

Handbook of Magma Functions

Part I: The Magma Language

Version 2.30-1

Sydney

November 10, 2026

Editors

John Cannon *Wieb Bosma* *Claus Fieker* *Allan Steel*

Handbook Contributors

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

Production Editors

Wieb Bosma *Claus Fieker* *Allan Steel* *Nicole Sutherland*

HTML Production

Claus Fieker *Allan Steel*

Copyright © 1994–2026 The University of Sydney. All rights reserved.

Contents

1 Statements and Expressions **1**
- 1.1 Introduction . . . 1
- 1.2 Starting, Interrupting, and Terminating . . . 1
- 1.3 Identifiers . . . 1
- 1.4 Assignment . . . 1
- 1.5 Boolean Values . . . 5
- 1.6 Coercion . . . 7
- 1.7 The where ... is Construction . . . 7
- 1.8 Conditional Statements and Expressions . . . 9
- 1.9 Error Handling Statements . . . 11
- 1.10 Iterative Statements . . . 12
- 1.11 Runtime Evaluation: the eval Expression . . . 15
- 1.12 Comments and Continuation . . . 16
- 1.13 Timing . . . 17
- 1.14 Types, Category Names, and Structures . . . 19
- 1.15 Random Object Generation . . . 22
- 1.16 Miscellaneous . . . 24
- 1.17 Bibliography . . . 24

2 Functions, Procedures, and Packages **25**
- 2.1 Introduction . . . 25
- 2.2 Functions and Procedures . . . 25
- 2.3 Packages . . . 30
- 2.4 Attributes . . . 38
- 2.5 User-defined Verbose Flags . . . 40
- 2.6 User-Defined Types . . . 42

3 Input and Output **49**
- 3.1 Introduction . . . 49
- 3.2 Character Strings . . . 49
- 3.3 Printing . . . 55
- 3.4 Input . . . 60
- 3.5 I/O Objects . . . 62
- 3.6 I/O Operations . . . 66
- 3.7 Managing Multiple I/O Objects . . . 72
- 3.8 Asynchronous I/O . . . 74
- 3.9 Loading Program Files . . . 78
- 3.10 Saving and Restoring Workspaces . . . 78
- 3.11 Logging a Session . . . 79
- 3.12 Memory Usage . . . 79
- 3.13 System Calls . . . 79
- 3.14 Creating Names . . . 80

4 Environment and Options **81**
- 4.1 Introduction . . . 81
- 4.2 Command Line Options . . . 81
- 4.3 Environment Variables . . . 83

4.4 Set and Get . . . 84
4.5 Verbose Levels . . . 88
4.6 Other Information Procedures . . . 89
4.7 The Magma Line Editor . . . 89
4.8 History Files . . . 94
4.9 History Commands . . . 101
4.10 The Magma Help System . . . 104

5 Parallelism 107
5.1 Introduction . . . 107
5.2 Magma Kernel Distributed Parallelism . . . 108
5.3 GPUs . . . 109
5.4 Linear Algebra . . . 109
5.5 Gröbner Bases . . . 112
5.6 Polynomial System Solving . . . 116
5.7 Polynomial Rings . . . 117
5.8 Distributed Computation for Internal Algorithms . . . 118
5.9 Integer Factorisation . . . 120
5.10 Ideal Class Group of Algebraic Number Fields . . . 120
5.11 Integral Lattices . . . 120
5.12 Linear Codes . . . 126
5.13 User-Implemented Distributed Parallelism . . . 128
5.14 Timing . . . 141
5.15 Bibliography . . . 143

6 Magma Semantics 144
6.1 Introduction . . . 144
6.2 Terminology . . . 144
6.3 Assignment . . . 144
6.4 Uninitialized Identifiers . . . 145
6.5 Evaluation in Magma . . . 145
6.6 Scope . . . 149
6.7 Procedure Expressions . . . 152
6.8 Reference Arguments . . . 153
6.9 Dynamic Typing . . . 153
6.10 Traps for Young Players . . . 154
6.11 Appendix A: Precedence . . . 155
6.12 Appendix B: Reserved Words . . . 156

7 The Magma Profiler 157
7.1 Introduction . . . 157
7.2 Profiler Basics . . . 157
7.3 Exploring the Call Graph . . . 159
7.4 Recursion and the Profiler . . . 160

8 Debugging Magma Code 163
8.1 Introduction . . . 163
8.2 Using the Debugger . . . 163

References 165

Index 166

Chapter 1

Statements and Expressions

1.1 Introduction

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

When a statement typed at the `>` prompt is not finished, the next prompt names the constructs still open, innermost last. After `print 9 - 3` it is `print>`, waiting for the semicolon. Inside a `while` loop inside a function it is `function|while>`, waiting for `end while;` and then `end function;`.

1.2 Starting, Interrupting, and Terminating

If Magma has been installed correctly, it may be activated by typing 'magma'.

`<Ctrl>-C`

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

`quit;`

`<Ctrl>-D`

Terminate the current Magma-session.

`<Ctrl>-\`

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

1.3 Identifiers

Identifiers (names for user variables, functions etc.) must begin with a letter, and this letter may be followed by any combination of letters or digits, provided that the name is not a *reserved word* (see Chapter *Magma Semantics* for a complete list of reserved words). In this definition the underscore `_` is treated as a letter; but note that a single underscore is a reserved word. Identifier names are case-sensitive; that is, they are distinguished from one another by lower and upper case. *Intrinsic* Magma functions usually have names beginning with capital letters (current exceptions include `pCore, pQuotient` and the like, where the `p` indicates a prime). Note that these identifiers are *not* reserved words; that is, one may use names of intrinsic functions for variables.

1.4 Assignment

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

1.4.1 Simple Assignment

`x := expression;`

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

Example: Identifiers and Intrinsic Names (ex-b15b93)

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

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

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

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

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

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

x_1, x_2, ..., x_n := expression;

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

`_ := expression;`

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

`assigned x: Var -> BoolElt`

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

Example: Multiple Return Values (ex-24e461)

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

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

To obtain the multipliers as well, type

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

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

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

1.4.2 Indexed Assignment

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

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

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

Example: Modifying Nested Sequence Entries (ex-36ee87)

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

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

1.4.3 Generator Assignment

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

E<x$_1$, x$_2$, ...x$_n$> := expression;

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

```
E := expression;
```

```
x_1 := E.1; x_2 := E.2; ... x_n := E.n;
```

E<[x]> := expression;

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

Example: Naming Generators with a Sequence (ex-b26be4)

We demonstrate the sequence method of generator naming.

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

(continues on next page)

(continued from previous page)

```
    X[1],
    X[2],
    X[3],
    X[4],
    X[5]
]
> &+X;
X[1] + X[2] + X[3] + X[4] + X[5]
> (&+X)^2;
X[1]^2 + 2*X[1]*X[2] + 2*X[1]*X[3] + 2*X[1]*X[4] +
    2*X[1]*X[5] + X[2]^2 + 2*X[2]*X[3] + 2*X[2]*X[4] +
    2*X[2]*X[5] + X[3]^2 + 2*X[3]*X[4] + 2*X[3]*X[5] +
    X[4]^2 + 2*X[4]*X[5] + X[5]^2
```

AssignNames(~S, [s_1, ... s_n]): Str, [MonStgElt]

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

Example: Renaming Group Generators (ex-92cebe)

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

1.4.4 Mutation Assignment

x o:= expression;

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

Example: Mutation Assignment (ex-f76a1d)

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

```
> x := 1;
> S := { };
```

(continues on next page)

(continued from previous page)

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

1.4.5 Deletion of Values

`delete x: Var`

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

The statement `clear;` deletes the value of every identifier, as `delete` does for one.

1.5 Boolean Values

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

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

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

1.5.1 Creation of Booleans

`Booleans() -> Bool`

The Boolean structure.

`# B: Bool -> RngIntElt`

Cardinality of Boolean structure (2).

`true`

`false`

The Boolean elements.

`Random(B): Bool -> BoolElt`

Return a random Boolean.

1.5.2 Boolean Operators

`x and y: BoolElt, BoolElt -> BoolElt`

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

`x or y: BoolElt, BoolElt -> BoolElt`

Returns `true` if at least one of x and y is `true`, `false` otherwise. If x is `true`, the expression for y is not evaluated.

`x xor y: BoolElt, BoolElt -> BoolElt`

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

`not x: BoolElt -> BoolElt`

Negate the truth value of x.

1.5.3 Equality Operators

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

(a) x and y are both elements of a structure S or there is a structure S such x and y will be coerced into S by automatic coercion;

(b) There is an equality test for elements of S defined within Magma. The possible automatic coercions are listed in the descriptions of the various Magma modules. For instance, the table in the introductory chapter on rings shows that integers can be coerced automatically into the rational field so an integer and a rational are comparable.

`x eq y: Any, Any -> BoolElt`

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

`x ne y: Any, Any -> BoolElt`

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

`x cmpeq y: Any, Any -> BoolElt`

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

`x cmpne y: Any, Any -> BoolElt`

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

Example: Semantics of Equality Operators (ex-838d04)

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

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

1.5.4 Iteration

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

Example: Iterating over the Booleans (ex-e85cda)

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

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

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

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

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

1.6 Coercion

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

`S ! x: Str, Elt -> Elt`

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

`IsCoercible(S, x): Str, Elt -> Bool, Elt`

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

1.7 The where ... is Construction

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

`expression`$_1$ `where identifier is expression`$_2$

`expression`$_1$ `where identifier := expression`$_2$

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

the construction is part of an expression list — see below. The `where` operator is left-associative. This means that there can be multiple uses of `where ... is` constructions and each expression can refer to variables bound in the enclosing constructions. Another important feature is found in a set or sequence constructor. If there are `where ... is` constructions in the predicate, then any variables bound in them may be referred to in the expression at the beginning of the constructor. If the whole predicate is placed in parentheses, then any variables bound in the predicate do not extend to the expression at the beginning of the constructor. The `where` operator also extends left in expression lists. That is, if there is an expression E in a expression list which is a `where` construction (or chain of where constructions), the identifiers bound in that where construction (or chain) will be defined in all expressions in the list which are to the left of E. Expression lists commonly arise as argument lists to functions or procedures, return arguments, print statements (with or without the word 'print') etc. A where construction also overrides (hides) any where construction to the right of it in the same list. Using parentheses around a where expression ensures that the identifiers bound within it are not seen outside it.

Example: Local Bindings with Where (ex-f8d9ed)

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

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

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

```
> x := 1;
> y := 2;
> x + y where x is 5 where y is 6;
11
> (x + y where x is 5) where y is 6; // the same
11
> x + y where x is (5 where y is 6);
7
> x + y where x is y where y is 6;
12
> (x + y where x is y) where y is 6; // the same
12
> x + y where x is (y where y is 6);
8
```

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

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

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

```
> // A simple use:
> [a, a where a is 1];
[ 1, 1 ]
```

(continues on next page)

(continued from previous page)

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

1.8 Conditional Statements and Expressions

The conditional statement has the usual form `if ... then ... else ... end if;`. It has several variants. Within the statement, a special prompt will appear, indicating that the statement has yet to be closed. Conditional statements may be nested. The conditional expression, `select ... else`, is used for in-line conditionals.

