J. Symbolic Computation (1997) **24**, 235–265

The Magma Algebra System I: The User Language[†]

WIEB BOSMA[‡], JOHN CANNON[§] AND CATHERINE PLAYOUST[¶]

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

In the first of two papers on MAGMA, a new system for computational algebra, we present the MAGMA language, outline the design principles and theoretical background, and indicate its scope and use. Particular attention is given to the constructors for structures, maps, and sets.

1. Introduction

MAGMA is a new software system for computational algebra, the design of which is based on the twin concepts of algebraic structure and morphism. The design is intended to provide a mathematically rigorous environment for computing with algebraic structures (groups, rings, fields, modules and algebras), geometric structures (varieties, special curves) and combinatorial structures (graphs, designs and codes).

The philosophy underlying the design of MAGMA is based on concepts from Universal Algebra and Category Theory. Key ideas from these two areas provide the basis for a general scheme for the specification and representation of mathematical structures. The user language includes three important groups of constructors that realize the philosophy in syntactic terms: *structure constructors*, *map constructors* and *set constructors*. The utility of MAGMA as a mathematical tool derives from the combination of its language with an extensive kernel of highly efficient C implementations of the fundamental algorithms for most branches of computational algebra. In this paper we outline the philosophy of the MAGMA design and show how it may be used to develop an *algebraic programming* paradigm for language design. In a second paper we will show how our design philosophy allows us to realize natural computational "environments" for different branches of algebra.

An early discussion of the design of MAGMA may be found in Butler and Cannon (1989, 1990). A terse overview of the language together with a discussion of some of the implementation issues may be found in Bosma *et al.* (1994).

[†] This research was supported in part by the Australian Research Council.
[‡] E-mail: wieb@maths.usyd.edu.au
[§] E-mail: john@maths.usyd.edu.au
[¶] E-mail: playoust@maths.usyd.edu.au

0747–7171/97/030235 + 31 $25.00/0 sy960125

2. The Magma Philosophy: Design Criteria

The process of designing a computer algebra system involves choosing a mathematical viewpoint and then selecting one or more programming language paradigms (procedural, functional, rewrite rule-based etc). A given system will be based on a particular viewpoint chosen from among the many different ways of looking at an area of mathematics. Not only are different approaches evident in different branches, but even within a single branch, quite different viewpoints often co-exist.

Modern algebra is characterized by the process of abstracting minimal sets of axioms satisfied by some particular class of structures and then attempting to produce a classification of the entire class of structures satisfying that axiomatic system. The internal structure of an algebraic structure plays a central role in classification efforts. Important tools are structure-preserving mappings (*morphisms*) and *actions* of structures.

The design of MAGMA is firmly rooted in this structural view of algebra. The fundamental notions underlying the system design are those of *algebraic structure* and *morphism*. Having chosen a mathematical viewpoint, it is now appropriate to state a number of more specific criteria:

- The language should be "*universal-algebraic*" in the sense that its design philosophy should strive to be equally appropriate for all branches of algebra and related fields.
- Algebraic structures and their morphisms are to be *first class objects*.
- The language should ensure that the specification of mathematical objects and their attributes is *precise* and *unambiguous*. The semantics associated with each object definable in the language should be as close as possible to the standard mathematical interpretation.
- The user language should support common *mathematical notation* as far as possible.
- "Mathematical" data structures, such as sets, sequences and mappings, are used rather than the more standard computer science data structures such as arrays, lists and trees.
- *Efficiency* is a paramount concern.

In the following paragraphs we expand on some of these points.

Universality. A system design that provides a satisfactory computational environment for all areas of mathematics has not yet appeared, and for good reason: it is probably impossible. For example, an environment supporting computation in analysis, where the problem of expression simplification is central, will differ very significantly from an environment for constructive combinatorics where backtrack search is the main tool. On the other hand the many excellent systems developed for particular branches of mathematics (KANT, LiE, PARI, Macaulay, Snappea etc) suffer from the disadvantage that many practical computations will involve the use of tools from neighbouring areas. We have chosen to design a system for an area of mathematics that is, hopefully, sufficiently broad to encompass a large class of mathematical computations that rely heavily on algebraic calculation, but which is sufficiently constrained that the adoption of a single mathematical viewpoint will work equally well throughout the area.

First class status for structures. Since, operationally, algebraic structures and individual "elements" of a structure are each mathematical entities possessing properties which we seek to investigate, structures need to have first class status in an algebraic language. Thus, it should be as easy to define a polynomial ring R and perform operations on R

as it would be to define and perform arithmetic with a polynomial. This is in sharp contrast to the approach taken in systems such as MACSYMA, REDUCE, Maple and Mathematica which have adopted an "element-centred" model of algebra and provide virtually no support for structural computation.

Precision and the avoidance of ambiguity. The central notion here is that of "context": the answer to mathematical questions (and indeed, their appropriateness) often depends upon the context in which they are asked. As a simple example, consider the problem of factoring $x^3 - 1$. At first sight this is an innocuous and appropriate question for a symbolic algebra package. Upon reflection, however, it is entirely unreasonable to expect a solution from any system without further context: if, "in the current context" x is an identifier with the integer 4 assigned to it, then evaluation should yield the answer $x^3 - 1 = 3^2 \cdot 7$. A computer algebra system would be able to decide that no "value" has been assigned to x yet, but ambiguities arise if it assumes next that x refers to a transcendental element: it will then have to decide over which structure to factor the polynomial $x^3 - 1$. One possible solution, often adopted by conventional symbolic algebra systems in some form, is to make the notion of context explicit: the user specifies a context in which computation is to be performed, so that a change of context changes the domain of computation. Serious drawbacks to this approach include the difficulty of computing simultaneously in various structures, and the counterintuitive change of "meaning" of objects whenever the exterior context is changed. While such an approach may work for elementary applications in which computation takes place in a single fixed structure, in more sophisticated algebraic computation, where objects of one structure interact with those of another, it becomes difficult if not impossible. The solution chosen for Magma is to associate with every object its unique context, in the form of the "parent structure" to which it belongs. It is impossible to define $x^3 - 1$ in Magma without (explicitly) designating the structure to which it belongs. If, for example, it is defined to be an element of the polynomial ring $\mathbf{Z}[x]$, factorization results in its factorization into a product of irreducible polynomials over $\mathbf{Z}$.

Efficiency. Since Magma is intended as a heavy-duty research tool, its fundamental algorithms need to be as fast as possible. The difficulties of achieving satisfactory speed of execution will be clear to anyone who has undertaken the development of mathematical software. Since the optimal choice of data structures is crucial for many algorithms, the implementation of such algorithms in a high-level general algebraic language suffers the drawback that the use of generic data structures rather than the optimal data structures for a given problem will often result in the loss of a considerable degree of efficiency. This is true regardless of whether the language is interpreted or compiled into some low-level language such as C or Lisp. One approach to this problem is to install fundamental or particularly time-critical algorithms in the C kernel. To avoid the cost of re-implementing every relevant algebraic algorithm, the Magma internal system architecture makes it possible for certain classes of specialist software written independently of Magma to be installed in the kernel at the cost of a modest amount of effort.

3. Theoretical Foundations

In order to produce a coherent design, we require a model of algebraic computation that specifies the possible classes of objects that may be defined within the scheme, and allows the valid operators for a given class of objects to be recognized.

An essential prerequisite for mathematical rigour is a *strongly typed* system. Ideas from Universal Algebra and Category Theory have been used to develop a theory of formal algebraic specification of data structures and their operations (Burstall and Gougen, 1981; Gougen, 1989). This work underpins semantic models for Computer Algebra systems such as AXIOM (Jenks and Sutor, 1992) that support strong typing. Recently, Hearn and Schrüfer have used the theory of order-sorted algebras to develop a proposal for a strongly typed language for the Computer Algebra system REDUCE (Hearn and Schrüfer, 1995). This body of work is chiefly concerned with defining a satisfactory system of types in a particular context and explicating the relationships between the different types.

While no description known to us of the semantics of algebraic computation seems to be completely satisfactory, we have found that ideas from Universal Algebra and Category Theory provide a useful framework. Although the description of our model takes its inspiration from these sources, the MAGMA notions of algebra, category and variety do not strictly adhere to the standard definitions. We use these concepts not only to develop a theory of algebraic types and their representation, but also to identify operations that are common to large classes of algebraic and geometric structures.

In the next sections we briefly outline the basic ideas of multi-sorted algebras and categories in order to provide the reader with an understanding of the foundations of the MAGMA model. Our overview of universal algebra follows that of Meinke and Tucker (1992), and the reader is referred to this excellent exposition for a detailed account. For the notions of category theory we refer the reader to Mac Lane (1971).

3.1. MULTI-SORTED ALGEBRAS

To define multi-sorted algebras, we fix a non-empty set S of *sorts*. Although many familiar algebraic structures may be described as single-sorted algebras, it will be convenient to consider multi-sorted algebras.

A *multi-sorted algebra* A consists of a family $\{A_s : s \in S\}$ of non-empty sets A_s called *carrier sets*, together with a *signature* Σ; the signature identifies certain distinguished elements of the carrier sets as well as the collection of allowable operations in A, and is formalized as follows. By S^* we denote the (possibly empty) set of *words* in the elements of S; by λ we will denote the empty word. A *signature* Σ is an $S^* \times S$ indexed collection of sets $\{\Sigma^A_{w,s}, w \in S^*, s \in S\}$. For each sort S, the set $\Sigma^A_{\lambda,s} \subset A_s$ consists of the *constants* of sort s in A, while for each non-empty word $w = s_1 \cdots s_k$ and sort s the set $\Sigma^A_{w,s}$ consists of the *operations* $\sigma\colon A_{s_1} \times \cdots \times A_{s_k} \to A_s$, of arity k, with domain $A_{s_1} \times \cdots \times A_{s_k}$ and co-domain A_s.

We should therefore speak of "multi-sorted Σ-algebras", but in the remainder of this section the abbreviated term "algebras" will be used.

