>
<td>5-1367, 6-1853</td>
</tr>
<tr>
<td>DoubleCosetRepresentatives</td>
<td>5-1367</td>
</tr>
<tr>
<td>DoubleCosets</td>
<td>6-1853</td>
</tr>
<tr>
<td>DoubleGenusOneModel</td>
<td>10-3408</td>
</tr>
<tr>
<td>DoublyCirculantQRCode</td>
<td>13-4213</td>
</tr>
<tr>
<td>DoublyCirculantQRCodeGF4</td>
<td>13-4213</td>
</tr>
<tr>
<td>Dual</td>
<td>3-590, 4-1203, 4-1214, 6-1762, 7-2175, 7-2218, 7-2256, 7-2282, 7-2321, 8-2514, 8-2694, 11-3805, 12-3863, 12-3987, 13-4182, 13-4192, 13-4288, 13-4305, 13-4310</td>
</tr>
<tr>
<td>DualAtkinLehner</td>
<td>11-3653</td>
</tr>
</tbody>
</table>
INDEX OF INTRINSICS

FactoredIndex, 5–1253, 5–1321, 5–1431, 5–1528, 6–1758, 6–1819, 6–1971
FactoredInverseDefiningPolynomials, 9–3044
FactoredMCPPolynomials, 2–510
FactoredMinimalAndCharacteristicPolynomials, 2–510
FactoredMinimalPolynomial, 2–509
FactoredModulus, 2–325
FactoredProjectiveOrder, 2–517, 5–1418, 8–2472
Factorial, 2–287, 12–3913
FactorisationOverSplittingField, 2–366
FactorisationToInteger, 2–276
FactorizationOverSplittingField, 2–366
FactorizationToInteger, 2–276, 2–302
Falphi, 8–2646, 2647
FaltingsHeight, 10–3293
Fano, 9–3223, 3224
FanoBaseGenus, 9–3223
FanoDatabase, 9–3224
FanoGenus, 9–3223
FanoIndex, 9–3223
FareySymbol, 11–3552
FewGenerators, 5–1296
Fibonacci, 2–288, 12–3913
Field, 12–3863, 13–4181, 13–4303, 13–4350
FieldAutomorphism, 7–2323
FieldMorphism, 3–933
FieldOfDefinition, 11–3728, 11–3772, 11–3818, 11–3829
FieldOfGeometricIrreducibility, 9–3145
FindCommonEmbeddings, 11–3792
FindDependencies, 2–312
FindFirstGenerators, 9–3204
FindGenerators, 3–862
FindN, 9–3224
FindRelations, 2–311
FindRelationsInCWIFormat, 2–315
FindWord, 11–3543
FineEquidimensionalDecomposition, 9–2822
FiniteAffinePlane, 12–3853, 3854, 12–3866, 12–3884
FiniteField, 2–354, 355
FiniteProjectivePlane, 12–3853, 3854, 12–3884
FireCode, 13–4212
FirstIndexOfColumn, 12–3938
FirstIndexOfRow, 12–3937
FirstWeights, 9–3214
FittingGroup, 5–1538, 6–1981
FittingLength, 6–1981
FittingSeries, 6–1981
FittingSubgroup, 5–1263, 5–1353, 5–1538, 6–1981
Fix, 5–1338, 8–2695, 13–4240
FixedArc, 11–3550
FixedField, 3–798, 3–846
FixedGroup, 3–798
FixedPoints, 11–3549, 11–3577
Flat, 1–209, 1–211, 1–229, 3–728, 3–947
Flexes, 9–3123
FlipCoordinates, 9–3128
Floor, 2–282, 2–349, 2–452
Flow, 12–4164
Flush, 1–75
FormalGroupHomomorphism, 10–3354
FormalGroupLaw, 10–3354
FormalLog, 10–3354
FormalPoint, 9–3118
FormalSet, 1–169
Format, 1–235
FormType, 5–1598
forward, 1–41
FourCoverPullback, 10–3308
FourDescent, 10–3306
FourToTwoCovering, 10–3409
FPAlgebra, 8–2399
FPGroupStrong, 5–1370, 5–1456, 6–1781
FPQuotient, 5–1370
FrattiniSubgroup, 5–1263, 5–1353, 5–1468, 5–1538, 6–1756
FreeAbelianGroup, 6–1733, 6–1967
FreeAbelianQuotient, 6–1751, 6–1984
FreeAlgebra, 8–2384, 8–2409
FreeGroup, 6–1770
FreeLieAlgebra, 8–2581
FreeMonoid, 6–2095
FreeNilpotentGroup, 6–1967
FreeProduct, 6–1786, 6–2101
FreeResolution, 9–2868, 9–2884, 9–2908
FreeSemigroup, 6–2095
Frobenius, 2–369, 370, 10–3393, 10–3436, 10–3450, 12–3967
FrobeniusActionOnPoints, 10–3393
FrobeniusActionOnReducibleFiber, 10–3393
FrobeniusActionOnTrivialLattice, 10–3393
FrobeniusAutomorphism, 3–850
INDEX OF INTRINSICS

GreatestCommonDivisor, 2-284, 2-329, 2-396, 2-430, 3-757, 3-781, 3-976, 4-1035, 4-1097, 6-2024, 9-3163
GreatestCommonLeftDivisor, 9-2965
GreatestCommonRightDivisor, 9-2965
GriesmerBound, 13-4228
GriesmerLengthBound, 13-4230
GriesmerMinimumWeightBound, 13-4229
Groebner, 8-2392, 9-2779, 9-2880
GroebnerBasis, 8-2392, 9-2782, 2783, 9-2829, 9-3007
GroebnerBasisUnreduced, 9-2782
GroundField, 2-357, 3-701
Group, 5-1296, 5-1336
GroupIdeal, 9-2917, 9-2923
GroupOfLieType, 7-2153, 7-2224, 7-2261, 7-2289, 7-2290-2301
GroupOfLieTypeFactorizedOrder, 7-2197
GroupOfLieTypeHomomorphism, 7-2224, 7-2325
GroupOfLieTypeOrder, 7-2197
GrowthFunction, 6-2079
GRSCode, 13-4214
GSet, 5-1296, 5-1336
GSetFromIndexed, 5-1335
gt, 1-63, 1-203, 2-264, 2-281, 2-306, 2-348, 2-388, 2-450, 3-977, 6-1774, 6-2097, 9-3162, 11-3690
GuessAltsymDegree, 5-1380, 5-1592
H2.G.A, 3-845
H2.G.QmodZ, 6-1762
HadamardAutomorphismGroup, 12-4016
HadamardCanonicalForm, 12-4013
HadamardColumnDesign, 12-4015
HadamardDatabase, 12-4016
HadamardDatabaseInformation, 12-4018
HadamardDatabaseInformationEmpty, 12-4018
HadamardGraph, 12-4053
HadamardInvariant, 12-4013
HadamardMatrixFromInteger, 12-4014
HadamardMatrixToInteger, 12-4014
HadamardNormalize, 12-4013
HadamardRowDesign, 12-4015
HadamardTransformation, 13-4357
HalfIntegralWeightForms, 11-3597
HallSubgroup, 12-1531
HamiltonianLieAlgebra, 8-2593
HammingAsymptoticBound, 13-4230
HammingCode, 13-4180
HammingWeightEnumerator, 13-4286
HarmonicNumber, 12-3914
HasAdditionAlgorithm, 10-3457
HasAffinePatch, 9-3025
HasAllPQuotientsMetacyclic (G), 5-1648
HasAllPQuotientsMetacyclic (G, p), 5-1648
HasAttribute, 2-359, 4-1127, 5-1464, 5-1466, 5-1544, 5-1697, 9-2875
HasClique, 12-4073
HasClosedCosetTable, 6-1895
HasCM, 11-3801
HasComplement, 5-1365, 6-1759, 8-2438
HasCompleteCosetTable, 6-1895
HasComplexConjugate, 3-718
HasComplexMultiplication, 10-3292, 10-3336
HasComputableAbelianQuotient, 6-1802
HasComputableLCS, 6-1981
HasDefinedModuleMap, 4-1221
HasDefiningMap, 4-1079
HasDenseAndSparseRep, 12-4037
HasDenseRep, 12-4037
HasDenseRepOnly, 12-4037
HasElementaryBasis, 12-3961
HasEmbedding, 8-2552
HasFiniteDimension, 9-2852
HasFiniteKernel, 11-3774
HasFiniteOrder, 2-517, 5-1417
HasFunctionField, 9-2997, 9-3143
HasGCD, 2-260
HasGNB, 4-1074
HasGrevlexOrder, 9-2797
HasGrocberBasis, 9-2784
Hash, 1-174
HasHomogeneousBasis, 12-3961
HasInfiniteComputableAbelianQuotient, 6-1802
HasIntersectionProperty, 12-3904
HasIntersectionPropertyN, 12-3904
HasInverse, 3-933
HasIrregularFibres, 9-3192
HasIsometricEmbedding, 3-627
HasKnownInverse, 9-3038
HasLeviSubalgebra, 8-2607
HasLinearGrayMapImage, 13-4275
HasMonomialBasis, 12-3961
HasMultiplicityOne, 11-3734
HasNegativeWeightCycle, 12-4145
HasNonsingularPoint, 9-3015
HasOddDegreeModel, 10-3422
HasOnlyOrdinarySingularities, 9-3114
HasOnlyOrdinarySingularitiesMonteCarlo, 9-3114
HasOrder, 10-3449
HasOutputFile, 1-74
HasParallelClass, 12-4000
HasParallelism, 12-3999
HasPlace, 3-937, 3-978, 9-3154
HasPointsOverExtension, 9-3017
HasPolynomial, 4-1048
INDEX OF INTRINSICS

HasPolynomialFactorization, 2–400
HasPowerSumBasis, 12–3961
HasPreimage, 1–245
HasPRoot, 4–1080
HasRandomPlace, 3–937, 3–970
HasRationalPoint, 10–3243
HasRationalSolutions, 9–2972
HasResolution, 12–3999
HasRoot, 2–392, 4–1062, 4–1101
HasRootOfUnity, 4–1080
HasSchurBasis, 12–3961
HasseWittInvariant, 3–941, 3–991, 9–3166
HasSingularPointsOverExtension, 9–3123
HasSparseRep, 12–4037
HasSparseRepOnly, 12–4037
HasSquareSha, 10–3464
HasSupplement, 5–1365
HasTwistedHopfStructure, 8–2643
HasValidCosetTable, 6–1895
HasValidIndex, 6–1897
HasWeakIntersectionProperty, 12–3904
HBinomial, 8–2658
HeckeAlgebra, 11–3656, 11–3777
HeckeBound, 11–3656
HeckeEigenvalueField, 11–3657
HeckeEigenvalueRing, 11–3657
HeckeOperator, 11–3611, 11–3652, 11–3691, 11–3707, 11–3836
HeckePolynomial, 11–3612, 11–3653, 11–3837
HeegnerDiscriminants, 10–3318
HeegnerForms, 10–3318, 3319
HeegnerPoint, 10–3316, 3317
HeegnerPoints, 10–3319
HeegnerTorsionElement, 10–3319
Height, 2–349, 10–3298, 10–3337, 10–3386, 10–3460
HeightConstant, 10–3460
HeightOnAmbient, 9–3027
HeightPairing, 10–3299, 10–3386, 10–3461
HeightPairingLattice, 10–3386
HeightPairingMatrix, 10–3299, 10–3337, 10–3386, 10–3461
HenselLift, 2–404, 2–459, 4–1099, 4–1102, 4–1139
HermiteForm, 2–514, 4–1210, 8–2478
HermitePolynomial, 2–408
HermitianCode, 13–4248
HermitianFunctionField, 3–906
HesseCovariants, 10–3411
HesseModel, 10–3403
HessenbergForm, 2–512, 8–2472
HessePolynomials, 10–3412
Hessian, 10–3411
HessianMatrix, 9–3008, 9–3110
Hexacode, 13–4330
HighestCoroot, 7–2168, 7–2205
HighestLongCoroot, 7–2168, 7–2205
HighestRoot, 7–2168, 7–2205, 7–2247, 7–2318
HighestWeightModule, 8–2614, 8–2640, 8–2740
HighestWeightRepresentation, 7–2329, 8–2613, 8–2640, 8–2739, 2740
HighestWeights, 8–2758
HighestWeightsAndVectors, 8–2641, 8–2755
HighestWeightVectors, 8–2758
HighMap, 8–2530
HighProduct, 8–2530
Hilbert90, 2–370, 3–824
HilbertClassField, 3–842, 3–1008
HilbertClassPolynomial, 3–664, 11–3533
HilbertFunction, 9–3022
HilbertGroebnerBasis, 9–2835
HilbertIdeal, 9–2918
HilbertNumerator, 9–3203, 9–3213
HilbertPolynomial, 9–2833
HilbertPolynomialOfCurve, 9–3215
HilbertSeries, 9–2832, 9–2882, 9–2908, 9–3202, 9–3214
HilbertSeriesApproximation, 9–2908
HilbertSeriesMultipliedByMinimalDenominator, 9–3203
HilbertSpace, 13–4350
HilbertSymbol, 8–2550, 10–3242
HirschNumber, 6–1971
Holes, 3–617
Holomorph, 5–1700
Hom, 2–544, 4–1178, 1179, 4–1206, 6–1760, 8–2445, 9–2870, 11–3776
hom<>, 2–342, 2–445
HomGenerators, 5–1560, 6–1760
HomogeneousComponent, 9–2829
HomogeneousComponents, 9–2829
HomogeneousComponentTest, 9–2840, 9–2911
HomogeneousModuleTestBasis, 9–2841
HomogeneousToElementaryMatrix, 12–3973
HomogeneousToMonomialMatrix, 12–3973
HomogeneousToPowerSumMatrix, 12–3973
HomogeneousToSchurMatrix, 12–3973
Homogenization, 9–2811
HomologicalDimension, 9–2884, 9–2909
Homology, 4–1215, 11–3751
IntegralUEA, 8–2656
IntegralUEA, 8–2656
IntegralUniversalEnvelopingAlgebra, 8–2656
Interior, 12–3873
InternalEdges, 11–3552
Interpolation, 2–394, 2–428, 2–480
Intersection, 9–3003, 9–3077, 11–3541, 11–3793
IntersectionArray, 12–4092
IntersectionGroup, 11–3667
IntersectionMatrix, 12–4069
IntersectionNumber, 9–3122, 12–3993
IntersectionOfImages, 11–3793
IntersectionPairing, 11–3657, 11–3807
IntersectionPairingIntegral, 11–3807
IntersectionWithNormalSubgroup, 5–1322
IntersectKernels, 6–1937
intrinsic, 1–43
InverseJeuDeTaquin, 7
InverseRSKCorrespondenceSingleWord, 12–3945
InverseRSKCorrespondenceDoubleWord, 12–3945
InverseRSKCorrespondenceMatrix, 12–3946
InverseRSKCorrespondenceSingleWord, 12–3945
InverseSqrt, 4–1096
InverseSquareRoot, 4–1096
InverseWordMap, 5–1371, 5–1457
Involution, 8–2674, 10–3434
Iroot, 2–282
IrreducibleCartanMatrix, 7–2146
IrreducibleCoxeterGraph, 7–2146
IrreducibleCoxeterGroup, 7–2230
IrreducibleCoxeterMatrix, 7–2146
IrreducibleDynkinDigraph, 7–2146
IrreducibleLowTermGF2Polynomial, 2–372
IrreducibleMatrixGroup, 5–1671
IrreducibleModule, 8–2504
IrreducibleModules, 8–2696
IrreducibleModulesBurnside, 8–2699
IrreducibleModulesInit, 8–2702
IrreducibleModulesSchur, 5–1558, 8–2701
IrreduciblePolynomial, 2–372
IrreducibleReflectionGroup, 7–2268
IrreducibleRepresentationsInit, 8–2702
IrreducibleRepresentationsSchur, 5–1558
IrreducibleRootDatum, 7–2189
IrreducibleRootSystem, 7–2162
IrreducibleSecondaryInvariants, 9–2901
IrreducibleSolubleSubgroups, 5–1623
IrreducibleSparseGF2Polynomial, 2–372
IrreducibleSubgroups, 5–1623
IrregularLDPCEnsemble, 13–4255
Is2T1, 12–3905
IsA, 1–28
ISABaseField, 3–915
IsAbelianVariety, 11–3734
IsAbsoluteField, 3–719
IsAbsolutelyIrreducible, 5–1451, 7–2199, 8–2432, 9–3145
IsAbsoluteOrder, 3–718, 3–943
IsAdditiveOrder, 5–1468
IsAffine, 7–2173, 7–2212
IsAffineLinear, 9–3005, 3006
IsAffineLinear, 9–3045
IsAlgebraic, 7–2324
IsAlgebraicallyDependent, 2–420
IsAlgebraicallyIsomorphic, 7–2307
IsAlgebraicDifferentialField, 9–2940
IsAlgebraicField, 3–717
IsAlgebraicGeometric, 13–4250
IsAlternating, 5–1377
IsAltsym, 5–1378
IsAmbient, 9–3005, 11–3687
IsAmbientSpace, 11–3605, 11–3704
IsAnalyticallyIrreducible, 9–3119
IsAnisotropic, 7–2201
IsArc, 12–3872
IsAssociate, 8–2349
IsAttachedToModularSymbols, 11–3734, 11–3754
IsAttachedToNewform, 11–3734
IsAutomaticGroup, 6–2066, 2067
IsAutomorphism, 9–3052
IsBalanced, 12–3998
IsBasePointFree, 9–3073
IsBiconnected, 12–4060, 12–4135
IsBijective, 4–1187
IsBipartite, 12–4056, 12–4131
IsBlock, 5–1346, 12–3996
IsBlockTransitive, 12–4008
IsBogomolovUnstable, 9–3223
IsBoundary, 4–1048
IsCanonical, 3–977, 3–992, 9–3163, 9–3208, 9–3210, 3211
INDEX OF INTRINSICS