1.8.1 The Simple Conditional Statement

`if Boolean expression then statements`$_1$`else statements`$_2$`end if;`

`if Boolean expression then statementsend if;`

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

`if Boolean expression`$_1$` then statements`$_1$`elif Boolean expression`$_2$` then statements`$_2$`else statements`$_3$`end if;`

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

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

is equivalent to

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

Example: A Simple Conditional Statement (ex-6d1681)

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

1.8.2 The Simple Conditional Expression

`Boolean expression select` $\texttt{expr}_1$ `else` $\texttt{expr}_2$

This is an expression, of which the value is that of expr_1 or expr_2, depending on whether Boolean expression is `true` or `false`.

Example: Inline Conditional Expressions (ex-bbc380)

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

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

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

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

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

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

1.8.3 The Case Statement

`case expr : when` $\texttt{expr}_i$ `: statements end case`

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

Example: Case Statement on a Sign (ex-62b01f)

```
> x := 73;
> case Sign(x):
>    when 1:
>       x, "is positive";
>    when 0:
```

(continues on next page)

(continued from previous page)

```
>       x, "is zero";
>    when -1:
>       x, "is negative";
> end case;
73 is positive
```

1.8.4 The Case Expression

case< expr | expr_left,1 : expr_right,1, ..., expr_left,n : expr_right,n, default : expr_def>

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

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

1.9 Error Handling Statements

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

1.9.1 The Error Objects

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

Error(x): Any -> Err

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

e`Position: Err -> MonStgElt

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

e`Traceback: Err -> MonStgElt

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

e`Object: Err -> Any

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

e`Type: Err -> MonStgElt

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

1.9.2 Error Checking and Assertions

error expression, ..., expression;

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

`error if Boolean expression, expression, ..., expression;`

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

`assert Boolean expression;`

`assert2 Boolean expression;`

`assert3 Boolean expression;`

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

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

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

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

1.9.3 Catching Errors

`try statements catch e statements end try`

The `try`/`catch` statement lets users handle raised errors. The semantics of a `try`/`catch` statement are as follows: the block of statements $statements_1$ is executed. If no error is raised during its execution, then the block of statements $statements_2$ is not executed; if an error is raised at any point in $statements_1$, execution *immediately* transfers to $statements_2$ (the remainder of $statements_1$ is not executed). When transfer is controlled to the `catch` block, the variable named e is initialized to the error that was raised by $statements_1$; this variable remains in scope until the end of the `catch` block, and can be both read from and written to. The catch block can, if necessary, reraise e, or any other error object, using an `error` statement.

Example: Error Objects and Try-Catch (ex-33665a)

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

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

1.10 Iterative Statements

Four types of iterative statement are available: the `for`-statement that provides definite iteration, and the `for random`-, `while`-, and `repeat`-statements that provide indefinite iteration. Iteration may be performed over an arithmetic progression of integers or over any finite enumerated structure. Iterative statements may be nested. If nested iterations occur over the same enumerated structure, abbreviations

such as `for x, y in X do` may be used; the leftmost identifier will correspond to the outermost loop, etc. (For nested iteration in sequence constructors, see Chapter Sequences.) Early termination of the body of loop may be specified through use of the 'jump' commands `break` and `continue`.

1.10.1 Definite Iteration

`for i := expr`$_1$ `to expr`$_2$ `by expr`$_3$ `do`

`for i := expr`$_1$ `to expr`$_2$ `do`

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

In the second form above, where the step size is omitted, it is taken to be 1.

`for x in S do  statementsend for;`

Each of the elements of the finite enumerated structure S will be assigned to x in succession, and each time the statements will be executed.

1.10.2 Indefinite Iteration

`while Boolean expression do  statementsend while;`

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

Example: Collatz Iteration with a While Loop (ex-9a6f96)

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

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

`repeat  statementsuntil Boolean expression;`

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

Example: Collatz Steps with a Repeat Loop (ex-bdac72)

This example is similar to the previous one, except that it only prints x and the number of steps taken before x becomes 1. We use a `repeat` loop, and show that the use of a `break` statement

sometimes makes it unnecessary that the Boolean expression following the `until` ever evaluates to `true`. Similarly, a `while true` statement may be used if the user makes sure the loop will be exited using `break`.

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

`for random x in S do statementsend for;`

For each iteration of the loop a random element of S is assigned to x and the statements are executed. The loop will repeat with a new random selection each time until a `break` command is executed as part of the statements.

1.10.3 Dual Iteration

Some objects (such as sequences, indexed sets, associative arrays, etc.) have an index or other value associated with each element. The standard iteration would loop through the elements in the object, but sometimes this other value is also wanted. Magma provides a special syntax to make it easy to get both the values at once. For conciseness, the explanations below assume that the other value is an index; the actual interpretation of the dual iteration variables is described in the Handbook section for each type which supports dual iteration.

`for i -> x in S do statementsend for;`

Each of the elements of the finite enumerated structure S will be assigned to x in succession, with the associated index assigned to i, and each time the statements will be executed.

`for random i -> x in S do statementsend for;`

For each iteration of the loop a random element of S is assigned to x, and its associated index assigned to i, and the statements are executed. The loop will repeat with a new random selection each time until a `break` command is executed as part of the statements.

1.10.4 Early Exit from Iterative Statements

`continue;`

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

`continue identifier;`

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

`break;`

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

break identifier;

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

Example: Breaking Out of Nested Loops (ex-c95fc5)

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

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

```
46 89
89 46
```

1.11 Runtime Evaluation: the eval Expression

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

eval expression

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

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

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

Example: Basic Use of Eval (ex-9398c3)

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

```
> x := eval "1+1";  // OK
> x;
2
> eval "1+1;"; // not OK
2
```

(continues on next page)

(continued from previous page)

```
>> eval "1+1;"; // not OK
   ^
Runtime error: eval must return a value

> eval "return 1+1;"; // OK
2
> eval "x + 1"; // OK
3
> eval "x := x + 1; return x";
>> eval "x := x + 1; return x";
   ^

In eval expression, line 1, column 1:
>> x := x + 1; return x;
   ^
    Located in:
    >> eval "x := x + 1; return x";
       ^
User error: Imported environment value 'x' cannot be used as a local
```

Example: Building Objects with Eval (ex-c351ee)

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

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

> M2 := eval s;
> assert M eq M2;
```

1.12 Comments and Continuation

`//`

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

`/* */`

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

`\`

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

Example: Comments and Line Continuation (ex-bc5419)

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

1.13 Timing

Cputime() -> FldReElt

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

Cputime(t): FldReElt -> FldReElt

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

Realtime() -> FldReElt

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

Realtime(t): FldReElt -> FldReElt

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

ClockCycles() -> RngIntElt

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

Time() -> MonStgElt

Return a string which should be assigned to a variable (say T) which can be passed to the next function after executing a series of statements.

Time(T): MonStgElt -> MonStgElt

Given a string T previously returned by the call `Time()`, return a string which gives the time elapsed since T was assigned. As for the `time` statement above, if a multi-threaded algorithm is used within the intervening statements at any point, and the real time r taken for all the statements is less than the CPU time c taken, then the resulting time is r, and the tag `[r]` is appended (to indicate that the time printed is a real time); otherwise the resulting time is the CPU time c. In this way, one can also tell whether a multi-threaded algorithm is actually used in the statements executed. See Example *Timing Multithreaded Computations* below.

time statement;

Execute the statement and print the time taken when the statement is completed. Since V2.25, if a multi-threaded algorithm is used within the executed statement and the real time r taken is less than the CPU time c taken, then the time printed is r, and the tag `[r]` is appended (to indicate that the time printed is a real time); otherwise the CPU time c is printed (see example below). See Example *Timing Multithreaded Computations* below.

vtime flag: statement;

vtime flag, n: statement:

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

SetShowRealTime(v): BoolElt

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

GetShowRealTime() -> BoolElt

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

Example: Timing a Factorisation (ex-4cec25)

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

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

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

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

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

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

Example: Timing Multithreaded Computations (ex-de9abd)

In this example, we first time a matrix multiplication statement in the default single-threaded mode, so the time which is printed (or returned by the second `Time` call) is the CPU time.

```
> X := Random(MatrixRing(GF(5), 10000));
> time P := X*X; // single thread, normal CPU time
Time: 4.060
> T := Time(); P := X*X; Time(T); // similar with Time() function
4.060
```

Then we specify 8 threads for the multiplication algorithm, and note that the time has the tag `[r]` appended, to indicate that the time which is printed (or returned by the second `Time` call) is the real time.

```
> SetNthreads(8);
> time P := X*X; // multi-threads, so real time shown by [r]
Time: 1.050[r]
> T := Time(); P := X*X; Time(T); // similar with Time() function
1.050[r]
```

1.14 Types, Category Names, and Structures

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

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

`Type(x): Elt -> Cat`

`Category(x): Elt -> Cat`

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

`ExtendedType(x): Elt -> ECat`

`ExtendedCategory(x): Elt -> ECat`

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

`ISA(T, U): Cat, Cat -> BoolElt`

`ISA(T, U): Cat, ECat -> BoolElt`

`ISA(T, U): ECat, Cat -> BoolElt`

`ISA(T, U): ECat, ECat -> BoolElt`

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

`BaseType(T): ECat -> Cat`

Returns the underlying base type of the extended type T.

`# T: ECat -> RngIntElt`

Returns the number of parametrizing objects for the extended type T.

`T[i]: ECat, RngIntElt -> .`

The i-th parametrizing object of the extended type T. Although this will usually be another extended type, as the example below shows it may not be.

Example: Extended Type Accessors (ex-1892d6)

We show some of the possibilities of the extended types. First we define some simple objects and show their extended types.

```
> Z := Integers();
> Zx<x> := PolynomialRing(Z);
> S := [ x ];
> f1 := map<Zx -> Z | f :-> Evaluate(f, 1)>;
> D := CremonaDatabase();
>
> ExtendedType(Z);
RngInt
> ExtendedType(Zx);
RngUPol[RngInt]
> ExtendedType(S);
SeqEnum[RngUPolElt[RngInt]]
> ExtendedType(f1);
Map[RngUPol[RngInt], RngInt]
> ExtendedType(D);
DB["CrvEll"]
```

We can see a few points of note from the above. If the type is not parameterized at all, like `Z`, then the extended type prints as just the base type. The printing for `S` shows that extended types may themselves be further parameterized. The printing for `f1` shows that there may be more than one parametrizing object. The printing for `D` shows that parametrizing objects are not necessarily extended types.

We can use further investigate these with the intrinsics `BaseType`, `'#'`, and indexing.

```
> tS := ExtendedType(S);
> BaseType(tS);
SeqEnum
> #tS;
1
> tS[1];
RngUPolElt[RngInt]
> #tS[1];
1
> tS[1][1];
RngInt
> Type(tS[1][1]);
ECat
>
> tD := ExtendedType(D);
> BaseType(tD);
DB
> #tD;
1
> tD[1];
CrvEll
> Type(tD[1]);
MonStgElt
```

This information could be put together to generate a tree of the type information (represented as a list).

```
> function tree(t)
>     if Type(t) ne ECat then
```

(continues on next page)

(continued from previous page)

```
>         return [* t *];
>     end if;
>     return [* BaseType(t) *] cat [* $$(t[i]) : i in [1..#t] *];
> end function;
> function type_tree(object)
>     return tree(ExtendedType(object));
> end function;
>
> type_tree(Z);
[* RngInt *]
> type_tree(Zx);
[* RngUPol, [* RngInt *] *]
> type_tree(S);
[* SeqEnum, [* RngUPolElt, [* RngInt *] *] *]
> type_tree(f1);
[* Map, [* RngUPol, [* RngInt *] *], [* RngInt *] *]
> type_tree(D);
[* DB, [* CrvEll *] *]
```

Remember that the "CrvEll" in the last tree is a string, not a type.