There are three standard constructions which, when applied to algebras, produce new algebras. These are the *subalgebra*, *quotient algebra* and *direct product* constructions. We briefly sketch their definitions. Let A and B be S-sorted algebras. Then a *subalgebra* B of A is defined as follows:

(i) The carrier sets of B are subsets of the carrier sets of A: i.e., $B_s \subseteq A_s$, for $s \in S$.
(ii) The constants of B are identical with those of A: i.e., $\Sigma^A_{\lambda,s} = \Sigma^B_{\lambda,s}$, for $s \in S$.
(iii) The operations on B are obtained by restricting the operations on A to B: i.e., $\sigma_B(b_1, \ldots, b_n) = \sigma_A(b_1, \ldots, b_n)$, for $\sigma \in \Sigma_{w,s}$, $b_i \in B_{s_i}$.

If B is a subalgebra of A then we write $B \leq A$. The notion of a subalgebra leads to a concept that is fundamental for computation: *generating set*. Let A be an algebra and let $X \subseteq A$. Then

$$\langle X \rangle_A = \bigcap_{\substack{X \subseteq B \\ B \leq A}} B$$

is the subalgebra *generated* by $X \subseteq A$. It is not hard to show that this intersection defines a subalgebra provided the intersection of carrier sets is non-empty. The statement that A is generated by $X \subseteq A$ is equivalent to $\langle X \rangle_A = A$. Further, we say that A is finitely generated if and only if $\langle X \rangle_A = A$, for some finite set X. The idea of a generating set provides us with an extremely compact method of representing algebras: rather than explicitly listing the elements of carrier sets, we will describe them in terms of (usually) small sets of generators.

A (Σ-)*congruence* is an S-sorted family of equivalence relations on the carrier sets of an algebra A which is compatible with the operations on A. Given a congruence $\cong$ on an algebra A, it is straightforward to define the *quotient algebra* $Q = A/\cong$ of A with respect to the congruence $\cong$ as follows (where $[\,]$ denotes for any R the canonical map $R \to Q$ that sends an element of R to its class in Q):

(i) The carrier sets of Q are quotients of the carrier sets of A: i.e., $Q_s = A_s/\cong_s$, for $s \in S$.
(ii) The constants of Q are the classes containing constants from A: $\Sigma^Q_{\lambda,s} = \{[c] : c \in \Sigma^A_{\lambda,s}\}$, for $s \in S$.
(iii) The operations on Q are obtained in the obvious way from those on A: $\sigma_Q([a_1], \ldots, [a_n]) = [\sigma_A(a_1, \ldots, a_n)]$, for $\sigma \in \Sigma_{w,s}$, and $a_i \in A_{s_i}$.

If A is an S-sorted Σ-algebra and $\cong$ is a Σ-congruence on A, then $Q = A/\cong$ is also an S-sorted Σ-algebra.

The *direct product* of Σ-algebras A and B is the Σ-algebra P defined as follows:

(i) The carrier sets of P are direct products of the carrier sets of A and B: i.e., $P_s = A_s \times B_s$, for $s \in S$.
(ii) The constants of P are the products of corresponding constants, i.e., $c_{(A\times B)_s} = (c_{A_s}, c_{B_s})$, for $s \in S$.
(iii) The operations on P are obtained componentwise, so for $\sigma \in \Sigma_{w,s}$, $a_i \in A_{s_i}, b_i \in B_{s_i}$: $\sigma_P((a_1, b_1), \ldots, (a_n, b_n)) = (\sigma_A(a_1, \ldots, a_n), \sigma_B(b_1, \ldots, b_n))$.

This construction is easily extended to products of arbitrary families of algebras.

A (Σ-)*homomorphism* between algebras A and B is an S-indexed family of mappings

$$\phi = \{\phi_s : A_s \to B_s \mid s \in S\}$$

such that $c_B = \phi(c_A)$, for each constant symbol c, and

$$\phi_s(\sigma_A(a_1, \ldots, a_n)) = \sigma_B(\phi_{s_1}(a_1), \ldots \phi_{s_n}(a_n))$$

for each operator σ. Related terms such as *homomorphic image*, Σ-*isomorphism*, Σ-*endomorphism* and Σ-*automorphism* are defined in the obvious manner. It is straightforward to show that for a Σ-algebra A, the set $\mathrm{End}(A)$, resp. $\mathrm{Aut}(A)$, of all endomorphisms, resp. automorphisms, of A forms a monoid, resp. group, under composition of mappings.

A class K of algebras is said to be *closed* under the formation of subalgebras if, and only if, whenever $A \in K$ and $B \leq A$, then $B \in K$. Similarly, we may define closure of a class K with respect to the formation of quotient algebras, direct products and homomorphic images.

A class of Σ-algebras closed under the formation of subalgebras, direct products and homomorphic images is called a *variety*. Such a class will also be closed under the formation of quotient algebras.

It turns out that varieties arise as classes of Σ-algebras with "the same operations" on them: algebras satisfying certain polynomial identities on (derived) operators. By $\mathbf{Alg}(\Sigma, P)$ we will denote the *equational class* of all Σ-algebras whose operations satisfy the relations specified by P.

A famous theorem by Birkhoff asserts that a class of (multi-sorted Σ-)algebras forms a variety if and only if it forms an equational class for some set of equations P.

Example: Commutative Rings

With the above notations we can describe *commutative rings* as single-sorted algebras as follows, if we adopt the convention that for single-sorted algebras we abbreviate $\Sigma_{s^n,s} = \Sigma_n$. The algebra consists of a non-empty carrier set R together with signature

$$\Sigma_0 = \{0, 1\}, \Sigma_1 = \{-\}, \Sigma_2 = \{+, \cdot\}, \Sigma_{n\geq 3} = \emptyset.$$

Thus, commutative rings contain $0, 1$ (it is convention to assume $0 \neq 1$), and allow the unary operation $-$ for negation, as well as the binary operations $+$ and $\cdot$.

Commutative rings form a variety, hence an equational class, which can be obtained from the above Σ-algebra by imposing the equations that express their axioms, i.e.,

$$\begin{array}{ll}
(x + y) + z = x + (y + z) & \text{(associativity of } +) \\
x + 0 = x = 0 + x & \text{(identity for } +) \\
x + (-x) = 0 = (-x) + x & \text{(inverse for } +) \\
x + y = y + x & \text{(commutativity of } +) \\
(x \cdot y) \cdot z = x \cdot (y \cdot z) & \text{(associativity of } \cdot) \\
x \cdot 1 = x = 1 \cdot x & \text{(identity for } \cdot) \\
x \cdot (y + z) = x \cdot y + x \cdot z & \text{(distributivity of } \cdot \text{ over } +) \\
x \cdot y = y \cdot x & \text{(commutativity of } \cdot)
\end{array}$$

Semigroups, groups, abelian groups, and not-necessarily-commutative rings can be obtained as classes of algebras and as equational classes in a similar fashion by choosing the appropriate subsets of constants, operations and equations from the above. Note that *fields* will not form a single-sorted algebra this way, because the additional unary operation $^{-1}$ of inversion is not everywhere defined on the carrier set.

Although many of the standard algebraic structures such as semigroups, rings and R-modules can thus be treated as single-sorted algebras, modelling computation requires the introduction of additional carrier sets and operations. For example, exponentiation of elements introduces the integers as a carrier set, membership testing introduces the booleans, etc. The introduction of such subsidiary carrier sets can be accomplished neatly by a process known as *expansion* (see Meinke and Tucker, 1992).

3.2. CATEGORIES

In category theory, the concepts of similar algebraic structures and their structure preserving mappings are formalized as follows. A *category* $\mathcal{C}$ consists of

(i) A collection $\mathrm{Ob}_{\mathcal{C}}$ of objects.
(ii) For each pair of objects A, B, a set $\mathrm{Hom}(A, B)$ of *morphisms* from A to B.
(iii) For each ordered triple A, B, C of objects of $\mathcal{C}$, a *composition* map $\mathrm{Hom}(B, C) \times \mathrm{Hom}(A, B) \to \mathrm{Hom}(A, C)$.

The following conditions are required to hold:

- The sets $\mathrm{Hom}(A, B)$ are disjoint for distinct pairs of objects A, B.
- Composition is associative.
- For each object A, $\mathrm{Hom}(A, A)$ contains an identity morphism id_A such that for all objects B and all f in $\mathrm{Hom}(A, B)$ and g in $\mathrm{Hom}(B, A)$, $\mathrm{id}_A \circ f = f$ and $g \circ \mathrm{id}_A = g$.

Many familiar classes of algebraic structures and their mappings form categories; in particular, every variety $\mathbf{Alg}(\Sigma, P)$ forms a category. However, there exist categories that are not varieties; for example, the category of fields (with the field homomorphisms between them) is not a variety, since fields are not even algebras.

A morphism of categories $\mathcal{C}$ and $\mathcal{D}$ is called a *functor*: it consists of a function F which acts on objects $C \mapsto FC$ and on morphisms $f \mapsto Ff$, such that $Ff \in \mathrm{Hom}(FA, FB)$ if $f \in \mathrm{Hom}(A, B)$, satisfying $F\,\mathrm{id}_C = \mathrm{id}_{FC}$ and $F(g \circ f) = Fg \circ Ff$. An important class of functors are the so-called *forgetful* functors, which simply forget some of the structure of the objects. Thus, the functors sending a field to the additive group of all its elements, and that of sending it to the multiplicative group of its non-zero elements, provide two examples of forgetful functors between the category of fields and that of abelian groups. As another example, the direct product construct that we encountered in the previous section is a functor from $\mathcal{C}$ to $\mathcal{C}$ for various categories $\mathcal{C}$.

A desirable property of varietal categories is the existence of free objects. In terms of algebras, a Σ-algebra F is *free* on the indexed family X_s of subsets $i_s \colon X_s \hookrightarrow F_s$ for a class of Σ-algebras $\mathcal{C}$ if and only if for any $A \in \mathcal{C}$ and any S-indexed family of maps $\phi_s \colon X_s \to A_s$ there exists a unique Σ-homomorphism ψ which agrees with ϕ on X, that is, such that $\psi_s \circ i_s = \phi_s$ on X_s. In short, every Σ-homomorphism from X factors through F.