IsCapacitated, 12-4115
IsCartanEquivalent, 7-2139, 7-2147, 
7-2163, 7-2192, 7-2236, 7-2277, 
7-2307
IsCartanMatrix, 7-2137
IsCentral, 3-845, 5-1261, 5-1321, 5-1431, 
5-1529, 6-1978, 7-2315, 8-2607, 2608
IsCentralCollineation, 12-3882
IsChainMap, 4-1221
IsCharacter, 8-2711
IsClassicalType, 8-2607
IsCluster, 9-3005
IsCM, 11-3801
IsCoercible, 1-13, 9-3014, 12-3958
IsCohenMacaulay, 9-2908
IsCollinear, 12-3871
IsCommutative, 2-259, 2-278, 2-326, 
2-346, 2-365, 2-388, 2-418, 2-449, 
3-717, 3-876, 3-892, 4-1119, 4-1130, 
8-2348, 8-2401, 11-3787, 12-3961
IsCompactHyperbolic, 7-2241
IsComplete, 1-201, 6-1850, 9-3073, 
12-3872, 12-3998, 12-4056, 12-4132
IsComplex, 3-770
IsConcurrent, 12-3871
IsConditioned, 5-1565
IsConfluent, 6-2055, 6-2114
IsCongruence, 11-3542
IsConic, 9-3005, 10-3234
IsConjugate, 5-1261, 5-1268, 5-1314, 
5-1339, 5-1428, 5-1517, 5-1521, 
5-1529, 6-1842, 6-1978, 6-1984, 
6-2021, 8-2566
IsConnected, 12-4060, 12-4135
IsConsistent, 2-504, 5-1504, 6-1963, 
8-2484
IsConstant, 3-949, 10-3353
IsConstantCurve, 10-3386
IsConway, 2-365
IsCotangentSpace, 7-2202
IsCoxeterAffine, 7-2144
IsCoxeterCompactHyperbolic, 7-2150, 2151
IsCoxeterFinite, 7-2144
IsCoxeterGraph, 7-2135
IsCoxeterHyperbolic, 7-2150, 2151
IsCoxeterIrreducible, 7-2134, 7-2140
IsCoxeterIsomorphic, 7-2134, 7-2138, 
7-2147, 7-2236, 7-2277
IsCoxeterMatrix, 7-2133
IsCrystallographic, 7-2140, 7-2165, 
7-2199, 7-2241, 7-2281
IsCurve, 9-3005, 9-3106
IsCusp, 9-3119, 11-3548
IsCuspidal, 11-3605, 11-3649, 11-3690
IsCyclic, 3-719, 5-1261, 5-1298, 5-1424, 
5-1506, 6-1757, 6-1977, 13-4194, 
13-4295
IsDecomposable, 8-2438
IsDefault, 2-365
IsDeficient, 10-3464
IsDefined, 1-201, 1-219, 1-223
IsDefinite, 8-2554
IsDenselyRepresented, 13-4350
IsDesarguesian, 12-3865
IsDesign, 12-3998
IsDiagonal, 2-506, 8-2470
IsDifferenceSet, 12-3991
IsDifferentialField, 9-2939
IsDifferentialIdeal, 9-2951
IsDifferentialLaurentSeriesRing, 9-2940
IsDifferentialOperatorRing, 9-2955
IsDifferentialSeriesRing, 9-2940
IsDirected, 12-4132
IsDirectSummand, 8-2438
IsDiscriminant, 3-656
IsDisjoint, 1-178
IsDistanceRegular, 12-4091
IsDistanceTransitive, 12-4091
IsDivisibleBy, 2-279, 2-394, 2-420, 
3-949, 10-3283
IsDivisionRing, 2-259, 2-278, 2-326, 
2-346, 2-366, 2-388, 2-418, 2-449, 
3-876, 3-892, 3-942, 4-1119, 4-1130, 
12-3961
IsDomain, 2-260, 2-278, 2-326, 2-346, 
2-366, 2-388, 2-418, 2-449, 3-718, 
3-876, 3-892, 3-942, 4-1119, 4-1130, 
9-2939, 12-3961
IsDominant, 7-2214, 7-2251, 7-2288, 
9-3045
IsDoublePoint, 9-3119
IsDoublyEven, 13-4195
IsDualComputable, 11-3805
IsDynkinDigraph, 7-2141
IsEdgeCapacitated, 12-4115
IsEdgeLabelled, 12-4115
IsEdgeTransitive, 12-4091
IsEdgeWeighted, 12-4116
IsEffective, 3-977, 9-3161, 9-3215
IsEisenstein, 4-1075, 11-3605, 11-3650, 
11-3690
IsEisensteinSeries, 11-3606, 11-3614
IsElementaryAbelian, 5-1261, 5-1298, 
5-1424, 5-1506, 6-1757, 6-1977
IsEllipticCurve, 10-3266, 10-3426
IsEllipticWeierstrass, 9-3125
IsEmpty, 1-177, 1-202, 1-218, 3-615, 
5-1561, 5-1642, 5-1661, 5-1666, 
5-1675, 6-1794, 6-1833, 6-2031, 
9-3020, 9-3028, 12-4057, 12-4132
IsEmptySimpleQuotientProcess, 6-1798
IsEmptyWord, 6-2020
IsEndomorphism, 9-3052, 11-3774
IsEof, 1-75
IsEquationOrder, 3-718, 3-943
INDEX OF INTRINSICS

IsInt, 3–806
IsInTangentVariety, 9–3062
IsInteger, 11–3774
IsIntegral, 2–280, 2–348, 2–450, 3–587,
3–722, 3–754, 3–961, 4–1090, 10–3287,
10–3424
IsIntegralDomain, 2–260
IsIntegralModel, 10–3268, 3269
IsInterior, 4–1048
IsIntersection, 9–3122
IsIntrinsic, 1–32
IsIsomorphism, 7–2302
IsInvariant, 3–809, 9–2891
IsInvertible, 9–3038
IsIrreducible, 2–263, 2–280, 2–327,
2–348, 2–368, 2–390, 2–403, 2–420,
2–433, 2–450, 3–722, 3–878, 3–949,
4–1103, 4–1131, 5–1450, 7–2165,
7–2199, 7–2241, 7–2310, 8–2387,
8–2432, 8–2712, 9–3021, 9–3111,
11–3647
IsIrregularSingularPlace, 9–2970
IsIsogenous, 7–2192, 7–2307, 10–3275,
10–3483, 11–3734
IsIsogenousPeriodMatrices, 10–3483
IsIsogeny, 7–2221, 11–3774
IsIsolated, 9–3208, 9–3211
IsIsometric, 3–626, 627
IsIsomorphic, 3–626, 627, 3–719, 3–931,
4–1104, 4–1107, 5–1371, 5–1459,
5–1549, 7–2163, 7–2192, 7–2236,
8–2379, 8–2448, 8–2566, 2567, 9–3132,
10–3275, 10–3439, 10–3482, 11–3735,
12–3865, 12–4004, 12–4084, 13–4243
IsIsomorphicBigPeriodMatrices, 10–3482
IsIsomorphicOverQ, 3–930
IsIsomorphicSmallPeriodMatrices, 10–3482
IsIsomorphism, 4–1221, 9–3045, 10–3348,
11–3774
IsKEdgeConnected, 12–4065, 12–4138
IsKnullEquivalent, 12–3923
IsKVertexConnected, 12–4065, 12–4137
IsLabelled, 12–4113, 12–4115
IsLargeReeGroup, 5–1615
IsLDPC, 13–4256
IsLE, 6–2021
IsLE, 6–2021
IsLeaf, 8–2582
IsLeftIdeal, 8–2376, 8–2672
IsLeftIsomorphic, 8–2379, 8–2567
IsLeftModule, 8–2452
IsLexicographicallyOrdered, 12–3944
IsLie, 8–2349
IsLinear, 8–2712, 9–3008, 9–3045
IsLinearGroup, 5–1601
IsLinearlyDependent, 10–3387
IsLinearlyEquivalent, 9–3164
IsLinearlyIndependent, 10–3288, 10–3301,
10–3387
IsLinearSpace, 12–3998
IsLineRegular, 12–3998
IsLineTransitive, 12–3883
IsLittlewoodRichardson, 12–3939
IsLocallySolvable, 9–3030
IsLocallyTwoTransitive, 12–3905
IsLocalNorm, 3–852, 853
IsLongRoot, 7–2172, 7–2211, 7–2250,
7–2319
IsLowerSymmetric, 2–506
IsMagmaEuclideanRing, 2–259
IsMatrixRing, 8–2554
IsMaximal, 3–718, 3–943, 5–1262, 5–1325,
5–1431, 5–1529, 6–1758, 6–1842,
8–2379, 8–2546, 9–2797
IsMaximisingFunction, 13–4377
IsMaximumDistanceSeparable, 13–4194
IsMDS, 13–4194
IsMemberBasicOrbit, 5–1386
IsNetCyclicPGroup (P), 5–1648
IsMinimalModel, 10–3269
IsMinusOne, 2–263, 2–280, 2–327, 2–348,
2–368, 2–390, 2–403, 2–420,
2–433, 2–450, 3–722, 3–878, 3–949,
4–1103, 4–1131, 5–1450, 7–2165,
7–2199, 7–2241, 7–2310, 8–2387,
8–2432, 8–2712, 9–3021, 9–3111,
11–3647
IsMixed, 6–1757
IsModularCurve, 9–3006
IsModuleHomomorphism, 8–2445, 8–2509
IsNmonic, 2–390, 9–2957
IsMorphism, 3–933, 11–3773
IsNearLinearSpace, 12–3998
IsNearlyPerfect, 13–4195
IsNeat, 6–1758
IsNegative, 7–2171, 7–2210, 7–2248
IsNegativeDefinite, 3–620
IsNegativeSemiDefinite, 3–621
IsNew, 11–3606, 11–3650
IsNewform, 11–3606
IsNilpotent, 2–263, 2–280, 2–327, 2–348,
2–368, 2–390, 2–420, 2–450, 3–722,
3–878, 3–894, 3–949, 4–1035, 4–1120,
4–1131, 4–1142, 4–1154, 8–2344, 8–2387,
8–2470, 8–2712, 9–3010, 12–3964
IsNodalCurve, 9–3111
IsNode, 9–3119
IsNonsingular, 9–3019, 3020, 9–3111,
9–3118
IsNorm, 3–853
IsNormal, 2–368, 3–719, 3–845, 4–1107,
5–1262, 5–1321, 5–1431, 5–1529,
6–1842, 6–1978
IsNormalised, 5–1723
IsNormalising, 7–2301
INDEX OF INTRINSICS

<table>
<thead>
<tr>
<th>Function</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsNull</td>
<td>1-177, 1-202, 12-4057, 12-4132</td>
</tr>
<tr>
<td>IsNullHomotopy</td>
<td>8-2531</td>
</tr>
<tr>
<td>IsNumberField</td>
<td>3-717</td>
</tr>
<tr>
<td>Iso</td>
<td>1-246, 10-3436</td>
</tr>
<tr>
<td>iso</td>
<td>6-1763, 9-3034</td>
</tr>
<tr>
<td>IsOdd</td>
<td>2-279, 2-303, 2-334</td>
</tr>
<tr>
<td>IsogeniousCurves</td>
<td>10-3293</td>
</tr>
<tr>
<td>Isogeny</td>
<td>11-3530</td>
</tr>
<tr>
<td>IsogenyFromKernel</td>
<td>10-3350</td>
</tr>
<tr>
<td>IsogenyFromKernelFactored</td>
<td>10-3350</td>
</tr>
<tr>
<td>IsogenyGroup</td>
<td>7-2198, 7-2239, 7-2279, 7-2310</td>
</tr>
<tr>
<td>IsogenyMapOmega</td>
<td>10-3352</td>
</tr>
<tr>
<td>IsogenyMapPhi</td>
<td>10-3352</td>
</tr>
<tr>
<td>IsogenyMapPhiMulti</td>
<td>10-3352</td>
</tr>
<tr>
<td>IsogenyMapPai</td>
<td>10-3351</td>
</tr>
<tr>
<td>IsogenyMapPaiMulti</td>
<td>10-3351</td>
</tr>
<tr>
<td>IsogenyMapPaiSquared</td>
<td>10-3351</td>
</tr>
<tr>
<td>IsolGroup</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolGroupDatabase</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolGroupOfDegreeFieldSatisfying</td>
<td>5-1674</td>
</tr>
<tr>
<td>IsolGroupOfDegreeSatisfying</td>
<td>5-1674</td>
</tr>
<tr>
<td>IsolGroupSatisfying</td>
<td>5-1673</td>
</tr>
<tr>
<td>IsolGroupsOfDegreeFieldSatisfying</td>
<td>5-1674</td>
</tr>
<tr>
<td>IsolGroupsOfDegreeSatisfying</td>
<td>5-1674</td>
</tr>
<tr>
<td>IsolGroupsSatisfying</td>
<td>5-1674</td>
</tr>
<tr>
<td>IsolGuardian</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolInfo</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolIsPrimitive</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolMinBlockSize</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolNumberOfDegreeField</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolOrder</td>
<td>5-1672</td>
</tr>
<tr>
<td>IsolProcess</td>
<td>5-1675</td>
</tr>
<tr>
<td>IsolProcessOfDegree</td>
<td>5-1675</td>
</tr>
<tr>
<td>IsolProcessOfDegreeField</td>
<td>5-1675</td>
</tr>
<tr>
<td>IsolProcessOfField</td>
<td>5-1675</td>
</tr>
<tr>
<td>IsometricCircle</td>
<td>11-3571</td>
</tr>
<tr>
<td>IsomorphicProjectionToSubspace</td>
<td>9-3065</td>
</tr>
<tr>
<td>Isomorphism</td>
<td>6-1763, 8-2567, 10-3347</td>
</tr>
<tr>
<td>IsomorphismData</td>
<td>10-3348</td>
</tr>
<tr>
<td>Isomorphisms</td>
<td>3-931, 932, 9-3132</td>
</tr>
<tr>
<td>IsomorphismToIsogeny</td>
<td>10-3348</td>
</tr>
<tr>
<td>IsOne</td>
<td>2-263, 2-280, 2-303, 2-327, 2-348, 2-368, 2-390, 2-420, 2-450, 2-506, 3-722, 3-754, 3-766, 3-877, 3-894, 3-949, 3-961, 4-1035, 4-1090, 4-1120, 4-1131, 4-1142, 4-1154, 6-2097, 8-2344, 8-2387, 8-2470, 8-2712, 9-2941, 9-2957, 9-3010, 12-3964</td>
</tr>
<tr>
<td>IsOneCoboundary</td>
<td>5-1710</td>
</tr>
<tr>
<td>IsOneCocycle</td>
<td>5-1724</td>
</tr>
<tr>
<td>IsOnlyMotivic</td>
<td>11-3735</td>
</tr>
<tr>
<td>IsOptimal</td>
<td>11-3774</td>
</tr>
<tr>
<td>IsOrbit</td>
<td>5-1344</td>
</tr>
<tr>
<td>IsOrder</td>
<td>10-3287</td>
</tr>
<tr>
<td>IsOrdered</td>
<td>2-259, 2-278, 2-326, 2-346, 2-365, 2-388, 2-418, 2-449, 3-717, 3-876, 3-892, 4-1119, 4-1130, 12-3961</td>
</tr>
<tr>
<td>IsOrderTerm</td>
<td>9-2941</td>
</tr>
<tr>
<td>IsOrdinary</td>
<td>10-3359</td>
</tr>
<tr>
<td>IsOrdinaryProjective</td>
<td>9-3007</td>
</tr>
<tr>
<td>IsOrdinaryProjectiveSpace</td>
<td>9-3005</td>
</tr>
<tr>
<td>IsOrdinarySingularity</td>
<td>9-3019, 9-3119</td>
</tr>
<tr>
<td>IsOrthogonalGroup</td>
<td>5-1601</td>
</tr>
<tr>
<td>IsotropicSubspace</td>
<td>2-518, 3-652</td>
</tr>
<tr>
<td>IsOuter</td>
<td>7-2201</td>
</tr>
<tr>
<td>IsOverQ</td>
<td>11-3788</td>
</tr>
<tr>
<td>IsOverSmallerField (G, k: -)</td>
<td>5-1486</td>
</tr>
<tr>
<td>IsOverSmallerField (G: -)</td>
<td>5-1486</td>
</tr>
<tr>
<td>IsParabolicSubgroup</td>
<td>7-2254</td>
</tr>
<tr>
<td>IsParallel</td>
<td>12-3871</td>
</tr>
<tr>
<td>IsParallelClass</td>
<td>12-4000</td>
</tr>
<tr>
<td>IsParallellism</td>
<td>12-4000</td>
</tr>
<tr>
<td>IsPartialRoot</td>
<td>4-1052</td>
</tr>
<tr>
<td>IsPartition</td>
<td>12-3919</td>
</tr>
<tr>
<td>IsPartitionRefined</td>
<td>12-4083</td>
</tr>
<tr>
<td>IsPath</td>
<td>12-4057</td>
</tr>
<tr>
<td>IsPerfect</td>
<td>3-919, 5-1262, 5-1298, 5-1424, 5-1506, 6-1803, 6-1977, 13-4195, 13-4311</td>
</tr>
<tr>
<td>IspGroup</td>
<td>6-1757</td>
</tr>
<tr>
<td>IspID</td>
<td>2-259, 2-278, 2-326, 2-346, 2-365, 2-388, 2-418, 2-449, 3-718, 3-876, 3-892, 3-942, 4-1119, 4-1130, 12-3961</td>
</tr>
<tr>
<td>IspIntegral</td>
<td>10-3424</td>
</tr>
<tr>
<td>IsPir</td>
<td>2-260</td>
</tr>
<tr>
<td>IsPlanar</td>
<td>9-3007, 12-4077, 12-4139</td>
</tr>
<tr>
<td>IsPlaneCurve</td>
<td>9-3005</td>
</tr>
<tr>
<td>IsPslAlgebra</td>
<td>8-2617</td>
</tr>
<tr>
<td>IsPSlGroup</td>
<td>8-2379, 8-2546</td>
</tr>
<tr>
<td>IspMaximal</td>
<td>10-3425</td>
</tr>
<tr>
<td>IspNormal</td>
<td>10-3424</td>
</tr>
<tr>
<td>IsPoint</td>
<td>4-1048, 10-3282, 10-3432, 10-3475</td>
</tr>
<tr>
<td>IsPointRegular</td>
<td>12-3998</td>
</tr>
<tr>
<td>IsPointTransitive</td>
<td>12-3883, 12-4008</td>
</tr>
<tr>
<td>IsPolygon</td>
<td>12-4057</td>
</tr>
<tr>
<td>IsPolynomial</td>
<td>9-3044</td>
</tr>
<tr>
<td>IsPositive</td>
<td>3-977, 7-2171, 7-2210, 7-2248, 9-3161</td>
</tr>
<tr>
<td>IsPositiveDefinite</td>
<td>3-620</td>
</tr>
<tr>
<td>IsPositiveSemiDefinite</td>
<td>3-620</td>
</tr>
<tr>
<td>IsPower</td>
<td>2-280, 2-371, 3-721, 722, 3-758, 3-870, 3-959, 4-1096</td>
</tr>
<tr>
<td>IsPRI</td>
<td>12-3904</td>
</tr>
<tr>
<td>IsPrimary</td>
<td>9-2796, 9-2850</td>
</tr>
<tr>
<td>IsPrime</td>
<td>2-263, 2-280, 2-290, 2-303, 2-327, 2-348, 2-368, 2-390, 2-390, 2-420, 2-450, 2-450, 3-722, 3-754, 3-766, 3-877, 3-894, 3-949, 3-961, 4-1035, 4-1090, 4-1120, 4-1131, 4-1142, 4-1154, 6-2097, 8-2344, 8-2387, 8-2470, 8-2712, 9-2941, 9-2957, 9-3010, 12-3964</td>
</tr>
<tr>
<td>IsPrimeCertificate</td>
<td>2-290</td>
</tr>
</tbody>
</table>
INDEX OF INTRINSICS

