Appendix: The Magma Language

Geoff Bailey

School of Mathematics and Statistics
The University of Sydney
Sydney, Australia

Introduction

The following is a brief introduction to the Magma language. It is not a comprehensive description of the language, nor is it a programming tutorial (although, inevitably, it has much in common with one). Instead, it is intended to describe the basic language features that account for the vast majority of code written in the Magma language. For a more in-depth treatment, see the language section of the Magma Handbook [1].

Some of the examples below will use language features that are not explained until a later point; it is hoped in each case that the meaning will be clear enough for this not to cause problems. A certain familiarity with basic programming concepts such as expressions and statements will be assumed.

1 Basics

1.1 Statements and printing

Throughout the book, code fragments will use a format similar to the example below. Lines beginning with > (the Magma prompt) indicate input to Magma (the prompt is *not* part of the input), while lines without the prompt indicate output produced by the preceding instructions.

```
>    1 + 1
>    + 1;
          3
```

This trivial example already illustrates two important features of the language. Firstly, statements in Magma must end with a semicolon. Thus, no output occurred after the first line of input, as the statement was still incomplete at that point. Secondly, when a Magma statement is an expression, then the result of evaluating that expression will be printed. If multiple expressions

occur (separated by commas) in a single statement, then they will each be printed, separated by spaces.

```
> "You are number", 6;
      You are number 6
```

Note the use of a *string* (text surrounded by double quotes) in this example. A string prints as the text it contains.

There are also two explicit print statements in Magma.

print *value*, . . . , *value*
printf *format-string*, *value*, . . . , *value*

The **print** statement prints its values, exactly as would occur if the keyword **print** were omitted. It is thus rarely seen, although its use can improve code clarity. (It is also useful when debugging, as it can be easily searched for and removed once the problem has been solved.)

The **printf** statement behaves similarly to the corresponding C function. The string argument is printed, with two types of substitutions made. Each instance of '%o' in the string is replaced by the output from printing the corresponding value argument to **printf** (the first instance uses the first value argument, and so on). Also, two-character strings beginning with backslash are changed in the usual manner. For example, '\t' is replaced with a tab character and '\n' is replaced with a newline character.

```
> printf "%o green %o...\n", 99, "bottles";
      99 green bottles...
> print "...hanging on the wall";
      ...hanging on the wall
```

1.2 Comments

A *comment* is a way of specifying that some specific text should be ignored by Magma. There are two types of comments in Magma, matching those of the C and C++ languages.

// *comment text*
/* *comment text* */

The short form // applies to one line only; all text following the // on the same line is ignored. The long form /* . . . */ can extend over multiple lines; all text between the /* and the */ is ignored.

```
> 1 + 1    // ; this semicolon is ignored
> /* As will be the one on the next line
> +1;
> */
> + 1;    // Now the statement is finally finished
      3
```

1.3 Intrinsic functions

A *function* is an object that uses some values (the *arguments*) to perform certain operations and produce some other values (the *return values*). A *procedure* is similar except that it has no return values. In Magma, a function or procedure is called by typing its name, followed by the values of its arguments (if any) enclosed within parentheses.

name(*arguments*)

If there are multiple arguments then commas are used to separate them. Magma contains a great many inbuilt, or intrinsic, functions and procedures, which are called *intrinsics* for short.

The names of intrinsics are usually descriptive of their purpose. For instance, the intrinsic GreatestCommonDivisor computes the greatest common divisor of its two arguments.

```
> GreatestCommonDivisor(91, 224);
7
```

Many intrinsics have synonyms that reflect different spellings, usages, or convenient shorthand forms. For example, the names GCD and Gcd are synonyms of GreatestCommonDivisor.

Functions may have more than one return value. One example is Quotrem, which returns the quotient and remainder (respectively) arising from Euclidean division of its arguments.

```
> Quotrem(16, 3);
5 1
```

If a function returns multiple values but not all are used then the unused return values are silently discarded. For example, if a function call is part of an expression then only the first return value is used.

```
> 3*Quotrem(16, 3);
15
```

Functions and procedures are described further in a later section which also describes how it is possible to create new functions and procedures, and how procedures may modify their arguments.

1.4 Arithmetic

Magma includes the standard arithmetic operators $+$, $-$, and $*$, as well as parentheses () for grouping.

```
> (2 + 3)*(4 - 1);
15
```

There are two versions of division, however, depending on what sort of object one is working with. The operators **div** and **mod** apply when working over Eu-

clidean rings (such as $\mathbb{Z}$), and return the quotient and remainder (respectively) of the division.

```
> 16 div 3, 16 mod 3;
    5 1
```

The operator / applies when working over fields, where the (exact) quotient is returned. It can also be used when working over rings; in this case the result will be returned as an element of the appropriate field of fractions.

```
> 16.0 / 3.0;
    5.3333333333333333333333333333
> 16 / 3;
    16/3
```

Note that this latter example took two elements from the Euclidean ring $\mathbb{Z}$ but returned a result in its field of fractions $\mathbb{Q}$.

```
> Parent(16);
    Integer Ring
> Parent(16 / 3);
    Rational Field
```

(The intrinsic Parent returns the structure that contains the specified object.)