Existence of Free Algebras: *Let $\mathcal{C}$ be a class of Σ-algebras that forms a variety. Then there exist free algebras in $\mathcal{C}$, and up to isomorphism such a free algebra is characterized by its indexed family of generator cardinalities $\#X_s$. In particular, in single sorted varieties of algebras there exists an essentially unique free algebra for any positive integer.*

The existence of free algebras for varieties is a consequence of the left adjoint theorem (Mac Lane, 1971), which implies the existence of a *left adjoint* for the forgetful functor **Alg** $\to$ **Set** in the case of a variety of single-sorted algebra; this left adjoint produces the free algebra in **Alg** on the carrier set X.

Since algebraic varieties have such nice properties, we use them to model the "ideal case". However, we do not restrict ourselves to working with categories belonging to

varieties. It usually turns out that the ideas developed for varietal categories can be suitably reinterpreted in the case of other structures.

We first use the varietal condition to provide us with a compact representation of algebras.

Existence of Generators: *Let A be a member of a class of algebras $\mathcal{C}$ that forms an algebraic variety V. Then A possesses a generating set X. Moreover, A can be obtained as a quotient of a free algebra in $\mathcal{C}$.*

Since we are interested in morphisms between algebras, the fact that an algebra is defined in terms of generating sets allows us to use the Homomorphism Representation Theorem.

Homomorphism Representation Theorem: *Let A be an algebra belonging to a variety. Then any homomorphism f of A is uniquely determined by the set $\{f(x) : x \in X\}$, where X is any generating set for A.*

3.3. THE MAGMA MODEL

The ideas developed in the previous sections provide the basis for developing a computational model of algebraic structures. We begin by formalizing the notion of an algebraic structure as a Σ-algebra. In order to avoid confusion with other notions of *algebra*, the term *magma* will be used when referring to a Σ-algebra. (This name was introduced by Bourbaki (Bourbaki, 1970), who defines a magma to be a set with a law of composition.† However, we will use it more generally to refer to a Σ-algebra.) We introduce a two-level classification of magmas in MAGMA:

(i) A class of magmas satisfying a set of identical relations P will be called a *variety*, written $\mathrm{Var}(\Sigma, P)$.
(ii) A class of magmas belonging to the variety V and sharing a common "representation" R will be called a *category*, written $\mathrm{Cat}(V, R)$.

The notion of variety permits the specification of *generic functions*, functions that are independent of the representation of a magma and which, consequently, may be applied to magmas belonging to any category in the variety. For example, the standard method of constructing the normal closure of a subgroup of a group requires only arithmetic with group elements and the ability to determine whether an element lies in a subgroup. Some varieties in MAGMA are groups (`Grp`), commutative rings (`Rng`), and modules (`Mod`).

At the level of category, a magma is realized in concrete form. Thus, the category to which a magma belongs determines the representation of its carrier sets. Associated with a category $\mathcal{C}$ belonging to the variety $\mathcal{V} = \mathrm{Var}(\Sigma, P)$ are functions implementing the operations given by Σ. Non-defining operations that depend upon a knowledge of the representation R are also attached to $\mathcal{C}$. Most categories in MAGMA have names consisting of (usually) three letters indicating the variety to which it belongs and usually three more to distinguish the category; for example, `RngMPol` is the name in MAGMA for the category of multivariate polynomial rings. (Technically, `RngMPol` is a family of categories indexed

† "Un ensemble muni d'une loi de composition est appelé un magma." (Bourbaki, 1970), ch. 1, p. 1.

by the coefficient ring, but we will abuse terminology and refer to `RngMPol` and similar indexed families of categories as though they were single categories.)

Recall that a magma is said to be *finitely generated* if it possesses a finite generating set. Since the vast majority of commonly occurring algebraic structures are finitely generated, we do not sacrifice much by restricting our model to describe finitely generated magmas. The occasional infinitely generated magma can be handled by other means within the language. Exceptions to the finitely generated paradigm are such notorious structures as the field of real numbers, rings of Laurent series, etc.

We further note that structure algorithms for algebraic structures typically work with generating sets. Hence, wherever possible we will choose to represent magmas in terms of finite generating sets. The property of being freely finitely generated is in many cases preserved under taking substructures and forming direct products, whereas taking quotients or homomorphic images yields (finite) presentations. Since the word problem is unsolvable for many interesting varieties, computation in a finitely presented category is often limited.

As we saw, free magmas exist within any class of magmas that form a variety; it will be convenient to extend the notion of a free magma somewhat, to include "the largest possible magma with the given parameters" in certain categories in which free objects strictly speaking do not exist. For example, within the category of matrix groups over a field F, the generalized free algebras are the general linear groups $\mathrm{GL}(n, F)$, for positive integers n. We will then be able to create most magmas from a "free" magma in this sense using constructions such as substructure, quotient, homomorphic image and direct product formation.

Many of the functions that create new (free) magmas will take existing magmas as parameters. Thus, for example, the creation of polynomial rings or of matrix groups uses some form of recursion since they must be defined over a coefficient ring (magma). Indeed, these creation functions may be viewed as functors from certain categories of ring to other categories of group or ring.

The category to which a magma belongs determines:

- The representation of elements of the carrier sets.
- The representation of carrier sets.
- The operations that are allowed on elements of the magma.
- The operations that are allowed on the magma.

The implementation of a category involves installing functions which perform the following tasks:

- Creation and deletion of magmas, their elements and their morphisms.
- Creation and deletion of mappings and coercion operations involving the magmas.
- The Σ-operations.
- The representation-dependent operations associated with the magma.

The fundamental action in our language will be the application of an operator to some list of objects. Let A be an S-sorted Σ-algebra and let

$$\sigma_A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$$

be an operation belonging to Σ^A. Suppose we attempt to apply σ_A to the arguments $b_1, \ldots, b_n$. This will involve four distinct steps:

(i) Test compatibility of sorts: That is, we must check that the sort corresponding to b_i is s_i for $i = 1, \ldots, n$. If any of these tests fail, the operation is illegal.
(ii) Test compatibility of values: In the case of n-ary operations for $n \geq 2$, it is necessary to check that $b_i \in A_{s_i}$ for $i = 1, \ldots, n$.
(iii) Locate the function that performs the operation σ_A. This is obtained from the category of A.
(iv) Invoke the function to evaluate $\sigma_A(b_1, \ldots, b_n)$.

Testing compatibility of sorts is what is usually referred to as type checking. A simple instance in which testing compatibility of values arises is in the evaluation of the expression $x * y$, where x and y are ring elements. The magma semantics insists that ring products may only be evaluated when both arguments belong to a common ring. In order to determine compatibility of values efficiently, we need to know if x and y belong to a common set.

We shall use the term type to mean a set of values. Any set constructible in the Magma language is ultimately defined in terms of one or more magmas. For simplicity, we restrict discussion to sets defined in terms of a single magma. As noted above, a narrow definition of a particular type of algebraic structure involves typically one or two carrier sets. However, actual computation introduces many further carrier sets, such as the set of integers, the power set of the magma, etc. The carrier sets that appear in the axiomatic definition of an algebraic structure will be referred to as *fundamental carrier sets*, while the expanded list required for computation will be referred to as *ancillary carrier sets*. The ancillary carrier sets for a magma are the same for every category. Roughly speaking, the types to which a magma A gives rise are as follows:

- The carrier sets of A.
- The set of all subalgebras of A.
- The set of all ideals of A.
- An n-order iterated power set of each of the above sets.

The situation is slightly more complicated than this since, as we shall subsequently see, Magma distinguishes between ordered and unordered sets, simple sets and multisets.

Thus, each possible set of values definable in the Magma language must arise as a carrier set associated with some magma. Hence, the types correspond precisely to the collection of all possible carrier sets. In Magma the carrier sets associated with a given magma form a related set of types. Compatibility of sorts and compatibility of values may be quickly determined if the pair $\langle s, A_s \rangle$ is associated with each object, i.e., its sort and the carrier set to which it belongs.

3.4. The constructors for magmas, elements, and mappings

In this section we discuss the language constructs for defining magmas and their elements. The first issue concerns the form of representation adopted. Since our objective is to work with specific magmas we require a concrete realization of the carrier sets of the magma. Rather than try to design in the abstract, we look at the way in which the fundamental varieties of algebraic structures are realized. What do the actual elements of carrier sets look like? While there are endless possibilities, it turns out that most of the computationally interesting categories of magmas arise as follows:

- As one of a small number of *primitive magmas* (e.g., the ring of integers).
- As a set of *mappings* from one magma to another.
- By means of *generators and relations*.
- As *substructures*, *quotient structures*, *extensions* and *direct products* of magmas.

For example, the finite field $\mathrm{GF}(p^3)$ can be specified by a single generator and a single relation over the prime field $\mathrm{GF}(p)$, which itself is a quotient of the integers. The bimodule of $m \times n$ integral matrices is the set of linear transformations from the module $\mathbf{Z}^m$ to the module $\mathbf{Z}^n$. The latter modules are direct products of copies of $\mathbf{Z}$.

We are now in a position where we can present a general mechanism for specifying a magma M. This involves two conceptual steps:

(i) The definition of an appropriate *free magma* F.
(ii) The construction of the desired magma M from F by forming a succession of substructures, quotient structures and extensions until M is obtained.

For each category, the implementer must install a function that creates the appropriate free magma. The system provides standard constructors which, given any magma M and a set X of elements belonging to M, construct either the substructure or quotient structure defined by the submagma generated by X.

The above ideas lead to the following generic constructions for creating magmas:

- A *free magma constructor*.
- A *submagma constructor*, which takes an existing magma M together with a set X of elements of M and creates the submagma of M generated by X.
- A *quotient magma constructor*, which takes an existing magma M together with a set X of elements of M and creates the quotient of M by the ideal generated by X.
- An *extension constructor*, which forms an extension of one magma by some other magma (the form of this is rather dependent upon the variety to which the magmas belong).

In addition to the above constructors we will need:

- An *element constructor*, for elements in a given magma.
- A *map constructor*, to construct morphisms between given magmas.

Mappings play many roles in Magma and are an important programming tool. Just as in mathematics, they are used to represent the following kinds of associations between sets and magmas:

- A natural *relationship* holding between two magmas (e.g., an embedding).
- A general *homomorphism* between two magmas (possibly an *endomorphism*).
- An *action* of magma A on magma B.
- An *association* between two sets.