IsWeaklyAG, 13−4250
IsWeaklyAGDual, 13−4250
IsWeaklyConnected, 12−4061, 12−4136
IsWeaklyEqual, 4−1131, 4−1155, 9−2941, 9−2957
IsWeaklyMonic, 9−2957
IsWeaklyPrimitive, 12−3904
IsWeaklyPrimitive, 4−1131, 4−1155, 9−2941, 9−2957
IsWealyPrimitive, 12−3904
IsWeierstrassModel, 10−3268
IsWeierstrassPlace, 3−972, 3−985, 9−3157
IsWeighted, 12−4116
IsWildlyRamified, 3−718, 719, 3−755, 3−943, 3−962
IsWPRI, 12−3904
IsWreathProduct, 5−1290
IsZero, 2−263, 2−280, 2−327, 2−348, 2−368, 2−390, 2−420, 2−450, 2−506, 2−527, 2−548, 3−583, 3−722, 3−754, 3−766, 3−877, 3−894, 3−949, 3−961, 3−977, 3−994, 3−1018, 4−1035, 4−1099, 4−1120, 4−1131, 4−1142, 4−1154, 4−1174, 4−1221, 8−2342, 8−2344, 8−2373, 8−2387, 8−2397, 8−2401, 8−2428, 8−2470, 8−2647, 8−2712, 8−2796, 9−2850, 9−2865, 9−2867, 9−2877, 9−2879, 9−2941, 9−2957, 9−3010, 9−3150, 9−3163, 10−3286, 10−3353, 10−3448, 11−3774, 11−3820, 12−3964, 13−4188, 13−4295, 13−4308
IsZeroAt, 11−3841
IsZeroComplex, 4−1218
IsZeroDimensional, 9−2797
IsZeroDivisor, 2−263, 2−280, 2−327, 2−348, 2−368, 2−390, 2−420, 2−450, 3−722, 3−878, 3−894, 3−949, 4−1035, 4−1120, 4−1131, 8−2344, 8−2387
IsZeroMap, 4−1218
IsZeroTerm, 4−1218
Jacobi, 6−1910
Jacobi, 10−3315, 10−3408, 10−3442
JacobiIdeal, 9−2798, 9−3008, 9−3110
JacobiMatrix, 2−427, 9−3008, 9−3110
JacobiOrdersByDeformation, 10−3456
JacobiSymbol, 2−286
JacobiTheta, 2−471
JacobiThetaNullIK, 2−471
JacobsonRadical, 8−2340, 8−2362, 8−2436, 8−2442, 8−2504
JBessel, 2−477
JenningsLieAlgebra, 8−2618
JenningsSeries, 5−1263, 5−1353, 5−1468, 5−1541
JeuDeTaquin, 12−3941
jFunction, 11−3532
JH, 11−3720, 3721
jInvariant, 2−472, 3−663, 10−3273
jInvariants, 10−3428, 3429
JohnsonBound, 13−4228
join, 1−178, 7−2174, 7−2216, 9−3003, 12−4049, 12−4127, 4128
JOne, 11−3720
JordanForm, 2−511, 8−2479
jParameter, 11−3583
Js, 11−3720
JustesenCode, 13−4215
Justuxtaposition, 13−4219, 13−4320
JZero, 11−3719
K3Copy, 9−3216
K3Database, 9−3220
K3Surface, 9−3216, 9−3221, 3222, 9−3225, 3226
K3SurfaceRaw, 9−3226
K3SurfaceToRecord, 9−3225
kAarc, 12−3872
kBessel, 2−477
kBessel12, 2−477
KBinomial, 8−2638
KCubeGraph, 12−4034
KDegree, 8−2639
KeepAbelian, 6−1936
KeepElementary, 6−1936
KeepElementaryAbelian, 6−1936
KeepGeneratorAction, 6−1936
KeepGeneratorOrder, 6−1936
KeepPrimePower, 6−1935
KeepSplit, 6−1936
KeepSplitAbelian, 6−1936
KeepSplitElementaryAbelian, 6−1937
KerdockCode, 13−4276
Kernel, 1−244, 2−503, 2−531, 2−563, 4−1187, 4−1220, 5−1300, 5−1411, 5−1508, 6−1790, 8−2473, 8−2509, 8−2611, 8−2710, 10−3352, 11−3650, 11−3700, 11−3762, 12−3898
KernelMatrix, 2−503, 2−531
Kernels, 12−3898
Keys, 1−223
KillingMatrix, 8−2604
KissingNumber, 3−606
KLPolynomial, 8−2760
KMatrixSpace, 2−544, 2−557
KMatrixSpaceWithBasis, 4−1181
KModule, 2−544, 2−557
KModuleWithBasis, 2−560
Knot, 3−853, 12−3873
KnownAutomorphismSubgroup, 13−4201
KnownIrreducibles, 8−2707
KodairaSymbol, 10−3291
KodairaSymbols, 10−3291, 10−3385
KostkaNumber, 12−3949
KrawchoukPolynomial, 13−4238
KrawchoukTransform, 13−4238
<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LeadingCharacter, 2-333</td>
<td></td>
</tr>
<tr>
<td>LeadingProduct, 2-501</td>
<td></td>
</tr>
<tr>
<td>LeadingSymbol, 2-286</td>
<td></td>
</tr>
<tr>
<td>KSpace, 2-544, 545, 2-557, 3-708, 8-2499</td>
<td></td>
</tr>
<tr>
<td>KSpaceWithBasis, 2-560</td>
<td></td>
</tr>
<tr>
<td>KummerSurface, 10-3474</td>
<td></td>
</tr>
<tr>
<td>L, 9-2960, 11-3841</td>
<td></td>
</tr>
<tr>
<td>Label, 12-4113, 12-4116</td>
<td></td>
</tr>
<tr>
<td>Labelling, 5-1336</td>
<td></td>
</tr>
<tr>
<td>Labels, 11-3552, 12-4113, 12-4116</td>
<td></td>
</tr>
<tr>
<td>LaguerrePolynomial, 2-407, 408</td>
<td></td>
</tr>
<tr>
<td>Lang, 7-2315</td>
<td></td>
</tr>
<tr>
<td>Laplace, 4-1134</td>
<td></td>
</tr>
<tr>
<td>LargeReeElementToWord, 5-1615</td>
<td></td>
</tr>
<tr>
<td>LargeReeGroup, 5-1590</td>
<td></td>
</tr>
<tr>
<td>LargeReeSylow, 5-1621</td>
<td></td>
</tr>
<tr>
<td>LargestConductor, 10-3332</td>
<td></td>
</tr>
<tr>
<td>LargestDimension, 3-645, 5-1667, 5-1669</td>
<td></td>
</tr>
<tr>
<td>LastIndexOfColumn, 12-3938</td>
<td></td>
</tr>
<tr>
<td>LastIndexOfRow, 12-3937</td>
<td></td>
</tr>
<tr>
<td>Lattice, 3-573, 3-577, 578, 3-580, 3-635, 3-646, 3-663, 3-707, 3-761, 5-1668-1670, 11-3642, 11-3752, 11-3785, 11-3828</td>
<td></td>
</tr>
<tr>
<td>LatticeCoordinates, 11-3822</td>
<td></td>
</tr>
<tr>
<td>LatticeData, 3-646</td>
<td></td>
</tr>
<tr>
<td>LatticeDatabase, 3-644</td>
<td></td>
</tr>
<tr>
<td>LatticeName, 3-645</td>
<td></td>
</tr>
<tr>
<td>LatticeWithBasis, 3-574, 3-635</td>
<td></td>
</tr>
<tr>
<td>LatticeWithGram, 3-575, 3-635</td>
<td></td>
</tr>
<tr>
<td>LaurentSeriesRing, 4-1126</td>
<td></td>
</tr>
<tr>
<td>LayerBoundary, 5-1568</td>
<td></td>
</tr>
<tr>
<td>LayerLength, 5-1567</td>
<td></td>
</tr>
<tr>
<td>LazyPowerSeriesRing, 4-1146</td>
<td></td>
</tr>
<tr>
<td>LazySeries, 4-1148</td>
<td></td>
</tr>
<tr>
<td>LCFRequired, 10-3510</td>
<td></td>
</tr>
<tr>
<td>LCM, 2-285, 2-329, 2-397, 2-431, 3-757, 3-781, 3-977, 4-1120, 6-2025, 9-3163</td>
<td></td>
</tr>
<tr>
<td>lcm, 2-285, 2-302, 2-329, 2-397, 2-431, 3-757, 3-781, 3-963, 3-977, 6-2025, 9-3163</td>
<td></td>
</tr>
<tr>
<td>LDPCBinarySymmetricThreshold, 13-4261</td>
<td></td>
</tr>
<tr>
<td>LDPCCode, 13-4255</td>
<td></td>
</tr>
<tr>
<td>LDPCDecode, 13-4258</td>
<td></td>
</tr>
<tr>
<td>LDPCDensity, 13-4257</td>
<td></td>
</tr>
<tr>
<td>LDPCEnsembleRate, 13-4257</td>
<td></td>
</tr>
<tr>
<td>LDPCGaussianThreshold, 13-4262</td>
<td></td>
</tr>
<tr>
<td>LDPCGirth, 13-4257</td>
<td></td>
</tr>
<tr>
<td>LDPCMMatrix, 13-4257</td>
<td></td>
</tr>
<tr>
<td>LDPCSimulate, 13-4260</td>
<td></td>
</tr>
<tr>
<td>le, 1-63, 1-203, 2-264, 2-281, 2-306, 2-348, 2-388, 2-450, 3-977, 5-1278, 6-1774, 6-2021, 6-2097, 9-3162</td>
<td></td>
</tr>
<tr>
<td>LeadingCoefficient, 2-390, 2-421, 3-1018, 4-1097, 4-1132, 4-1152, 8-2387, 9-2957, 11-3843</td>
<td></td>
</tr>
<tr>
<td>LeadingExponent, 5-1566, 6-1957</td>
<td></td>
</tr>
<tr>
<td>LeadingGenerator, 5-1566, 6-1772, 6-1957</td>
<td></td>
</tr>
<tr>
<td>LeadingMonomial, 2-422, 8-2388</td>
<td></td>
</tr>
<tr>
<td>LeadingMonomialIdeal, 9-2795</td>
<td></td>
</tr>
<tr>
<td>LeadingTerm, 2-391, 2-423, 4-1132, 4-1152, 5-1565, 6-1957, 8-2388, 9-2958</td>
<td></td>
</tr>
<tr>
<td>LeadingTotalDegree, 2-425, 8-2389</td>
<td></td>
</tr>
<tr>
<td>LeadingWeightedDegree, 9-2828</td>
<td></td>
</tr>
<tr>
<td>LeastCommonLeftMultiple, 9-2966</td>
<td></td>
</tr>
<tr>
<td>LeastCommonMultiple, 2-285, 2-329, 2-397, 2-431, 3-757, 3-781, 3-977, 4-1097, 6-2025, 9-3163</td>
<td></td>
</tr>
<tr>
<td>LeeBrickellsAttack, 13-4224</td>
<td></td>
</tr>
<tr>
<td>LeeDistance, 13-4283</td>
<td></td>
</tr>
<tr>
<td>LeeWeight, 13-4187, 13-4283</td>
<td></td>
</tr>
<tr>
<td>LeeWeightDistribution, 13-4283</td>
<td></td>
</tr>
<tr>
<td>LeeWeightEnumerate, 13-4286</td>
<td></td>
</tr>
<tr>
<td>LeftAnnihilator, 8-2359, 8-2672</td>
<td></td>
</tr>
<tr>
<td>LeftConjugate, 6-2017</td>
<td></td>
</tr>
<tr>
<td>LeftCosetSpace, 6-1848, 6-1905</td>
<td></td>
</tr>
<tr>
<td>LeftDescentSet, 7-2244, 7-2280</td>
<td></td>
</tr>
<tr>
<td>LeftDiv, 6-2017</td>
<td></td>
</tr>
<tr>
<td>LeftExactExtension, 4-1216</td>
<td></td>
</tr>
<tr>
<td>LeftGCD, 6-2024</td>
<td></td>
</tr>
<tr>
<td>LeftGcd, 6-2024</td>
<td></td>
</tr>
<tr>
<td>LeftGreatestCommonDivisor, 6-2024</td>
<td></td>
</tr>
<tr>
<td>LeftIdeal, 8-2558</td>
<td></td>
</tr>
<tr>
<td>LeftIdealClasses, 8-2379, 8-2561</td>
<td></td>
</tr>
<tr>
<td>LeftInverse, 11-3809</td>
<td></td>
</tr>
<tr>
<td>LeftInverseMorphism, 11-3809</td>
<td></td>
</tr>
<tr>
<td>LeftIsomorphism, 8-2567</td>
<td></td>
</tr>
<tr>
<td>LeftLCM, 6-2025</td>
<td></td>
</tr>
<tr>
<td>LeftLcm, 6-2025</td>
<td></td>
</tr>
<tr>
<td>LeftLeastCommonMultiple, 6-2025</td>
<td></td>
</tr>
<tr>
<td>LeftMixedCanonicalForm, 6-2013</td>
<td></td>
</tr>
<tr>
<td>LeftNormalForm, 6-2013</td>
<td></td>
</tr>
<tr>
<td>LeftOrder, 8-2374, 8-2560</td>
<td></td>
</tr>
<tr>
<td>LeftRepresentationMatrix, 8-2373</td>
<td></td>
</tr>
<tr>
<td>LeftString, 7-2172, 7-2210, 7-2248</td>
<td></td>
</tr>
<tr>
<td>LeftStringLength, 7-2172, 7-2211, 7-2249</td>
<td></td>
</tr>
<tr>
<td>LeftZeroExtension, 4-1217</td>
<td></td>
</tr>
<tr>
<td>LegendreModel, 10-3240</td>
<td></td>
</tr>
<tr>
<td>LegendrePolynomial, 2-407, 10-3239</td>
<td></td>
</tr>
<tr>
<td>LegendresMethod, 10-3245</td>
<td></td>
</tr>
<tr>
<td>LegendreSymbol, 2-286</td>
<td></td>
</tr>
<tr>
<td>Length, 2-421, 3-583, 3-724, 3-1013, 4-1210, 5-1279, 7-2243, 7-2287, 8-2388, 9-2998, 12-3923, 12-3926, 12-3965, 13-4181, 13-4273, 13-4304</td>
<td></td>
</tr>
<tr>
<td>LengthenCode, 13-4216</td>
<td></td>
</tr>
<tr>
<td>Lengths, 9-2998</td>
<td></td>
</tr>
<tr>
<td>LeonsAttack, 13-4224</td>
<td></td>
</tr>
<tr>
<td>Level, 8-2557, 11-3526, 11-3542, 11-3608, 11-3686, 11-3702, 11-3728</td>
<td></td>
</tr>
<tr>
<td>LevenshteinBound, 13-4228</td>
<td></td>
</tr>
<tr>
<td>LexicographicalOrdering, 12-3944</td>
<td></td>
</tr>
<tr>
<td>LexProduct, 12-4050</td>
<td></td>
</tr>
<tr>
<td>LFSRSequence, 13-4363</td>
<td></td>
</tr>
</tbody>
</table>
INDEX OF INTRINSICS

LFSRStep, 13–4363
LFunction, 10–3391
LGetCoefficients, 10–3510
LHS, 6–1734, 6–1775, 6–2098
ldeal, 6–2100, 8–2337, 8–2374, 8–2391, 8–2464, 8–2558, 8–2668
LieAlgebra, 7–2153, 7–2176, 7–2224, 7–2260, 7–2289, 7–2329, 8–2336, 8–2359, 8–2577–2580, 8–2583, 8–2586–2589, 8–2740
LieAlgebraHomorphism, 7–2224
LieBracket, 8–2361
LieCharacteristic, 5–1593
LieConstant_C, 7–2223
LieConstant_epsilon, 7–2222
LieConstant_eta, 7–2223
LieConstant_M, 7–2223
LieConstant_N, 7–2222
LieConstant_p, 7–2222
LieConstant_q, 7–2222
LieEMaximalSubgroups, 8–2763
LieRepresentationDecomposition, 8–2737
LieType, 5–1594
Lift, 3–952, 3–973, 9–3157
LiftCharacter, 8–2717
LiftCharacters, 8–2717
LiftCoCycle, 5–1713
LiftDescendant, 10–3304
LiftHomomorphism, 8–2508
LiftMap, 9–2901
LiftNonSplitExtension, 6–1933
LiftNonSplitExtensionRow, 6–1933
LiftPoint, 9–3031
LiftSplitExtension, 6–1932
LiftSplitExtensionRow, 6–1932
LiftToChainmap, 8–2531
Line, 9–3107, 12–3996
LinearCharacters, 5–1460, 8–2708
LinearCode, 12–3886, 12–4009, 13–4176, 4177, 13–4218, 13–4267, 4268
LinearRelation, 2–460
LinearRelations, 3–818, 819
LinearSpace, 12–3981, 12–4003
LinearSystem, 9–3068, 9–3070, 3071, 9–3073
LinearSystemTrace, 9–3072
LineAtInfinity, 9–3124
LineGraph, 12–3885, 12–4048, 12–4053
LineGroup, 12–3877
LineOrbits, 5–1440
Lines, 12–3860
LineSet, 12–3856
Linking, 9–3192
LinkingNumbers, 9–3192
ListAttributes, 1–53
ListCategories, 1–98
ListSignatures, 1–97
ListTypes, 1–98
ListVerbose, 1–97
LittlewoodRichardsonTensor, 8–2751
LLL, 3–596, 3–600, 3–703
LLL Basis Matrix, 3–600
LLL Gram, 3–599
LLL Gram Matrix, 3–600
loc, 2–266
LocalCoxeterGroup, 7–2255
LocalDegree, 3–771
LocalFactorization, 4–1103
LocalGenera, 3–629
LocalHeight, 10–3298, 10–3337, 10–3386
LocalInformation, 10–3290, 3291, 10–3335, 10–3354, 10–3385
Localizations, 2–266, 9–2962, 2963
LocalRing, 3–708, 3–1014, 4–1108
LocalTwoSelmerMap, 10–3346
LocalUniformizer, 3–973
Log, 2–373, 2–461, 3–662, 4–1092, 4–1136, 6–1750, 10–3377, 3378
LogarithmicFieldExtension, 9–2949
LogDerivative, 2–476
LogGamma, 2–476
LogIntegral, 2–479
Logs, 3–724
LongestElement, 7–2243, 7–2279
LongExactSequenceOnHomology, 4–1224
LowerFaces, 4–1043
LowerSlopes, 4–1047
LowerTriangularMatrix, 2–489, 490
LowerVertices, 4–1044
LowIndexNormalSubgroups, 6–1836
LowIndexProcess, 6–1832
LowIndexSubgroups, 5–1328, 5–1433, 6–1828
L Polynomial, 3–937, 9–3147
LPProcess, 13–4376
LRatio, 11–3662, 11–3841
LRatioOddPart, 11–3662
L Series, 10–3495–3497, 10–3504, 11–3662, 11–3838
L Series Data, 10–3510
L Series Leading Coefficient, 11–3662
L Set Coefficients, 10–3508
L Set Precision, 10–3512
L Star, 10–3500
LT Taylor, 10–3500
Lucas, 2–288, 12–3913
MacWilliamsTransform, 13–4204, 13–4316
MagicNumber, 9–3210
MakeBasket, 9–3211
MakeCoPrime, 3–761
MakePCMap, 9–3050
INDEX OF INTRINSICS

INDEX OF INTRINSICS

xxxv

MakeProjectiveClosureMap, 9–3050
MakeResolutionGraph, 9–3187
MakeSpliceResolutionGraph, 9–3191
MakeType, 1–28
Manifold, 5–1680
ManifoldDatabase, 5–1680
ManinConstant, 10–3319
ManinSymbol, 11–3638
MantissaExponent, 2–451
map, 1–241, 9–3034, 9–3036
Mapping, 7–2322
Maps, 1–246
MargulisCode, 13–4255
MarkGroebner, 8–2393, 9–2784
MasseyProduct, 8–2530
Match, 6–1885, 6–2103
MatchTo, 5–1678
MatchToCharacteristics, 5–1678
MatchToDegrees, 5–1678
MatchToFieldSizes, 5–1678
MatchToKeys, 5–1678
Matrices, 10–3406, 12–4016
MatrixAlgebra, 2–364, 8–2336, 8–2362, 8–2402, 8–2459, 8–2461, 8–2554, 9–2853, 11–3755
MatrixGroup, 5–1238, 5–1407, 5–1678, 8–2424
MatrixLieAlgebra, 8–2580, 8–2587–2589
MatrixOfElement, 5–1707
MatrixOfIsomorphism, 8–2623
MatrixRepresentation, 8–2556
MatrixRing, 8–2459, 8–2461, 8–2554
MatrixUnit, 8–2460
MattsonSolemsonTransform, 13–4238
Max, 1–174, 1–193
Maxdeg, 12–4057, 12–4059, 12–4132, 12–4134
MaximalSubfields, 3–822
MaximalSubgroups, 5–1279, 5–1325, 5–1436, 5–1532, 5–1625, 6–1758, 8–2763
MaximalSubgroupsData (str : -), 5–1626
MaximalSubmodules, 8–2436, 8–2442
MaximalZeroOneSolution, 13–4374
MaximumClique, 12–4074
MaximumDegree, 12–4057, 12–4059, 12–4132, 12–4134
MaximumFlow, 12–4164
MaximumInDegree, 12–4059, 12–4133
MaximumIndependentSet, 12–4075
MaximumMatching, 12–4063, 12–4137
MaximumOutDegree, 12–4059, 12–4134
MaxIndeg, 12–4059, 12–4133
MaxNorm, 2–398, 2–437
MaxOutdeg, 12–4059, 12–4134
MaxParabolics, 12–3898
McElieceEtAllAsymptoticBound, 13–4229
McElieceAttack, 13–4224
MCPolynomials, 2–509
MDSCode, 13–4215
MEANS, 5–1366, 1367
Meataxe, 8–2432
meet=:, 2–559, 5–1527, 6–1755, 6–1976
MergeFields, 3–682
MergeFiles, 2–312
MergeUnits, 3–738
MetacyclicPCGroups, 5–1647
Min, 1–174, 1–193, 3–749, 3–952
Mindeg, 12–4058, 4059, 12–4133, 4134
MinimalAlgebraGenerators, 9–2840, 9–2910
MinimalAndCharacteristicPolynomials, 2–509
MinimalBaseRingCharacter, 2–335
MinimalBasis, 9–2882, 9–3007
MinimalCyclotomicField, 3–788
MinimalDegreeModel, 10–3386
MinimalElementConjugatingToPositive, 6–2034
MinimalElementConjugatingToSuperSummit, 6–2034
MinimalElementConjugatingToUltraSummit, 6–2034
INDEX OF INTRINSICS