The other important arithmetic operator is ^, which is used for exponentiation. However, in this book exponentiation has been indicated in the conventional way as a superscript and the ^ is not shown. So, whereas one would type 4^2 into Magma, in the examples it would appear as follows.

```
> 4^2;
    16
```

1.5 Variables, assignment, mutation, and where

A *variable* is a name that may have a value associated with it. Variables can be given a value by using an assignment statement.

name := *value*;
name, ..., *name* := *values*;

(As is common with interpreted languages, variables in Magma do not need to be declared; they are defined by use and the type of a variable is that of the value it currently contains.)

```
> seconds := 24*60*60;    // seconds per day
> seconds;
    86400
```

Magma also has *mutation* operations, which are a convenient shorthand for the most common kind of variable modification. These have the form

variable op:= *value*;

and are equivalent to the longer form *variable* := *variable op value*. Here most binary operators may be used in place of *op*.

```
> seconds *:= 365 + 1/4;    // Average seconds per year
> seconds;
    31557600
> seconds /:= 12;    // Average seconds per month
> seconds;
    2629800
```

The special symbol _ may appear on the left hand side of an assignment statement instead of a variable, and indicates that the value that would have been assigned should be discarded instead. It is most commonly used to get rid of unwanted return values from functions.

```
> p := 17;
> x := 12;
> _,_,xinv := XGCD(p, x);    // Extended GCD
> x*xinv mod p;
    1
```

There is a special kind of 'temporary' assignment available in Magma, using the **where** clause. There are two minor variations of this clause.

expression **where** *name* := *value*
expression **where** *name* **is** *value*

In each case the expression is evaluated as though *name* were a variable with the given value. The previous value of the variable (if any) is unchanged by this process; the changed value applies only while evaluating the expression.

```
> x*xinv mod p where p is 13;
    7
> p;
    17
```

The use of **where** is particularly desirable when a lengthy or computationally expensive sub-expression occurs multiple times within the one expression.

```
> Modexp(3, p-1, p) where p is 2^n + 1 where n is 2^5;
    3029026160
```

(The intrinsic Modexp(x, n, m) efficiently computes x^n **mod** m.) Note the use of more than one **where** in this example.

1.6 Generators and generator assignment

Many structures in Magma are generated by a few special elements. These *generators* can be retrieved with the dot operator.

structure . *integer*

The integer must be a valid generator number for the given structure. These numbers range from 1 to the number of generators.[1]

```
> K := QuadraticField(5);    // sets K to ℚ[√5], with generator √5
> α := K.1;
> α;
    K.1
> α^2-5;
    0
```

Since it is extremely common to want to use the generators after defining the structure, Magma provides a convenient way to assign these generators to variables when the structure is created.

variable<*names*> := *structure*;

This construct sets the variables specified by *names* to the corresponding generators of the structure. It also has another important effect: The names provided will be used when printing elements of the structure.

```
> P<x> := PolynomialRing(Integers());
> (x+1)*(x-1);
    x^2 - 1
> u := x;
> (u^3-1) div (u-1);
    x^2 + x + 1
```

1.7 Coercion

Given two mathematical structures, it is often the case that there is a natural choice of map from one to the other. In Magma, the act of applying this map is called *coercion*, and is achieved by using the coercion operator ' ! '.

structure ! *element*

The appropriate map is applied to the element, producing a value lying in the specified structure. In Magma terminology, the element has been *coerced* into the structure.

[1] In certain cases other values are allowed; for instance, if the structure is a group then the negative numbers yield the inverses of the generators and zero gives the identity element.

```
> K := GF(7);    // GF(q) produces the finite field with q elements
> x := K ! 23;
> x;
    2
> Parent(x);
    Finite field of size 7
```

Note that after the coercion we can perform operations on x that might return different values or just be nonsensical if attempted prior to the coercion.

```
> Order(x);
    3
```

Coercion is of fundamental importance in Magma, since it allows the easy transfer of objects from one mathematical structure to a related one where questions of interest may more readily be answered.

1.8 Boolean expressions

Magma has the inbuilt Boolean constants TRUE and FALSE. The logical operators ***and***, ***or***, ***not*** and ***xor*** operate on these in the expected manner. Boolean values may arise in other ways; for instance, the six relational operators ***eq***, ***ne***, ***lt***, ***le***, ***gt***, and ***ge*** take two comparable objects and return the truth-value of the appropriate comparison. These actions are summarised in the following table.

Operator	Usage	Meaning
not	***not*** *a*	TRUE if *a* is FALSE, else FALSE
and	*a* ***and*** *b*	TRUE if both *a* and *b* are TRUE, else FALSE
or	*a* ***or*** *b*	TRUE if either *a* or *b* is TRUE, else FALSE
xor	*a* ***xor*** *b*	TRUE if exactly one of *a* and *b* is TRUE, else FALSE
eq	*x* ***eq*** *y*	TRUE if *x* is equal to *y*, else FALSE
ne	*x* ***ne*** *y*	TRUE if *x* is not equal to *y*, else FALSE
lt	*x* ***lt*** *y*	TRUE if *x* is less than *y*, else FALSE
le	*x* ***le*** *y*	TRUE if *x* is less than or equal to *y*, else FALSE
gt	*x* ***gt*** *y*	TRUE if *x* is greater than *y*, else FALSE
ge	*x* ***ge*** *y*	TRUE if *x* is greater than or equal to *y*, else FALSE

Functions may also return Boolean values, of course.

```
> G<a,b> := Sym(4);    // Symmetric group on four elements
> Order(a) eq 4 or Order(b) eq 4;
    true
> IsAlternating(G);
    false
> IsSymmetric(G) and IsSoluble(G);
    true
```

1.9 Conditionals: if and select

To perform different commands based on some condition, an **if** statement is used.