For example, if some structural invariant of a magma A is to be computed using a divide-and-conquer algorithm which replaces the computation in A by computations in (smaller) submagmas and quotient magmas of A then the natural morphisms relating the

submagmas and quotient magmas to A are used to pull back solutions of the subproblems into A.

We need to consider the problem of representing mappings. For a small set or magma A it may be possible to represent a map $f: A \to B$ by listing all pairs $(a, f(a))$. In some situations, it is possible to define a mapping by giving a rule for calculating the image of a general element of the domain. However, both of these cases are the exception. Since we are mainly interested in morphisms between magmas, the fact that our magmas are defined in terms of finite generating sets allows us to use the Homomorphism Representation Theorem. Hence the constructor will require only the images of the generators of the domain.

3.5. COERCION

In a strongly typed algebra system such as MAGMA, a particular magma M may be embeddable in one or more other magmas. This embedding may be essentially unique (e.g. $\mathbf{Z}$ in $\mathbf{Q}$) or there may be a number of distinct possibilities (e.g. $\mathbf{Q}(\alpha)$ in $\mathbf{C}$). Since for most binary operations MAGMA requires both arguments to belong to a common magma, the user can soon become overwhelmed with the detail of continually having to cast objects into a different magma (i.e., to change types). The design of MAGMA is organized so as to mitigate this by providing both automatic and forced coercion. *Coercion* is an operation that, given an element x of a magma M and some magma N such that there is an interpretation of x in N, returns this "image" of x in N. *Automatic coercion* will only be performed when there is a canonical relationship between M and N. Knowledge of the existence of such a relationship will be available to MAGMA in two sets of circumstances.

Firstly, as a submagma or quotient magma is created, MAGMA automatically notes the relationship between the magmas and stores the associated morphism (that is, the inclusion homomorphism or the natural homomorphism onto a quotient) in its internal tables. When an attempt is made, say, to multiply two elements that are not already in the same magma, the MAGMA processor will use this relationship information in attempting to locate a common overstructure O which contains both elements. If one is found then both elements will be automatically cast as elements of O and the operation performed. Secondly, in certain situations where from convention or for mathematical reasons an unambiguous canonical map between M and N exists, this knowledge has been built into the MAGMA processor and automatic coercion will occur.

Forced coercion refers to the situation in which M does not have a unique and canonical relationship with N, but nevertheless an element x does have a natural image in N. In this case the user will have to apply forced coercion via `N ! x`. This occurs, for example, when only a subset of the elements of M have a natural interpretation in N.

4. The Magma Language

The MAGMA language is an imperative programming language having standard imperative style statements and procedures. The language has a functional subset providing functions as first class objects, higher order functions, partial evaluation, etc. We summarize and comment on some of the more interesting language elements in this chapter.

An object is specified in the MAGMA language by means of an *expression*. Expressions are constructed in the usual manner through the application of operators, functions and

Table 1. Expressions.

Identifier	Map constructor	Cartesian product constructor
Literal	Magma constructor	Co-product constructor
Function invocation	Element constructor	Tuple constructor
Function/procedure definition	Sequence constructor	Record format constructor
`select`, `case` expressions	Set constructor	Record constructor

constructors to less complex expressions. The major kinds of expressions in MAGMA are shown in Table 1. In contrast to many algebraic languages, any identifier appearing in a MAGMA expression must have a value. The type of an expression is inferred at run-time from the types of its subexpressions. In general, MAGMA employs *call-by-value evaluation*, in that it evaluates each subexpression before commencing the evaluation of the outermost expression.

Much of the power of the language derives from its constructors. These are basically functions with special syntax and semantics. Constructors are used to specify magmas, elements of magmas, mappings and aggregates (sets, sequences, products and records). They will receive particular attention in this survey, because of their novel nature and their importance to the language.

Before proceeding to the constructors, we note briefly that there are three classes of *identifiers* in MAGMA:

- *variable identifiers*: those identifiers which are declared as local, either implicitly by the first use rule, or explicitly through a local declaration. They may be assigned values.
- *value identifiers*: placeholders for values to be substituted during evaluation. They are effectively constants, and may not be reassigned. The value identifiers are:
  - the formal value arguments and parameters of a function or procedure.
  - all loop identifiers.
  - the pseudo-identifiers `$` ("current" structure) and `$$` (function/procedure being defined by current expression).
  - all identifiers whose first use in the statement body of a function or procedure is as a value (i.e., not on the left-hand side of a `:=` symbol, nor as an actual reference argument to a procedure).
- *reference identifiers*: identifiers preceded by a tilde $\sim$ in a procedure definition or invocation (page 260). A reference identifier in a procedure definition may be assigned a new value in the procedure. This will have the effect of changing the value of the corresponding reference identifier in the calling context.

4.1. MAGMA CONSTRUCTORS

4.1.1. FREE MAGMA CONSTRUCTORS

The standard model for many categories includes a constructor for *free magmas*. As Table 2 below indicates, this is true not only for varietal categories (such as `GrpFP`) in which true free objects exist, but also in some other magma categories in which it is convenient to create a magma from a general "free-like" object.

Table 2. Free magma constructors.

Category	*Constructor*	*Magma*
`SgpFP`	`FreeSemigroup(n)`	Free semigroup of rank n
`MonFP`	`FreeMonoid(n)`	Free monoid of rank n
`GrpFP`	`FreeGroup(n)`	Free finitely-presented group of rank n
`GrpAb`	`FreeAbelianGroup(n)`	Free abelian group of rank n
`RngMPol`	`PolynomialRing(R, n)`	Polynomial ring in n indeterminates over the ring R
`ModFP`	`FreeRModule(R, n)`	Free R-module of rank n
`AlgFP`	`FreeAlgebra(R, M)`	Free associative algebra over the ring R and the monoid M
`AlgMat`	`MatrixAlgebra(R, n)`	Full matrix algebra of degree n over the ring R
`GrpPerm`	`Sym(X)`	Symmetric group acting on the set X
`GrpMat`	`GL(n, R)`	General linear group of degree n over the ring R
`ModTup`	`RModule(R, n)`	Free R-module of n-tuples

A free algebra is normally defined as a term algebra over some set of variables. The notion of a (term algebra) variable causes semantic problems in many computer algebra systems since such a variable has different properties to an ordinary variable identifier. For this reason we avoid the notion of a (term algebra) variable or indeterminate in Magma. Rather, we use the notion of a *generator*. If F is a free algebra, the expression `F.i` denotes the ith generator of F. Semantically, this is simply a particular element of F. (In cases where a free algebra has not been assigned to an identifier, the form `$.i` refers to its ith generator in certain contexts.) The expression `F.i` is called the *standard name* for the ith generator of F; the generator assignment statement allows the user to choose identifiers for the generators as well.

We consider two examples. In the first we construct the free group F of rank 3. We will use an assignment statement to assign the group to the identifier F.

```
> F := FreeGroup(3);
> w := F.1^2*F.2*F.1*F.2^-1*F.3;
> w;
F.1^2 * F.2 * F.1 * F.2^-1 * F.3
```

It is usually desirable to be able to refer to generators by means of normal identifiers. Further, when elements of a free algebra are printed, we would like to see them presented as strings in generator names. This is handled by a special form of the Magma assignment. Any constructor that defines a magma in terms of generators may have names assigned to the generators by an assignment of the form

F`<`$a_1, \ldots, a_n$`> := `*magma constructor*`;`

This has the following effect. The magma F has stored, as part of its definition, the (ordered) set $X = \{x_1, \ldots, x_n\}$ of generators on which it is defined. When the above assignment is executed, two things happen. Firstly, the identifier a_i is assigned the ith generator of F as its value, for $i = 1, \ldots, n$. Secondly, when a word of F is printed, it will be printed as a string in the symbols $a_1, \ldots, a_n$ and their inverses. Note that any of these identifiers may be reassigned at any time. If a_i is assigned a new value, this has no effect on the generator that was its previous value. However, while a_i will no longer refer

Table 3. The submagma, quotient and extension constructors.

`sub<` *expr* `|` $expr_1$`, ...,` $expr_k$ `>`	(semi)groups, fields, modules, codes, graphs
`ncl<` *expr* `|` $expr_1$`, ...,` $expr_k$ `>`	Groups
`ideal<` *expr* `|` $expr_1$`, ...,` $expr_k$ `>`	Semigroups, rings
`lideal<` *expr* `|` $expr_1$`, ...,` $expr_k$ `>`	Semigroups, algebras
`rideal<` *expr* `|` $expr_1$`, ...,` $expr_k$ `>`	Semigroups, algebras
`quo<` *expr* `|` $expr_1$`, ...,` $expr_k$ `>`	(semi)groups, rings, modules, graphs
`ext<` *expr* `|` $expr_1$`, ...,` $expr_k$ `>`	Fields
`ExtensionField<` *expr*`,` *idfr* `|` $expr_1$`, ...,` $expr_k$ `>`	Finite fields

to x_i, the name a_i will still appear in words involving the generator x_i. We illustrate this mechanism by continuing the previous example.

```
> F<a,b,c> := FreeGroup(3);
> v := a^2*b*a*b^-1*c;
> v;
a^2 * b * a * b^-1 * c
> w := F.1^2*F.2*F.1*F.2^-1*F.3;
> v eq w;
true
```

In our second example, we construct the full matrix ring of 2×2 matrices over the integers. In the case of R-algebras, we may write down elements directly if we wish, since we have some concrete representation of their elements. Thus, the matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right)$ is created by the second statement below:

```
> M := MatrixRing(Integers(), 2);
> v := M ! [1, 2, 3, 4];
> v;
[1 2]
[3 4]
```

4.1.2. SUBMAGMA, QUOTIENT AND EXTENSION CONSTRUCTORS

After a free magma is constructed, the desired magma will normally be obtained from F by applying a sequence of submagma, quotient magma, and extension constructors.

The main form of the submagma constructor is `sub< >` as shown in Table 3. Given a magma M defined by an expression and a set Y of elements $m_i \in M$ defined by expressions on the right-hand side, it creates the submagma of M generated by Y. Thus, a magma in the same category as that of M is constructed.