```
> Type($1[2][1]);
MonStgElt
```

MakeType(S): MonStgElt -> Cat

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

ElementType(S): Str -> Cat

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

CoveringStructure(S, T): Str, Str -> Str

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

ExistsCoveringStructure(S, T): Str, Str -> BoolElt, Str

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

Example: Type and Structure Functions (ex-55c1aa)

We demonstrate the type and structure functions.

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

(continues on next page)

(continued from previous page)

```
> ISA(RngIntElt, GrpElt);
false
> ISA(FldRat, Fld);
true
```

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

```
> Q := RationalField();
> CoveringStructure(Z, Q);
Rational Field
> ExistsCoveringStructure(Z, DihedralGroup(3));
false
> ExistsCoveringStructure(Z, CyclotomicField(5));
true Cyclotomic Field of order 5 and degree 4
> ExistsCoveringStructure(CyclotomicField(3), CyclotomicField(5));
true Cyclotomic Field of order 15 and degree 8
> ExistsCoveringStructure(GF(2), GF(3));
false
> ExistsCoveringStructure(GF(2^6), GF(2, 15));
true Finite field of size 2^30
```

Our last example demonstrates the use of extended types:

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

1.15 Random Object Generation

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

Since V2.7 (June 2000), Magma contains an implementation of the *Monster* random number generator of G. Marsaglia [Mar00]. The period of this generator is $2^{29430} - 2^{27382}$ (approximately 10^{8859}), and passes all of the stringent tests in Marsaglia's *Diehard* test suite [Mar95].

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

SetSeed(s, c): RngIntElt, RngIntElt

SetSeed(s): RngIntElt

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

GetSeed() -> RngIntElt, RngIntElt

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

`Random(S): Str -> Elt`

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

`Random(a, b): RngIntElt, RngIntElt -> RngIntElt`

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

`Random(b): RngIntElt -> RngIntElt`

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

Example: Restoring a Random State (ex-a0e14a)

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

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

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

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

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

```
> SetSeed(23);
> GetSeed();
23 0
> time S := {* Random(1, 4): i in [1..100000] *};
Time: 0.500
> S;
{* 1^^24962, 2^^24923, 3^^24948, 4^^25167 *}
> GetSeed();
23 100000
> [Random(1, 100): i in [1 .. 10]];
[ 3, 93, 11, 62, 6, 73, 46, 52, 100, 30 ]
> GetSeed();
23 100013
```

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

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

1.16 Miscellaneous

`IsIntrinsic(S): MonStgElt -> Bool, Intrinsic`

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

Example: Checking Whether a Name Is Intrinsic (ex-c50516)

We demonstrate the function `IsIntrinsic`.

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

1.17 Bibliography

- [Mar95] G. Marsaglia. DIEHARD: a battery of tests of randomness. http://stat.fsu.edu/pub/diehard/, 1995.
- [Mar00] G. Marsaglia. The Monster, a random number generator with period over 10^{2857} times as long as the previously touted longest-period one. Preprint, 2000.

Chapter 2

Functions, Procedures, and Packages

2.1 Introduction

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

2.2 Functions and Procedures

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

2.2.1 Functions

`f := function(`x_1`, ..., `x_n` : parameters)  statementsend function;`

`function f(`x_1`, ..., `x_n` : parameters)  statementsend function;`

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

The function may return multiple values. Usually one uses the form `return expression, ..., expression;`. If one wishes to make some return value(s) undefined (so that the number of return values for the function is the same in all 'branches' of the function) the underscore symbol (`_`) may be used. (The undefined symbol may only be used for final values of the list.) This construct allows behaviour similar to the intrinsic function `IsSquare`, say, which returns `true` and the square root of its argument if that exists, and `false` and the undefined value otherwise. See also the example below.

If there are parameters given, they must consist of a comma-separated list of clauses each of the form `identifier := value`. The identifier gives the name of the parameter, which can then be treated as a normal value argument within the statements. The value gives a default value for the parameter, and may depend on any of the arguments or preceding parameters; if, when the function is called, the parameter is not assigned a value, this default value will be assigned to

the parameter. Thus parameters are always initialized. If no parameters are desired, the colon following the last argument, together with parameters, may be omitted.

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

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

In a call to a function, a procedure, or an intrinsic, a parameter name given on its own sets that parameter to `true`: `f(x : Verbose)` is the same as `f(x : Verbose := true)`.

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

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

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

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

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

Example: Recursive Functions (ex-eb1a18)

This example illustrates recursive functions.

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

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

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

(continues on next page)

(continued from previous page)

```
> fibo(10)+fibo(12);
199
```

Example: Function Parameters (ex-6cad0c)

This example illustrates the use of parameters.

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

Example: Returning Undefined Values (ex-f4d48f)

This example illustrates the returning of undefined values.

```
> f := function(x)
>    if IsOdd(x) then
>        return true, x;
>    else
>        return false, _;
>    end if;
> end function;
>
> f(1);
true 1
> f(2);
false
> a, b := f(1);
> b;
1
> a, b := f(2);
> b;   // Undefined value

>> b;
   ^
User error: Identifier 'b' has not been assigned
```

Example: Variadic Functions (ex-1b08eb)

This example illustrates the use of variadic functions.

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

(continues on next page)

(continued from previous page)

```
> f(1, 2, 3);
x:  1
y:  [* 2, 3*]
[ 3, 4 ]
> f(1, 2, 3, 4);
x:  1
y:  [* 2, 3, 4*]
[ 3, 4, 5 ]
```

2.2.2 Procedures

p := procedure(x$_1$, ..., x$_n$: parameters) statementsend procedure;

procedure p(x$_1$, ..., x$_n$: parameters) statementsend procedure;

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

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

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

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

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

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

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

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

Example: Procedures (ex-711cd6)

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

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

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

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

2.2.3 The forward Declaration

`forward f;`

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

The `forward` statement must occur on the 'main' level, that is, outside other functions or procedures. (See also Chapter *Magma Semantics*.)

Example: Mutual Recursion with Forward Declarations `(ex-09aa09)`

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

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

```
> function strongTest(primdiv, n)
>     return exists{ x : x in [2..n-1] | \\
>       Modexp(x, n-1, n) eq 1 and
>       forall{ p : p in primdiv | Modexp(x, (n-1) div p, n) ne 1 }
>     };
> end function;
```

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

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

(continues on next page)

(continued from previous page)

```
>    end if;
> end function;
```

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

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

2.3 Packages

2.3.1 Introduction

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

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

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

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

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

The syntax for the definition of an intrinsic function is similar to that of an ordinary function except that the function header must define the function's signature together with text summarizing the semantics of the function. As noted above, an intrinsic function definition must reside in a package file. It is necessary for Magma to know the location of all necessary package files. A package may be attached or detached through use of the `Attach` or `Detach` procedures. More generally, a family of packages residing in a directory tree may be specified through provision of a **spec** file which specifies the locations of a collection of packages relative to the position of the spec file. Automatic attaching of the packages in a

spec file may be set by means of an environment variable (`MAGMA_SYSTEM_SPEC` for the Magma system packages and `MAGMA_USER_SPEC` for a users personal packages).

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

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

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

2.3.2 Intrinsics

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

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

`intrinsic name(arguments) [ -> return-values ]{ comment-text }  statementsend␣ intrinsic;`

The syntax of a intrinsic declaration is as above, where: *name* is the name of the intrinsic, which can be any identifier or a non-alphanumeric name enclosed within single quotes (like '+'); *arguments* is the argument list (possibly variadic and optionally including parameters preceded by a colon); optionally there is an arrow and return type list *return-values*; the comment text is any text within the braces (use `\}` to get a right brace within the text, and use `"` to repeat the comment from the immediately preceding intrinsic); and statements is a list of statements making up the body of the intrinsic.

arguments is a list of comma-separated arguments of the form

```
 name::type
~name::type
~name
```

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

```
.                   Any type
<category-name>     Simple type name
[]                  Sequence type
{}                  Set type
{[ ]}               Set or sequence type
{@ @}               Iset type
{* *}               Multiset type
<>                  Tuple type
```

or a *composite type*:

```
[ <type> ]    Sequences over <type>
{ <type> }    Sets over <type>
```

(continues on next page)

(continued from previous page)

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

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

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

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

Example: Defining an Intrinsic for GCDs (ex-82f54a)

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

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

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

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

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

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

2.3.3 Resolving Calls to Intrinsics

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

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

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

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

Example: Intrinsic Lookup for Overloaded Signatures (ex-45ec7c)

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

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

```
intrinsic overloaded(x::RngUPolElt[RngInt], y::RngUPolElt) -> RngIntElt
   { Overload 2 }
      return 2;
   end intrinsic;
```

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

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

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

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

2.3.4 Attaching and Detaching Package Files

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

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

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

`Attach(F): MonStgElt`

Procedure to attach the package file F.

`Detach(F): MonStgElt`

Procedure to detach the package file F.

`freeze;`

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

2.3.5 Related Files

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

```
file.sig     sig file containing signature information;

file.lck     lock file.
```

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

2.3.6 Importing Constants

`import "filename": ident_list;`

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

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

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

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

Example: Importing Constants from a Package (ex-2ff439)

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

```
MY_LIMIT := 10000;

   function fred(x)
      return 1/x;
   end function;
```

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

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

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

2.3.7 Argument Checking

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

`require condition: print_args;`

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

`requirerange v, L, U;`

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

`requirege v, L;`

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

Example: Argument Checking with require (ex-216764)

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

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

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

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

2.3.8 The Number of Results

Since V2.28, a function intrinsic can tell how many results are requested by the caller of the function. This means the intrinsic can possibly be more efficient in the case that less results are required than the default full list of results.

`NumberOfResults() -> RngIntElt, [ BoolElt ]`

`Nresults() -> RngIntElt, [ BoolElt ]`

Within a function or intrinsic function, return the number n of results requested by the caller of the current function, and optionally return a sequence B of booleans indicating which specific results are requested. If the function is called in the context of printing, then n is zero (and B is empty).

Example: Number of Results Requested (ex-2485c6)

Suppose that the file "test.m" contains the following intrinsic function.

```
intrinsic Func(x::RngIntElt) -> ., .
   {Test function}
     n := Nresults();
     printf "[Nresults: %o]\n", n;
     if n eq 1 then
       return x;
     else // includes print mode
       // Could potentially do something very expensive
       return x, x^2;
     end if;
   end intrinsic;
```

The intrinsic function can thus be called:

```
> Attach("test.m");
> Func(3); // print (returns all by default)
[Nresults: 0]
3 9
> c := Func(5); // Avoid potentially expensive computation of 2nd result
[Nresults: 1]
> c;
5
> a, b := Func(4);
[Nresults: 2]
> a, b;
4 16
```

Now suppose that the file "test.m" also contains the following intrinsic function. The function simulates an algorithm to compute the Smith form S of the input matrix A, with optional unimodular transformation matrices P and Q so that $S = P \times A \times Q$ (here we return strings to simulate the actual matrices which would be computed). The algorithm only computes P and/or Q if the caller requests these, based on the LHS of the assignment statement (the internal intrinsic function SmithForm has similar optimisations).