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

If the condition is TRUE then the statements between **then** and **else** will be executed; otherwise, the statements between **else** and **end if** will be executed.

```
> p := 341;
> if 2^(p-1) mod p eq 1 then
>     p, "is a pseudo-prime to base 2";
> else
>     p, "is definitely composite";
> end if;
    341 is a pseudo-prime to base 2
```

In this example the expression evaluated to TRUE, so the statements between **then** and **else** were executed.

The **else** section is optional; without such a section no statements will be executed if the condition is FALSE. Also, an adjacent **else** and **if** can be combined together into **elif**; the reason to do this is that then only one **end if** will be required. For long chains of **if**...**else if**... sequences this is highly desirable.

```
> p := 191;
> if not IsPrime(p) then
>     p, "is not prime";
> elif IsPrime(2*p + 1) then
>     p, "is a Germain prime";
> else
>     p, "is prime (but not a Germain prime)";
> end if;
    191 is a Germain prime
```

Sometimes it is desirable to have an expression whose value depends on some condition. This is particularly useful when creating recursive sequences (described later), or in **where** clauses, or even just as a shorter form than the corresponding **if** statement would be. The **select** expression is used for this purpose.

condition **select** $expression_1$ **else** $expression_2$

If the condition is TRUE then the entire expression has the value of $expression_1$, otherwise it has the value of $expression_2$.

```
> m := 17;
> x := 56 mod m;
> modnegx := x eq 0 select 0 else m - x;
> modnegx;
    12
```

2 Sets and sequences

Sets and sequences are extremely important in Magma. Sequences in particular are the second-most commonly used type of object (integers are the most common), and it is rare to find a significant piece of Magma code that does not use them. An understanding of their use often leads to compact yet expressive and understandable programs.

Sets and sequences are collections of objects belonging to the same structure (this structure is said to be the *universe* of the set or sequence). Sets are unordered collections, and so may not contain duplicated values.[2] In contrast, sequences are ordered collections and duplications are allowed.

Sets and sequences use brackets in their creation and printing; the brackets involved are { } for sets and [ ] for sequences.

2.1 Creation of sets and sequences

The simplest way to create a set or sequence is to explicitly list its elements.

```
> [ FALSE, TRUE, TRUE, FALSE, TRUE, FALSE, TRUE ];
    [ false, true, true, false, true, false, true ]
> { 2, 7, 1, 8, 2, 8 };
    { 1, 2, 7, 8 }
```

In many cases, explicit enumeration of elements in this manner is not necessary because one of the special constructors described below can be used. Magma has two special constructors for sets and sequences; the first has the form

{ *a*..*b* **by** *k* }
[*a*..*b* **by** *k*]

where *a*, *b*, and *k* are integers. This creates the set or sequence containing the values from the arithmetic progression *a*, *a*+*k*, *a*+2*k*, ... and bounded by *b*. (It is not required that *b* be part of the progression.) The **by** *k* part may be omitted, in which case the increment is taken to be 1.

The second special constructor has the following form.

{ *expression in x* : *x* ***in*** *D* | *condition on x* }
[*expression in x* : *x* ***in*** *D* | *condition on x*]

Using this form, the result consists of the values of *expression in x* evaluated for each *x* in *D* such that the *condition on x* evaluates to TRUE.

```
> [ p : p in [1..100] | IsPrime(p) ];
    [ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
    53, 59, 61, 67, 71, 73, 79, 83, 89, 97 ]
```

[2] There is a related *multiset* type in Magma that allows duplicated values; it will be briefly described later.

The condition part of the constructor may be omitted; this is equivalent to using a condition of `TRUE`.

```
> K := GF(7);
> { x^2 : x in K };
    { 0, 1, 2, 4 }
```

It is possible to iterate with respect to more than one variable in the same constructor.

```
> V := VectorSpace(K, 2);
> m := Matrix(K, [ [1, 2], [4, 1] ]);
> m;
    [1 2]
    [4 1]
> { v : v in V, a in K | not IsZero(a*v) and v*m eq a*v };
    {
        (4 1),
        (3 6),
        (2 4),
        (6 5),
        (1 2),
        (5 3)
    }
```

There is actually one more part to these special constructors. The universe of the set or sequence may be specified, and elements arising during the construction will then be coerced into this universe.