The `ncl` constructor applies to objects in the variety of groups, and constructs the normal closure of the subgroup generated by the elements within the given group; again the result lies in the same category as the original group.

The various (two-sided and one-sided) `ideal` constructors listed apply in certain semigroup, ring and algebra categories. Although the effect is slightly dependent on the particular category, the effect is usually similar to that of any other `sub`, in the sense that an "ideal" generated by the given elements is created within the same category; thus a sub-object closed under some action by the given magma is constructed, rather than an object of some "ideal"-type.

The quotient magma constructor `quo` has similar syntactic structure. There is an important semantic difference: the quotient constructor may construct an object Q in a category different from that of the original magma M.

The expressions on the right-hand side of both `quo` and `sub` are allowed to be more general than simply defining elements of M: each $expr_i$ may evaluate to an element of M, a sequence defining an element of M, a submagma of M, an ideal of M, a set of elements of M, or a sequence of elements of M. Each term of the list defines one or more elements m_i of M, and Y is the set of all the m_i. In the case where an expression evaluates to a submagma or ideal, it contributes its generators to Y.

The `quo` constructor allows us to create any finitely presented algebra in a natural manner. For instance, the group G with presentation $\langle c, d \mid c^2, d^3, (cd)^4 \rangle$ can be created as a quotient of the free group F of rank 2:

```
> F<a, b> := FreeGroup(2);
> G<x, y> := quo< F | a^2, b^3, (a*b)^4 >;
> G;
GrpFP: G on 2 generators
Relations
     x^2 = Id(G)
     y^3 = Id(G)
     (x * y)^4 = Id(G)
> Generators(G);
{ x, y }
```

The ternary Golay code, constructed as a six-dimensional subspace of $K^{(11)}$ where K equals GF(3), provides an example of the use of the `sub` constructor.

```
> K11 := RModule(FiniteField(3), 11);
> G3 := sub< K11 |
>     [1,0,0,0,0,0,1,1,1,1,1], [0,1,0,0,0,0,0,1,2,2,1],
>     [0,0,1,0,0,0,1,0,1,2,2], [0,0,0,1,0,0,2,1,0,1,2],
>     [0,0,0,0,1,0,2,2,1,0,1], [0,0,0,0,0,1,1,2,2,1,0] >;
> G3;
KModule G3 of dimension 6 with base ring GF(3)
```

The third fundamental construction in algebra is the formation of extensions. Owing to the lack of a general theory of extensions even in the case of a particular variety such as groups, we have not been able to design a general extension constructor. The constructor `ext` is supported only for certain categories, and its semantics are heavily dependent upon the category. The `ExtensionField` constructor was devised to construct algebraic extensions in a single statement via a transcendental extension followed by a quotient.

A critical issue in algebraic system design concerns the ability to transfer objects between related structures. This, in turn, introduces questions concerning the representation of "derived magmas", i.e., magmas produced through the operations of forming submagmas, quotient magmas and extension magmas. In MAGMA, the following principle has been adopted:

Principle of Magma Autonomy: *If N is a magma constructed as a submagma, quotient magma or extension magma of M, then N will be autonomous, in the sense that any operation involving N alone can be performed without reference to M.*

As a consequence of the principle of autonomy, the magma M may be deleted without affecting the definition of N. This principle has some important consequences. In the case when N is finitely presented, then the elements of N are given in terms of the generators of N rather than in terms of the generators of M. If N is a magma belonging to a category in which magmas are always given with respect to a standard basis, then N will be given with respect to the appropriate standard basis. In both cases, it may not be easy to relate elements of N to elements of M. In order to have this information available, the `sub` and `quo` constructors return morphisms as a second return value. Thus the constructor `sub< M | ...>` returns:

(i) the submagma S defined by the constructor.
(ii) the inclusion monomorphism $\phi : S \to M$.

and the constructor `quo< M | ...>` returns:

(i) the quotient magma Q defined by the constructor.
(ii) the natural homomorphism $\phi : M \to Q$.

4.1.3. DIRECT PRODUCT AND DIRECT SUM CONSTRUCTORS

For a category of magmas belonging to a single algebraic variety, direct products may be formed using the intrinsic function `DirectProduct` (or `DirectSum`, in certain categories). This function takes a sequence Q of n magmas belonging to a single category and constructs their direct product (or sum). It returns the following objects:

(i) The product (or sum) magma $A = Q[1] \times \cdots \times Q[n]$.
(ii) The sequence I of injections such that $I[i] : Q[i] \to A$.
(iii) The sequence P of projections such that $P[i] : A \to Q[i]$.

This function should not be confused with the general Cartesian product constructor `car< >`, which merely returns a set (of type `SetCart`) containing *tuples* of elements with entries in the factor magmas. Its factors may be of any kind.

4.1.4. SPECIFIC MAGMA CONSTRUCTORS

In addition to the generic magma constructors above, MAGMA also provides various constructors for category-specific magmas. Table 4 lists most of them. Besides these *constructors* (with arguments enclosed by `< >`) there are many *functions* (with arguments in `( )`) for the creation of category-specific magmas.

Example:
We illustrate the iterated use of constructors with code that creates the Hecke algebra of type E_6. The algebra is obtained as the quotient of a finitely presented associative algebra H by a certain right ideal I. A free associative algebra FA over a cyclotomic field $Q5$ is defined on generators corresponding to those of a monoid M. Thus, the multiplicative relations between the generators of FA are inherited from M. Additional relations are now imposed on FA to give the required associative algebra H. The only serious computation involved occurs when constructing the algebra quotient H/I, which is performed using Linton's vector enumerator.

Table 4. Specific magma constructors.

`Monoid<` *generators* `|` *relations* `>`
`Semigroup<` *generators* `|` *relations* `>`
`AbelianGroup<` *generators* `|` *relations* `>`
`Group<` *generators* `|` *relations* `>`
`PolycyclicGroup<` *generators* `|` *relations* `>`
`MatrixGroup<` *degree*, *ring* `|` *generators* `>`
`PermutationGroup<` *degree* or *G-set* `|` *generators* `>`
`MatrixRing<` *ring*, *degree* `|` *generators* `>`
`MatrixAlgebra<` *ring*, *degree* `|` *generators* `>`
`LinearCode<` *field*, *length* `|` *generators* `>`
`IncidenceStructure<` *points* `|` *blocks* `>`
`Design<` *t*, *points* `|` *blocks* `>`
`ProjectivePlane<` *points* `|` *lines* `>`
`AffinePlane<` *points* `|` *lines* `>`
`Graph<` *vertex set* `|` *edge set* `>`
`Digraph<` *vertex set* `|` *edge set* `>`

Table 5. Literals.

Description	*Remarks*	*Examples*	*Type*
Boolean		`true false`	`BoolElt`
String		`"abc0 12"`	`MonStgElt`
Integer	Optional sign	`-7, +5, 11`	`RngIntElt`
Rational	n/d for literal integers n and d	`5/84, -1/3`	`FldRatElt`
Real	Optional sign, exponent, precision	`-7.123e-5p32`	`FldPrElt`
Literal sequence	Efficient enumerated sequence	`\[1,2]`	`SeqEnum`
Literal cycle	Efficient permutation	`\(1,2)`	`Cyc`

```
> FM<x, y, z, t, u, v, q> := FreeMonoid(7);
> M := quo< FM |
>            x*y*x = y*x*y, z*y*z = y*z*y, x*z = z*x,
>            x*t = t*x, y*t = t*y, t*z*t = z*t*z,
>            u*x = x*u, u*y = y*u, u*z*u = z*u*z,
>            u*t = t*u, x*v = v*x, y*v = v*y,
>            z*v = v*z, t*v = v*t, u*v*u = v*u*v >;
>
> Q5<w> := CyclotomicField(5);
> FA<x, y, z, t, u, v, q> := FreeAlgebra(Q5, M);
> H<x, y, z, t, u >, f := quo< FA | x^2 = (q-1)*x + q,
>     y^2 = (q-1)*y + q, z^2 = (q-1)*z + q, t^2 = (q-1)*t + q,
>     u^2 = (q-1)*u + q, v^2 = (q-1)*v + q, q = FA ! w >;
>
> I := rideal<H | x+1, y+1, z+1, t+1, u+1>;
> Q, im, f := QuotientModule(H, I);
> E6 := sub< Universe(Q) | Q >;
> E6;
Matrix Algebra of degree 27 with 7 generators
over Cyclotomic Field of order 5 and degree 4
```

Table 6. Element constructors.

`elt<` *expr* `|` *expr*$_1$`, ...,` *expr*$_k$ `>`
expr `! [` *expr*$_1$`, ...,` *expr*$_k$ `]`

4.2. ELEMENT CONSTRUCTORS

When viewing *elements* in MAGMA it is important to recall that there are no autonomous elements. An element may only be created in the context of a unique magma. Thus, a polynomial cannot exist as an independent object: it can only exist as an element of a polynomial ring. Consequently, prior to the construction of an element, its parent magma must exist. As many standard algebraic structures have a single fundamental carrier set, we will consider this situation first. There are three standard constructions for elements of a magma M:

- In the case of the most elementary sets, such as the set of integers or the set of boolean values, the elements are self-identifying; their parent magma can be deduced from the syntax. See Table 5.
- If M is a finitely generated algebra defined over the generating set X, an element of M may be specified in the form of an expression built up from the elements of X through application of the fundamental operators of M.
- An element of M may be constructed from a list $a_1, \ldots, a_n$ of more elementary objects using the *element constructor* `elt< >`. This constructor has an abbreviated form `M ! Q`, where Q is the sequence containing $a_1, \ldots, a_n$. See Table 6.

Example:
Consider the 4-dimensional vector space V over the finite field $F = \mathbf{F}_{3^3}$. First, we create this space by applying the function `VectorSpace` to the field $\mathbf{F}_{3^{12}}$ and its subfield F. Note that we apply a functor from the category of finite fields to that of vector spaces.

```
> G<g> := FiniteField(3, 12);
> F<f> := FiniteField(3, 3);
> V, v := VectorSpace(G, F);
> V, v;
Full Vector space of degree 4 over GF(3^3)
Mapping from: FldFin: G to ModTupFld: V
```

Note also that a second value, the map v from G to V, is returned, The map v and its inverse may be used to transfer elements between G and V. We may also create elements of G as polynomials in f and g.

```
> elt< V | 1, f, f^2, f^3 >;
(   1    f  f^2  f^3)
> e := V ! [ f, f^11, f^2+f+1, 7 ];
> e;
(   f f^11  f^6    1)
> e @@ v;
g^86726
> f + f^11*g + (f^2+f+1)*g^2 + 7*g^3;
g^86726
```

Table 7. The map constructors.