```
intrinsic MySmith(A::Mtrx) -> ., .
   {Test function}
     n, B := Nresults();
     printf "[Nresults: %o, B: %o]\n", n, B;
     if n le 1 then
       // Compute Smith S only (includes print mode)
       return "S", _, _;
     elif n eq 2 then
       // Compute Smith S and P so S = P*X*?
       return "S", "P", _;
     else // n eq 3
       if not B[2] then
         // Compute Smith S and Q so S = ?*X*Q
         return "S", _, "Q"; // S = ?*X*Q
       else
```

(continues on next page)

(continued from previous page)

```
        // Compute Smith S and P, Q so S = P*X*Q
        return "S", "P", "Q"; // S = P*X*Q
      end if;
    end if;
  end intrinsic;
```

After attaching the file, the intrinsic function can thus be called:

```
> A := MatrixRing(IntegerRing(), 2)!1; // arbitrary matrix
> MySmith(A); // print mode
[Nresults: 0, B: []]
S
> S := MySmith(A);
[Nresults: 1, B: [ true ]]
> S, P := MySmith(A);
[Nresults: 2, B: [ true, true ]]
> S, _, Q := MySmith(A);
[Nresults: 3, B: [ true, false, true ]]
> S, P, Q := MySmith(A);
[Nresults: 3, B: [ true, true, true ]]
```

2.3.9 Package Specification Files

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

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

`AttachSpec(S): MonStgElt`

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

`DetachSpec(S): MonStgElt`

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

Example: Package Specification File `(ex-69025d)`

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

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

(continues on next page)

(continued from previous page)

```
            galois.m
      }
   }
}
```

Then there should be the files

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

and if one typed within Magma

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

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

2.3.10 User Startup Specification Files

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

Example: User Startup Specification Files `(ex-4f513b)`

One could have

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

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

2.4 Attributes

This section is placed beside the section on packages because the use of attributes is most common within packages. For any structure within Magma, it is possible to have *attributes* associated with it. These are simply values stored within the structure and are referred to by named fields in exactly the same manner as Magma records.

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

2.4.1 Predefined System Attributes

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

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

is completely equivalent to the code:

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

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

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

is completely equivalent to the code:

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

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

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

2.4.2 User-defined Attributes

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

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

`AddAttribute(C, F): Cat, MonStgElt -> ;`

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

`declare attributes C: F1,...,Fn;`

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

2.4.3 Accessing Attributes

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

`S`fieldname: Str, Fieldname -> Elt`

`S`` N: Str, Str -> Elt`

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

`assigned S`fieldname: Str, Fieldname -> BoolElt`

assigned S`` N: Str, Str -> BoolElt

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

S`fieldname := expression;

S`` N := expression;

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

delete S`fieldname;

delete S`` N;

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

GetAttributes(C): Cat -> [MonStgElt]

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

ListAttributes(C): Cat

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

2.5 User-defined Verbose Flags

Verbose flags may be defined by users within packages.

declare verbose F, m;

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

2.5.1 Examples

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

Example: System Attributes (ex-ca5e5c)

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

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

>> G`Order;
    ^
Runtime error in `: Attribute 'Order' for this structure
is valid but not assigned
> // Force computation of order by intrinsic Order.
```

(continues on next page)

(continued from previous page)

```
> Order(G);
168
> // Check Order field again.
> assigned G`Order;
true
> G`Order;
168
> G`` "Order"; // String form for field
168
> o := "Order";
> G`` o;
168
> // Create code C and set its minimum weight.
> C := QRCode(GF(2), 31);
> C`MinimumWeight := 7;
> C;
[31, 16, 7] Quadratic Residue code over GF(2)
...
```

Example: Interactive User Attributes (ex-659b09)

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

```
> // Add attribute field MyStuff for matrix groups.
> AddAttribute(GrpMat, "MyStuff");
> // Create group G.
> G := GL(2, 3);
> // Try illegal field.
> G`silly;

>> G`silly;
    ^
Runtime error in `: Invalid attribute 'silly' for this structure
> // Try legal but unassigned field.
> G`MyStuff;

>> G`MyStuff;
    ^
Runtime error in `: Attribute 'MyStuff' for this structure is valid but not
assigned
> // Assign field and notice value.
> G`MyStuff := [1, 2];
> G`MyStuff;
[ 1, 2 ]
```

Example: Package User Attributes (ex-280c05)

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

```
declare attributes GrpMat: PermRep, PermRepMap;

intrinsic PermutationRepresentation(G::GrpMat) -> GrpPerm
{A permutation group representation P of G, with homomorphism f: G -> P};
    // Only compute rep if not already stored.
    if not assigned G`PermRep then
```

(continues on next page)

(continued from previous page)

```
        G`PermRepMap, G`PermRep := CosetAction(G, sub<G|>);
    end if;
    return G`PermRep, G`PermRepMap;
end intrinsic;
```

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

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

Suppose the following line were also in the package file:

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

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

2.6 User-Defined Types

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

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

2.6.1 Declaring User-Defined Types

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

`declare type T;`

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

`declare type T: P`$_1$`, ..., P`$_n$`;`

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

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

`declare type T[E];`

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

`declare type T[E]:` $\texttt{P}_1$`, ...,` $\texttt{P}_n$`;`

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

2.6.2 Creating an Object

`New(T): Type -> T`

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

`Clone(X): Any -> Any`

Given an object X having (user-defined) type T, return a copy C of X with type T and with all assigned attributes copied from X. This allows one to modify the attributes of the new object C without affecting the attributes of X.

2.6.3 Special Intrinsics Provided by the User

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

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

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

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

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

Note: `Print` is used by Magma in error-handling tracebacks, being called with the "Minimal" parameter if available. In particular, `Print` in this form should be kept rather uncomplicated, notably avoiding the usage of (failing) try/catch constructions, as these will cause a crash from recursive errors being detected.

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

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

parent of X, assuming it has one. In such a case, typically the attribute `Parent` will be defined for X and so `X`Parent` should simply be returned.

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

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

```
intrinsic IsCoercible(X::T, y::.) -> BoolElt, .
{Return whether y is coercible into X and the result if so}
    // Code: do tests on the type of y to see whether coercible

    // On failure, do:
    //    return false, "Illegal coercion"; // Or more particular message

    // Assumed coercible now; set x to result of coercion into X
    return true, x;
end intrinsic;
```

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

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

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

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

Providing this intrinsic can greatly speed the checking of equality of objects of type T, and in particular if you wish to work with sets of reasonable cardinality (more than 1000 elements) it should be made

available. The requirement is that if X and Y are equal, then their hashes should be the same, regardless of their internal representation.

2.6.4 Examples

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

Example: User-Defined Type for Rationals (ex-6e336c)

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

```
declare type MyRat;

declare attributes MyRat: Numer, Denom;

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

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

intrinsic Print(r::MyRat)
{Print r}

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

intrinsic '+'(r::MyRat, s::MyRat) -> MyRat
{Return r + s}

    rn := r`Numer;
    rd := r`Denom;
    sn := s`Numer;
    sd := s`Denom;
    return MyRational(rn*sd + sn*rd, rd*sd);
end intrinsic;

intrinsic '*'(r::MyRat, s::MyRat) -> MyRat
{Return r * s}

    rn := r`Numer;
    rd := r`Denom;
    sn := s`Numer;
    sd := s`Denom;
    return MyRational(rn*sn, rd*sd);
end intrinsic;
```

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

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

Example: User-Defined Type for Direct Products (ex-5c6834)

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

```
/* Declare types and attributes */


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


/* Special intrinsics for DirProd */


intrinsic DirectProduct(Rings::Tup) -> DirProd
{Create the direct product of given rings (a tuple)}

    require forall{R: R in Rings | ISA(Type(R), Rng)}:
        "Tuple entries are not all rings";
    D := New(DirProd);
    D`Rings := Rings;
    return D;
end intrinsic;


intrinsic Print(D::DirProd)
{Print D}

    Rings := D`Rings;
    printf "Direct product of %o", Rings; // NOTE: no newline!
end intrinsic;


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

(continues on next page)

(continued from previous page)

```
end function;


intrinsic IsCoercible(D::DirProd, x::.) -> BoolElt, .
{Return whether x is coercible into D and the result if so}

    Rings := D`Rings;
    n := #Rings;
    if Type(x) eq DirProdElt then
        if x`Parent cmpeq D then
            return true, x;
        end if;
        x := x`Elements;
    end if;
    if Type(x) ne Tup then
        return false, "Coercion RHS must be a tuple";
    end if;
    if #x ne n then
        return false, "Wrong length of tuple for coercion";
    end if;
    Elements := <>;
    for i := 1 to n do
        l, t := IsCoercible(Rings[i], x[i]);
        if not l then
            return false, Sprintf("Tuple entry %o not coercible", i);
        end if;
        Append(~Elements, t);
    end for;

    y := CreateElement(D, Elements);
    return true, y;
end intrinsic;


/* Special intrinsics for DirProdElt */


intrinsic Print(x::DirProdElt)
{Print x}

    printf "%o", x`Elements; // NOTE: no newline!
end intrinsic;


intrinsic Parent(x::DirProdElt) -> DirProd
{Parent of x}

    return x`Parent;
end intrinsic;


intrinsic '+'(x::DirProdElt, y::DirProdElt) -> DirProdElt
{Return x + y}

    D := Parent(x);
    require D cmpeq Parent(y): "Incompatible arguments";
```

(continues on next page)

(continued from previous page)

```
    Ex := x`Elements;
    Ey := y`Elements;
    return CreateElement(D, <Ex[i] + Ey[i]: i in [1 .. #Ex]>);
end intrinsic;


intrinsic '*'(x::DirProdElt, y::DirProdElt) -> DirProdElt
{Return x * y}

    D := Parent(x);
    require D cmpeq Parent(y): "Incompatible arguments";

    Ex := x`Elements;
    Ey := y`Elements;
    return CreateElement(D, <Ex[i] * Ey[i]: i in [1 .. #Ex]>);
end intrinsic;
```

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

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

Chapter 3

Input and Output

3.1 Introduction

This chapter is concerned with the various facilities provided for communication between Magma and its environment. The first section describes character strings and their operations. Following this, the various forms of the `print`-statement are presented. Next the `IO` type is introduced and its subtypes described, followed by a summary of the allowed I/O operations on these types. The chapter concludes with a section listing system calls. These include facilities that allow the user to execute an operating system command from within Magma or to run an external process.

3.2 Character Strings

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

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

3.2.1 Representation of Strings

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

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

To obtain a literal backslash, one simply types two backslashes; for characters other than double-quotes and backslash it does not make a difference when a backslash precedes them inside a string, with the exception of `n`, `r`, and `t`. Any occurrence of `\n` or `\r` inside a string is converted into a `<new-line>` while `\t` is converted into a `<tab>`. For example:

```
> "The first line,\\nthe second line, and then\\ran\\tindented line";
The first line,
```

(continues on next page)

(continued from previous page)