{ *universe* | *rest of constructor* }
[*universe* | *rest of constructor*]

This is preferable to using explicit coercion for clarity reasons, and also because it ensures that the universe is correctly set even when the resulting set or sequence is empty.

```
> T := { RationalField() | x : x in K | x^2 eq 3 };
> T;
    {}
> Universe(T);
    Rational Field
```

In the context of a sequence constructor, the function Self returns the specified element of the sequence that is being created. In conjunction with the **select** expression to handle base cases, Self allows sequences to be created recursively.

```
> [ n le 2 select 1 else Self(n-1) + Self(n-2) : n in [1..15] ];
    [ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610 ]
```

The set and sequence constructors can clearly be used to convert between the two types by explicit iteration. Another way to achieve the same goal is to use one of the intrinsics SetToSequence or SequenceToSet.

```
> S := [ Order(x) : x in K | x ne 0 ];
> S;
    [ 1, 3, 6, 3, 6, 2 ]
> T := { x : x in S };
> T;
    { 1, 2, 3, 6 }
> SequenceToSet(S);
    { 1, 2, 3, 6 }
> SetToSequence(T);
    [ 1, 2, 3, 6 ]
```

Many mathematical objects can be naturally viewed as including a sequence of elements from some underlying structure. For example, polynomials have their coefficients and geometric points have their coordinates. In such cases the intrinsic ElementToSequence, usually abbreviated to Eltseq, returns this sequence.

```
> Zx<x> := PolynomialRing(Integers());
> f := (x+1)^2*(x-2);
> f;
    x^3 - 3*x - 2
> Eltseq(f);
    [ -2, -3, 0, 1 ]
```

(Note that for polynomials the ensuing sequence starts with the coefficient of the constant term and finishes with the coefficient of the leading term.)

Elements can also be created by coercing such sequences into the parent structure of the original object.

```
> Zx ! [ 41, 1, 1 ];
    x^2 + x + 41
> V ! [3, 5];
    (3 5)
```

There are two other set-like types in Magma: *indexed sets* and *multisets*. These can be created like normal sets, except that they use different brackets—indexed sets use {@ @} and multisets use {* *}. Indexed sets are ordered but may not contain duplicates, while multisets are unordered and may contain duplicates (each element has an associated multiplicity). Most operations that apply to normal sets also apply to indexed sets and multisets. Additionally, indexed sets support many operations that apply to sequences.

2.2 Indexing and indexed assignment

Sequences support an indexing operation (using [ ]) to extract particular elements. Valid indices are integers from 1 to the number of elements in the sequence (also called its *length*).

```
> S := [ "r", "e", "l", "a", "t", "i", "n", "g" ];
> S[4];
      a
```

It is also possible to index one sequence by another sequence. In this case the result will be a sequence whose elements are the result of indexing the first sequence by the elements of the second.

```
> indices := [ 6, 7, 5, 2, 8, 1, 4, 3 ];
> S[indices];
      [ i, n, t, e, g, r, a, l ];
> S[ [5, 1, 6, 4, 7, 8, 3, 2] ];
      [ t, r, i, a, n, g, l, e ];
```

Indexing may also be used to set specific elements of a sequence.

```
> S[5] := "x";
> S;
      [ r, e, l, a, x, i, n, g ]
```

If a sequence contains sequences (or other indexable objects) then multi-level indexing can take place.

```
> M := [ [1, 2], [3, 4] ];
> M[2];
      [ 3, 4 ]
> M[2][1];
      3
```

In such situations there is a valid alternate syntax, in which the indices are flattened into a single list.

```
> M[2, 1];
      3
```

Note that this is *not* the same as indexing by a sequence.

```
> M[[2, 1]];
      [
          [ 3, 4 ],
          [ 1, 2 ]
      ]
```

When setting elements of a sequence, the index used may be larger than the current size of the sequence. If so, the sequence is extended to include the new element at the appropriate index.

```
> M[3] := [ 5, 6 ];
> M;
[
    [ 1, 2 ],
    [ 3, 4 ],
    [ 5, 6 ]
]
```

2.3 Operators on sets and sequences

There are a few important operators that can be applied to sets and/or sequences. One with more general applicability is #, which in Magma returns the cardinality of an object. In the case of sets or sequences this is the number of elements. These operators are summarised in the following table.

Operator	Usage	Meaning
#	#*S*	the number of elements in *S*
in	*x* ***in*** *S*	TRUE if the element *x* is in *S*, else FALSE
notin	*x* ***notin*** *S*	TRUE if the element *x* is not in *S*, else FALSE
cat	S_1 ***cat*** S_2	the concatenation of sequences S_1 and S_2
join	S_1 ***join*** S_2	the union of sets S_1 and S_2
meet	S_1 ***meet*** S_2	the intersection of sets S_1 and S_2
diff	S_1 ***diff*** S_2	the set of elements in set S_1 but not in set S_2
sdiff	S_1 ***sdiff*** S_2	the symmetric difference of sets S_1 and S_2
subset	S_1 ***subset*** S_2	TRUE if S_1 is a subset of S_2, else FALSE

There is an extremely important class of operators called *reduction operators*. These have the form &*op* and act on a set or sequence to reduce it to a single element by repeated application of the binary operator *op*. Not all binary operators have a corresponding reduction operator; the valid reduction operators are as follows.

&+ &∗ &***and*** &***or*** &***meet*** &***join*** &***cat***

If *S* contains the elements $\alpha_1, \ldots, \alpha_n$ in that order[3] then

$$\&op\ S = (\ldots((\alpha_1\ op\ \alpha_2)\ op\ \alpha_3)\ldots)\ op\ \alpha_n.$$

Note that most of these operators are commutative (at least under common circumstances). The exception is &***cat***, which concatenates the elements of the set/sequence.