`map<` $expr_1$ `->` $expr_2$ `|` *graph* `>`
`pmap<` $expr_1$ `->` $expr_2$ `|` *graph* `>`
`hom<` $expr_1$ `->` $expr_2$ `|` *graph* `>`

In a general multi-sorted algebra A, the concept of an element of A is not well defined. Instead, we have a number of sorts s, each with their associated carrier sets A_s. The problem then arises of specifying an element of a particular carrier set A_s. In MAGMA, such an object may be constructed by means of the third method above (`elt` constructor or coercion of a sequence), where the element target is the appropriate carrier set, rather than A itself.

Example:
In the category of undirected graphs, each magma has two fundamental carrier sets: the *vertex-set* and the *edge-set*. We define the graph of the 4-dimensional cube, create two adjacent vertices p and q and then form the edge e connecting them.

```
> G := KCubeGraph(4);
> V := VertexSet(G); E := EdgeSet(G);
> p := V ! 1; q := V ! 9;
> e := E ! { p, q };
> e;
{1, 9}
> Parent(p), Parent(e);
The vertex-set of graph G
The edge-set of graph G
```

4.3. MAP CONSTRUCTORS

The MAGMA syntax distinguishes three main classes of mappings (see Table 7):

- A *mapping* $f\colon A \to B$, i.e., a subgraph G of the graph $A \times B$, such that every element of A appears in exactly one pair of G.
- A *partial mapping* $f\colon A \to B$, i.e., a subgraph G of the graph $A \times B$, such that every element of A appears in at most one pair of G.
- A *homomorphism* $f\colon A \to B$, where A and B are magmas.

The graph may be specified in various ways. In the case of partial maps and maps, the pairs comprising the graph may be enumerated. However, a more useful method is to give the image of a generic element as in the following example:

```
> R := RealField();
> t := map< R -> R | x :-> x - Sin(x) >;
> t( Pi(R)/2 );
0.57079632679489661923132169162
```

If f is a mapping defined with the general mapping constructor, it is not possible, in general, to compute pre-images with respect to f. However, many particular mappings created by intrinsic functions do allow the computation of pre-images. The image of an

element x is represented syntactically either by $x@f$ or $f(x)$, while the pre-image of x (when defined for the mapping f) is represented by $x@@f$.

When defining a homomorphism, it suffices to specify images for the defining generators of the domain, by virtue of the Homomorphism Representation Theorem. We illustrate this form by defining isomorphisms between Sym(4) defined as a finitely presented group G and as a permutation group H. Notice that the homomorphism can be applied to subgroups as well as to elements.

```
> G<a, b> := Group< a, b | a^2 = b^3 = (a*b)^4 = 1>;
> H<x, y> := PermutationGroup< 4 | (1, 2), (2, 3, 4) >;
> f := hom< G -> H | a->x, b->y >;
> g := hom< H -> G | x->a, y->b >;
> f(a^b);
(1, 3)
> // We construct the Klein 4-group in H and find its image in G
> K := sub< H | (1, 4)(3, 2), (1, 3)(2, 4) >;
> V4 := g(K);
> V4;
Finitely presented group V4 on 2 generators
Generators as words in group G
     V4.1 = (a * b)^2
     V4.2 = (a * b^-1)^2
```

4.4. SETS AND SEQUENCES

The data structures provided in the MAGMA language for aggregating objects together are based on the standard mathematical notions of unordered sets, ordered sets, unordered multisets, ordered multisets (sequences), products and co-products (see Table 8). There are conversion functions between the various kinds of aggregate structures.

Most aggregate data structures in MAGMA are required to satisfy a *homogeneity* principle:

Homogeneity Principle: *The members of any set or sequence must all belong to some common magma U which is known as the universe of the set.*

This principle allows sets to be stored in the form of a hash-addressed table. It also simplifies the storage of elements of sets and sequences, since the parent of each element does not have to be stored with it.

The most important aggregate structures are enumerated sets and sequences. An *enumerated set* is a finite unordered collection of objects of a common magma, stored as a list of elements (usually in the form of a hash table). This allows random insertion and deletion of elements, selection of elements and iteration over the set. An *enumerated sequence* is a finite ordered collection of objects (with possible duplicates), all belonging to a common magma. Its implementation maximizes the speed of accessing the ith element. Enumerated sets and sequences are specified in the same way, the only difference being the type of delimiters used; the general constructor has the following form:

{ U | e : x in E | P } (or the appropriate delimiters)

where U is the common magma to which all elements of the set will belong, e is an

Table 8. Sets, sequences, and other aggregates.

Brackets	*Characteristics*	*Description*
`{ }`	Homogeneous, finite, unordered, no duplicates, fast ∈ test	Enumerated set
`{* *}`	Homogeneous, finite, unordered, fast ∈ test	Enumerated multiset
`{@ @}`	Homogeneous, finite, ordered, no duplicates, fast ∈ test	Indexed set
`{! !}`	Homogeneous, unordered, no duplicates, ∈ test via predicates	Formal set
`[ ]`	Homogeneous, finite, ordered, fast indexed access	Enumerated sequence
`[* *]`	Inhomogeneous, finite, ordered	List
`< >`	Inhomogeneous, finite, ordered (element of Cartesian product)	Tuple
`car< >`	Iterable if components are iterable	Cartesian product
`cop< >`		Co-product

expression usually involving the (local) parameter x which ranges over the magma (or set or sequence) E, which must be enumerable, and restricted by the predicate P usually also involving x. Note that it is quite possible that the expression e defines objects not belonging to the magma E. Often U and the following `|` may be omitted, when MAGMA will be able to determine the common overstructure by itself. If all elements of E should be incorporated, it is permissible to omit the `|` P section of the constructor.

An *indexed set* X is a finite collection of n distinct objects from a common magma, with an associated bijection (the *index map*) between X and the set $\{1, \ldots, n\}$. As in enumerated sets duplicates are discarded, and as in sequences the elements are linearly ordered and can be indexed. Operators are provided that allow the adjunction of further elements to X in such a way that the index map for the larger set is an extension of the index map for X. This data structure provides fast membership testing while associating a unique index with each element. These two facilities are critical when writing efficient code to form the closure of a set under some action.

A *formal set* is a possibly infinite subset of the element-set of a magma M; it is stored as M together with a predicate defining its elements. Membership testing is a simple predicate evaluation, but iteration over formal sets is impossible.

A *tuple* is an element of a Cartesian product of finitely many magmas (or sets). The factors may be chosen quite independently of one another.

We mention a few additional features of the constructors for enumerated sets and sequences:

- Instead of enumerating over a single domain E, it is possible to enumerate over a finite number of domains simultaneously.
- A small set may be created by simply listing the elements within the delimiters: $\{t_1\,,\, t_2\,,\, \ldots\,,\, t_n\}$.
- The *arithmetic progression constructor* $\{a\ \texttt{..}\ b\ \texttt{by}\ k\}$ makes it easy to enumerate the integer-set $\{x : x = a + kn,\ a \leq x \leq b\}$.
- Recursion on sequence constructors is also possible. Inside the sequence constructor, the ith entry of the sequence under construction is `Self(i)`, and the whole sequence computed so far is `Self()`.

It is possible to use *existential and universal quantifiers* on set constructors. Each of the expressions below returns a Boolean value, with the obvious interpretation:

Table 9. Some set functions.

`eq, ne`	Equality testing	`Random(S)`	Random element
`in, notin`	Membership	`Representative(S)`	Representative element
`subset, notsubset`	Subset?	`Include(S, s)`	Include s in S
`join, meet`	Union, intersection	`Exclude(S, x)`	Remove element s
`diff, sdiff`	(symmetric) difference	`exists`	Existential quantification
`IsEmpty`	Empty?	`forall`	Universal quantification

Table 10. Some sequence functions.

`eq, ne`	Equality testing	`Random(S)`	Random element
`in, notin`	Membership	`Representative(S)`	Representative element
`cat`	Concatenate	`Exclude(S, x)`	Remove element x
`IsEmpty`	Empty?	`Include(S, x)`	Include x if not in S
`IsSubsequence(S, T)`	S subsequence of T?	`Remove(S, i)`	Remove ith element
`IsComplete(S)`	All entries defined?	`Insert(S, i, x)`	Put x in ith position
`IsDefined(S, i)`	ith entry defined?	`Append(S, s)`	Append element s
`Undefine(S, i)`	Undefine entry i	`Prune(S)`	Remove last element
`Position(S, x)`	Find index of x in S	`Rotate(S, p)`	Cyclically permute p places
`Sort(S)`	Sort (ordered magma)	`Reverse(S)`	S backwards
`Explode(S)`	Terms of S	`&•S`	Reduce S using operator $\bullet$

```
exists(idfr){ e : x in E | P }
forall(idfr){ e : x in E | P }
```

Moreover, if `exists` returns true, the *idfr* will be assigned the value obtained by evaluating e with x taking the value of an example, and similarly if `forall` returns false, the *idfr* will be assigned the value obtained by evaluating e with x taking the value of a counter-example. As soon as the return value is established, the evaluation of the right-hand side is terminated.

Entries in sequences can be modified by certain procedures, or by accessing them directly using indexing (on the left-hand side of an assignment):

```
> s := [];
> s[3] := 11;
> s;
[ undef, undef, 11 ]
```

As the above example shows, sequences are not necessarily *complete*. That is, using left-hand side indexing, it is possible to create sequences for which not all entries have been assigned values ("sequences with holes"). Predicates are provided to test whether sequences are complete and whether particular entries have been assigned.

The final important concept is that of *reduction*: it allows a (binary) symmetric operator to be applied successively to all elements of a set or sequence. For example, the value of `&*s` is equal to the product of all the terms of s.