```
the second line, and then
an      indented line
```

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

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

3.2.2 Creation of Strings

`"abc" -> MonStgElt`

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

`BinaryString(s): MonStgElt -> BStgElt`

`BString(s): MonStgElt -> BStgElt`

Create a binary string from the character string s.

`s cat t: MonStgElt, MonStgElt -> MonStgElt`

`s cat t: BStgElt, BStgElt -> BStgElt`

`s * t: MonStgElt, MonStgElt -> MonStgElt`

Concatenate the strings s and t.

`s cat:= t: MonStgElt, MonStgElt -> MonStgElt`

`s cat:= t: BStgElt, BStgElt -> BStgElt`

`s *:= t: MonStgElt, MonStgElt -> MonStgElt`

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

`&cat s: [ MonStgElt ] -> MonStgElt`

`&cat s: [ BStgElt ] -> BStgElt`

`&* s: [ MonStgElt ] -> MonStgElt`

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

`s ^ n: MonStgElt, RngIntElt -> MonStgElt`

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

`s[i]: MonStgElt, RngIntElt -> MonStgElt`

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

`s[i]: BStgElt, RngIntElt -> RngIntElt`

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

`ElementToSequence(s): MonStgElt -> [ MonStgElt ]`

`Eltseq(s): MonStgElt -> [ MonStgElt ]`

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

`ElementToSequence(s): BStgElt -> [ BStgElt ]`

`Eltseq(s): BStgElt -> [ BStgElt ]`

Returns the sequence of numeric values representing the characters of s.

`Substring(s, n, k): MonStgElt, RngIntElt, RngIntElt -> MonStgElt`

`Substring(s, n, k): BStgElt, RngIntElt, RngIntElt -> BStgElt`

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

3.2.3 Integer-Valued Functions

`# s: MonStgElt -> RngIntElt`

`# s: BStgElt -> RngIntElt`

The length of the string s.

`Index(s, t): MonStgElt, MonStgElt -> RngIntElt`

`Position(s, t): MonStgElt, MonStgElt -> RngIntElt`

This function returns the position (an integer p with $0 < p \leq \#s$) in the string s where the beginning of a contiguous substring t occurs. It returns 0 if t is not a substring of s. (If t is the empty string, position 1 will always be returned, even if s is empty as well.)

3.2.4 Character Conversion

To perform more sophisticated operations, one may convert the string into a sequence and use the extensive facilities for sequences described in the next part of this manual; see the examples at the end of this chapter for details.

`StringToCode(s): MonStgElt -> RngIntElt`

Returns the code number of the first character of string s. This code depends on the computer system that is used; it is ASCII on most UNIX machines.

`CodeToString(n): RngIntElt -> MonStgElt`

Returns a character (string of length 1) corresponding to the code number n, where the code is system dependent (see previous entry).

`StringToInteger(s): MonStgElt -> RngIntElt`

Returns the integer corresponding to the string of decimal digits s. All non-space characters in the string s must be digits $(0, 1, \ldots, 9)$, except the first character, which is also allowed to be $+$ or $-$. An error results if any other combination of characters occurs. Leading zeros are omitted. Equivalently, a string of decimal digits may be coerced directly into the integer ring: `IntegerRing()!s` returns the same result as `StringToInteger(s)`.

`StringToInteger(s, b): MonStgElt, RngIntElt -> RngIntElt`

Returns the integer corresponding to the string of digits s, all assumed to be written in base b. The base b must be in the range $[2 \ldots 36]$. All non-space characters in the string s must be digits less than b (if b is greater than 10, 'A' is used for 10, 'B' for 11, and so on up to 'Z' for 35; upper and lower case are both accepted), except the first character, which is also allowed to be $+$ or $-$. An error results if any other combination of characters occurs.

`StringToIntegerSequence(s): MonStgElt -> [ RngIntElt ]`

Returns the sequence of integers corresponding to the string s of space-separated decimal numbers. All non-space characters in the string s must be digits $(0, 1, \ldots, 9)$, except the first character after each space, which is also allowed to be $+$ or $-$. An error results if any other combination of characters occurs. Leading zeros are omitted. Each number can begin with a sign ($+$ or $-$) without a space.

`IntegerToString(n): RngIntElt -> MonStgElt`

Convert the integer n into a string of decimal digits; if n is negative the first character of the string will be $-$. (Note that leading zeros and a $+$ sign are ignored when Magma builds an integer, so the resulting string will never begin with $+$ or 0 characters.)

`IntegerToString(n, b): RngIntElt, RngIntElt -> MonStgElt`

Convert the integer n into a string of digits with the given base (which must be in the range $[2 \ldots 36]$); if n is negative the first character of the string will be $-$.

3.2.5 Boolean Functions

`s eq t: MonStgElt, MonStgElt -> BoolElt`

`s eq t: BStgElt, BStgElt -> BoolElt`

Returns `true` if and only if the strings s and t are identical. Note that blanks are significant.

`s ne t: MonStgElt, MonStgElt -> BoolElt`

`s ne t: BStgElt, MonStgElt -> BoolElt`

Returns `true` if and only if the strings s and t are distinct. Note that blanks are significant.

`s in t: MonStgElt, MonStgElt -> BoolElt`

Returns `true` if and only if s appears as a contiguous substring of t. Note that the empty string is contained in every string.

`s notin t: MonStgElt, MonStgElt -> BoolElt`

Returns `true` if and only if s does not appear as a contiguous substring of t. Note that the empty string is contained in every string.

`s lt t: MonStgElt, MonStgElt -> BoolElt`

`s lt t: BStgElt, BStgElt -> BoolElt`

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

`s le t: MonStgElt, MonStgElt -> BoolElt`

`s le t: BStgElt, BStgElt -> BoolElt`

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

`s gt t: MonStgElt, MonStgElt -> BoolElt`

`s gt t: BStgElt, BStgElt -> BoolElt`

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

`s ge t: MonStgElt, MonStgElt -> BoolElt`

`s ge t: BStgElt, BStgElt -> BoolElt`

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

Example: Concatenating and Quoting Strings (ex-04834d)

```
> "Mag" cat "ma";
Magma
```

Omitting double-quotes usually has undesired effects:

```
> "Mag cat ma";
Mag cat ma
```

And note that there are two different equalities involved in the following!

```
> "73" * "9" * "42" eq "7" * "3942";
true
> 73 * 9 * 42 eq 7 * 3942;
true
```

The next line shows how strings can be concatenated quickly, and also that strings of blanks can be used for formatting:

```
> s := ("Mag" cat "ma? ")^2;
> s, " "^30, s[4]^12, "!";
Magma? Magma?                              mmmmmmmmmmmm !
```

Here is a way to list (in a sequence) the first occurrence of each of the ten digits in the decimal expansion of π, using `IntegerToString` and `Position`.

```
> pi := Pi(RealField(1001));
> dec1000 := Round(10^1000*(pi-3));
> I := IntegerToString(dec1000);
> [ Position(I, IntegerToString(i)) : i in [0..9] ];
[ 32, 1, 6, 9, 2, 4, 7, 13, 11, 5 ]
```

Using the length `#` and string indexing `[ ]` it is also easy to count the number of occurrences of each digit in the string containing the first 1000 digits.

```
> [ #[i : i in [1..#I] | I[i] eq IntegerToString(j)] : j in [0..9] ];
[ 93, 116, 103, 102, 93, 97, 94, 95, 101, 106 ]
```

We would like to test if the ASCII-encoding of the string 'Magma' appears. This could be done as follows, using `StringToCode` and `in`, or alternatively, `Position`. To reduce the typing, we first abbreviate `IntegerToString` to `its` and `StringToCode` to `sc`.

```
> sc := StringToCode;
> its := IntegerToString;
> M := its(sc("M")) * its(sc("a")) * its(sc("g")) * its(sc("m")) * its(sc("a"));
> M;
779710310997
> M in I;
false
> Position(I, M);
0
```

So 'Magma' does not appear this way. However, we could be satisfied if the letters appear somewhere in the right order. To do more sophisticated operations (like this) on strings, it is necessary to convert the string into a sequence, because sequences constitute a more versatile data type, allowing many more advanced operations than strings.

```
> Iseq := [ I[i] : i in [1..#I] ];
> Mseq := [ M[i] : i in [1..#M] ];
> IsSubsequence(Mseq, Iseq);
false
> IsSubsequence(Mseq, Iseq: Kind := "Sequential");
true
```

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

```
> "Pi" le "magma";
true
> "magma" lt "pi";
true
```

3.2.6 Parsing Strings

`Split(S, D): MonStgElt, MonStgElt -> [ MonStgElt ]`

`Split(S): MonStgElt -> [ MonStgElt ]`

```
IncludeEmpty: BoolElt                    Default: false
```

Given a string S, together with a string D describing a list of separator characters, return the sequence of strings obtained by splitting S at any of the characters contained in D. That is, S is considered as a sequence of fields, with any character in D taken to be a delimiter separating the fields. If D is omitted, it is taken to be the string consisting of the newline character alone (so S is split into the lines found in it). If S is desired to be split into space-separated words, the argument `" \t\n"` should be given for D.

By default, empty fields are not returned. This may be changed by setting the parameter `IncludeEmpty` to `true`.

Example: Splitting Strings (ex-461123)

We demonstrate elementary uses of `Split`.

```
> Split("a b c d", " ");
[ a, b, c, d ]
> // Note that adjacent separators do not produce
> // extra fields by default:
> Split("a||b|c", "|");
[ a, b, c ]
> // But they can be made to appear with IncludeEmpty:
> Split("a||b|c", "|" : IncludeEmpty := true);
[ a, , b, c ]
> Split("abxcdyefzab", "xyz");
[ ab, cd, ef, ab ]
> // Note that no splitting happens if the delimiter
> // is empty:
> Split("abcd", "");
[ abcd ]
```

`Regexp(R, S): MonStgElt, MonStgElt -> BoolElt, MonStgElt, [ MonStgElt ]`

Given a string R specifying a regular expression, together with a string S, return whether S matches R. If so, return also the matched substring of S, together with the sequence of matched substrings of S corresponding to the parenthesized expressions of R. This function is based on the freely distributable reimplementation of the V8 regexp package by Henry Spencer. The syntax and interpretation of the characters `|`, `*`, `+`, `?`, `^`, `$`, `[]`, `\` is the same as in the UNIX command `egrep`. The parenthesized expressions are numbered in left-to-right order of their opening parentheses. Note that the parentheses should not have an initial backslash before them as the UNIX commands `grep` and `ed` require.

Example: Regular Expression Matching (ex-c69b5a)

We demonstrate some elementary uses of `Regexp`.