[3]For sets, the order is the internal iteration order, which corresponds to the order used when printing.

```
> &+[ k^2 : k in [1..24] ];
    4900
> F11 := GF(11);
> _<t> := PolynomialRing(F11);
> &*{ t - a : a in F11 | not IsSquare(a) };
    t^5 + 1
> P<a,b,c> := PolynomialRing(Integers(), 3);
> sympols := [ &+[ &*S : S in Subsets({a,b,c}, k) ] : k in [0..3] ];
> sympols;
    [
        1,
        a + b + c,
        a*b + a*c + b*c,
        a*b*c
    ]
> &and [ IsSymmetric(pol) : pol in sympols ];
    true
> &cat [ "Hello,", " ", "world!" ];
    Hello, world!
> &meet [ { n : n in [2..10^4] | Modexp(x,n,n) eq x } : x in {2,3,5} ]
>         diff { n : n in [2..10^4] | IsPrime(n) };
    { 561, 1105, 1729, 2465, 2821, 6601, 8911 }
```

When a set or sequence is empty then it may not be obvious what the result of applying a reduction operator to it should be. The guiding principle is that the result should be consistent, in the sense that (&*op S*) *op x* should produce *x* when *S* is empty. Thus &+ produces 0 and &∗ produces 1, &***and*** produces TRUE and &***or*** produces FALSE, and &***join*** and &***cat*** produce empty objects of the appropriate types. It is not permissible to call &***meet*** on an empty structure.

In the case of &+ and &∗, if they are called on an empty set or sequence then the universe must have already been set. This is so that Magma can determine in which structure the result should lie.

2.4 Important set and sequence intrinsics

The commonly used intrinsics for modifying sets and sequences are summarised in the table below. The intrinsics Include and Exclude can be used on both types, while Append, Prune, and Remove apply to sequences only. Each of these intrinsics has both a procedural and a functional form; the usage with tilde (∼) is the procedural version that modifies the set or sequence directly, while the functional version returns a new object and leaves the original unchanged. For more information on procedures, see the later section on creating functions and procedures.

Intrinsic	Usage	Meaning
Include	Include($\sim S$, x) Include(S, x)	Puts x into S
Exclude	Exclude($\sim S$, x) Exclude(S, x)	Removes the first occurrence of x from S
Append	Append($\sim S$, x) Append(S, x)	Appends x to the sequence S
Prune	Prune($\sim S$) Prune(S)	Removes the last element from the sequence S
Remove	Remove($\sim S$, i) Remove(S, i)	Removes the ith element from the sequence S

Another important intrinsic is Index(S, x), which returns the index of the first occurrence of x in the sequence S (or 0 if x is not in S). Thus, assuming that the sequence S contains x, the effects of Exclude(S, x) and Remove(S, Index(S, x)) are the same.

```
> S := [ 1, 4, 9, 16 ];
> Index(S, 16);
    4
> Remove(~S, 4);
> S;
    [ 1, 4, 9 ]
> Append(S, 2);    // Functional version—does not change S
    [ 1, 4, 9, 2 ]
> S;
    [ 1, 4, 9 ]
> T := { "e", "a", "t" };
> Include(~T, "r");
> T;
    { t, e, a, r }
> Exclude(T, "e");
    { t, a, r }
```

Another handy intrinsic is Explode, which returns the elements of a sequence. This may not seem like it accomplishes much, but it is a great convenience when assigning these elements to variables. Without it, each assignment would have to be written explicitly, index and all.

```
> first, second, third := Explode([ 3, 1, 4 ]);
> first; second; third;
    3
    1
    4
```

2.5 Quantifiers: exists and forall

There are a few ways to test whether some or all elements of a set satisfy a certain property. For example, one could construct a sequence of Boolean values indicating whether each element satisfies the property, and then apply one of the reduction operators &***or*** or &***and*** (as appropriate) to this sequence. A better idea would be to use a loop (described later) over the set.

However, both of these approaches require the set itself to have been fully constructed, which may be needlessly time-consuming. A better method is available in Magma, using one of the quantifiers **exists** or **forall**. These allow the test and the set construction to be aborted early if an element is found that satisfies (**exists**) or does not satisfy (**forall**) the given property. In such cases the exceptional element may be assigned to a variable.[4]

The syntax for quantifiers is quite similar to that of set constructors.

exists(*variable*) { *expression in x* : *x* ***in*** *D* | *condition on x* }
forall(*variable*) { *expression in x* : *x* ***in*** *D* | *condition on x* }

In the case of **exists**, each element of *D* is checked; if one is found that satisfies the condition then TRUE is returned (and no more elements of *D* are checked). If this occurs then the variable is assigned the value of the expression for that particular element. If no elements satisfy the condition then FALSE is returned.

```
> exists(u) { x^3 - 13 : x in [1..100] | IsSquare(x^3 - 13) };
    true
> u;
    4900
```

In the case of **forall**, the early stoppage occurs if the condition is *not* satisfied for some *x*. If this occurs then FALSE is returned and the variable is assigned the value of the expression for that particular element. If all elements satisfy the condition then TRUE is returned.

```
> forall(u) { x : x in [1..50] | Factorial(x-1) mod x in {0, x-1} };
    false
> u;
    4
```