We summarize the most important functions on enumerated sets and sequences in Tables 9 and 10.

Example:
We give a concise definition of a function that returns all integer pairs (x, y) with $-10 \leq x \leq 10$ and $0 \leq y \leq 15$ such that $y^2 = x^3 + ax + b$, for a given pair (a, b).

```
> n := func< a, b |
    [ <x, y> : x in [-10..10], y in [0..15] | y^2 eq x^3+a*x+b ] >;
```

We may look for pairs (a, b) in some finite interval such that there are at least six solutions to the given equation:

```
> exists(p){ <a,b> : a, b in [-10..10] | #n(a,b) ge 6 };
true
> p;
<-7, 10>
> n(-7, 10);
[ <-3, 2>, <1, 2>, <2, 2>, <-2, 4>, <-1, 4>, <3, 4>, <5, 10> ]
```

The next construction uses recursion to obtain 50 elements of the Fibonacci sequence:

```
> F := [n in {1, 2} select 1 else Self(n-1)+Self(n-2) : n in [1..50]];
```

We now define a function that returns the degree-k elementary symmetric polynomial in the polynomial ring $R = \mathbf{Z}[x_1, \ldots, x_m]$. This polynomial is simply obtained by summing the elements of the G-set consisting of the images of $x_1 x_2 \cdots x_k$ under the action of the symmetric group of degree m. This would be a one-line function, if we had not insisted in the code below that the indeterminates of R print as the strings `x[1]`, `x[2]` etc.

```
> elSym := function(k, m)
>     R<[x]> := PolynomialRing(Integers(), m);
>     return &+( &*[ R | R.i : i in [1..k] ] ^ Sym(m) );
> end function;
> elSym(2, 4);
x[1]*x[2] + x[1]*x[3] + x[1]*x[4] + x[2]*x[3] + x[2]*x[4] + x[3]*x[4]
```

Note that the universe R was explicitly included in the `return` statement to ensure that the function works for $k = 0$: omitting the universe will lead to a null sequence for $k = 0$ (rather than an empty R-sequence) and the reduction `&*[ ]` is not defined generally.

Finally, we give an example of the use of *Cartesian products* and *co-products*, which are created by means of the `car` and `cop` constructors. Let G be a permutation group with two actions X and Y, represented as G-sets. We want to build the action on the disjoint union of X and Y. Given the action of an element g on each of the components, the function below uses `UniversalMap` to generate the action on the co-product of X and Y. (The function could be applied in a situation where X and Y are two non-compatible and unfaithful actions, and the user wishes to construct from them a faithful action of fairly small degree.)

Table 11. Operators.

Arity	*Operators*	*Remarks*
1	+ - not #	Unary
1	&+ &* &and &or &meet &join &cat	Reduction
2	+ - * div mod / ^	Ordinary arithmetical
2	and or xor	Logical
2	in notin subset notsubset	Membership
2	meet join diff sdiff cat	Set-theoretical
2	eq ne gt lt ge le	Comparative
2	adj notadj	Adjacency
2	=	Relator
2	. ‘ ‘‘	Access
2	@ @@ :-> ->	Map application
2	!! !	Coercion
3	select else	Switch

```
> CompositeAction := function(G, X1, X2)
>     f1 := Action(X1);
>     f2 := Action(X2);
>     XW, I := cop<X1, X2>;
>     X := Set(XW);
>     f := map<
>         car<X, G> -> X | z :-> UniversalMap(XW, X, [h1, h2])(x)
>         where h1 is map<X1 -> X | x :-> f1(x, g)>
>         where h2 is map<X2 -> X | x :-> f2(x, g)>
>         where x is z[1] where g is z[2] >;
>     Y := GSet(G, X, f);
>     return Y;
> end function;
```

4.5. FUNCTIONS AND PROCEDURES

There are three kinds of functions and procedures in MAGMA: user-defined functions and procedures, user intrinsics, and system intrinsics. All of these are *first-class objects* in the MAGMA language. Thus, a function or procedure may be passed as an argument to another function or procedure, returned as the result of a function, or assigned to an identifier, and it can be invoked “on the fly” by means of any expression defining it. All function arguments are passed by value, as a consequence of *call-by-value semantics*, but procedure arguments may be passed by value or by reference. A *function invocation* is an expression, returning at least one value, whereas a *procedure invocation* is a statement, designed to perform input/output or to change the calling context by assigning or reassigning reference arguments. *Parameters* are available for both functions and procedures.

4.5.1. OPERATORS

Operators (see Table 11) are treated by MAGMA as functions with special syntax. If an operator is to be regarded as a normal function, it must be enclosed within ’ characters. The Boolean operators are exceptional in using call-by-name semantics; they only evaluate their operands as required.

Table 12. User-defined functions and procedures.

`function(`arg_1`, ..., `arg_k`)`
statements
`end function`
	`procedure(`arg_1`, ..., `arg_k`)`
	statements
	`end procedure`

`func<` arg_1`, ..., `arg_m `|` $expr_1$`, ..., `$expr_k$ `>`	`proc<` arg_1`, ..., `arg_m `|` *procedure call* `>`

4.5.2. INVOCATION OF FUNCTIONS AND PROCEDURES

The syntax for a function-invocation expression is f`(`arg_1`, ..., `arg_k`)` where f is an expression returning the function, and $arg_1, \ldots, arg_k$ are expressions whose values are taken to be the actual value arguments of the function. If the function has parameters, then the user may assign any non-default parameter values as follows:

$$f\texttt{(}arg_1\texttt{, ..., }arg_k\texttt{ : }param\_idfr_\alpha\texttt{:= }expr_\alpha\texttt{, ..., }param\_idfr_\lambda\texttt{ := }expr_\lambda\texttt{)}$$

Every function invocation in MAGMA returns at least one value. If it appears on the right side of a multiple-value assignment, as an element of a `print` statement or as the sole item in a function `return` statement, then it returns all its values; otherwise, it returns only the first or principal value. (For some intrinsic functions, only the principal value will be printed from a simple `print` statement invoking the function.)

The syntax for a procedure-invocation statement is p`(`arg_1`, ..., `arg_k`);` where p is an expression returning the procedure, and $arg_1, \ldots, arg_k$ are the actual arguments, or, if there are parameters being assigned non-default values:

$$p\texttt{(}arg_1\texttt{, ..., }arg_k\texttt{ : }param\_idfr_\alpha\texttt{:= }expr_\alpha\texttt{, ..., }param\_idfr_\lambda\texttt{ := }expr_\lambda\texttt{);}$$

The arguments of a procedure may be passed either by value or by reference, according to the way the procedure was defined. Value arguments behave in the same way as for functions; i.e., if the ith argument is a value argument, then arg_i may be any expression, and its value will be passed to the procedure. If the ith argument is a reference argument, then arg_i must have the form $\sim idfr$, where *idfr* is an identifier; *idfr* may have a (new) value assigned to it as a result of the procedure invocation.

4.5.3. DEFINITION OF FUNCTIONS AND PROCEDURES

Table 12 lists the constructions for user-defined functions (on the left) and for user-defined procedures (on the right), where $arg_1, \ldots, arg_k$ denote (zero or more) formal arguments. For a function, each argument is an identifier, acting as a value argument. For a procedure, each argument is either an identifier, acting as a value argument, or a tilde $\sim$ followed by an identifier, acting as a reference argument. These constructions are expressions, whose value is a function or procedure; the values of these expressions are usually assigned immediately to identifiers, but may be called directly or passed as an argument to another function or procedure.

Functions must be created in such a way that when they are invoked they will return

one or more values. In the `func` form of a function expression, the return values are the values of $expr_1, \ldots, expr_k$ when the actual arguments given in the function invocation are substituted for the formal arguments. In the other form, the statements in the function body must include one or more statements of the form

`return` $expr_1$`,` `...`, $expr_k$`;`

During the execution of a procedure invocation, any instances of a formal reference argument in the statements of the procedure body are understood to refer to the corresponding actual reference argument. In particular, any changes to a formal reference argument within the procedure body will change the actual reference argument in the calling context. The statement `return`; may be used (zero or more times) within a procedure definition to cause an immediate end to the execution of the procedure and a return to the calling context.

When a function/procedure with parameters is created, the list of arguments must be followed by a colon and a list of *parameter identifiers*, each of which is followed by `:=` and an expression, as follows:

arg_1`,` `...`, arg_k `:` $param\_idfr_1$`:=` $expr_1$`,` `...`, $param\_idfr_n$ `:=` $expr_n$

The default values of the parameters are the values of the corresponding expressions. The parameters will normally occur in the statement body of the function/procedure definition (or the right-hand side of the `func` or `proc` constructor). They are classed as value identifiers, so they may not be changed within the statement body. However, the value used for the parameters depends on the function/procedure invocation. If the invocation includes parameters and values then the parameters will have those values; otherwise, they will have the default values.

Recursive functions and procedures pose a particular problem in Magma since functions and procedures are first-class objects. Not every function or procedure is assigned to an identifier. Even if it is, the identifier cannot be used within the function or procedure since, by the assignment rules, identifiers appearing on the left-hand side of an assignment statement are not considered to have a value on the right-hand side, unless they were previously assigned a value. Magma solves this problem by using the `$$` pseudo-identifier as a placeholder for the function/procedure value denoted by the function/procedure expression inside which the `$$` occurs.

Lexical scoping of identifiers is employed. Formal value arguments, formal reference arguments and parameters automatically have local scope. A "first-use" rule decides which other identifiers have local scope: if the first textual use of an identifier inside a function or procedure body is on the left-hand side of a `:=` symbol, then the identifier is considered to be local; otherwise, its value is imported from the environment (the value of the identifier in the calling context when the function/procedure is defined, not when it is invoked). The calling context cannot be changed from within the function/procedure, except by the standard means of reference arguments. The problem of mutual recursion is handled through the provision of a `forward` statement that allows reference to functions or procedures prior to their lexical appearance.