```
> Regexp("b.*d", "abcde");
true bcd []
> Regexp("b(.*)d", "abcde");
true bcd [ c ]
> Regexp("b.*d", "xyz");
false
> date := "Mon Jun 17 10:27:27 EST 1996";
> _, _, f := Regexp("([0-9][0-9]):([0-9][0-9]):([0-9][0-9])", date);
> f;
[ 10, 27, 27 ]
> h, m, s := Explode(f);
> h, m, s;
10 27 27
```

3.3 Printing

3.3.1 The print-Statement

print expression;

print expression, ..., expression;

print expression : parameters;

Print the value of the expression. Some limited ways of formatting output are described in the section on strings. Four levels of printing (that may in specific cases coincide) exist, and may be indicated after the colon: `Default` (which is the same as the level obtained if no level is indicated), `Minimal`, `Maximal`, and `Magma`. The last of these produces output representing the value of the identifier as valid Magma-input (when possible).

3.3.2 Formatted Printing

printf format, expression, ..., expression;

Print values of the expressions under control of format. The first argument, the *format string*, must be a string which contains two types of objects: plain characters, which are simply printed, and conversion specifications (indicated by the `%` character), each of which causes conversion and printing of zero or more of the expressions. (Use `%%` to get a literal percent character.) Currently, the only conversion specifications allowed are: `%o` and `%O`, which stand for "object", `%m`, which stands for "magma", and `%h`, which stands for "hexadecimal".

The hexadecimal conversion specification will print its argument in hexadecimal; currently, it only supports integer arguments. The object and magma conversion specifications each print the corresponding argument; they differ only in the printing mode used. The `%o` form uses the default printing mode, while the `%O` form uses the printing mode specified by the next argument (as a string). The "magma" conversion specification uses a printing mode of `Magma`. It is thus equivalent to (but shorter than) using `%O` and an extra argument of `"Magma"`.

For each of these conversion specifications, the object can be printed in a field of a particular width by placing extra characters immediately after the `%` character: digits describing a positive integer, specifying a field with width equal to that number and with right-justification; digits describing a negative integer, specifying a field with width equal to the absolute value of the number and with left-justification; or the character `*` specifying a field width given by the next appropriate expression argument (with justification determined by the sign of the number). This statement is thus like the C language function `printf()`, except that `%o` (and `%O` and `%m`) covers all kinds of objects — it is not necessary to have different conversion specifications for the different types of Magma objects. Note also that this statement does *not* print a newline character after its arguments while the `print` statement does (a `\n` character should be placed in the format string if this is desired). A newline character will be printed just before the next prompt, though, if there is an incomplete line at that point.

Example: Simple Uses of printf (ex-f268ee)

The following statements demonstrate simple uses of *printf*.

```
> for i := 1 to 150 by 33 do printf "[%3o]\\n", i; end for;
[  1]
[ 34]
[ 67]
[100]
[133]
> for i := 1 to 150 by 33 do printf "[%-3o]\\n", i; end for;
[1  ]
[34 ]
[67 ]
[100]
[133]
```

(continues on next page)

(continued from previous page)

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

Example: Further Uses of printf (ex-2f73a2)

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

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

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

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

Example: Printing to a Sorting Pipe with fprintf (ex-97d95e)

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

```
> p := 1000000000000000000000000000057;
> F := POpen("sort -n", "w");
> for i := 100 to 110 do
>    fprintf F, "%30o (2^%o mod p)\\n", 2^i mod p, i;
> end for;
> // Close F and then see output on standard output:
> delete F;
 371073168534535663120411155l9 (2^109 mod p)
 706024009129176059868128212l9 (2^102 mod p)
 742146337069071326240822310l8 (2^110 mod p)
129638414606681695789005139447 (2^106 mod p)
141204801825835211973625642438 (2^103 mod p)
259276829213363391578010278894 (2^107 mod p)
267650600228229401496703205319 (2^100 mod p)
282409603651670423947251284876 (2^104 mod p)
518553658426726783156020557788 (2^108 mod p)
535301200456458802993406410638 (2^101 mod p)
564819207303340847894502569752 (2^105 mod p)
```

3.3.3 Verbose Printing

The following statements allow convenient printing of information conditioned by whether an appropriate verbose flag is turned on.

vprint flag: expression, ..., expression;

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

If the verbose flag *flag* (see the function `SetVerbose`) has a level greater than or equal to n, print the expressions to the right of the colon exactly as in the `print` statement. If the flag has level 0 (i.e. is not turned on), do nothing. In the first form of this statement, where a specific level is not given, n is taken to be 1. This statement is useful in Magma code found in packages where one wants to print verbose information if an appropriate verbose flag is turned on.

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

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

If the verbose flag *flag* (see the function `SetVerbose`) has a level greater than or equal to n, print using the format and the expressions to the right of the colon exactly as in the `printf` statement. If the flag has level 0 (i.e. is not turned on), do nothing. In the first form of this statement, where a specific level is not given, n is taken to be 1. This statement is useful in Magma code found in packages where one wants to print verbose information if an appropriate verbose flag is turned on.

3.3.4 Automatic Printing

Magma allows *automatic printing* of expressions: basically, a statement consisting of an expression (or list of expressions) alone is taken as a shorthand for the `print`-statement. Some subtleties are involved in understanding the precise behaviour of Magma in interpreting lone expressions as statements. The rules Magma follows are outlined here. In the following, a *call-form* means any expression of the form $f(arguments)$; that is, anything which could be a procedure call *or* a function call.

(a) Any single expression followed by a semicolon which is not a call-form is printed, just as if you had 'print' in front of it.

(b) For a single call-form followed by a semicolon (which could be a function call or procedure call), the first signature which matches the input arguments is taken and if that is procedural, the whole call is taken as a procedure call, otherwise it is taken as function call and the results are printed.

(c) A comma-separated list of any expressions is printed, just as if you had 'print' in front of it. Here any call-form is taken as a function call only so procedure calls are impossible.

(d) A print level modifier is allowed after an expression list (whether the list has length 1 or more). Again any call-form is taken as a function call only so procedure calls are impossible.

(e) Any list of objects printed, whether by any of the above rules or by the 'print' statement, is placed in the previous value buffer. `$1` gives the last printed list, `$2` the one before, etc. Note that multi-return values stay as a list of values in the previous value buffer. The only way to get at the individual values of such a list is by assignment to a list of identifiers, or by `where` (this is of course the only way to get the second result out of `Quotrem`, etc.). In other places, a `$1` expression is evaluated with principal value semantics.

Magma also provides procedures to manipulate the previous value buffer in which $1, etc. are stored.

ShowPrevious()

Show all the previous values stored. This does *not* change the contents of the previous value buffer.

ShowPrevious(i): RngIntElt

Show the i-th previous value stored. This does *not* change the contents of the previous value buffer.

ClearPrevious()

Clear all the previous values stored. This is useful for ensuring that no more memory is used than that referred to by the current identifiers.

SetPreviousSize(n): RngIntElt

Set the size of the previous value buffer (this is not how many values are defined in it at the moment, but the maximum number that will be stored). The default size is 3.

GetPreviousSize() -> RngIntElt

Return the size of the previous value buffer.

Example: Automatic Printing (ex-feb63d)

Examples which illustrate point (a):

```
> 1;
1
> x := 3;
> x;
3
```

Examples which illustrate point (b):

```
> 1 + 1;            // really function call '+'(1, 1)
2
> Q := [ 0 ];
> Append(~Q, 1);    // first (in fact only) match is procedure call
> Append(Q, 1);     // first (in fact only) match is function call
[ 0, 1, 1 ]
> // Assuming fp is assigned to a procedure or function:
> fp(x);            // whichever fp is at runtime
> SetVerbose("Meataxe", true);  // simple procedure call
```

Examples which illustrate point (c):

```
> 1, 2;
1 2
> // Assuming f assigned:
> f(x), 1;                              // f only can be a function
> SetVerbose("Meataxe", true), 1;       // type error in 'SetVerbose'
>                                       // (since no function form)
```

Examples which illustrate point (d):

```
> 1: Magma;
1
> Sym(3), []: Maximal;
Symmetric group acting on a set of cardinality 3
Order = 6 = 2 * 3
[]
> SetVerbose("Meataxe", true): Magma; // type error as above
```

Examples which illustrate point (e):

```
> 1;
1
> $1;
1
> 2, 3;
2 3
> $1;
2 3
> Quotrem(124124, 123);
```

(continues on next page)

(continued from previous page)

```
1009 17
> $1;
1009 17
> a, b := $1;
> a;
1009
```

3.3.5 Indentation

Magma has an indentation level which determines how many initial spaces should be printed before each line. The level can be increased or decreased. Each time the top level of Magma is reached (i.e. a prompt is printed), the level is reset to 0. The level is usually changed in verbose output of recursive functions and procedures. The functions `SetIndent` and `GetIndent` are used to control and examine the number of spaces used for each indentation level (default 4).

`IndentPush()`

Increase (push) the indentation level by 1. Thus the beginning of a line will have s more spaces than before, where s is the current number of indentation spaces.

`IndentPush(C): RngIntElt`

Increases the indentation level by C.

`IndentPop()`

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

`IndentPop(C): RngIntElt`

Decreases the indent level by C.

3.3.6 Printing to a File

`PrintFile(F, x): MonStgElt, Any`

`Write(F, x): MonStgElt, Any`

```
Overwrite: BoolElt                    Default: false
```

Print x to the file specified by the string F. If this file already exists, the output will be appended, unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten.

`WriteBinary(F, s): MonStgElt, BStgElt`

```
Overwrite: BoolElt                    Default: false
```

Write the binary string s to the file specified by the string F. If this file already exists, the output will be appended, unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten.

`PrintFile(F, x, L): MonStgElt, Any, MonStgElt`

`Write(F, x, L): MonStgElt, Any, MonStgElt`

```
Overwrite: BoolElt                    Default: false
```

Print x in format defined by the string L to the file specified by the string F. If this file already exists, the output will be appended unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten. The level L can be any of the print levels on the `print` command above (i.e., it must be one of the strings `"Default"`, `"Minimal"`, `"Maximal"`, or `"Magma"`).

`PrintFileMagma(F, x): MonStgElt, Any`

```
Overwrite: BoolElt                    Default: false
```

Print x in Magma format to the file specified by the string F. If this file already exists, the output will be appended, unless the optional parameter `Overwrite` is set to true, in which case the file is overwritten.

3.3.7 Printing to a String

Magma allows the user to obtain the string corresponding to the output obtained when printing an object by means of the `Sprint` function. The `Sprintf` function allows formatted printing like the `printf` statement.

`Sprint(x): Elt -> MonStgElt`

`Sprint(x, L): Elt, MonStgElt -> MonStgElt`

Given any Magma object x, this function returns a string containing the output obtained when x is printed. If a print level L is given also (a string), the printing is done according to that level (see the `print` statement for the possible printing levels).

`Sprintf(F, ...): MonStgElt, ... -> MonStgElt`

Given a format string F, together with appropriate extra arguments corresponding to F, return the string resulting from the formatted printing of F and the arguments. The format string F and arguments should be exactly as for the `printf` statement – see that statement for details.

Example: Printing to a String with Sprintf (ex-66c7f1)

We demonstrate elementary uses of `Sprintf`.

```
> Q := [Sprintf("{%4o<->%-4o}", x, x): x in [1,10,100,1000]];
> Q;
[ {   1<->1   }, {  10<->10  }, { 100<->100 }, {1000<->1000} ]
```

3.3.8 Redirecting Output

`SetOutputFile(F): MonStgElt`

```
Overwrite: BoolElt                    Default: false
```

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

`UnsetOutputFile()`

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

`HasOutputFile() -> BoolElt`

If Magma currently has an output or log file F, return `true` and F; otherwise return `false`.

3.4 Input

3.4.1 Reading a Complete File

`Read(name): MonStgElt -> MonStgElt`

Returns the contents of the file named *name* as a string. Note that *name* may be an expression returning a string.

`ReadBinary(name): MonStgElt -> BStgElt`

Returns the contents of the file named *name* as a binary string.

Example: Reading a Complete File (ex-63220d)

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

```
> Read("mystery.c");
#include <stdio.h>

main(argc, argv)
int     argc;
char    **argv;
{
    int n, i;

    n = atoi(argv[1]);

    for (i = 1; i <= n; i++)
        printf("%d\\n", i * i);

    return 0;
}
> System("cc mystery.c -o mystery");
> mysteryMagma := function(n)
>    System("./mystery " cat IntegerToString(n) cat " >outfile");
>    output := Read("outfile");
>    return StringToIntegerSequence(output);
> end function;
> mysteryMagma(5);
[ 1, 4, 9, 16, 25 ]
```

3.4.2 Reading Command Output

The intrinsics in this section are not currently available on Windows platforms. This will be addressed in a future release.

`Pipe(cmd, input): MonStgElt, MonStgElt -> MonStgElt`

Given a shell command *cmd* and an input string *input*, run *cmd* in a new process with *input* as its input and return the output of this command as a string. Note that for many commands, *input* should finish with a new line character if it consists of only one line.

`Pipe(cmds, inputs): SeqEnum, SeqEnum -> SeqEnum`

Given a sequence of shell commands *cmds* and a sequence of input strings *inputs*, run each command in *cmds* in a new process with the corresponding element of *inputs* as its input. Returns the sequence of outputs (as strings) of the commands.

Example: Reading Output through a Pipe (ex-b6f57d)

We revisit the `mystery` program from the earlier example on file reading. Using a pipe means that we no longer need a temporary file.