More than one variable may be given in the quantifier's arguments. In this case, the expression must evaluate to a tuple (see the next section for an explanation of tuples) with the matching number of elements. Each element of the tuple is then assigned to the appropriate variable.

```
> p := 89;
> exists(x,y) { <x, y> : x,y in [1..10] | x^2 + y^2 eq p };
    true
```

[4] The effects of a quantifier can also be achieved by using a suitable loop; this is more work, however.

```
> x; y; x^2 + y^2;
    5
    8
    89
```

3 Tuples

A *tuple* is an element of some cartesian product. As such, it has a number of components, each potentially lying in a different structure. This makes tuples handy for keeping related data of different types together. Tuples are created by enclosing the components in angled brackets $< >$.

$<$ *first component*, ..., *last component* $>$

Tuples have a certain degree of similarity to sequences. They are indexable in much the same way, for instance (except that indices larger than the number of components may not be used), and they support the intrinsics Explode and Append, although uses of the latter intrinsic are uncommon.

```
> tup := < 7, TRUE >;
> tup[1] +:= 4;
> tup;
    <11, true>
> a, b := Explode(tup);
> a; b;
    11
    true
```

One place where tuples commonly arise is when factoring. In Magma, the intrinsic Factorisation (or its synonym Factorization) takes a ring element and returns a sequence of tuples containing primes and exponents that give the factorisation of the element (up to units).

```
> P<t> := PolynomialRing(GF(7));
> g := t^15 + 6*t^11 + 2*t^8 + t^7 + 5*t^4 + 2;
> F := Factorisation(g);
> F;
    [
        <t + 2, 7>,
        <t^2 + t + 4, 1>,
        <t^2 + 2*t + 2, 1>,
        <t^2 + 5*t + 2, 1>,
        <t^2 + 6*t + 4, 1>
    ]
> g eq &*[ t[1]^t[2] : t in F ];
    true
```

4 Creating functions and procedures

New functions can be created using the **function** statement.

function *name*(*arguments*) *statements* **end function**;

Here *name* is the name of the function and *arguments* is a comma-separated list of variable names. When the function is called, the statements between the arguments and the **end function** are executed (these statements are called the *function body*). While executing the function body, the argument names are treated as variables whose initial values are those given when the function was called.

At some point during the execution of the function a **return** statement must be executed.

return *values*;

Here *values* is a comma-separated list of values. When such a statement is executed the function immediately finishes and its return values are those specified in the **return** statement.

```
> function minmax(x, y)
>     if x le y then
>         return x, y;
>     else
>         return y, x;
>     end if;
> end function;
> minmax(8, 6);
    6 8
```

When defining a function, the name may be omitted; this turns the statement into an expression whose value is the unnamed function. In Magma, functions are first-class objects, so this function may then be assigned to a variable. The result is indistinguishable from using the statement form, and both methods are commonly employed.

```
> area := function(a, b, c)
>     s := (a + b + c) / 2;
>     return Sqrt(s*(s - a)*(s - b)*(s - c));
> end function;
> area(3, 4, 5);
    6.000000000000000000000000000
```

There is a convenient shorthand for the common special case of a function with a single return value that is a simple expression in the arguments.

func<*arguments* | *expression*>

For example:

```
> harmonic_mean := func<a, b | 2 / (1/a + 1/b)>;
> harmonic_mean(3, 6);
    4
```

A *procedure* is a function that does not return any values. Procedures are created similarly to functions, except that the relevant keyword is **procedure**.

```
> procedure print_num(n, obj)
>     if n eq 1 then
>         print n, obj;
>     else
>         print n, obj cat "s";
>     end if;
> end procedure;
> print_num(39, "step");
    39 steps
```

Procedures may include a **return** statement that causes them to finish immediately (no values are returned, of course).

Procedures have a feature that functions in Magma do not have—the ability to modify their arguments. An argument that is to be modified is indicated by prefixing it with a tilde (∼), both in the definition and when called.

```
> remove_gcd := procedure(~x, ~y)
>     g := GCD(x, y);
>     x div:= g;
>     y div:= g;
> end procedure;
> a := 15;
> b := 20;
> remove_gcd(~a, ~b);
> a, b;
    3 4
```

5 Loops: for, while, and repeat

Magma has three looping constructs, which are similar to those of other languages. Each of the loops has a group of statements called the *loop body* that is executed repeatedly until the loop ends.