4.5.4. USER INTRINSICS AND PACKAGE FILES

A *user intrinsic* is a special kind of user-defined function/procedure that has a *signature* attached to it, giving the categories of the arguments and return values. The syntax for a user intrinsic is a modified form of the function/procedure expression, bounded by an `intrinsic` statement. A file containing one or more intrinsics constitutes a *package file*; also present may be some assigned values that are private to the package file. When the package file is *attached* to MAGMA, the user intrinsics are incorporated into the system, together with the kernel intrinsics. Identifiers private to a package file may be accessed by other packages if they are explicitly imported using the `import` statement.

4.6. COMMON SUBEXPRESSION EVALUATION

The `where` clause may be attached to the right end of a list of expressions $expr_1, \ldots, expr_n$, in order to perform common subexpression evaluation. The general form of an expression-list ending with a `where` clause is

$expr_1$`, ...,` $expr_n$ `where` $idfr_1$`, ...,` $idfr_k$ `is` $expr_w$

The expression $expr_w$ must have at least k return values; in most cases, it has one return value, and there is only one identifier.

To evaluate the whole expression-list, MAGMA temporarily assigns to $idfr_1, \ldots, idfr_k$ the first k return values of $expr_w$. It then evaluates the expressions $expr_1, \ldots, expr_n$, substituting the values for $idfr_1, \ldots, idfr_k$ whenever they occur in the expressions. The scope of $idfr_1, \ldots, idfr_k$ is limited to the expression-list.

Within the predicate of a set/sequence constructor, the `where` clause has an additional property: $idfr_1, \ldots, idfr_k$ may also be used in the expression used to calculate the set or sequence elements. In this case, the scope of $idfr_1, \ldots, idfr_k$ is limited to the constructor.

4.7. STATEMENTS

The principal kinds of statements in MAGMA are assignment statements, input and output statements, iterative statements, conditional statements, and procedure invocations. The last of these has already been discussed; the others will be summarized below.

4.7.1. ASSIGNMENT STATEMENTS

There is no assignment expression in MAGMA, and it is not possible to print values at the same time as they are assigned. Variable identifiers and reference identifiers may be the targets of an assignment, but not value identifiers.

Functions may return $m \geq 1$ values; the first k (where $1 \leq k \leq m$) can be assigned to identifiers by the *multiple-value assignment*. It is possible to ignore some of the first k return values by using the special throwaway identifier `_`, i.e., the underscore character.

The *mutation assignment statement* *idfr* `•:=` *idfr*`;` is equivalent to *idfr* `:=` *idfr* `•` *expr*`;` in its overall effect. However, the mutation version may sometimes result in faster execution since it allows the obvious optimization. Mutation of certain objects (such as sequences) is also possible.

It is often desirable for the user to provide names for the generators, and so the

Table 13. Assignment statements.

Statement	*Description*	*Remarks*
idfr `:=` *expr*`;`	Simple assignment	
idfr$_1$`, ...,` *idfr*$_k$ `:=` *expr*`;`	Multiple-value assignment	Use `_` for throwaway
idfr •`:=` *expr*`;`	Mutation assignment	• is a binary operator
idfr`<`*idfr*$_1$`, ...,` *idfr*$_k$`>` `:=` *expr*`;`	Generator name assignment	

Table 14. Input and output statements.

`print` *expr*$_1$`, ...,` *expr*$_k$`;`	Print values of expressions
expr$_1$`, ...,` *expr*$_k$`;`	As above [not in function/procedure definition]
`print` *expr*$_1$`, ...,` *expr*$_k$`:` *level*`;`	*level* is one of `Minimal`, `Default`, `Maximal`, `Magma`
expr$_1$`, ...,` *expr*$_k$`:` *level*`;`	As above [not in function/procedure definition]
`printf` *expr*`,` *expr*$_1$`, ...,` *expr*$_k$`;`	Print values of expressions in *expr* format
`read` *idfr*	Assign the next input line to *idfr* as a string
`read` *idfr*`,` *expr*	As above, giving value of *expr* as prompt
`readi` *idfr*	Assign the (legal) next input line to *idfr* as an integer
`readi` *idfr*`,` *expr*	As above, giving value of *expr* as prompt

generator assignment statement provides a mechanism for the user to assign a magma to an identifier, and simultaneously to specify particular identifiers (and printnames, for free magmas and their quotients) for the generators of a magma. See page 248.

4.7.2. INPUT AND OUTPUT STATEMENTS

The word `print` is optional, except within the definition of a function or procedure. Therefore, the user can obtain the evaluation of a list of expressions at the command-line simply by typing the comma-separated expressions and following the list with a semicolon.

The optional qualifier *level* may be used to control the amount and format of information printed for the designated objects. For example, the qualifier value `Magma` causes the values to be printed in MAGMA-readable format, so that they may be used as MAGMA input.

There is also a `printf` statement for formatted output. The *expr* immediately following `printf` must evaluate to a string. The statement prints this string, substituting the value of *expr*$_i$ at the ith occurrence of `%o` within the string.

A number of other string and file-handling facilities are provided for input and output.

4.7.3. ITERATIVE STATEMENTS

The `while` and `repeat` statements have the usual interpretations. In the `for` statement, the *idfr* is a local-scope value identifier that assumes successively the values of the elements of the domain. The domain may be any iterable magma (i.e., a finite algebraic structure or aggregate whose elements can be enumerated by MAGMA) and if it is an ordered aggregate, such as a sequence, then iteration will occur in index order.

Table 15. Iterative statements.

`while` *expr* `do`
  statements
`end while;`
	`repeat`
	  statements
	`until` *expr*`;`
		`for` *idfr* `in` *domain* `do`
		  statements
		`end for;`

Table 16. Conditionals.

Construction	*Remarks*

`if` *expr* `then` *statements*
  `elif` *expr* `then` *statements*
  `else` *statements*
`end if;`
	conditional statement
	Optional, may be repeated
	Optional

`case` *expr*
  `when` $expr_1$`, ...,` $expr_k$`:` *statements*
  `else` *statements*
`end case;`
	case-statement
	Optional, may be repeated
	Optional

expr `select` $expr_1$ `else` $expr_2$	**conditional expression**
`case<`*expr* `|` $expr_1$`:`$expr'_1$`, ...,` $expr_k$`:`$expr'_k$`, default:`$expr_d$`>`	**case-expression** `default` is compulsory

4.7.4. CONDITIONAL STATEMENTS AND EXPRESSIONS

As shown in Table 16, the MAGMA language offers both conditional statements and conditional expressions: a standard `if` statement, a `case` statement, a `select` expression, and a `case` expression. The two conditional expressions use call-by-name semantics.

5. Closing Remarks

This paper has attempted to show how ideas from universal algebra and category theory can provide the general foundations for an algebraic programming language. The organization of algebraic structures, firstly into varieties, and then within varieties into categories, has shown itself to be a natural hierarchy for algebraic computation. As more exotic classes of structures are included, we find that the model holds up extremely well. The use of algebraic data structures such as sets, sequences and mappings provides a very concise and natural means of specifying algebraic computations.

An interpreter and run-time system for the MAGMA language have been developed by Graham Matthews and Allan Steel. MAGMA V1 was released at the end of 1993 and V2 in 1996. The language will be extended in the near future to support user-defined categories and varieties.

An elementary overview of the system is given in Cannon and Playoust (1996a). A full description of the language and installed categories may be found in Cannon and Playoust (1996b, c) and Bosma and Cannon (1996). Some aspects of the implementation are touched on in Bosma *et al.* (1994). Applications of the system are described in the

proceedings of the first MAGMA conference (this issue), the proceedings of the second MAGMA conference (to appear as a Special Issue of *JSC*) and in Bosma *et al.* (1995).

References

Bosma, W., Cannon, J.J. (1996). *Handbook of Magma Functions*. Sydney: School of Mathematics and Statistics, University of Sydney.

Bosma, W., Cannon, J.J., Matthews, G. (1994). Programming with algebraic structures: design of the Magma language. In Giesbrecht, M., ed., *Proc. of the 1994 International Symposium on Symbolic and Algebraic Computation, Oxford*, 52–57. Association for Computing Machinery.

Bosma, W., Cannon, J.J, Playoust, C., Steel, A. (1995). *Solving Problems with Magma*. Sydney: School of Mathematics and Statistics, University of Sydney.

Bourbaki, N. (Nouvelle Édition 1970). *Algèbre I: Chapitres 1 à 3 (Éléments de Mathématique)*. Paris: Hermann.

Burstall, R.M., Gougen, J.A. (1981). An informal introduction to specifications using CLEAR. In Boyer, R.S., Moore, J.S., eds, *The Correctness Problem in Computer Science*. London: Academic Press, 185–213.

Butler, G., Cannon, J.J. (1989). Cayley version 4: The user language. [The early name for Magma was Cayley V4.] In Gianni, P., ed., *Proc. of the 1988 International Symposium on Symbolic and Algebraic Computation, Rome, July 4–8, 1988, LNCS 358*, 456–466. New York: Springer.

Butler, G., Cannon, J.J. (1990). The design of Cayley, a language for modern algebra. In Miola, A., ed., *Design and Implementation of Symbolic Computation Systems, LNCS 429*, 10–19.

Cannon, J.J., Playoust, C. (1996a). *Magma: A New Computer Algebra System*. Euromath Bulletin, **2**, 1.

Cannon, J.J., Playoust, C. (1996b). *Algebraic Programming with Magma: The Language*. Springer-Verlag, To appear.

Cannon, J.J., Playoust, C. (1996c). *Algebraic Programming with Magma: The Categories*. Springer-Verlag, To appear.

Gougen, J.A. (1989). Principles of Parameterized Programming. In Biggerstaff, T., Perlis, A., eds, *Software Reuseability, Vol 1: Concepts and Models*. Reading, MA: ACM Press, Addison-Wesley, 159–225.

Hearn, A.C, Schrüfer, E. (1995). A computer algebra system based on order-sorted algebra. *J. Symbolic Comput.* **19**, 65–77.

Jenks, R.J, Sutor, R. (1992). *AXIOM: The Scientific Computation System*. New York: Springer-Verlag.

Mac Lane, S. (1971). *Categories for the working mathematician*. New York: Springer-Verlag.

Meinke, K., Tucker, J.V. (1992). Universal Algebra. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds, *Handbook of Logic in Computer Science* **1**. Oxford: Clarendon Press, 189–411.

Originally received 10 January 1996
Accepted 7 November 1996