```
> mysteryMagma := function(n)
>    cmd := Sprintf("./mystery %o", n);
>    output := Pipe(cmd, "");
>    return StringToIntegerSequence(output);
> end function;
> mysteryMagma(5);
[ 1, 4, 9, 16, 25 ]
```

Example: Alphabetic Shifts with External Commands (ex-7c1133)

In this example, we use external commands to apply alphabetic shifts to input words. First, we write a function of n to return the appropriate command to (cyclically) shift each letter forward n places in the alphabet. The UNIX `tr` command is used for this purpose.

```
> letters := "abcdefghijklmnopqrstuvwxyz";
> shiftcmd := func<n | Sprintf("tr 'a-z' '%o-za-%o'", letters[n+1], letters[n])>;
```

Then we form the commands for shifting by amounts from 1 to 13 and apply them to some well-chosen inputs.

```
> cmds := [ shiftcmd(n) : n in [1..13] ];
> inputs :=
> [
>     "steeds", "pyic", "dolt", "pecan", "fizzy", "fusion", "inkier", "talk",
>     "sleep", "cubed", "spots", "road", "nowhere"
> ];
> Pipe(cmds, inputs);
[ tuffet, rake, grow, tiger, kneed, layout, purply, bits, bunny, melon, dazed,
damp, abjurer ]
```

3.4.3 Interactive Input

`read identifier;`

`read identifier, prompt;`

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

`readi identifier;`

`readi identifier, prompt;`

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

3.5 I/O Objects

Magma provides the `IO` type to represent objects that can have I/O performed on them, such as files, pipes, and sockets. A uniform set of I/O operations are provided for all I/O objects, as well as some special operations for specific types. We will often use the term *channel* to refer to an I/O object.

`IOType(I): IO -> MonStgElt`

Given an I/O object I, returns the subtype of I as a string; currently this will be one of `"file"`, `"pipe"`, or `"socket"`, as appropriate.

3.5.1 Files

Magma provides a file I/O object for interacting with external files. The string returned by `IOType` for such an object is `"file"`. Note that there are also several intrinsics that interact with files using the file name alone (for examples, see the sections on printing and input).

`Open(name, mode): MonStgElt, MonStgElt -> IO`

Returns a Magma file object associated with the file named *name* for access of type *mode*. Tilde expansion is performed on *name*. The possible values for *mode* are those of the standard C library function `fopen()` on the current operating system, and have the same interpretation. Thus one should give the value `"r"` for *mode* to open the file for reading and give the value `"w"` for *mode*

to open the file for writing, etc. (Note that in the Windows version of Magma, the character "b" should be appended to *mode* if the file is desired to be opened in binary mode.)

Once a file object is created the various I/O operations described later can be performed on it. A file is closed by deleting it (i.e., by use of the delete statement or by reassigning the variable associated with the file); there is no Fclose function. This ensures that the file is not closed while there are still multiple references to it. (The function is called Open instead of Fopen to follow Perl-style conventions.)

OpenTest(name, mode): MonStgElt, MonStgElt -> BoolElt, IO

Attempts to create a file object as described in Open above. If it is successful then true will be returned together with the new file object. If the file cannot be opened then false is returned.

3.5.2 Pipes

Magma provides a pipe I/O object for interacting with external programs. The string returned by IOType for such an object is "pipe". Note that there are also some intrinsics that interact with programs using the program name alone (for examples, see the sections on input and system calls).

Pipes are not currently available on Windows platforms. This will be addressed in a future release.

POpen(cmd, mode): MonStgElt, MonStgElt -> IO

Runs the shell command *cmd* in a new process and returns a Magma pipe object associated with this command with access of type *mode*. The possible values for *mode* are those of the standard C library function popen() on the current operating system, and have the same interpretation. Thus one should give the value "r" for *mode* when wanting to read from the pipe and give the value "w" for *mode* when wanting to write to the pipe.

Bidirectional pipes are not currently supported, but see the Pipe intrinsic for a method for sending input to and receiving output from a single command.

Example: Current Time from a Pipe (ex-aff26a)

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

```
> function GetTime()
>     I := POpen("date", "r");
>     date := Read(I);
>     _, _, f := Regexp("([0-9][0-9]):([0-9][0-9]):([0-9][0-9])", date);
>     h, m, s := Explode(f);
>     return h, m, s;
> end function;
> h, m, s := GetTime();
> h, m, s;
14 30 01
> h, m, s := GetTime();
> h, m, s;
14 30 04
```

3.5.3 Sockets

Magma provides a socket I/O type for communicating over the network. The string returned by IOType for such an object is "socket".

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

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

The network address selects on which of the available network interfaces communication will take place; it is a string identifying the machine on that network, in either domain name or dotted-decimal format. For example, both `"localhost"` and `"127.0.0.1"` identify the machine on the loopback interface (which is only accessible from the machine itself), whereas `"foo.bar.com"` or `"10.0.0.3"` might identify the machine in a local network, accessible from other machines on that network. Currently only IPv4 addresses are supported.

The port number is just an integer that identifies the socket on a particular network interface. It must be less than 65 536. A value of 0 will indicate that the port number should be chosen by the operating system.

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

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

`Socket(H, P : parameters): MonStgElt, RngIntElt -> IOSocket`

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

Attempts to create a (client) socket connected to port P of host H. Note: these are the *remote* values; usually it does not matter which local values are used for client sockets, but for those rare occasions where it does they may be specified using the parameters `LocalHost` and `LocalPort`. If these parameters are not set then suitable values will be chosen by the operating system. Also note that port numbers below 1 024 are usually reserved for system use, and may require special privileges to be used as the local port number.

`Socket(: parameters) -> IOSocket`

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

Attempts to create a server socket on the current machine, that can be used to accept connections. The parameters `LocalHost` and `LocalPort` may be used to specify which network interface and port the socket will accept connections on; if either of these are not set then their values will be determined by the operating system. Note that port numbers below 1 024 are usually reserved for system use, and may require special privileges to be used as the local port number.

`WaitForConnection(S): IO -> IO`

This may only be used on server sockets. It waits for a connection attempt to be made, and then creates a new socket to handle the resulting communication channel. Thus S may continue to be used to accept connection attempts, while the new socket is used for communication with whatever entity just connected. Note: this new socket is *not* a server socket.

3.5.4 Other Socket Operations

`SocketInformation(S): IO -> Tup, Tup`

This routine returns the identifying information for the socket as a pair of tuples. Each tuple is a <*host, port*> pair — the first tuple gives the local information and the second gives the remote information. Note that this second tuple will be undefined for server sockets.

`IsServerSocket(S): IO -> BoolElt`

Returns whether S is a server socket or not.

Example: Sending a Message over a Socket (ex-7f96d9)

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

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

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

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

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

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

3.5.5 A Note on Socket Security

A server socket cannot tell which user is connecting to it. (Indeed, the very concept is not well-defined if the connection is made from another machine.) Thus it is possible for someone on another machine on the same network to connect to "your" socket, whether by accident or otherwise, and this cannot be directly distinguished from an intended connection from a process under your control. Problems may then arise if the data from this unexpected connection does not meet whatever requirements your code has.

If you are reasonably sure that no-one else with access to your machine or its network will act in such a way, then you can simply use the socket functions as described above. Otherwise, some kind of authentication process would be desirable. A simple method is to require that client connections first send some data known only to the server, and have the server reject connections which do not do so.

A more robust and convenient version of authentication is being worked upon, and will appear in a later patch release.

3.6 I/O Operations

The underlying I/O routines of the operating system act on collections of *bytes*, which are 8-bit quantities that can be thought of as integers in the range 0..255. The Magma I/O routines provide three different interpretations of these collections of bytes: as strings (textual data), as byte sequences, and as representations of Magma objects. The first two of these are unstructured — any amount of data is a valid instance — while the third viewpoint requires that the byte values conform to a certain protocol. This protocol is described in a separate document (Magma Object Transfer Protocol), available from the Magma website.

To make error handling easier, most of the I/O operations that read or write data also have checked variants that will not raise an error due to I/O failure but instead return whether the operation was successful.

3.6.1 End of File Marker

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

`Eof() -> MonStgElt`

Creates the special EOF string.

`IsEof(S): MonStgElt -> BoolElt`

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

`AtEof(I): IO -> BoolElt`

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

Example: End of File Marker (ex-7ebbe1)

We create the special EOF string and test it.

```
> eof := Eof();
> eof;
EOF
> IsEof(eof);
true
```

Note that the special EOF string is different from the string `"EOF"`, which is a normal string.

```
> E := "EOF";
> E;
EOF
> IsEof(E);
false
```

3.6.2 Strings

`Read(I : parameters): IO -> MonStgElt`

```
Max: RngIntElt                    Default: 0
```

Waits for data to become available for reading from I and then returns it as a string. If the parameter `Max` is set to a positive value then at most that many bytes will be read. Note that fewer than `Max` bytes may be returned, depending on the amount of currently available data.

If no data can be read from I then the special EOF string is returned.

`ReadCheck(I : parameters): IO -> BoolElt, MonStgElt`

Max: RngIntElt	Default: 0

Performs the same `Read` action as above. If an error would arise then instead `false` is returned; otherwise `true` is returned along with the result of the read.

`Read(I, n): IO, RngIntElt -> MonStgElt`

Attempts to read exactly n bytes of data from I and then returns it as a string; will wait if necessary until either n bytes have become available or the input has terminated. Note that the returned string may contain fewer than n bytes if the input has terminated.

If the input terminates and no bytes have been read then the special EOF string is returned.

`ReadCheck(I, n): IO, RngIntElt -> BoolElt, MonStgElt`

Performs the same `Read` action as above. If an error would arise then instead `false` is returned; otherwise `true` is returned along with the result of the read.

`Write(I, s): IO, MonStgElt`

Writes the characters of the string s to I.

`WriteCheck(I, s): IO, MonStgElt -> BoolElt`

Performs the same `Write` action as above. Returns whether the write completed without error.

3.6.3 Legacy String Operations

The I/O operations in this section are retained for compatibility with older versions of Magma. They reflect the semantics of the C standard I/O library at the time. Their use is somewhat discouraged, although there is currently no good alternative for getting a line of input at a time (as provided by `Gets`).

`Put(I, s): IO, MonStgElt`

Writes the string s to the channel I. This is equivalent to `Write(I, s)`.

`Puts(I, s): IO, MonStgElt`

Writes the string s to the channel I, followed by a newline character. This is equivalent to `Write(I, s cat "\n")`.

`Getc(I): IO -> MonStgElt`

Reads a single character from I and returns it as a string. This is equivalent to `Read(I, 1)`. If no data is available from I then the special EOF string is returned instead.

`Gets(I): IO -> MonStgElt`

Reads a line of input from I and returns it as a string. The line is considered to end when either a newline character is received or all data in I has been received. If a newline character is received then it is omitted from the returned string. If no data at all is available from I then the special EOF string is returned instead.

This routine is only currently available for files and pipes.

`Ungetc(I, c): IO, MonStgElt`

Pushes the given character (a string of length one) c into the start of the input buffer of I. The next read operation on I should return this character as the first byte.

This routine is only currently available for files and pipes.

Use of this intrinsic may interact unpredictably and poorly with asynchronous I/O, and especially so when reading objects. Its use is strongly discouraged and it will probably be removed in a future release.

Example: Counting Lines in a File (ex-81a099)

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

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

3.6.4 Byte Sequences

A *byte sequence* is a sequence of integers in the range 0..255. Note that byte sequences have the ring of integers as their universe; they will never be null.

ReadBytes(I : parameters): IO -> SeqEnum