MinimalField, 2–344, 345, 3–788, 5–1451, 8–2433
MinimalFreeResolution, 9–2868, 9–2884, 9–2908
MinimalHeckePolynomial, 11–3838
MinimalIdeals, 8–2340, 8–2605
MinimalInteger, 3–749
MinimalIntegerSolution, 13–4374
MinimalLeftIdeals, 8–2340
MinimalModel, 10–3240, 10–3268, 10–3386
MinimalNormalSubgroup, 5–1541
MinimalNormalSubgroups, 5–1355
MinimalOverfields, 3–822
MinimalOvergroup, 6–1837
MinimalOvergroups, 5–1279
MinimalParabolics, 12–3898
MinimalPartition, 5–1346
MinimalPartitions, 5–1346
MinimalQuadraticTwist, 10–3272
MinimalRelations, 8–2525
MinimalRightIdeals, 8–2340
MinimalSolution, 13–4374
MinimalSubmodule, 8–2436
MinimalSubmodules, 8–2436
MinimalSupermodules, 8–2442
MinimalSyzygyModule, 9–2883
MinimalWeierstrassModel, 10–3423
MinimalZeroOneSolution, 13–4374
Minimise, 3–789, 10–3407
MinimiseWeights, 9–3215
Minimize, 3–789, 8–2690
MinimizeGenerators, 9–2924
MinimumCut, 12–4163
MinimumDegree, 12–4058, 4059, 12–4133, 4134
MinimumDistance, 13–4197, 13–4282, 13–4312
MinimumDominatingSet, 12–4058
MinimumEuclideanDistance, 13–4284
MinimumEuclideanWeight, 13–4284
MinimumInDegree, 12–4059, 12–4133
MinimumLeeDistance, 13–4283
MinimumLeeWeight, 13–4283
MinimumOutDegree, 12–4059, 12–4134
MinimumWeight, 13–4197, 13–4282, 13–4312, 13–4341
MinimumWeightBounds, 13–4198
MinimumWeightTree, 12–4145
MinimumWord, 13–4199
MinimumWords, 13–4199
Minindeg, 12–4059, 12–4133
MinkowskiBound, 3–731
MinkowskiLattice, 3–578, 3–707, 3–761
MinkowskiSpace, 3–579, 3–707
Minor, 2–508
MinorBoundary, 5–1567
MinorLength, 5–1567
Minors, 2–508
Minoutdeg, 12–4059, 12–4134
MinParabolics, 12–3898
MinusInfinity, 2–305
MinusTamagawaNumber, 11–3674
MinusVolume, 11–3662
MixedCanonicalForm, 6–2013
mod: =, 2–279
ModelToString, 10–3406
ModelType, 11–3526
ModifySelfintersection, 9–3189
ModifyTransverseIntersection, 9–3189
Modinv, 2–303, 3–758, 3–948
Modorder, 2–303
Modsqrt, 2–303
ModularCurve, 11–3525
ModularCurveDatabase, 11–3528
ModularCurveQuotient, 11–3534
ModularDegree, 10–3326, 11–3672, 11–3811
ModularEmbedding, 11–3740
ModularEquation, 11–3702
ModularForm, 11–3599, 11–3624
ModularForms, 11–3595
ModularKernel, 11–3666
ModularParameterization, 11–3740
ModularParametrisation, 10–3317, 3318
ModularParametrization, 10–3317, 3318
ModularPolarization, 11–3805
ModularSolution, 2–532
ModularSymbolToIntegralHomology, 11–3744
ModularSymbolToRationalHomology, 11–3744
Module, 3–753, 3–954, 3–995, 4–1192, 5–1707, 8–2350, 8–2368, 8–2442, 8–2450, 8–2610, 8–2671, 9–2873, 9–2903, 9–3145, 9–3150
ModuleMap, 4–1220
ModuleOverSmallerField, 8–2690
Modules, 6–1930
ModulesOverCommonField, 8–2691
ModulesOverSmallerField, 8–2690
ModuleWithBasis, 8–2452
INDEX OF INTRINSICS

<table>
<thead>
<tr>
<th>Function</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumberOfComponents</td>
<td>1-63, 1-177, 1-202, 2-262, 2-266, 2-279, 2-327, 2-329, 2-347, 2-367, 2-389, 2-406, 2-419, 2-450, 2-558, 3-754, 3-878, 3-893, 3-949, 3-961, 3-972, 3-977, 4-1034, 4-1089, 4-1120, 4-1131, 4-1176, 5-1254, 5-1320, 5-1368, 5-1421, 5-1525, 6-1753, 6-1842, 6-1849, 6-1971, 6-2020, 6-2032, 6-2089, 8-2344, 8-2370, 8-2376, 8-2387, 8-2397, 8-2475, 8-2548, 8-2711, 9-2799, 9-3156, 9-3162, 12-3868, 12-3995, 12-4040, 12-4056, 12-4131, 13-4194, 13-4295, 13-4311</td>
</tr>
<tr>
<td>nsubset</td>
<td>1-178, 2-266, 2-329, 2-406, 2-558, 4-1176, 5-1254, 5-1320, 1321, 5-1421, 5-1526, 6-1753, 6-1842, 6-1972, 6-2089, 8-2342, 8-2397, 8-2475, 8-2596, 9-2796, 12-3868, 12-3995, 12-4040, 13-4194, 13-4295, 13-4311</td>
</tr>
<tr>
<td>NPCGenerators</td>
<td>5-1505</td>
</tr>
<tr>
<td>NPCgens</td>
<td>5-1505, 6-1970</td>
</tr>
<tr>
<td>Nqubits</td>
<td>13-4350</td>
</tr>
<tr>
<td>Nrels</td>
<td>6-1863, 6-2054, 6-2113</td>
</tr>
<tr>
<td>Nrows</td>
<td>2-493, 2-526, 2-548, 8-2469, 11-3766</td>
</tr>
<tr>
<td>Nsgens</td>
<td>5-1386, 5-1467</td>
</tr>
<tr>
<td>NthPrime</td>
<td>2-292</td>
</tr>
<tr>
<td>nTorsionSubgroup</td>
<td>11-3823</td>
</tr>
<tr>
<td>NuclearRank</td>
<td>6-1911</td>
</tr>
<tr>
<td>NullGraph</td>
<td>12-4034</td>
</tr>
<tr>
<td>NullHomotopy</td>
<td>8-2531</td>
</tr>
<tr>
<td>Nullity</td>
<td>11-3772</td>
</tr>
<tr>
<td>NullSpace</td>
<td>2-563, 4-1187, 8-2473</td>
</tr>
<tr>
<td>Nullspace</td>
<td>2-503, 2-531, 8-2611</td>
</tr>
<tr>
<td>NullspaceMatrix</td>
<td>2-503, 2-531</td>
</tr>
<tr>
<td>NullspaceTranspose</td>
<td>2-503, 2-531, 8-2473, 8-2611</td>
</tr>
<tr>
<td>Number</td>
<td>9-3220</td>
</tr>
<tr>
<td>NumberField</td>
<td>3-677, 678, 3-689, 3-767, 3-770, 3-776, 3-822, 3-846</td>
</tr>
<tr>
<td>NumberFieldSieve</td>
<td>2-307</td>
</tr>
<tr>
<td>NumberingMap</td>
<td>5-1255, 5-1309, 5-1422, 5-1518, 6-1754</td>
</tr>
<tr>
<td>NumberOfActionGenerators</td>
<td>3-636, 8-2424, 8-2687</td>
</tr>
<tr>
<td>NumberOfAffinePatches</td>
<td>9-3025</td>
</tr>
<tr>
<td>NumberOfAlgebraicGenerators</td>
<td>7-2307</td>
</tr>
<tr>
<td>NumberOfAntisymmetricForms</td>
<td>3-638</td>
</tr>
<tr>
<td>NumberOfBlocks</td>
<td>12-3992</td>
</tr>
<tr>
<td>NumberOfClasses</td>
<td>5-1268, 5-1315, 5-1428, 5-1521, 12-4093</td>
</tr>
<tr>
<td>NumberOfColumns</td>
<td>2-493, 2-526, 2-547, 8-2469</td>
</tr>
<tr>
<td>NumberOfComponents</td>
<td>1-210, 10-3386</td>
</tr>
<tr>
<td>NumberOfConstantWords</td>
<td>13-4205</td>
</tr>
<tr>
<td>NumberOfConstraints</td>
<td>13-4376</td>
</tr>
<tr>
<td>NumberOfCoordinates</td>
<td>9-2998</td>
</tr>
<tr>
<td>NumberOfCurves</td>
<td>10-3332</td>
</tr>
<tr>
<td>NumberOfDivisors</td>
<td>2-285, 2-302</td>
</tr>
<tr>
<td>NumberOfExtensions</td>
<td>4-1112</td>
</tr>
<tr>
<td>NumberOfFaces</td>
<td>12-4078, 12-4140</td>
</tr>
<tr>
<td>NumberOfFixedSpaces</td>
<td>5-1442</td>
</tr>
<tr>
<td>NumberOfGenerators</td>
<td>2-333, 2-558, 4-1198, 5-1251, 5-1296, 5-1409, 5-1505, 5-1576, 5-1690, 6-1736, 6-1740, 6-1788, 6-1863, 6-1970, 6-2003, 6-2054, 6-2071, 6-2086, 6-2099, 6-2113, 7-2238, 7-2278, 7-2307, 8-2462, 8-2499, 9-3134, 10-3296, 10-3368, 13-4181, 13-4274, 13-4305</td>
</tr>
<tr>
<td>NumberOfGradings</td>
<td>9-2997</td>
</tr>
<tr>
<td>NumberOfGraphs</td>
<td>12-4093</td>
</tr>
<tr>
<td>NumberOfGroups</td>
<td>5-1652, 5-1656, 5-1668, 1669</td>
</tr>
<tr>
<td>NumberOfInclusions</td>
<td>5-1279</td>
</tr>
<tr>
<td>NumberOfInvariantForms</td>
<td>3-637</td>
</tr>
<tr>
<td>NumberOfIrreducibleMatrixGroups</td>
<td>5-1671</td>
</tr>
<tr>
<td>NumberOfIsogenyClasses</td>
<td>10-3332</td>
</tr>
<tr>
<td>NumberOfLattices</td>
<td>3-645, 5-1668, 1669</td>
</tr>
<tr>
<td>NumberOfLines</td>
<td>12-3864</td>
</tr>
<tr>
<td>NumberOfMatrices</td>
<td>12-4016</td>
</tr>
<tr>
<td>NumberOfMetacyclicGroups</td>
<td>(p, n), 5-1648</td>
</tr>
<tr>
<td>NumberOfNewformClasses</td>
<td>11-3616</td>
</tr>
<tr>
<td>NumberOfNonZeroEntries</td>
<td>2-493, 2-526</td>
</tr>
<tr>
<td>NumberOfPartitions</td>
<td>2-288, 12-3919</td>
</tr>
<tr>
<td>NumberOfPCGGenerators</td>
<td>5-1505, 6-1911, 6-1970</td>
</tr>
<tr>
<td>NumberOfPermutations</td>
<td>12-3913</td>
</tr>
<tr>
<td>NumberOfPlacesDegECF</td>
<td>3-936, 3-971, 9-3146</td>
</tr>
<tr>
<td>NumberOfPlacesDegreeOne</td>
<td>3-1012</td>
</tr>
<tr>
<td>NumberOfPlacesDegreeOneECF</td>
<td>3-936, 3-971, 9-3147</td>
</tr>
<tr>
<td>NumberOfPlacesDegreeOneECFBound</td>
<td>3-937, 3-971, 9-3147</td>
</tr>
<tr>
<td>NumberOfPlacesDegreeOneOverExactConstantField</td>
<td>3-936, 3-971, 9-3147</td>
</tr>
<tr>
<td>NumberOfPlacesDegreeOneOverExactConstantFieldBound</td>
<td>3-937, 3-971, 9-3147</td>
</tr>
<tr>
<td>NumberOfPlacesDegreeOneOverExactConstantField</td>
<td>3-936, 3-971, 9-3146</td>
</tr>
<tr>
<td>NumberOfPoints</td>
<td>12-3864, 12-3992</td>
</tr>
<tr>
<td>NumberOfPointsAtInfinity</td>
<td>10-3434</td>
</tr>
<tr>
<td>NumberOfPointsOnSurface</td>
<td>10-3392</td>
</tr>
<tr>
<td>NumberOfPositiveRoots</td>
<td>7-2139, 7-2148, 7-2167, 7-2203, 7-2238, 7-2246, 7-2283, 7-2316</td>
</tr>
<tr>
<td>NumberOfPrimePolynomials</td>
<td>2-398</td>
</tr>
<tr>
<td>NumberOfPrimitiveAffineGroups</td>
<td>5-1663</td>
</tr>
<tr>
<td>NumberOfPrimitiveAlmostSimpleGroups</td>
<td>5-1663</td>
</tr>
<tr>
<td>NumberOfPrimitiveDiagonalGroups</td>
<td>5-1663</td>
</tr>
<tr>
<td><strong>INDEX OF INTRINSICS</strong></td>
<td>xli</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--</td>
</tr>
<tr>
<td>OrthogonalSum, 3–591</td>
<td></td>
</tr>
<tr>
<td>Orthonormalize, 3–619</td>
<td></td>
</tr>
<tr>
<td>OutDegree, 12–4058, 12–4133</td>
<td></td>
</tr>
<tr>
<td>OuterFaces, 4–1044</td>
<td></td>
</tr>
<tr>
<td>OuterFG, 5–1693</td>
<td></td>
</tr>
<tr>
<td>OuterOrder, 5–1691</td>
<td></td>
</tr>
<tr>
<td>OuterShape, 12–3936</td>
<td></td>
</tr>
<tr>
<td>OuterVertices, 4–1044</td>
<td></td>
</tr>
<tr>
<td>OutNeighbors, 12–4060, 12–4135</td>
<td></td>
</tr>
<tr>
<td>OutNeighbors, 12–4060, 12–4135</td>
<td></td>
</tr>
<tr>
<td>OvalDerivation, 12–3884</td>
<td></td>
</tr>
<tr>
<td>Overdatum, 7–2255, 7–2282</td>
<td></td>
</tr>
<tr>
<td>OverDimension, 2–558, 4–1169</td>
<td></td>
</tr>
<tr>
<td>Overgroup, 7–2254, 7–2282</td>
<td></td>
</tr>
<tr>
<td>p, 2–447</td>
<td></td>
</tr>
<tr>
<td>p, 2–447</td>
<td></td>
</tr>
<tr>
<td>PadCode, 13–4216, 13–4290, 13–4319</td>
<td></td>
</tr>
<tr>
<td>pAdicEllipticLogarithm, 10–3324</td>
<td></td>
</tr>
<tr>
<td>pAdicEmbeddings, 11–3620</td>
<td></td>
</tr>
<tr>
<td>pAdicField, 4–1071, 1072, 4–1079</td>
<td></td>
</tr>
<tr>
<td>pAdicQuotientRing, 4–1072</td>
<td></td>
</tr>
<tr>
<td>pAdicRing, 4–1071, 1072, 4–1079</td>
<td></td>
</tr>
<tr>
<td>PairReduce, 3–602, 603</td>
<td></td>
</tr>
<tr>
<td>PairReduceGram, 3–603</td>
<td></td>
</tr>
<tr>
<td>PaleyClass, 12–4052</td>
<td></td>
</tr>
<tr>
<td>PaleyTournament, 12–4052</td>
<td></td>
</tr>
<tr>
<td>ParallelClass, 12–3871</td>
<td></td>
</tr>
<tr>
<td>ParallelClasses, 12–3871</td>
<td></td>
</tr>
<tr>
<td>Parameters, 12–3993</td>
<td></td>
</tr>
<tr>
<td>Parametrization, 3–986, 9–3157, 10–3249</td>
<td></td>
</tr>
<tr>
<td>ParametrizationMatrix, 10–3248</td>
<td></td>
</tr>
<tr>
<td>ParametrizationToPuiseux, 4–1056</td>
<td></td>
</tr>
<tr>
<td>ParametrizeDegree6DelPezzo, 9–3081</td>
<td></td>
</tr>
<tr>
<td>ParametrizeDegree6DelPezzo, 9–3079</td>
<td></td>
</tr>
<tr>
<td>ParametrizeDegree9DelPezzo, 9–3079</td>
<td></td>
</tr>
<tr>
<td>ParametrizeOrdinaryCurve, 10–3250</td>
<td></td>
</tr>
<tr>
<td>ParametrizeRationalNormalCurve, 10–3250</td>
<td></td>
</tr>
<tr>
<td>ParentGraph, 12–4040</td>
<td></td>
</tr>
<tr>
<td>ParentPlane, 12–3860</td>
<td></td>
</tr>
<tr>
<td>ParentRing, 4–1048</td>
<td></td>
</tr>
<tr>
<td>ParityCheckMatrix, 13–4182, 13–4274, 13–4305</td>
<td>PartialDual, 3–590</td>
</tr>
<tr>
<td>PartialFactorization, 2–301</td>
<td></td>
</tr>
<tr>
<td>PartialFractionDecomposition, 3–895</td>
<td></td>
</tr>
<tr>
<td>PartialWeightDistribution, 13–4202</td>
<td></td>
</tr>
<tr>
<td>Partition, 1–199</td>
<td></td>
</tr>
<tr>
<td>PartitionCovers, 12–3936</td>
<td></td>
</tr>
<tr>
<td>Partitions, 2–287, 12–3919</td>
<td></td>
</tr>
<tr>
<td>PartitionToWeight, 8–2763</td>
<td></td>
</tr>
<tr>
<td>PascalTriangle, 12–3994</td>
<td></td>
</tr>
<tr>
<td>Path, 12–4144</td>
<td></td>
</tr>
<tr>
<td>PathExists, 12–4144</td>
<td></td>
</tr>
<tr>
<td>PathGraph, 12–4034</td>
<td></td>
</tr>
<tr>
<td>Paths, 12–4144</td>
<td></td>
</tr>
<tr>
<td>PathTree, 8–2503</td>
<td></td>
</tr>
<tr>
<td>PCClass, 5–1566</td>
<td></td>
</tr>
<tr>
<td>pCentralSeries, 5–1264, 5–1353, 5–1467, 5–1540</td>
<td></td>
</tr>
<tr>
<td>PCExponents, 6–1971</td>
<td></td>
</tr>
<tr>
<td>PCGenerators, 5–1505, 6–1970</td>
<td></td>
</tr>
<tr>
<td>pClass, 5–1541, 6–1911</td>
<td></td>
</tr>
<tr>
<td>pClosure, 8–2617</td>
<td></td>
</tr>
<tr>
<td>PCM, 8–2523, 9–3026</td>
<td></td>
</tr>
<tr>
<td>pCore, 5–1261, 5–1353, 5–1432, 5–1531, 5–1538</td>
<td></td>
</tr>
<tr>
<td>pCover, 5–1279, 5–1373, 5–1714</td>
<td></td>
</tr>
<tr>
<td>pCoveringGroup, 6–1910</td>
<td></td>
</tr>
<tr>
<td>PCPrimes, 5–1566</td>
<td></td>
</tr>
<tr>
<td>pElementaryAbelianNormalSubgroup, 5–1366</td>
<td></td>
</tr>
<tr>
<td>Pencil, 12–3871</td>
<td></td>
</tr>
<tr>
<td>PerfectGroupDatabase, 5–1650</td>
<td></td>
</tr>
<tr>
<td>PerfectSubgroups, 5–1272, 5–1332</td>
<td></td>
</tr>
<tr>
<td>PeriodMapping, 11–3669, 11–3844</td>
<td></td>
</tr>
<tr>
<td>Periods, 10–3323, 11–3669, 11–3844</td>
<td></td>
</tr>
<tr>
<td>PermRep, 5–1678</td>
<td></td>
</tr>
<tr>
<td>PermRepDegrees, 5–1678</td>
<td></td>
</tr>
<tr>
<td>PermRepKeys, 5–1678</td>
<td></td>
</tr>
<tr>
<td>Permutation, 5–1386</td>
<td></td>
</tr>
<tr>
<td>PermutationAutomorphism, 9–3054</td>
<td></td>
</tr>
<tr>
<td>PermutationCharacter, 5–1280, 1281, 5–1376, 5–1461, 8–2718</td>
<td></td>
</tr>
<tr>
<td>PermutationCode, 13–4177, 13–4268</td>
<td></td>
</tr>
<tr>
<td>PermutationMatrix, 2–491</td>
<td></td>
</tr>
<tr>
<td>PermutationModule, 5–1281, 5–1376, 5–1462, 8–2423, 8–2683</td>
<td></td>
</tr>
<tr>
<td>PermutationRepresentation, 5–1651, 5–1692, 9–3134</td>
<td></td>
</tr>
<tr>
<td>Permutations, 1–180, 12–3915</td>
<td></td>
</tr>
<tr>
<td>PermutationSupport, 5–1692</td>
<td></td>
</tr>
<tr>
<td>PermuteWeights, 8–2743</td>
<td></td>
</tr>
<tr>
<td>pExcess, 3–652</td>
<td></td>
</tr>
<tr>
<td>Pfaffian, 2–508</td>
<td></td>
</tr>
<tr>
<td>Pfaffians, 2–508</td>
<td></td>
</tr>
</tbody>
</table>
INDEX OF INTRINSICS