The most commonly used type of loop is the **for** loop. It has two versions; the less used but possibly more familiar one has the following form.

for *variable* := *a* **to** *b* **by** *k* **do** *statements* **end for**;

In this form the values *a*, *b*, and *k* must be integers. The loop body is executed for each integer in the specified range; the first time with the variable set to *a*, the next time to *a*+*k*, and so on. The increment is allowed to be negative.

```
> for i := 13 to 1 by -3 do
>     i;
> end for;
    13
    10
    7
    4
    1
```

The **by** subclause is optional (and usually omitted) in this form; if it is omitted then the increment is taken to be 1. A common use is when it is desired to perform the same operation a specific number of times. The following example demonstrates that performing eight perfect out-shuffles returns a deck of 52 cards to their original order.

```
> os_image := [ 1..52 by 2 ] cat [ 2..52 by 2 ];
> out_shuffle := func<cards | cards[ os_image ]>;
> cards := [ 1..52 ];
> for i := 1 to 8 do
>     cards := out_shuffle(cards);
> end for;
> cards eq [ 1..52 ];
    true
```

The second version of the **for** loop allows the variable to take on values other than integers.

for *variable* ***in*** *domain* **do** *statements* **end for**;

In this form the loop body is executed once for each value in the specified domain, with the variable taking on these values.

```
> G<a,b> := AbelianGroup([2, 4]);    // Z2 x Z4
> for g in G do
>     printf "%o\thas order %o\n", g, Order(g);
> end for;
    0       has order 1
    a       has order 2
    b       has order 4
    a + b   has order 4
    2*b     has order 2
    a + 2*b has order 2
    3*b     has order 4
    a + 3*b has order 4
```

The first version of the **for** loop can be simply converted into the second form by using a sequence constructor, changing *variable* := *a* **to** *b* **by** *k* into *variable* ***in*** [*a*..*b* **by** *k*]. In fact, this is exactly how Magma handles these statements internally.

The other two types of loops are **while** loops and **repeat** loops.

while *condition* **do** *statements* **end while**;
repeat *statements* **until** *condition*;

In a **while** loop, the condition is first checked, and if it evaluates to TRUE then the loop body is executed. This process is continued until the condition becomes FALSE.

```
> collatz := func<x | IsOdd(x) select 3*x + 1 else x div 2>;
> x := 13;
> while x ne 1 do
>     x := collatz(x);
>     print x;
> end while;
40
20
10
5
16
8
4
2
1
```

In a **repeat** loop, the loop body is executed and then the condition is checked. If it evaluates to FALSE then the process will be repeated. This process is continued until the condition becomes TRUE.

```
> Zx<x> := PolynomialRing(Integers());
> newton := func<f,r | r - Evaluate(f, r)/Evaluate(Derivative(f), r)>;
> g := x^2 - x - 1;
> phi := 1.0;
> repeat
>     oldphi := phi;
>     phi := newton(g, phi);
> until phi eq oldphi;
> phi;
1.61803398874989484820458683
```

The **break** and **continue** statements can be used to cause the early termination of a loop (**break**) or of a particular execution of the loop body (**continue**).

break;
continue;

When a **break** statement is executed it causes the innermost loop to finish immediately; the flow of execution resumes after the loop, just as if the loop had terminated normally. When a **continue** statement is executed it causes the innermost loop body to finish immediately; the flow of execution resumes

at the end of the loop body. For **for** loops this causes the next variable in the iteration to be used; for **while** and **repeat** loops the corresponding condition is checked to see whether the loop should continue or not.

```
> for x in [0..10] do
>     if IsOdd(x) then continue; end if;
>     print x;
>     if not IsPrime(2^x + 1) then break; end if;
> end for;
```

```
0
2
4
6
```

Each of these statements has a variant that allows outer **for** loops or loop bodies to be terminated.

break *variable*;
continue *variable*;

In this form, the specified variable must be the loop variable of an enclosing **for** loop. It is the loop or loop body associated with this **for** loop that will be terminated.

```
> n := 91;
> for a in [-6..6] do
>     for b in [-6..6] do
>         if a^3 + b^3 eq n then
>             <a, b>;
>             break a;
>         end if;
>     end for;
> end for;
```

```
<-5, 6>
```

If the *a* were omitted from the statement **break** *a* then only the inner loop would have been terminated, and the other solutions <3, 4>, <4, 3> and <6, −5> would have been found.

6 Maps

Another highly important kind of object is the *map*. The map type in Magma corresponds to the mathematical notion of a map from one structure to another. While the application of a map could also be achieved by defining a suitable function, the use of the map type allows structural information to be retained on the map. It also enables special maps such as homomorphisms to be created in a simple fashion. How maps can be created will be explained a little later; some basic uses of maps will be described first.

6.1 Map operations

When an intrinsic computes one structure from another, it is quite common for a map to also be returned that describes how to move from the original structure to the new one. One such intrinsic is MinimalModel.

```
> E := EllipticCurve([3, 1, 4, 1, 5]);
> M, φ := MinimalModel(E);
> φ;
    Elliptic curve isomorphism from: CrvEll: E to CrvEll: M
    Taking (x : y : 1) to (x + 1 : y + x + 1 : 1)
```

The intrinsics Domain and Codomain can be used to extract the appropriate structures from a map.

```
> Domain(φ);
    Elliptic Curve defined by y^2 + 3*x*y + 4*y = x^3 + x^2 + x
    + 5 over Rational Field
> Codomain(φ);
    Elliptic Curve defined by y^2 + x*y + y = x^3 + 3*x + 4 over
    Rational Field
```

If the map supports it, the intrinsic Inverse returns the inverse of the map.

```
> Inverse(φ);
    Elliptic curve isomorphism from: CrvEll: M to CrvEll: E
    Taking (x : y : 1) to (x - 1 : y - x : 1)
```

(The alternative form ϕ^{-1} is equivalent to Inverse(ϕ).)