pFundamentalUnits, 3-738
PGammaL, 5-1390
PGammaU, 5-1391
PGL, 5-1389
PG, 5-1392
PGMinus, 5-1392
PGPlus, 5-1392
PGroupSection, 5-1940
PGroupStrong, 6-1781
PGU, 5-1390
PhaseFlip, 13-4357
phi, 11-3764, 11-3821
PHom, 8-2508
Pi, 2-453
PicardGroup, 3-730, 3-779
PicardNumber, 3-779
pIntegralModel, 10-3423
Pipe, 1-77
Place, 3-768, 3-968, 969, 9-3154
PlaceEnumCopy, 3-1027
PlaceEnumCurrent, 3-1028
PlaceEnumInit, 3-1027
PlaceEnumNext, 3-1028
PlaceEnumPosition, 3-1028
Places, 3-767, 3-916, 3-937, 3-969, 970, 3-976, 9-3153, 3154
PlacticIntegerMonoid, 12-3925
PlacticMonoid, 12-3925
PlaneToDisc, 11-3577
Plethysm, 8-2749
PlotkinAsymptoticBound, 13-4229
PlotkinBound, 13-4228
PlotkinSum, 13-4217, 13-4290, 13-4319
pMap, 8-2617
pmap, 1-242, 243
pMatrixRing, 8-2379, 8-2552
pMaximalOrder, 3-692, 3-913, 3-963, 8-2379, 8-2546
pMinimalWeierstrassModel, 10-3423
pMinimise, 10-3407
pMinus1, 2-297
pMultiplier, 5-1279, 5-1373, 5-1714
pMultiplierRank, 6-1911
pNormalModel, 10-3423
Point, 9-3207, 12-3985
PointDegree, 12-3995
PointDegrees, 12-3992
PointGraph, 12-4009, 12-4053
PointGroup, 12-3877, 12-4005
Points, 9-3211, 10-3244, 10-2377, 10-3281, 10-3367, 10-3390, 10-3432, 10-34355, 10-3445, 10-3451, 10-3458, 10-3475, 10-3477, 12-3860, 12-3897, 12-3992
PointsAtInfinity, 9-3124, 10-3282, 10-3432, 10-3434
PointsCubicModel, 9-3176
PointSearch, 9-3032
PointSet, 9-3013, 10-3279, 12-3856, 12-3985
PointsKnown, 10-3435
PointsOverSplittingField, 9-3017
PointsQI, 10-3308, 10-3390
Polarisation, 9-3207
PolarisedVariety, 9-3212
PolarToComplex, 2-451
Poles, 3-953, 3-969, 9-3155
PollardRho, 2-297
PolycyclicGenerators, 5-1467
PolycyclicGroup, 5-1239, 5-1502, 6-1960
PolygonGraph, 12-4034
Polylog, 2-461, 462
PolylogD, 2-462
PolylogDold, 2-462
PolylogP, 2-462
PolyMapKernel, 9-2838
Polynomial, 2-386, 3-1018, 4-1048
PolynomialAlgebra, 2-383, 2-414, 9-2775, 9-2828
PolynomialCoefficient, 4-1157
PolynomialMap, 9-3076
PolynomialRing, 2-383, 2-414, 9-2775, 9-2828, 9-2891, 10-3406
PolynomialSievert, 2-317
POmega, 5-1393
POmegaMinus, 5-1394
POmegaPlus, 5-1393
POpen, 1-77
Position, 1-61, 1-170, 1-193
PositiveConjugatesProcess, 6-2031
PositiveCoroots, 7-2167, 7-2203, 7-2246, 7-2284, 7-2316
PositiveDefiniteForm, 3-638
PositiveGammaOrbitsOnRoots, 7-2195
PositiveRelativeRoots, 7-2206
PositiveRoots, 7-2167, 7-2203, 7-2246, 7-2284, 7-2316
PositiveRootsPerm, 8-2637
PositiveSum, 2-480
PossibleCanonicalDissidentPoints, 9-3212
PossibleSimpleCanonicalDissidentPoints, 9-3212
Power, 3-657
PowerBigInteger, 1-168
PowerGroup, 6-1991
PowerIdeal, 2-265
PowerIndexedSet, 1-167
PowerMap, 5-1268, 5-1315, 5-1428, 5-1521
PowerMultiset, 1-168
PowerPolynomial, 2-395
PowerProduct, 3-760
PowerRelation, 2-460
PowerResidueCode, 13-4213
PowerSequence, 1-191
<table>
<thead>
<tr>
<th>Category</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerSeries</td>
<td>11-3602, 11-3659</td>
</tr>
<tr>
<td>PowerSeriesRing</td>
<td>4-1125</td>
</tr>
<tr>
<td>PowerSet</td>
<td>1-167</td>
</tr>
<tr>
<td>PowerSumToElementaryMatrix</td>
<td>12-3975</td>
</tr>
<tr>
<td>PowerSumToElementarySymmetric</td>
<td>3-821</td>
</tr>
<tr>
<td>PowerSumToHomogeneousMatrix</td>
<td>12-3975</td>
</tr>
<tr>
<td>PowerSumToMonomialMatrix</td>
<td>12-3975</td>
</tr>
<tr>
<td>PowerSumToSchurMatrix</td>
<td>12-3974</td>
</tr>
<tr>
<td>pPlus1</td>
<td>2-298</td>
</tr>
<tr>
<td>pPowerTorsion</td>
<td>10-3336</td>
</tr>
<tr>
<td>pPrimaryComponent</td>
<td>6-1752</td>
</tr>
<tr>
<td>pPrimaryInvariants</td>
<td>6-1752</td>
</tr>
<tr>
<td>pQuotient</td>
<td>5-1248, 5-1333, 5-1438, 5-1537, 5-1563, 6-1805, 8-2617</td>
</tr>
<tr>
<td>pQuotientProcess</td>
<td>6-1907</td>
</tr>
<tr>
<td>pRadical</td>
<td>3-692, 3-914, 3-964</td>
</tr>
<tr>
<td>pRank</td>
<td>12-3864, 12-3993</td>
</tr>
<tr>
<td>pRanks</td>
<td>5-1541</td>
</tr>
<tr>
<td>Precision</td>
<td>2-449, 2-452, 4-1080, 4-1091, 4-1130, 4-1141, 11-3602</td>
</tr>
<tr>
<td>PrecisionBound</td>
<td>11-3601</td>
</tr>
<tr>
<td>PreimageIdeal</td>
<td>8-2401, 9-2849</td>
</tr>
<tr>
<td>PreimageRing</td>
<td>2-407, 8-2401, 9-2849</td>
</tr>
<tr>
<td>PreparaataCode</td>
<td>13-4276</td>
</tr>
<tr>
<td>Preprune</td>
<td>4-1216</td>
</tr>
<tr>
<td>Presentation</td>
<td>7-2260, 8-2490</td>
</tr>
<tr>
<td>PresentationIsSmall</td>
<td>6-1963</td>
</tr>
<tr>
<td>PresentationLength</td>
<td>6-1788, 6-1863</td>
</tr>
<tr>
<td>PreviousPrime</td>
<td>2-292</td>
</tr>
<tr>
<td>PrimalityCertificate</td>
<td>2-290</td>
</tr>
<tr>
<td>Primary</td>
<td>3-782</td>
</tr>
<tr>
<td>PrimaryAlgebra</td>
<td>9-2905</td>
</tr>
<tr>
<td>PrimaryComponents</td>
<td>9-3021</td>
</tr>
<tr>
<td>PrimaryDecomposition</td>
<td>9-2814, 9-2850</td>
</tr>
<tr>
<td>PrimaryIdeal</td>
<td>9-2906</td>
</tr>
<tr>
<td>PrimaryInvariantFactors</td>
<td>2-512, 8-2480</td>
</tr>
<tr>
<td>PrimaryInvariants</td>
<td>6-1752, 9-2899</td>
</tr>
<tr>
<td>PrimaryRationalForm</td>
<td>2-511, 8-2479</td>
</tr>
<tr>
<td>Prime</td>
<td>3-630, 4-1078, 11-3702</td>
</tr>
<tr>
<td>PrimeBasis</td>
<td>2-293, 2-300</td>
</tr>
<tr>
<td>PrimeComponents</td>
<td>9-3021</td>
</tr>
<tr>
<td>PrimeDivisors</td>
<td>2-293, 2-300, 2-302</td>
</tr>
<tr>
<td>PrimeField</td>
<td>2-258, 2-344, 2-357, 2-363, 2-449, 3-706, 3-875, 3-914, 4-1079, 4-1118</td>
</tr>
<tr>
<td>PrimeForm</td>
<td>3-656</td>
</tr>
<tr>
<td>Primeideal</td>
<td>8-2558</td>
</tr>
<tr>
<td>PrimePolynomials</td>
<td>2-398</td>
</tr>
<tr>
<td>PrimePowerRepresentation</td>
<td>3-956</td>
</tr>
<tr>
<td>PrimeRing</td>
<td>2-258, 2-277, 2-325, 2-363, 2-387, 2-417, 3-706, 3-875, 3-892, 3-914, 4-1034, 4-1079, 4-1118, 8-2385, 12-3960</td>
</tr>
<tr>
<td>Primes</td>
<td>6-1929</td>
</tr>
<tr>
<td>PrimesInInterval</td>
<td>2-292</td>
</tr>
<tr>
<td>PrimesUpTo</td>
<td>2-292</td>
</tr>
<tr>
<td>PrimitiveElement</td>
<td>2-330, 2-361, 3-723, 3-749, 3-920</td>
</tr>
<tr>
<td>PrimitiveGroup</td>
<td>5-1663, 1664</td>
</tr>
<tr>
<td>PrimitiveGroupDatabaseLimit</td>
<td>5-1663</td>
</tr>
<tr>
<td>PrimitiveGroupDescription</td>
<td>5-1663</td>
</tr>
<tr>
<td>PrimitiveGroupIdentification</td>
<td>5-1667</td>
</tr>
<tr>
<td>PrimitiveGroupProcess</td>
<td>5-1665, 1666</td>
</tr>
<tr>
<td>PrimitiveGroups</td>
<td>5-1664</td>
</tr>
<tr>
<td>PrimitiveIdempotentData</td>
<td>8-2486</td>
</tr>
<tr>
<td>PrimitiveIdempotents</td>
<td>8-2486</td>
</tr>
<tr>
<td>PrimitivePart</td>
<td>2-397, 2-432</td>
</tr>
<tr>
<td>PrimitivePolynomial</td>
<td>2-372</td>
</tr>
<tr>
<td>PrimitiveQuotient</td>
<td>5-1351</td>
</tr>
<tr>
<td>PrimitiveWreathProduct</td>
<td>5-1304</td>
</tr>
<tr>
<td>PrincipalCharacter</td>
<td>8-2706</td>
</tr>
<tr>
<td>PrincipalDivisor</td>
<td>3-952, 9-3159</td>
</tr>
<tr>
<td>PrincipalDivisorMap</td>
<td>3-989</td>
</tr>
<tr>
<td>PrincipalIdealMap</td>
<td>3-939</td>
</tr>
<tr>
<td>PrincipalUnitGroup</td>
<td>4-1109</td>
</tr>
<tr>
<td>PrincipalUnitGroupGenerators</td>
<td>4-1109</td>
</tr>
<tr>
<td>PrintCollector</td>
<td>6-1925</td>
</tr>
<tr>
<td>PrintExtensions</td>
<td>6-1926</td>
</tr>
<tr>
<td>PrintFile</td>
<td>1-72, 73</td>
</tr>
<tr>
<td>PrintFileMagma</td>
<td>1-73</td>
</tr>
<tr>
<td>PrintModules</td>
<td>6-1925</td>
</tr>
<tr>
<td>PrintPrimes</td>
<td>6-1925</td>
</tr>
<tr>
<td>PrintProbabilityDistribution</td>
<td>13-4354</td>
</tr>
<tr>
<td>PrintProcess</td>
<td>6-1927</td>
</tr>
<tr>
<td>PrintQuotient</td>
<td>6-1925</td>
</tr>
<tr>
<td>PrintRelat</td>
<td>6-1926</td>
</tr>
<tr>
<td>PrintSeries</td>
<td>6-1926</td>
</tr>
<tr>
<td>PrintSortedProbabilityDistribution</td>
<td>13-4355</td>
</tr>
<tr>
<td>PrintSylowSubgroupStructure</td>
<td>7-2327</td>
</tr>
<tr>
<td>PrintTermsOfDegree</td>
<td>4-1151</td>
</tr>
<tr>
<td>PrintToPrecision</td>
<td>4-1151</td>
</tr>
<tr>
<td>Probability</td>
<td>13-4354</td>
</tr>
<tr>
<td>ProbabilityDistribution</td>
<td>13-4354</td>
</tr>
<tr>
<td>ProbableAutomorphismGroup</td>
<td>3-850</td>
</tr>
<tr>
<td>ProbableRadicalDecomposition</td>
<td>9-2814</td>
</tr>
<tr>
<td>ProcessLadder</td>
<td>5-1367</td>
</tr>
<tr>
<td>ProductCode</td>
<td>13-4216</td>
</tr>
<tr>
<td>ProductProjectiveSpace</td>
<td>9-2997</td>
</tr>
<tr>
<td>ProductRepresentation</td>
<td>3-728, 3-955, 8-2743</td>
</tr>
<tr>
<td>ProfileGraph</td>
<td>1-132</td>
</tr>
<tr>
<td>ProfileHTMLOutput</td>
<td>1-135</td>
</tr>
<tr>
<td>ProfilePrintByTotalCount</td>
<td>1-134</td>
</tr>
<tr>
<td>ProfilePrintByTotalTime</td>
<td>1-134</td>
</tr>
<tr>
<td>ProfilePrintChildrenByCount</td>
<td>1-134</td>
</tr>
<tr>
<td>ProfilePrintChildrenByTime</td>
<td>1-134</td>
</tr>
<tr>
<td>ProfileReset</td>
<td>1-131</td>
</tr>
<tr>
<td>Proj</td>
<td>9-2994, 9-3003</td>
</tr>
<tr>
<td>Projection</td>
<td>9-3036</td>
</tr>
<tr>
<td>ProjectionFromNonsingularPoint</td>
<td>9-3037</td>
</tr>
<tr>
<td>ProjectionOnto</td>
<td>11-3808</td>
</tr>
<tr>
<td>ProjectionOntoImage</td>
<td>11-3808</td>
</tr>
<tr>
<td>ProjectiveClosure</td>
<td>9-3025, 9-3050, 9-3124</td>
</tr>
</tbody>
</table>
INDEX OF INTRINSICS

ProjectiveClosureMap, 9–3026
ProjectiveCover, 8–2510
ProjectiveEmbedding, 12–3866
ProjectiveFunction, 9–3010, 9–3144
ProjectiveGammaLinearGroup, 5–1390
ProjectiveGammaUnitaryGroup, 5–1391
ProjectiveGeneralLinearGroup, 5–1389
ProjectiveGeneralOrthogonalGroup, 5–1392
ProjectiveGeneralOrthogonalGroupMinus, 5–1392
ProjectiveGeneralOrthogonalGroupPlus, 5–1392
ProjectiveGeneralUnitaryGroup, 5–1390
ProjectiveMap, 9–3037
ProjectiveModule, 8–2503, 2504
ProjectiveOmega, 5–1393
ProjectiveOmegaMinus, 5–1394
ProjectiveOmegaPlus, 5–1393
ProjectiveOrder, 2–517, 5–1418, 8–2472
ProjectivePlane, 9–3103
ProjectiveRationalFunction, 9–3010
ProjectiveResolution, 8–2510, 8–2524
ProjectiveResolutionPGroup, 8–2524
ProjectiveSGroupLinear, 5–1390
ProjectiveSymplecticGroup, 5–1392
ProjectiveSymplecticGroupMinus, 5–1391
ProjectiveSpace, 9–2994, 9–3103
ProjectiveSpecialLinearGroup, 5–1390
ProjectiveSpecialOrthogonalGroup, 5–1393
ProjectiveSpecialOrthogonalGroupMinus, 5–1393
ProjectiveSpecialOrthogonalGroupPlus, 5–1393
ProjectiveSpecialUnitaryGroup, 5–1391
ProjectiveSuzukiGroup, 5–1394
ProjectiveSymplecticGroup, 5–1391
Projectivity, 9–3056
Prospector, 5–1258
Prune, 1–196, 1–211, 1–218, 3–880, 4–1215, 9–3041
pSelmerGroup, 3–836, 10–3346
PseudoAdd, 10–3476
PseudoAddMultiple, 10–3476
PseudoAddBasis, 4–1202, 8–2368, 8–2375
PseudoBasis, 13–4274
PseudoDimensions, 4–1202
PseudoMatrix, 4–1209, 8–2368, 8–2375
PseudoMellinWeilGroup, 10–3342
PseudoRandom, 5–1578
PseudoReflection, 7–2266
PseudoRemainder, 2–395
Psi, 2–476
PSGroup, 5–1390
PSGroup, 5–1392
PSGroup, 5–1391
pSignature, 3–651
PSL, 5–1390
PSL2, 11–3541
PSO, 5–1392
PSOMinus, 5–1393
PSOPlus, 5–1393
PSp, 5–1391
PS, 5–1391
pSubalgebra, 8–2617
PSz, 5–1394
PuiseuxExpansion, 4–1049
PuiseuxExponents, 4–1054
PuiseuxExponentsCommon, 4–1054
PuiseuxSeriesRing, 4–1126
PuiseuxToParametrization, 4–1056
Pullback, 6–1871, 6–2035, 8–2508, 9–3045, 9–3047, 9–3078, 9–3130, 10–3438, 11–3780
PunctureCode, 13–4217, 13–4290, 13–4320, 13–4343
PureBraidGroup, 7–2259
PureLattice, 3–592
Pushforward, 9–3130
Pushout, 8–2508
PushThroughIsogeny, 10–3350
Put, 1–75
Puts, 1–75
QECC, 13–4345
QECLowerBound, 13–4347
QECCUpperBound, 13–4347
qEigenform, 11–3624, 11–3659
qExpansion, 11–3602
qExpansionBasis, 11–3600, 11–3659, 11–3692
qIntegralBasis, 11–3659
QMatrix, 2–403
QRCodeZ4, 13–3476
QRCode, 13–3476
QMatrix, 2–4213
QuadricIntersection, 13–3476
QuadeIdeal, 2–349, 3–721
QuadIdeal, 9–2924
QuadraticClassGroupTwoPart, 3–779
QuadraticField, 3–776
QuadraticForm, 3–586, 3–651, 3–784, 5–1598, 12–3873
QuadraticForms, 3–655
QuadraticOrder, 3–659
QuadraticTransformation, 9–3060
QuadraticTwist, 10–3269, 3270, 10–3425
QuadraticTwists, 10–3270, 10–3425
QuadricIntersection, 10–3307, 3308, 10–3406
QuantizedUEA, 8–2636
QuantizedUEAlgebra, 8–2636
QuantizedUniversalEnvelopingAlgebra, 8–2636
QuantumBasisElement, 13–4335
QuantumBinaryErrorGroup, 13–4336
QuantumCode, 13–4325, 13–4328, 4329
QuantumCyclicCode, 13–4331–4333
QuantumDimension, 8–2745
INDEX OF INTRINSICS

Rank, 2-388, 2-418, 2-508, 2-531, 2-562, 3-585, 3-875, 3-892, 4-1146, 4-1175, 4-1187, 7-2164, 7-2195, 7-2238, 7-2278, 7-2309, 8-2385, 8-2400, 8-2471, 9-2849, 10-3295, 11-3687, 11-3773, 11-3784, 12-3897

RankBound, 10-3340, 10-3388, 10-3465

RankBounds, 10-3295, 10-3388, 10-3465

RanksOfPrimitiveIdempotents, 8-2486

RationalCharacterTable, 8-2709

RationalCurve, 10-3234

RationalCuspidalSubgroup, 11-3833

RationalDifferentialField, 9-2934

RationalExtensionRepresentation, 3-916

RationalField, 2-343

RationalHomology, 11-3752

RationalMap, 10-3349

RationalMatrixGroupDatabase, 5-1667

RationalPoint, 10-3244

RationalPoints, 9-3014, 9-3017, 10-3244, 10-3277, 10-3281, 10-3367, 10-3432, 10-3435, 10-3445, 10-3451, 10-3458, 10-3476, 3477

RationalPointsByFibration, 9-3015

RationalReconstruction, 2-350, 3-956

Rationals, 2-343

RationalSequence, 8-2647

RationalSolutions, 9-2972

RawBasket, 9-3213

RayClassField, 3-839, 840

RayClassGroup, 3-833, 3-1005

RayClassGroupDiscLog, 3-1006

RayResidueRing, 3-835, 3-1005

Re, 2-451, 11-3574

Reachable, 12-4067, 12-4144

Read, 1-76, 1-78, 1-81

ReadBinary, 1-76

ReadBytes, 1-78, 1-81

Real, 2-451, 11-3548, 11-3574

RealEmbeddings, 3-769

RealField, 2-446

RealHomology, 11-3752

RealInjection, 7-2164

RealMatrix, 11-3766

RealPeriod, 10-3323

RealPlaces, 3-769

RealSigns, 3-770

RealTamagawaNumber, 11-3674

Realtime, 1-26

RealVectorSpace, 11-3752

RealVolume, 11-3662

rec, 1-234

recformat, 1-233

ReciprocalPolynomial, 2-395

RecogniseAlternating, 5-1379, 5-1591

RecogniseAlternatingOrSymmetric, 5-1378, 5-1590

RecogniseLargeRee, 5-1615

RecogniseRee, 5-1612

RecogniseSL3, 5-1604

RecogniseSymmetric, 5-1379, 5-1591

RecogniseSz, 5-1606

RecognizeClassical, 5-1600

RecognizeLargeRee, 5-1615

RecognizeRee, 5-1612

RecognizeSL2, 5-1602

RecognizeSL3, 5-1604

RecognizeSz, 5-1606

Rectify, 12-3941

RedoEnumeration, 6-1892

Reduce, 3-918, 4-1182, 8-2394, 9-2785, 10-3407

ReduceCharacters, 8-2719

ReducedAteTPairing, 10-3371

ReducedBasis, 8-2379, 8-2564, 8-2566, 10-3301, 10-3461

ReducedDiscriminant, 3-711

ReducedEtaTPairing, 10-3370

ReducedForm, 3-658

ReducedForms, 3-659

ReducedGramMatrix, 8-2564

ReducedLegendreModel, 10-3240

ReducedLegendrePolynomial, 10-3239

ReducedMinimalWeierstrassModel, 10-3424

ReducedModel, 10-3424

ReducedOrbits, 3-659

ReducedSubscheme, 9-3021

ReducedTatePairing, 10-3370

ReduceGenerators, 5-1388, 6-1859

ReduceGroebnerBasis, 9-2785

ReduceQuadrics, 10-3407

ReduceToTriangleVertices, 11-3582

ReduceVector, 2-559

Reduction, 3-658, 3-784, 3-982, 9-3074, 9-3166, 10-3245, 10-3335

ReductionOrbit, 3-658

Reductions, 11-3620

ReductionStep, 3-658

ReductionType, 10-3291

ReductiveLieAlgebra, 8-2588

ReductiveMatrixLieAlgebra, 8-2588

ReductiveRank, 7-2309

Reductum, 2-395, 2-429

ReeConjugacyClasses, 5-1623

ReedMullerCode, 13-4180

ReedMullerCodeZ4, 13-4276

ReedSolomonCode, 13-4214

ReeToWord, 5-1612

ReeGroup, 5-1589

ReeIrreducibleRepresentation, 5-1613

ReeMaximalSubgroups, 5-1617
INDEX OF INTRINSICS

ReeMaximalSubgroupsConjugacy, 5-1617
ReeSylow, 5-1621
ReeSylowConjugacy, 5-1621
RefineSection, 5-1361
Reflection, 7-2252, 7-2265, 7-2318
ReflectionGroup, 7-2153, 7-2176, 7-2224, 7-2234, 7-2258, 7-2260, 7-2267-2270, 7-2275
ReflectionMatrices, 7-2169, 7-2209, 7-2252, 7-2286
ReflectionMatrix, 7-2169, 7-2209, 7-2253, 7-2286
ReflectionPermutation, 7-2170, 7-2209, 7-2252, 7-2286
ReflectionPermutations, 7-2170, 7-2209, 7-2253, 7-2286
Reflections, 7-2252, 7-2318
ReflectionSubgroup, 7-2254
ReflectionWord, 7-2170, 7-2209, 7-2253, 7-2286
ReflectionWords, 7-2170, 7-2209, 7-2253, 7-2286
Regexp, 1-65
RegularLDPCEnsemble, 13-4255
RegularRepresentation, 8-2362, 8-2504
RegularSpliceDiagram, 9-3190
RegularSubgroups, 5-1272
Regulator, 3-711, 3-939, 10-3299, 10-3461
RegulatorLowerBound, 3-711
RelationIdeal, 9-2807, 9-2905
RelationMatrix, 3-732, 6-1736, 9-2866
Relations, 3-732, 3-954, 3-996, 6-1736, 6-1788, 6-2054, 6-2100, 6-2113, 9-2905, 9-3145, 9-3151, 11-3623
RelativeField, 3-702
RelativeInvariant, 3-808
RelativePrecision, 4-1091, 4-1132, 4-1142
RelativeRank, 7-2195
RelativeRootDatum, 7-2206
RelativeRootElement, 7-2306
RelativeRoots, 7-2206
RelativeRootSpace, 7-2203
Remove, 1-196, 1-223
RemoveColumn, 2-499
RemoveConstraint, 13-4377
RemoveEdge, 12-4047, 12-4126
RemoveEdges, 12-4047, 12-4126
RemoveFiles, 2-312
RemoveIrreducibles, 8-2719
RemoveRow, 2-499
RemoveRowColumn, 2-499
RemoveVertex, 12-4045, 12-4123
RemoveVertices, 12-4045, 12-4123
RemoveWeight, 9-3215, 9-3217
Rep, 1-172, 1-193, 1-210, 2-261, 3-821, 5-1255, 5-1310, 5-1519, 5-1578, 6-1754, 6-1972, 6-2003, 6-2031, 6-2059, 6-2077, 6-2090, 6-2118, 12-3857, 12-3868, 12-3985, 3986, 12-4041
RepetitionCode, 13-4178, 13-4270
ReplacePrimes, 6-1929
ReplaceRelation, 6-1883, 6-2102
ReplicationNumber, 12-3993
Representation, 6-1743, 8-2686, 9-2917, 9-2923
RepresentationDimension, 8-2745
RepresentationMatrix, 3-727, 3-950, 8-2361, 8-2373, 8-2402, 9-2853
RepresentationNumber, 3-663
RepresentationType, 8-2671
Representative, 1-172, 1-193, 2-261, 2-275, 2-326, 2-344, 2-361, 2-386, 2-417, 2-449, 3-629, 630, 3-695, 3-821, 3-869, 3-891, 3-947, 4-1084, 4-1117, 4-1129, 5-1255, 5-1310, 5-1519, 6-1972, 6-2003, 6-2031, 6-2059, 6-2077, 6-2118, 8-2385, 12-3857, 12-3868, 12-3985, 3986, 12-4041
RepresentativeCocycles, 5-1560
RepresentativePoint, 9-3156
Representatives, 3-631
Res_H2_0.QmodZ, 6-1762
ResetMaximumMemoryUsage, 1-84
ResetMinimumWeightBouns, 13-4198
Residual, 12-3988
Residue, 3-995, 9-3150, 9-3157, 12-3900
ResidueClassDegree, 3-968, 3-973
ResidueClassField, 2-266, 3-750, 3-771, 3-968, 3-973, 4-1079, 4-1129, 4-1141, 9-3157
ResidueClassRing, 2-323
ResidueField, 4-1119
ResolutionData, 8-2523
ResolutionGraph, 9-3183, 3184, 9-3186
ResolutionGraphVertex, 9-3183
RestrictDegree, 12-3969
RestrictedPartitions, 2-288, 12-3919
RestrictedSubalgebra, 8-2617
RestrictEndomorphism, 11-3758
RestrictField, 2-556, 5-1408, 13-4218
Restrict, 5-1712, 8-2694, 8-2717, 9-3011, 9-3040, 11-3758, 12-3988
RestrictionChainMap, 8-2526
RestrictionData, 8-2525
RestrictionMap, 8-2617
RestrictionMatrix, 8-2759, 8-2763
RestrictionOfGenerators, 8-2526
RestrictionOfScalars, 9-3028
RestrictionToImage, 11-3758
RestrictionToPatch, 9-3010, 9-3050
RestrictionPartitionLength, 12-3969
RestrictParts, 12-3969
INDEX OF INTRINSICS