The image of an element under a map can be obtained in one of two ways, either by treating the map like a function or by applying the @ operator.

map(*element*)
element @ *map*

Additionally, if supported by the map, the @@ operator can be used to return a preimage of an element of the codomain.

element @@ *map*

```
> P1 := E ![ -2, 1 ];
> P2 := E ![ 0, 1 ];
> φ(P1);
    (-1 : 0 : 1)
> P2 @ φ;
    (1 : 2 : 1)
> (M ![ 2, 3 ]) @@ φ;
    (1 : 1 : 1)
```

In fact, both image and preimage calculation can be applied to an entire sequence or set of elements at once.

```
> [ M | [ -1, 0 ], [ 1, 2 ] ] @@ φ;
    [ (-2 : 1 : 1), (0 : 1 : 1) ]
```

Given maps $f : A \to B$ and $g : B \to C$, the iterated application $g(f(x))$ is legal since the codomain of f is the same as the domain of g. The corresponding composite map $h = g \circ f : A \to C$ may be created as $h := f*g$ (note the order of f and g).

```
> φ * φ^-1;
    Elliptic curve isomorphism from: CrvEll: E to CrvEll: E
    Taking (x : y : 1) to (x : y : 1)
> φ^-1 * φ;
    Elliptic curve isomorphism from: CrvEll: M to CrvEll: M
    Taking (x : y : 1) to (x : y : 1)
```

6.2 Map creation

Maps can be created with either the **map** or **hom** constructor, specifying the domain and codomain of the map together with the information that determines how to find the image of an element of the domain.

map< *domain* → *codomain* | *image specification* >
hom< *domain* → *codomain* | *image specification* >

(The two character sequence −> is represented by → throughout this book; thus a code fragment like $A \to B$ would actually be typed as A−>B.)

There are two ways to define a map, either by specifying the images of each element (an *image map*) or by providing a rule that determines the image of an arbitrary element (a *rule map*). If the **hom** constructor is used with an image map then it is only necessary to provide images for each generator of the domain, since these determine the homomorphism uniquely.

There are two equivalent notations for explicitly specifying the image of an element.

element → *image*
< *element*, *image* >

An image map specification will contain either or both of these forms. They may be listed directly, or contained within a set or sequence. So in the following example the three map definitions are equivalent.

```
> K := GF(2);
> ψ1 := map< K -> Booleans() | 0 -> FALSE, 1 -> TRUE >;
> ψ2 := map< K -> Booleans() | <1, TRUE>, <0, FALSE> >;
> ψ3 := map< K -> Booleans() | [ x -> IsOne(x) : x in K ] >;
```

If specifying a homomorphism, a third form may be used whereby only the images are listed. These will be implicitly matched up with the generators of the domain (the first generator yields the first image, and so on).

```
> C<i> := QuadraticField(-1);
> conj := hom< C -> C | -i >;
> conj(3 - 4*i);
    4*i + 3
```

A rule in a rule map is given by a variable and an expression involving that variable. The following format is used:

x ↦ *expression in x*

(The three character sequence :−> is represented by ↦ throughout this book; thus a code fragment like $x \mapsto x + 1$ would be typed as x :−> x + 1.)

When the map is applied to an element, the value returned is the result of evaluating the expression with the variable temporarily assigned the value of the element.

A rule map may have a second rule, which specifies how to find the preimage of elements under the map. Continuing the example from the previous section, it turns out that the Mordell–Weil group of E is generated by the two points P_1 of order 2 and P_2 of infinite order. Thus this group is isomorphic to $\mathbb{Z}_2 \times \mathbb{Z}$. The following is a quick-and-dirty implemention of the map between this abstract group and E itself.

```
> G<a,b> := AbelianGroup([ 2, 0 ]);
> GtoE := func<g | t[1]*P1 + t[2]*P2 where t is Eltseq(g)>;
> EtoG := function(P)
>     n := Round(Sqrt(Height(P)/Height(P2)));
>     Q := n*P2;
>     m := Q[1] eq P[1] select 0 else 1;
>     Q -:= m*P1;
>     if Q[2] ne P[2] then n := -n; end if;
>     return G ! [ m, n ];
> end function;
> gmap := map<G -> E | x :-> GtoE(x), y :-> EtoG(y)>;
> gmap(a - b);
    (1 : 1 : 1)
> (P1 + 3*P2) @@ gmap;
    a + 3*b
```

There may also be special forms of map constructors allowed for particular combinations of domain and codomain. One important case is that of maps between schemes, which may be specified by providing the coordinates of the image as rational functions of the coordinates of the element (and similarly for the inverse, if supplied).

```
> Q := RationalField();
> P1<u,v> := ProjectiveSpace(Q, 1);
> P2<x,y,z> := ProjectiveSpace(Q, 2);
> C := Conic(P2, x^2 + y^2 - z^2);
> psi := map< C -> P1 | [ y, x + z ], [ v^2 - u^2, 2*u*v, v^2 + u^2 ] >;
> psi(C ! [ 3, 4, 5 ]);
    (1/2 : 1)
> Inverse(psi)(P1 ! [ 2, 3 ]);
    (5/13 : 12/13 : 1)
```

References

1. John Cannon, Wieb Bosma (eds.), *Handbook of Magma Functions*, Version 2.11, Volume **1**, Sydney, 2004.