RestrictResolution, 8-2526
Resultant, 2-403, 2-436
ResumeEnumeration, 6-1893
Retrieve, 1-228
Reverse, 1-196, 1-218, 4-1134
Reversion, 4-1134
RevertClass, 6-1910
Rewind, 1-75
Rewrite, 6-1825
ReynoldsOperator, 9-2894
RHS, 6-1734, 6-1776, 6-2098
rideal, 6-2100, 8-2338, 8-2374, 8-2391, 8-2464, 8-2558, 8-2669
RiemannRochSpace, 3-981, 9-3166
RiemannZeta, 10-3494
RightAction, 8-2424
RightActionGenerator, 8-2687
RightAdjointMatrix, 8-2609
RightAnnihilator, 8-2359, 8-2672
RightCosetSpace, 6-1848, 6-1905
RightDescentSet, 7-2244, 7-2280
RightExactExtension, 4-1216
RightGCD, 6-2024
RightGcd, 6-2024
RightGreatestCommonDivisor, 6-2024
RightIdeal, 8-2558
RightIdealClasses, 8-2379, 8-2561
RightInverse, 11-3810
RightInverseMorphism, 11-3810
RightIsomorphism, 8-2567
RightLCM, 6-2025, 2026
RightLcm, 6-2025, 2026
RightLeastCommonMultiple, 6-2025, 2026
RightMixedCanonicalForm, 6-2014
RightNormalForm, 6-2013
RightOrder, 8-2374, 8-2560
RightRegularModule, 8-2504
RightRepresentationMatrix, 8-2373
RightString, 7-2172, 7-2210, 7-2249
RightStringLength, 7-2172, 7-2211, 7-2249
RightTransversal, 5-1259, 5-1369, 5-1456, 5-1543, 6-1755, 6-1850, 6-1905, 6-1973
RightZeroExtension, 4-1217
Ring, 5-1707, 9-3013, 10-3279
RingClassGroup, 3-730
RingMap, 9-3013
RingOfFractions, 9-2856, 9-2935
RingOfIntegers, 2-274, 2-323, 2-343, 3-685, 3-687, 3-776, 3-891, 4-1077, 4-1129, 4-1141
RMatrixSpace, 4-1169, 4-1178, 9-2869, 11-3785
RMatrixSpaceWithBasis, 4-1169, 4-1181
RModule, 4-1168, 8-2422, 9-2863, 9-2873, 11-3601
RModuleWithAction, 11-3786
RModuleWithBasis, 4-1169
RombergQuadrature, 2-480
Root, 2-370, 2-454, 3-721, 3-758, 3-870, 3-959, 4-1096, 7-2167, 7-2203, 7-2246, 7-2284, 7-2317, 12-4070
RootAction, 7-2258
RootClosure, 7-2212
RootDatum, 7-2152, 7-2176, 7-2186, 7-2188, 7-2191, 7-2202, 7-2237, 7-2278, 7-2308, 8-2600, 8-2637, 8-2741
RootGSet, 7-2257
RootHeight, 7-2171, 7-2211, 7-2249, 7-2319
RootImages, 7-2221
RootLattice, 7-2001
RootNorm, 7-2172, 7-2211, 7-2249, 7-2319
RootNorms, 7-2171, 7-2211, 7-2249, 7-2319
RootNumber, 10-3325, 10-3335, 3336, 10-3355
RootOfUnity, 2-344, 2-366, 3-788, 789, 3-869
RootPermutation, 7-2221
RootPosition, 7-2167, 7-2204, 7-2246, 7-2284, 7-2317
Roots, 2-366, 2-391, 392, 2-457, 3-869, 3-954, 4-1061, 4-1101, 7-2167, 7-2203, 7-2246, 7-2283, 7-2316
RootSide, 12-4070
RootsInSplittingField, 2-366
RootsNonExact, 2-458
RootSpace, 7-2166, 7-2201, 7-2245, 7-2282, 7-2316
RootSystem, 7-2151, 7-2160, 2161, 7-2224, 7-2237, 7-2278, 8-2599
RootSystemMatrix, 7-2275
RootVertex, 9-3191
RosenhainInvariants, 10-3486
Rotate, 1-196, 2-548, 4-1173, 13-4188, 13-4294, 13-4307
RotateWord, 6-1885, 6-2103
Round, 2-282, 2-349, 2-391, 2-452
Row, 12-3937
RowInsert, 12-3942
RowLength, 12-3937
RowNullSpace, 8-2473, 8-2611
RowReductionHomomorphism, 7-2330
Rows, 11-3766, 12-3937
RowSequence, 2-494
RowSkewLength, 12-3937
RowSpace, 8-2473
RowSubmatrix, 2-496
RowSubmatrixRange, 2-496
RowWeight, 2-526
RowWeights, 2-526
RowWord, 12-3939
RPolynomial, 8-2760
<table>
<thead>
<tr>
<th>Index</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchurIndices,</td>
<td>8</td>
</tr>
<tr>
<td>Schur,</td>
<td>8</td>
</tr>
<tr>
<td>SchreierVectors,</td>
<td>12</td>
</tr>
<tr>
<td>SchurToHomogeneousMatrix,</td>
<td>12</td>
</tr>
<tr>
<td>SchurToElementaryMatrix,</td>
<td>5</td>
</tr>
<tr>
<td>SchreierSystem,</td>
<td>6</td>
</tr>
<tr>
<td>SchreierVector,</td>
<td>5</td>
</tr>
<tr>
<td>SchreierVectors,</td>
<td>5</td>
</tr>
<tr>
<td>Schur,</td>
<td>8</td>
</tr>
<tr>
<td>SchurIndex,</td>
<td>8</td>
</tr>
<tr>
<td>SchurIndices,</td>
<td>8</td>
</tr>
<tr>
<td>SchurToElementaryMatrix,</td>
<td>12</td>
</tr>
<tr>
<td>SchurToHomogeneousMatrix,</td>
<td>12</td>
</tr>
<tr>
<td>SchurToMonomialMatrix,</td>
<td>12</td>
</tr>
<tr>
<td>SchurToPowerSumMatrix,</td>
<td>12</td>
</tr>
<tr>
<td>SCGroup,</td>
<td>3</td>
</tr>
<tr>
<td>SCGroupAbelianInvariants,</td>
<td>3</td>
</tr>
<tr>
<td>SCGroupExactSequence,</td>
<td>3</td>
</tr>
<tr>
<td>SCGroupNumber,</td>
<td>3</td>
</tr>
<tr>
<td>Sdiff,</td>
<td>1–179</td>
</tr>
<tr>
<td>SEA,</td>
<td>10–3360</td>
</tr>
<tr>
<td>Search,</td>
<td>6–1862</td>
</tr>
<tr>
<td>SearchEqual,</td>
<td>6–1863</td>
</tr>
<tr>
<td>SearchForDecomposition,</td>
<td>5–1489</td>
</tr>
<tr>
<td>SearchForIsomorphism,</td>
<td>6–1799</td>
</tr>
<tr>
<td>SearchPGroups,</td>
<td>5–1646</td>
</tr>
<tr>
<td>Sec,</td>
<td>2–463</td>
</tr>
<tr>
<td>SecantVariety,</td>
<td>9–3063</td>
</tr>
<tr>
<td>Sech,</td>
<td>2–466</td>
</tr>
<tr>
<td>SecondaryInvariants,</td>
<td>9–2900</td>
</tr>
<tr>
<td>SectionCentraliser,</td>
<td>5–1323</td>
</tr>
<tr>
<td>SectionCentralizer,</td>
<td>5–1323</td>
</tr>
<tr>
<td>Sections,</td>
<td>9–3074</td>
</tr>
<tr>
<td>Sections (G),</td>
<td>8–2687</td>
</tr>
<tr>
<td>Seek,</td>
<td>1–75</td>
</tr>
<tr>
<td>Self,</td>
<td>1–204</td>
</tr>
<tr>
<td>SelfIntersections,</td>
<td>9–3189</td>
</tr>
<tr>
<td>SelmerGroup,</td>
<td>10–3339</td>
</tr>
<tr>
<td>Semidir,</td>
<td>5–1674</td>
</tr>
<tr>
<td>Semigroup,</td>
<td>6–2098</td>
</tr>
<tr>
<td>SemiLinearGroup,</td>
<td>5–1412</td>
</tr>
<tr>
<td>SemisimpleEFAModuleMaps,</td>
<td>6–1987</td>
</tr>
<tr>
<td>SemisimpleEFAModules,</td>
<td>6–1987</td>
</tr>
<tr>
<td>SemisimpleEFASeries,</td>
<td>6–1982</td>
</tr>
<tr>
<td>SemisimpleGeneratorData,</td>
<td>8–2488</td>
</tr>
<tr>
<td>SemisimpleLieAlgebra,</td>
<td>8–2586, 8–2588, 2589</td>
</tr>
<tr>
<td>SemisimpleMatrixLieAlgebra,</td>
<td>8–2587, 8–2589</td>
</tr>
<tr>
<td>SemisimpleRank,</td>
<td>7–2309</td>
</tr>
<tr>
<td>SemisimpleType,</td>
<td>8–2598</td>
</tr>
<tr>
<td>SeparatingElement,</td>
<td>3–922</td>
</tr>
<tr>
<td>SeparationVertices,</td>
<td>12–4062, 12–4136</td>
</tr>
<tr>
<td>Seq,</td>
<td>6–2060, 6–2078, 6–2119</td>
</tr>
<tr>
<td>Seqelt,</td>
<td>2–362</td>
</tr>
<tr>
<td>SeqFact,</td>
<td>2–302</td>
</tr>
<tr>
<td>SeqInt,</td>
<td>2–276</td>
</tr>
<tr>
<td>Seqlist,</td>
<td>1–218</td>
</tr>
<tr>
<td>Seqset,</td>
<td>1–200</td>
</tr>
<tr>
<td>SequenceOfRadicalGenerators,</td>
<td>8–2490</td>
</tr>
<tr>
<td>SequenceToElement,</td>
<td>2–362</td>
</tr>
<tr>
<td>SequenceToFactorization,</td>
<td>2–302</td>
</tr>
<tr>
<td>SequenceToInteger,</td>
<td>2–276</td>
</tr>
<tr>
<td>SequenceToList,</td>
<td>1–218</td>
</tr>
<tr>
<td>SequenceToMultiset,</td>
<td>1–176</td>
</tr>
<tr>
<td>SequenceToSet,</td>
<td>1–200</td>
</tr>
<tr>
<td>SerreBound,</td>
<td>3–936, 9–3147</td>
</tr>
<tr>
<td>Set,</td>
<td>1–169, 2–325, 2–363, 6–2060, 6–2078, 6–2119, 12–3869, 12–3996</td>
</tr>
<tr>
<td>SetAllInvariantsOfDegree,</td>
<td>9–2896</td>
</tr>
<tr>
<td>SetAssertions,</td>
<td>1–92</td>
</tr>
<tr>
<td>SetAttribute,</td>
<td>5–1697</td>
</tr>
<tr>
<td>SetAutoColumns,</td>
<td>1–92</td>
</tr>
<tr>
<td>SetAutoCompact,</td>
<td>1–92</td>
</tr>
<tr>
<td>SetBeep,</td>
<td>1–92</td>
</tr>
<tr>
<td>SetBufferSize,</td>
<td>10–3331</td>
</tr>
<tr>
<td>SetClassGroupBoundMaps,</td>
<td>3–736</td>
</tr>
<tr>
<td>SetClassGroupBounds,</td>
<td>3–736</td>
</tr>
<tr>
<td>SetColumns,</td>
<td>1–92</td>
</tr>
<tr>
<td>SetDebugOnError,</td>
<td>1–141</td>
</tr>
<tr>
<td>SetDefaultRealField,</td>
<td>2–445</td>
</tr>
<tr>
<td>SetDisplayLevel,</td>
<td>6–1911</td>
</tr>
<tr>
<td>SetEchoInput,</td>
<td>1–84, 1–93</td>
</tr>
<tr>
<td>SetElementPrintFormat,</td>
<td>6–2002</td>
</tr>
<tr>
<td>SetEntry,</td>
<td>2–528</td>
</tr>
<tr>
<td>SetForceCFP,</td>
<td>6–2002</td>
</tr>
<tr>
<td>SetGlobalTCPARAMETERS,</td>
<td>6–1823</td>
</tr>
<tr>
<td>SetHeckeBound,</td>
<td>11–3656</td>
</tr>
<tr>
<td>SetHelpExternalBrowser,</td>
<td>1–107</td>
</tr>
<tr>
<td>SetHelpExternalSystem,</td>
<td>1–107</td>
</tr>
<tr>
<td>SetHelpUseExternalBrowser,</td>
<td>1–107</td>
</tr>
<tr>
<td>SetHelpUseExternalSystem,</td>
<td>1–107</td>
</tr>
<tr>
<td>SetHistorySize,</td>
<td>1–93</td>
</tr>
<tr>
<td>SetIgnorePrompt,</td>
<td>1–93</td>
</tr>
<tr>
<td>SetIgnoreSpaces,</td>
<td>1–93</td>
</tr>
<tr>
<td>SetIndent,</td>
<td>1–93</td>
</tr>
<tr>
<td>SetIntegerSolutionVariables,</td>
<td>13–4377</td>
</tr>
<tr>
<td>SetKantPrecision,</td>
<td>3–700</td>
</tr>
<tr>
<td>SetKantPrinting,</td>
<td>3–700</td>
</tr>
<tr>
<td>Intrinsic</td>
<td>Page(s)</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------</td>
</tr>
<tr>
<td>SetLibraries</td>
<td>1-94</td>
</tr>
<tr>
<td>SetLibraryRoot</td>
<td>1-94</td>
</tr>
<tr>
<td>SetLineEditor</td>
<td>1-94</td>
</tr>
<tr>
<td>SetLogFile</td>
<td>1-84, 1-94</td>
</tr>
<tr>
<td>SetLowerBound</td>
<td>13-4377</td>
</tr>
<tr>
<td>SetMaximizeFunction</td>
<td>13-4377</td>
</tr>
<tr>
<td>SetMemoryLimit</td>
<td>1-94</td>
</tr>
<tr>
<td>SetObjectiveFunction</td>
<td>13-4377</td>
</tr>
<tr>
<td>SetOptions</td>
<td>6-1862</td>
</tr>
<tr>
<td>SetOrderMaximal</td>
<td>3-720, 3-910</td>
</tr>
<tr>
<td>SetOrderTorsionUnit</td>
<td>3-720</td>
</tr>
<tr>
<td>SetOrderUnitsAreFundamental</td>
<td>3-720</td>
</tr>
<tr>
<td>SetOutputFile</td>
<td>1-74, 1-94</td>
</tr>
<tr>
<td>SetPath</td>
<td>1-95</td>
</tr>
<tr>
<td>SetPowerPrinting</td>
<td>2-359</td>
</tr>
<tr>
<td>SetPrecision</td>
<td>11-3602</td>
</tr>
<tr>
<td>SetPresentation</td>
<td>6-2002</td>
</tr>
<tr>
<td>SetPreviousSize</td>
<td>1-71</td>
</tr>
<tr>
<td>SetPrimitiveElement</td>
<td>2-361</td>
</tr>
<tr>
<td>SetPrintKetsInteger</td>
<td>13-4351</td>
</tr>
<tr>
<td>SetPrintLevel</td>
<td>1-95</td>
</tr>
<tr>
<td>SetProcessParameters</td>
<td>6-1891</td>
</tr>
<tr>
<td>SetProfile</td>
<td>1-131</td>
</tr>
<tr>
<td>SetPrompt</td>
<td>1-95</td>
</tr>
<tr>
<td>SetQuitOnError</td>
<td>1-95</td>
</tr>
<tr>
<td>SetRows</td>
<td>1-95</td>
</tr>
<tr>
<td>SetSeed</td>
<td>1-30, 1-96</td>
</tr>
<tr>
<td>Setseq</td>
<td>1-199</td>
</tr>
<tr>
<td>SetsOfSingularPlaces</td>
<td>9-2970</td>
</tr>
<tr>
<td>SetToIndexedSet</td>
<td>1-176</td>
</tr>
<tr>
<td>SetToMultiset</td>
<td>1-176</td>
</tr>
<tr>
<td>SetToSequence</td>
<td>1-199</td>
</tr>
<tr>
<td>SetTraceback</td>
<td>1-95</td>
</tr>
<tr>
<td>SetUpperBound</td>
<td>13-4377</td>
</tr>
<tr>
<td>SetVerbose</td>
<td>1-96, 1-100</td>
</tr>
<tr>
<td>Seyersen</td>
<td>3-603, 604</td>
</tr>
<tr>
<td>SeyersenGram</td>
<td>3-603</td>
</tr>
<tr>
<td>SFA</td>
<td>12-3956</td>
</tr>
<tr>
<td>SFAElementary</td>
<td>12-3956</td>
</tr>
<tr>
<td>SFAHomogeneous</td>
<td>12-3956</td>
</tr>
<tr>
<td>SFAMonomial</td>
<td>12-3956</td>
</tr>
<tr>
<td>SFAPower</td>
<td>12-3956</td>
</tr>
<tr>
<td>SFASchur</td>
<td>12-3956</td>
</tr>
<tr>
<td>Shadow</td>
<td>12-3901</td>
</tr>
<tr>
<td>ShadowSpace</td>
<td>12-3901</td>
</tr>
<tr>
<td>Shape</td>
<td>8-2647, 12-3936</td>
</tr>
<tr>
<td>ShephardTodd</td>
<td>7-2276</td>
</tr>
<tr>
<td>Shift</td>
<td>4-1216</td>
</tr>
<tr>
<td>ShiftToDegreeZero</td>
<td>4-1216</td>
</tr>
<tr>
<td>ShimuraConjugates</td>
<td>11-3583</td>
</tr>
<tr>
<td>ShimuraReduceUnit</td>
<td>11-3581</td>
</tr>
<tr>
<td>ShortBasis</td>
<td>3-981, 9-3167</td>
</tr>
<tr>
<td>ShortCosets</td>
<td>5-1369, 5-1543</td>
</tr>
<tr>
<td>ShortenCode</td>
<td>13-4217, 13-4290, 13-4320, 13-4343</td>
</tr>
<tr>
<td>ShortestPath</td>
<td>12-4144</td>
</tr>
<tr>
<td>ShortestPaths</td>
<td>12-4144</td>
</tr>
<tr>
<td>ShortestVectors</td>
<td>3-606</td>
</tr>
<tr>
<td>ShortestVectorsMatrix</td>
<td>3-607</td>
</tr>
<tr>
<td>ShortVectors</td>
<td>3-609</td>
</tr>
<tr>
<td>ShortVectorsMatrix</td>
<td>3-609</td>
</tr>
<tr>
<td>ShortVectorsProcess</td>
<td>3-614</td>
</tr>
<tr>
<td>ShowIdentifiers</td>
<td>1-97</td>
</tr>
<tr>
<td>ShowMemoryUsage</td>
<td>1-97</td>
</tr>
<tr>
<td>ShowOptions</td>
<td>6-1861</td>
</tr>
<tr>
<td>ShowPrevious</td>
<td>1-70</td>
</tr>
<tr>
<td>ShowValues</td>
<td>1-97</td>
</tr>
<tr>
<td>ShrikhandeGraph</td>
<td>12-4054</td>
</tr>
<tr>
<td>ShrinkingGenerator</td>
<td>13-4364</td>
</tr>
<tr>
<td>Sieve</td>
<td>2-374</td>
</tr>
<tr>
<td>Sign</td>
<td>2-282, 2-306, 2-349, 2-398, 2-437, 2-453, 3-1027, 5-1307, 11-3728</td>
</tr>
<tr>
<td>Signature</td>
<td>2-278, 2-346, 3-711, 11-3571</td>
</tr>
<tr>
<td>SignDecomposition</td>
<td>9-3162</td>
</tr>
<tr>
<td>SisekBound</td>
<td>10-3300</td>
</tr>
<tr>
<td>SilvermanBound</td>
<td>10-3300</td>
</tr>
<tr>
<td>SimEQ</td>
<td>3-743</td>
</tr>
<tr>
<td>SimpleCanonicalDissidentPoints</td>
<td>9-3212</td>
</tr>
<tr>
<td>SimpleCohomologyDimensions</td>
<td>8-2515</td>
</tr>
<tr>
<td>SimpleCoreflationMatrices</td>
<td>7-2169, 7-2208, 7-2252, 7-2286</td>
</tr>
<tr>
<td>SimpleCoroots</td>
<td>7-2166, 7-2203, 7-2245, 7-2283, 7-2316</td>
</tr>
<tr>
<td>SimpleEpimorphisms</td>
<td>6-1798</td>
</tr>
<tr>
<td>SimpleExtension</td>
<td>3-702</td>
</tr>
<tr>
<td>SimpleGroupName</td>
<td>5-1594</td>
</tr>
<tr>
<td>SimpleGroupOfLieType</td>
<td>7-2300, 2301</td>
</tr>
<tr>
<td>SimpleHomologyDimensions</td>
<td>8-2510</td>
</tr>
<tr>
<td>SimpleLieAlgebra</td>
<td>8-2589</td>
</tr>
<tr>
<td>SimpleMatrixLieAlgebra</td>
<td>8-2590</td>
</tr>
<tr>
<td>SimpleModule</td>
<td>8-2504</td>
</tr>
<tr>
<td>SimpleOrders</td>
<td>7-2283</td>
</tr>
<tr>
<td>SimpleQuotientAlgebras</td>
<td>8-2485</td>
</tr>
<tr>
<td>SimpleQuotientProcess</td>
<td>6-1797</td>
</tr>
<tr>
<td>SimpleQuotients</td>
<td>8-1797</td>
</tr>
<tr>
<td>SimpleReflectionMatrices</td>
<td>7-2169, 7-2208, 7-2252, 7-2286</td>
</tr>
<tr>
<td>SimpleReflectionPermutations</td>
<td>7-2170, 7-2209, 7-2252, 7-2286</td>
</tr>
<tr>
<td>SimpleReflections</td>
<td>7-2252</td>
</tr>
<tr>
<td>SimpleRelativeRoots</td>
<td>7-2206</td>
</tr>
<tr>
<td>SimpleRoots</td>
<td>7-2166, 7-2203, 7-2245, 7-2283, 7-2316</td>
</tr>
<tr>
<td>SimpleSubgroups</td>
<td>5-1272, 5-1332</td>
</tr>
<tr>
<td>Simplex</td>
<td>9-2999</td>
</tr>
<tr>
<td>SimplexAlphaCode24</td>
<td>13-4277</td>
</tr>
<tr>
<td>SimplexBetaCode24</td>
<td>13-4277</td>
</tr>
<tr>
<td>SimplexCode</td>
<td>13-4180</td>
</tr>
</tbody>
</table>
INDEX OF INTRINSICS

SimplifiedModel, 10-3268, 10-3422
Simplify, 3-703, 3-880, 3-912, 4-1198, 6-1859, 6-1862, 12-3988
SimplifyLength, 6-1861, 1862
SimplifyPresentation, 6-1862
SimplyConnectedVersion, 7-2218
SimpsonQuadrature, 2-481
SimsSchreier, 5-1382
Sin, 2-462, 463, 4-1138
Sincos, 2-463, 4-1138
SingerDifferenceSet, 12-3990
SingletonAsymptoticBound, 13-4229
SingletonBound, 13-4228
SingularPoints, 9-3123
SingularRank, 9-3217
SingularSubscheme, 9-3021
Sinh, 2-466, 4-1138
SIntegralDesBovestPoints, 10-3330
SIntegralJunggrenPoints, 10-3330
SIntegralPoints, 10-3329
SIntegralQuarticPoints, 10-3330
Size, 9-3189, 9-3193, 12-4054, 12-4131
SkewHadamardDatabase, 12-4016
SkewShape, 12-3936
SkewWeight, 12-3937
SL2Characteristics, 5-1603
SL2ElementToWord, 5-1602
SL2Presentation, 5-1602
SL3ElementToWord (G, g), 5-1605
SL4Invariants, 10-3411
Slope, 12-3871
Slopes, 4-1047
SLPGroup, 6-2085
SmallerField, 5-1486
SmallerFieldBasis (G), 5-1487
SmallerFieldImage (G, g), 5-1487
SmallGroup, 5-1638, 1639
SmallGroupDatabase, 5-1637
SmallGroupDatabaseLimit, 5-1637
SmallGroupDecoding, 5-1644
SmallGroupEncoding, 5-1644
SmallGroupIsInsoluble, 5-1639
SmallGroupIsInSolvable, 5-1639
SmallGroupIsSoluble, 5-1638
SmallGroupIsSolvable, 5-1638
SmallGroupProcess, 5-1642
SmallGroups, 5-1639, 1640
SmallPeriodMatrix, 10-3478
SmallRoots, 2-392
SmithForm, 2-515, 8-2478
Socket, 1-79, 80
SocketInformation, 1-80
Socle, 5-1359, 8-2436, 8-2504
SocleAction, 5-1360
SocleFactor, 5-1359
SocleFactors, 5-1359, 8-2437
SocleImage, 5-1360
SocleKernel, 5-1360
SocleQuotient, 5-1360
SocleSeries, 5-1359, 8-2437
SocleQuotient, 5-1333, 5-1438, 5-1563, 6-1813, 1814, 6-1920
SocleQuotientProcess, 6-1924
SocleRadical, 5-1362, 5-1453, 7-2321, 8-2603
SocleResidual, 5-1264, 5-1352, 5-1452
SocleSchreier, 5-1382
SocleSubgroups, 5-1272
Solution, 2-304, 2-327, 2-504, 8-2484, 13-4377
Solutions, 3-744
SolvableLieAlgebra, 8-2622
SolvableQuotient, 5-1333, 5-1438, 5-1563, 6-1813, 1814, 6-1920
SolvableRadical, 5-1362, 5-1453, 8-2603
SolvableResidual, 5-1264, 5-1352, 5-1452
SolvableSchreier, 5-1382
SolvableSubgroups, 5-1272
SolveByRadicals, 3-817
Sort, 1-197
SortDecomposition, 11-3647, 11-3689
SpaceOfDifferentialsFirstKind, 3-992, 9-3149
SpaceOfHolomorphicDifferentials, 3-992, 9-3149
SpanningForest, 12-4069, 12-4139
SpanningTree, 12-4069, 12-4139
SparseIrreducibleRootDatum, 7-2190
SparseMatrix, 2-523, 524, 2-529, 530
SparseRootDatum, 7-2190, 2191
SparseStandardRootDatum, 7-2190
Spec, 9-2994, 9-3003
SpecialEvaluate, 3-1019
SpecialLieAlgebra, 8-2592
SpecialLinearGroup, 5-1583
SpecialOrthogonalGroup, 5-1585
SpecialOrthogonalGroupMinus, 5-1586
SpecialOrthogonalGroupPlus, 5-1586
SpecialPresentation, 5-1567
SpecialUnitaryGroup, 5-1584
SpecialWeights, 5-1567
Spectrum, 8-2749, 12-4054
Sphere, 12-4068
SpherePackingBound, 13-4228
SpinorCharacters, 3-629
SpinorGenera, 3-628
SpinorGenerators, 3-629
SpinorGenus, 3-628
SpinorNorm, 5-1599
SpinorRepresentatives, 3-631
Splice, 4-1216
SpliceDiagram, 9-3190, 3191, 9-3193, 3194
SpliceDiagramVertex, 9-3191
Split, 1-65
SplitAbelianSection, 6-1939
INDEX OF INTRINSICS

SplitCollector, 6-1928
Splitcomponents, 12-4062, 12-4136
SplitElementaryAbelianSection, 6-1939
SplitExtension, 5-1280, 5-1373, 5-1714
SplitExtensionSpace, 6-1930
SplitRealPlace, 11-3568
SplitRootDatum, 7-2219
SplitSection, 6-1939
SplittingField, 2-356, 3-681, 682, 4-1077
SPolynomial, 9-2799, 9-2877
SPrincipalDivisorMap, 3-990
Sprint, 1-73
Sprintf, 1-73
SQ.check, 6-1934
Sqrt, 2-328, 2-336, 2-370, 2-454, 3-721, 3-758, 3-870, 3-959, 4-1096, 4-1134, 4-1156
SquareFreeFactorization, 4-1103
SquarefreeFactorization, 2-282, 2-302, 2-402, 2-433
SquarefreePart, 2-433
SquarefreePartialFractionDecomposition, 3-895
SquareLatticeGraph, 12-4054
SquareRoot, 2-328, 2-370, 2-454, 3-721, 3-758, 3-870, 3-959, 4-1096, 4-1134, 4-1156
SQUFOF, 2-298
SRregulator, 3-990
SrivastavaCode, 13-4212
Stabiliser, 5-1340
StabiliserCode, 13-4335
StabiliserGroup, 13-4337
StabiliserMatrix, 13-4335
StabiliserOfSpaces, 5-1445
Stabilizer, 5-1340, 5-1440, 11-3549, 12-3878, 12-4007, 12-4088
StabilizerCode, 13-4355
StabilizerGroup, 13-4337
StabilizerLadder, 5-1368
StabilizerMatrix, 13-4335
StandardAction, 7-2259, 7-2282
StandardActionGroup, 7-2259, 7-2282
StandardForm, 8-2550, 13-4183, 13-4279
StandardFormConjugationMatrices, 8-2490
StandardGenerators (G, str : -), 5-1624
StandardGraph, 12-4032, 12-4109
StandardGroup, 5-1292
StandardLattice, 3-575
StandardMaximalTorus, 7-2321
StandardMetacyclicGroup (P), 5-1648
StandardParabolicSubgroup, 7-2254
StandardPresentation, 5-1549, 5-1625
StandardRepresentation, 7-2328, 8-2613, 8-2738, 8-2740
StandardRootDatum, 7-2189
StandardRootSystem, 7-2162
StandardTableaux, 12-3933
StandardTableauxOfWeight, 12-3933
StarInvolution, 11-3653
StartEnumeration, 6-1892
StartNewClass, 6-1908
Stauduhar, 3-805
SteenrodOperation, 9-2909
SteinitzClass, 4-1203
SteinitzForm, 4-1203
SternsAttack, 13-4225
StirlingFirst, 2-288, 12-3914
StirlingSecond, 2-288, 12-3914
StoreFactor, 2-295
StringToCode, 1-61
StringToInteger, 1-62
StringToIntegerSequence, 1-62
Strip, 5-1387
StrongApproximation, 3-1025
StrongGenerators, 5-1386, 5-1467
StronglyConnectedComponents, 12-4061, 12-4136
StronglyRegularGraphsDatabase, 12-4093
StructureConstant, 8-2714
StructureConstants, 7-2223
sub, 2-277, 2-325, 2-356, 357, 2-552, 3-588, 3-682, 3-689, 3-776, 4-1175, 4-1196, 4-1214, 5-1242, 5-1318, 5-1430, 5-1523, 6-1745, 1746, 6-1816, 6-1963, 6-2100, 7-2174, 7-2215, 8-2337, 8-2428, 8-2453, 8-2463, 8-2595, 8-2668, 9-2865, 9-2878, 12-3862, 12-4042, 12-4120, 12-4155, 13-4191, 13-4281, 13-4308
SubalgebraModule, 8-2452
SubcanonicalCurve, 9-3215
Subcode, 13-4191, 13-4281, 13-4308, 4309, 13-4330
SubcodeBetweenCode, 13-4191, 13-4309
SubcodeWordsOfWeight, 13-4191, 13-4309
SubfieldCode, 13-4219
SubfieldLattice, 3-821
SubfieldRepresentationCode, 13-4218
SubfieldRepresentationParityCode, 13-4218
Subfields, 3-821, 3-927
SubfieldSubcode, 13-4218
SubfieldSubplane, 12-3862
Subgroup, 6-1850, 6-1897, 11-3777, 3778, 11-3823
SubgroupClasses, 5-1270, 5-1326, 5-1434, 5-1532
SubgroupLattice, 5-1274, 5-1533
SubgroupsOfTorus, 11-3664
Subgroups, 5-1270, 5-1326, 5-1434, 5-1532, 5-1625, 6-1758
SubgroupScheme, 10-3276, 11-3530
SubgroupsData (str), 5-1626
SubgroupsLift, 5-1328, 5-1436
Sublattices, 3-641, 642
INDEX OF INTRINSICS

Submatrix, 2-495, 496, 8-2476
SubmatrixRange, 2-496
Submodule, 9-2866
SubmoduleAction, 5-1450
SubmoduleImage, 5-1450
SubmoduleLattice, 8-2440
SubmoduleLatticeAbort, 8-2440
Submodules, 8-2440
SubnormalSeries, 5-1264, 5-1353, 5-1452, 5-1540
SubOrder, 3-686, 3-915
Subring, 11-3778
Subsequences, 1-180, 12-3915
subset, 1-178, 2-266, 2-329, 2-406, 2-558, 3-718, 3-754, 3-757, 3-822, 3-844, 3-1012, 4-1176, 4-1200, 5-1254, 5-1278, 5-1320, 5-1421, 5-1526, 6-1753, 6-1842, 6-1971, 1972, 6-2089, 7-2174, 7-2216, 7-2306, 8-2342, 8-2376, 8-2397, 8-2401, 8-2430, 8-2442, 8-2474, 8-2596, 9-2796, 9-2850, 9-2867, 9-2879, 9-3014, 9-3077, 9-3134, 11-3542, 11-3690, 11-3704, 11-3739, 11-3788, 11-3831, 12-3866, 12-3868, 12-3995, 12-4040, 13-4194, 13-4295, 13-4311
Subsets, 1-179, 12-3915
Substitute, 6-1885, 6-2103
Substring, 1-61
SubsystemSubgroup, 7-2320
SubWeights, 8-2743
Subword, 6-1885, 6-2103
SuccessiveMinima, 3-615
SuggestedPrecision, 4-1103
Sum, 7-2171, 7-2210, 7-2248, 12-3988
SumNorm, 2-398, 2-437
SumOf, 11-3792
SumOfDivisors, 2-285, 2-302
SumOfImages, 11-3792
SumOfMorphismImages, 11-3792
UnitAction, 3-763
UnitCohomologyProcess, 3-824
UnitDiscLog, 3-764
UnitGroup, 3-762, 3-989
SuperScheme, 9-3006
SupersingularEllipticCurve, 10-3264
SupersingularModule, 11-3698
SuperSummitCanonicalLength, 6-2011
SuperSummitInfinum, 6-2011
SuperSummitProcess, 6-2031
SuperSummitRepresentative, 6-2028
SuperSummitSet, 6-2028
SuperSummitSupremum, 6-2011
Supplements, 5-1365
Support, 2-526, 2-548, 3-759, 3-769, 3-969, 3-980, 4-1174, 5-1337, 8-2351, 8-2428, 8-2610, 8-2673, 9-3160, 12-3861, 12-3965, 12-3992, 12-3996, 12-4032, 12-4109, 13-4187, 13-4293, 13-4307
Supremum, 6-2010
SurjectivePart, 11-3759
SuzukiGroup, 5-1587
SuzukiIrreducibleRepresentation, 5-1607
SuzukiMaximalSubgroups, 5-1617
SuzukiMaximalSubgroupsConjugacy, 5-1617
SuzukiSylow, 5-1619
SuzukiSylowConjugacy, 5-1619
SVPermutation, 5-1387
SVWord, 5-1387
SwapColumns, 2-498, 8-2477
SwapRows, 2-498, 8-2477
SwinnertonDyerPolynomial, 2-409
Switch, 12-4048
Sylow, 5-1261, 5-1323, 5-1432, 5-1531, 6-1756, 10-3457
SylowBasis, 5-1531
SylowConjClassical, 5-1618
SylowSubgroup, 5-1261, 5-1323, 5-1432, 5-1531, 6-1756, 7-2327
Sym, 5-1246, 5-1292, 5-1302, 6-1785
SymmetricBinaryForm, 2-429, 5-1597
SymmetricCharacter, 8-2729, 12-3968
SymmetricCharacterTable, 8-2729
SymmetricCharacterValue, 8-2729
SymmetricComponents, 8-2718
SymmetricForms, 3-637
SymmetricFunctionAlgebra, 12-3956
SymmetricFunctionAlgebraElementary, 12-3956
SymmetricFunctionAlgebraHomogeneous, 12-3956
SymmetricFunctionAlgebraMonomial, 12-3956
SymmetricFunctionAlgebraPower, 12-3956
SymmetricFunctionAlgebraSchur, 12-3956
SymmetricGroup, 5-1246, 5-1292, 5-1302, 6-1785
SymmetricMatrix, 2-490, 3-651
SymmetricNormaliser, 5-1323
SymmetricNormalizer, 5-1323
SymmetricPower, 8-2467, 8-2614, 8-2749, 8-2756, 9-2975
SymmetricRepresentation, 6-2040, 8-2727
SymmetricRepresentationOrthogonal, 8-2728
SymmetricRepresentationSeminormal, 8-2728
SymmetricSquare, 3-592, 8-2467, 8-2693
SymmetricWeightEnumerator, 13-4286
Symmetrization, 8-2717
SymplecticComponent, 8-2718
SymplecticComponents, 8-2718
SymplecticDual, 13-4338
SymplecticForm, 5-1597
SymplecticGroup, 5-1585
SymplecticInnerProduct, 13-4338
Syndrome, 13-4186
INDEX OF INTRINSICS

SyndromeSpace, 13–4184
System, 1–85
SystemNormaliser, 5–1531
SystemNormalizer, 5–1531
SystemOfEigenvalues, 11–3660
SyzygyMatrix, 9–2836
SyzygyModule, 8–2510, 9–2836, 9–2883
SzClassMap, 5–1623
SzClassRepresentative, 5–1623
SzConjugacyClasses, 5–1623
SzElementToWord, 5–1607
SzIsConjugate, 5–1623
SzPresentation, 5–1607
Tableau, 12–3930
TableauIntegerMonoid, 12–3928
TableauMonoid, 12–3928
Tableaux, 12–3968
TableauxOfShape, 12–3933
TableauxOnShapeWithContent, 12–3933
TableauxWithContent, 12–3933
Tails, 6–1908
TamagawaNumber, 10–3290, 11–3674, 11–3845
TamagawaNumbers, 10–3290
TameOrder, 8–2546
Tan, 2–463, 4–1138
Tangent, 12–3873
TangentAngle, 11–3550, 11–3576
TangentCone, 9–3019, 9–3119
TangentLine, 9–3119
TangentSpace, 9–3019
TangentVariety, 9–3062
Tanh, 2–466, 4–1138
TannerGraph, 13–4257
TateLichtenbaumPairing, 3–991
TatePairing, 10–3369
TeichmuellerLift, 4–1095
TeichmuellerSystem, 3–1014
Tell, 1–75
Tempname, 1–85
TensorBasis, 5–1481
TensorFactors, 5–1481
TensorInducedAction, 5–1483
TensorInducedBasis, 5–1483
TensorInducedPermutations, 5–1483
TensorPower, 8–2693, 8–2748
TensorProduct, 2–548, 2–559, 3–592,
8–2465, 8–2467, 8–2498, 8–2614,
8–2642, 8–2693, 8–2747, 2748, 8–2755,
10–3502, 12–4050
TensorWreathProduct, 5–1412
Term, 2–423, 4–1215
TerminalIndex, 9–3208
TerminalPolarisation, 9–3208
TerminalVertex, 12–4041, 12–4111
Terms, 2–391, 2–422, 4–1215, 8–2388,
9–2958
Theta, 2–474
ThetaOperator, 11–3654

ThetaSeries, 3–615, 3–663, 11–3692
ThreeDescent, 10–3311
ThreeDescentCubic, 10–3313
ThreeIsogenyDescent, 10–3313
ThreeIsogenyDescentCubic, 10–3314
ThreeIsogenySelmerGroups, 10–3314
ThreeSelmerElement, 10–3315
ThreeSelmerGroup, 10–3312
ThreeTorsionMatrices, 10–3316
ThreeTorsionPoints, 10–3316
ThreeTorsionType, 10–3315
Thue, 3–744
TietzeProcess, 6–1861
To2DUpperHalfSpaceFundamentalDomain,
10–3482
ToAnalyticJacobiann, 10–3480
ToddCoxeter, 6–1819
ToddCoxeterSchreier, 5–1382, 5–1465
ToLiE, 8–2762
Top, 3–822, 5–1276, 8–2441
TopQuotients, 5–1652
ToralRootDatum, 7–2190
ToralRootSystem, 7–2162
TorsionBound, 10–3336, 10–3387, 10–3458,
11–3672
TorsionFreeRank, 6–1751, 6–1803
TorsionFreeSubgroup, 6–1752
TorsionInvariants, 6–1751
TorsionLowerBound, 11–3834
TorsionMultiple, 11–3834
TorsionSubgroup, 6–1752, 10–3296, 10–3336,
10–3368, 10–3387, 10–3458, 11–3835
TorsionSubgroupScheme, 10–3276
TorsionUnitGroup, 3–737
TorusTerm, 7–2312
TotalDegree, 2–425, 3–894, 8–2389
TotalLinking, 9–3192
TotallyRamifiedExtension, 4–1075, 4–1140
TotalNumberOfCosets, 6–1897
Trace, 2–281, 2–348, 2–369, 2–508,
2–549, 3–726, 3–878, 3–950, 4–1094,
5–418, 8–2373, 8–2471, 8–2548,
8–2674, 10–3364, 11–3773, 13–4188,
13–4219, 13–4307
TraceAbs, 2–369, 3–726
Traceback, 1–97
TraceInnerProduct, 13–4307
TraceMatrix, 3–716
TraceOfProvenius, 10–3364
TracesOfProvenius, 10–3291
TraceZeroSubspace, 8–2368
TrailingCoefficient, 2–390, 2–421, 422,
8–2388
TrailingTerm, 2–391, 2–423, 8–2388
Transformation, 10–3437
TransformationMatrix, 3–714, 3–752,
3–920, 3–965
INDEX OF INTRINSICS

TransformForm, 5-1599
TransitiveGroup, 5-1658, 1659
TransitiveGroupDatabaseLimit, 5-1658
TransitiveGroupDescription, 5-1658
TransitiveGroupIdentification, 5-1662
TransitiveGroupProcess, 5-1661
TransitiveGroups, 5-1659
TransitiveQuotient, 5-1351
Transitivity, 5-1340
Translation, 9-2959, 9-3054, 9-3058, 3059, 9-3128
TranslationMap, 9-2961, 10-3349
TranslationOfSimplex, 9-3058
TranslationToInfinity, 9-3128
Transport, 6-2035
Transpose, 2-503, 4-1210, 8-2471
TransposePartition, 8-2763
Transversal, 2-559, 5-1259, 5-1369, 5-1456, 5-1543, 6-1755, 6-1838, 6-1850, 6-1905, 6-1973, 7-2255, 2256
TransversalElt, 7-2255, 2256
TransversalProcess, 5-1369
TransversalProcessNext, 5-1369
TransversalProcessRemaining, 5-1369
TransversalIds, 7-2255
TransverseIndex, 9-3210
TransverseIntersections, 9-3190
TransverseType, 9-3209
TrapezoidalQuadrature, 2-481
TrialDivision, 2-297, 3-782
TriangularDecomposition, 9-2820
TriangularGraph, 12-4054
TrivialLieRepresentationDecomposition, 8-2737
TrivialModule, 8-2679
TrivialOneCocycle, 5-1724
TrivialRepresentation, 8-2738, 8-2740
TrivialRootDatum, 7-2190
TrivialRootSystem, 7-2162
Truncate, 2-282, 2-349, 2-452, 4-1133, 9-2942
Truncation, 12-3901
TupleToList, 1-212, 1-218
Tuplist, 1-212, 1-218
TwistedCartanName, 7-2193
TwistedGroup, 5-1725
TwistedGroupOfLieType, 7-2305
TwistedPolynomials, 3-1015
TwistedQRCode, 13-4213
TwistedRootDatum, 7-2219
TwistedTori, 7-2326
TwistedToriOrders, 7-2326
TwistedTorus, 7-2326
TwistedTorusOrder, 7-2325
TwistedWindingElement, 11-3663
TwistedWindingSubmodule, 11-3664
TwistingDegree, 7-2195
Twists, 10-3270
TwoCycle, 5-1710
TwoCoverPullback, 10-3308
TwoDescendantsOverTwoIsogenyDescendant, 10-3304
TwoDescent, 10-3303, 10-3390
TwoElement, 3-753, 3-964
TwoElementNormal, 2-322, 3-753
TwoGenerators, 3-973, 9-3154
TwoGenus, 9-3217
TwoIsogeny, 10-3349
TwoIsogenyDescent, 10-3304
TwoIsogenySelmerGroups, 10-3390
TwoSelmerGroup, 10-3339, 10-3389, 10-3465
TwoSidedIdealClasses, 8-2379, 8-2561
TwoSidedIdealClassGroup, 8-2379, 8-2561
TwoTorsionPolynomial, 10-3289
TwoTorsionSubgroup, 3-660, 10-3458
TwoTransitiveGroupIdentification, 5-1378
Type, 1-28, 1-170, 2-258, 2-260, 3-585, 9-2936, 9-2940, 9-2954, 2955, 10-3235, 10-3275, 10-3277, 10-3279, 10-3282, 11-3686
Types, 12-3897
UltraSummitProcess, 6-2031
UltraSummitRepresentative, 6-2028
UltraSummitSet, 6-2028
UncapacitatedGraph, 12-4118
Define, 1-197
UnderlyingDigraph, 12-4051, 12-4129
UnderlyingElement, 5-1576
UnderlyingField, 3-916, 9-2937
UnderlyingGraph, 9-3188, 9-3191, 12-4051, 12-4129
UnderlyingMultiDigraph, 12-4130
UnderlyingMultiGraph, 12-4129
UnderlyingNetwork, 12-4130
UnderlyingRing, 3-916, 9-2937
UnderlyingVertex, 9-3191
Ungetc, 1-75
UniformizingElement, 3-749, 3-771, 3-973, 4-1079, 4-1085, 4-1128, 4-1141
UniformizingParameter, 9-3145, 9-3157
Union, 9-3003, 9-3107, 12-3988, 12-4049, 12-4127, 4128
UnipotentStabiliser, 5-1446
UnitFeet, 12-3875
UnitaryForm, 5-1598
UnitDisc, 11-3573
UnitEquation, 3-745
UnitGenerators, 2-333
UnitGroup, 2-277, 2-325, 2-330, 2-345, 2-363, 3-737, 3-765, 3-939, 4-1110, 8-2379, 8-2570
UnitGroupAsSubgroup, 3-737
UnitGroupGenerators, 4-1110
UnitRank, 3-711, 3-738, 3-939
Units, 8-2379, 8-2570
INDEX OF INTRINSICS

WeilRestriction, 3-918, 9-3028
WeylGroup, 7-2309, 8-2601
WeylWord, 8-2647
Widths, 11-3552
WindingElement, 11-3663
WindingLattice, 11-3664
WindingSubmodule, 11-3664
WittDesign, 12-3990
WittLieAlgebra, 8-2592
WittRing, 3-1013
Word, 12-3939
WordGroup, 5-1371, 5-1457
WordInStrongGenerators, 5-1387
WordProblem, 8-2492
WordProblemData, 8-2492
Words, 13-4205, 13-4316
WordsOfBoundedLeeWeight, 13-4283
WordsOfBoundedWeight, 13-4205, 13-4316
WordsOfLeeWeight, 13-4283
WordStrip, 5-1387
WordToSequence, 6-2009
WordToTableau, 12-3930
WreathProduct, 5-1305, 5-1412, 5-1511
Write, 1-72, 73, 1-79, 1-81
WriteBinary, 1-73
WriteBytes, 1-79, 1-81
WriteGModuleOver, 8-2691
WriteHadamardDatabase, 12-4018
WriteK3Data, 9-3226
WriteOverLargerField, 5-1479
WriteOverSmallerField, 5-1488, 8-2689
WriteRawHadamardData, 12-4019
WriteRepresentationOver, 8-2691
WronskianDeterminant, 9-2951
WronskianMatrix, 9-2951
WronskianOrders, 3-923, 3-985, 9-3146, 9-3167
X, 9-3013
XGCD, 2-284, 2-397, 4-1035, 4-1120
Xgcd, 2-284, 2-397, 4-1035
Xor, 1-201
xor, 1-11
YoungSubgroup, 5-1303
YoungSubgroupLadder, 5-1368
Z4CodeFromBinaryChain, 13-4278
ZBasis, 8-2368, 8-2375
ZechLog, 2-374
Zero, 2-261, 2-275, 2-326, 2-344, 2-361, 2-386, 2-417, 2-449, 2-546, 3-581, 3-695, 3-869, 3-891, 3-946, 3-1013, 3-1016, 4-1083, 4-1117, 4-1129, 4-1171, 8-2337, 8-2371, 8-2385, 8-2427, 8-2547, 8-2582, 8-2594, 8-2638, 8-2658, 8-2706, 9-2864, 9-2876, 9-2936, 9-2953
ZeroChainMap, 4-1220
ZeroCocycle, 5-1709
ZeroCode, 13-4178, 13-4270
ZeroComplex, 4-1213
Zeroes, 3-712, 3-952, 953
ZeroExtension, 4-1217
ZeroGammaOrbitsOnRoots, 7-2195
ZeroMap, 8-2508, 11-3757
ZeroMatrix, 2-489
ZeroModularAbelianVariety, 11-3725
ZeroModule, 8-2504
ZeroRootLattice, 7-2203
ZeroRootSpace, 7-2203
Zeros, 3-712, 3-952, 953, 3-969, 9-3155
ZeroSubgroup, 11-3823
ZeroSubspace, 11-3609
ZeroSubvariety, 11-3726
ZeroSumCode, 13-4178, 13-4270
ZetaFunction, 2-479, 3-937, 9-3147, 10-3365, 10-3435
ZetaFunctionsByDeformation, 10-3456
ZinovievCode, 13-4222