```
Max: RngIntElt                    Default: 0
```

Waits for data to become available for reading from I and then returns it as a byte sequence. If the parameter **Max** is set to a positive value then at most that many bytes will be read. Note that fewer than **Max** bytes may be returned, depending on the amount of currently available data.

If no data can be read from I then an empty byte sequence is returned.

ReadBytesCheck(I : parameters): IO -> BoolElt, SeqEnum

```
Max: RngIntElt                    Default: 0
```

Performs the same **ReadBytes** action as above. If an error would arise then instead **false** is returned; otherwise **true** is returned along with the result of the read.

ReadBytes(I, n): IO, RngIntElt -> SeqEnum

Attempts to read exactly n bytes of data from I and then returns it as a byte sequence; will wait if necessary until either n bytes have become available or the input has terminated. Note that the returned sequence may contain fewer than n bytes if the input has terminated.

If the input terminates and no bytes have been read then an empty byte sequence is returned.

ReadBytesCheck(I, n): IO, RngIntElt -> BoolElt, SeqEnum

Performs the same **ReadBytes** action as above. If an error would arise then instead **false** is returned; otherwise **true** is returned along with the result of the read.

WriteBytes(I, S): IO, SeqEnum

Writes the bytes of the byte sequence S to I.

WriteBytesCheck(I, S): IO, SeqEnum -> BoolElt

Performs the same **WriteBytes** action as above. Returns whether the write completed without error.

3.6.5 Objects

Some Magma objects can be used in I/O calls. The details of which objects can be handled, as well as specifications of the format, are provided in a separate document (Magma Object Transfer Protocol), available from the Magma website.

As of V25.1 the allowable object types include integers, rationals, reals, complexes, booleans, strings, finite field elements, polynomials (univariate and multivariate), sequences, sets, indexed sets, multisets, tuples, lists, records, matrices, vectors, lattices, permutations, number field elements, and all the associated parent structures. Support for more types of objects will continue to be added in patch releases.

`ReadObject(I): IO -> Any`

Reads a Magma object from I in Magma Object Format and returns it. An error will be raised if input terminates before a complete object can be read.

`ReadObjectCheck(I): IO -> BoolElt, Any`

Performs the same `ReadObject` action as above. If an error would arise then instead `false` is returned; otherwise `true` is returned along with the result of the read.

`WriteObject(I, x): IO, Any`

Writes the Magma object x to I, using the Magma Object Format.

`WriteObjectCheck(I, x): IO, Any -> BoolElt`

Performs the same `WriteObject` action as above. Returns whether the write completed without error.

3.6.6 Versions

As described in the Magma Object Transfer Protocol document (available from the Magma website), each program that can read or write Magma objects does so according to the Protocol. This Protocol will continue to evolve as more objects are implemented and as more features are added.

In order to manage the different versions of the Protocol, each is associated with a unique *version number* (or simply *version*), and this version forms part of the data for each object. Programs that understand one version of the Protocol are also expected to understand all previous versions. Thus programs can detect if a received object uses a later version of the Protocol and raise an error, or cooperate to ensure that communication takes place using only a mutually-understood version of the Protocol.

In the following, a version number is represented as a string of the form "$M.m$", where M and m are integers between 0 and 65 535 (inclusive). When two programs wish to exchange Magma objects, the first step should be for each to notify the other of the version of the Protocol that it understands. Each program can then send objects using the lower of the two versions, ensuring that the object data is correctly interpreted by the other program.

`GetMOTPMaxVersion() -> MonStgElt`

Returns the maximum (latest) version number of the Protocol that is understood by this copy of Magma.

`GetMOTPVersion() -> MonStgElt`

Returns the version number of the Protocol that will be used by Magma for newly-created I/O objects. This will usually be the same as the maximum version, but the user can explicitly lower it by using `SetMOTPVersion`.

`SetMOTPVersion(v): MonStgElt`

Sets the version of the Protocol to use with newly-created channels. This may not be set to a value higher than the maximum version (as returned by `GetMOTPMaxVersion`). Usually it is not desirable to change this value as version negotiation takes care of communication issues, but it can be useful when testing or when saving objects that are to be read later by programs that only understand an earlier version of the Protocol.

`SetMOTPVersion(I, v): IO, MonStgElt`

Sets the version of the Protocol to use from now on with the given channel. Usually it is not desirable to change this value as version negotiation takes care of communication issues, but it can be useful when testing or when saving objects that are to be read later by programs that only understand an earlier version of the Protocol.

`NotifyVersion(I): IO`

Sends a version Protocol message to the other end of the channel that contains the desired version number to use for communication. i.e., a version number that this program will be able to understand. This will be either the value returned by `GetMOTPVersion` when the channel was created, or the value used in a call to `SetMOTPVersion` on the channel.

It is allowed, but unusual, to send versions in this way after the initial exchange of versions occurs. The other end of the channel is expected to update its internal value to use for communication when it receives such a message. For more details see the Magma Object Transfer Protocol document.

`ExpectVersion(I): IO`

Expects a version Protocol message containing a version number understood by the other end of the communication channel. This version will be stored internally and Protocol messages sent to the other end of the channel will use a version no higher than this value. An error will arise if a different Protocol message was received when a version was expected.

`ExchangeVersions(I): IO`

Performs both a `NotifyVersion` and `ExpectVersion` on the channel. This is the preferred method to use; after a communication channel has been set up then the first thing both sides should do is to call `ExchangeVersions` (or its asynchronous version, see later) on the channel. Thereafter objects can be safely sent and received.

Note: It is often the case that one is using the same version of Magma on each machine, and thus that each process understands the same version of the Protocol. It may thus be tempting to omit the use of `ExchangeVersions` under such circumstances, and this is reasonable for interactive sessions. However, it is strongly encouraged to use `ExchangeVersions` in longer-lived code; doing so will save many headaches down the line when one finds oneself trying to get different versions of Magma to interact.

Example: Exchanging Objects between Processes (ex-9cbada)

We demonstrate the appropriate process to use for the simple exchange of objects between two Magma processes. First we create the server socket and wait for the connection. In this case we explicitly give the local host and port to use in order to slightly simplify the example.

```
> // First Magma process
> server := Socket(: LocalHost := "localhost", LocalPort := 4000);
> S1 := WaitForConnection(server);
```

The first Magma process is now waiting for a connection to be made. We create a client socket in another Magma process on the same machine and establish the connection with the first process. Since we intend to exchange objects over this connection, we also do the version exchange immediately.

```
> // Second Magma process
> S2 := Socket("localhost", 4000);
> ExchangeVersions(S2);
```

Now the second Magma process is waiting for the version exchange to be complete. The first process has received the connection and completes the version exchange.

```
> // First Magma process
> ExchangeVersions(S1);
```

Both processes are now ready to communicate using `ReadObject` and `WriteObject`. We demonstrate some simple exchanges.

```
> // First Magma process
> WriteObject(S1, [ IsPrime(x) : x in [1..10] ]);
```

```
> // Second Magma process
> ReadObject(S2);
[ false, true, true, false, true, false, true, false, false, false ]
> WriteObject(S2, IdentityMatrix(Rationals(), 2));
```

```
> // First Magma process
> ReadObject(S1);
[1 0]
[0 1]
> Parent($1);
Full Matrix Algebra of degree 2 over Rational Field
```

It is possible to use `eval` and Magma-level printing in order to send objects of types that not yet handled by the Protocol.

```
> // First Magma process
> E := EllipticCurve("37a1");
> E;
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
> WriteObject(S1, E);

>> WriteObject(S1, E);
               ^
Runtime error in 'WriteObject': Unable to serialize specified object
> WriteObject(S1, Sprint(E, "Magma"));
```

```
> // Second Magma process
> E := eval ReadObject(S2);
> E;
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

3.6.7 Positioning

Some I/O types have positioning information associated with them, and this information can be retrieved and modified to allow access to specific parts of the data. Essentially, we regard the object as equivalent to a large array of bytes and the position as being the index within that array that the next I/O call will start from.

I/O types that support this view are called *seekable*, after the C library function for manipulating the position. Currently only file objects are seekable. Note: The position starts at 0, not 1, reflecting the C conventions of the underlying operating system calls.

`Tell(I): IO -> RngIntElt`

Given a seekable channel I, returns the current position as an integer.

`Seek(I, offset): IO, RngIntElt`

`Seek(I, offset, origin): IO, RngIntElt, RngIntElt`

Adjusts the position of the seekable channel I to be at the given offset relative to the origin. The origin must be one of: 0 to mean relative to the beginning; 1 to mean relative to the current position; or 2 to indicate relative to the end of the channel. If the origin is not specified then it is taken to be 0.

`Rewind(I): IO`

Moves the position of the seekable channel I to the beginning. This is equivalent to `Seek(I, 0)`.

3.6.8 Other I/O Operations

`Flush(I): IO`

This routine ensures that all pending output on the given channel has been committed. It is useful to ensure that previously-committed asynchronous writes have completed, and also to ensure that any buffered data has actually been written.

3.7 Managing Multiple I/O Objects

Some care is needed when using multiple communication channels at once in order to maintain responsiveness, since any individual I/O operation might take arbitrarily long to complete. (For instance, reading from a channel will not complete until the other end of the channel has written the requisite data.) This is particularly important when implementing distributed parallelism, as processes which are waiting for I/O to occur are not doing useful work.

In particular, consider the manager/worker model of parallelism, where a manager process assigns pieces of work to individual worker processes, collects the results, and assigns new work, repeating until all of the desired work has been completed. If the manager waits for the results from a particular worker (i.e., performs a read operation on the associated channel) then they will do nothing else until that worker has finished their task. Other workers who finish their tasks in the meantime will not be given more work to do until the first worker is finished, potentially causing them to wait for a very long time doing nothing.

One solution to this problem is for the manager not to wait unconditionally on a specific worker, but instead to wait until at least one of the set of workers has results to report, and then process those. In this way the manager never spends time waiting for one worker while another worker has results to report, and thus workers never spend significant time waiting for the manager to receive their results and give them new work to do.

Reinterpreting this approach in terms of I/O channels, the above indicates that we want a way to wait until at least one of a set of channels is ready for I/O to be performed on it. (By "ready" we mean that an I/O operation of the appropriate type will make at least some progress on the channel without further waiting.) This is accomplished with the `WaitForIO` intrinsic, as described below.

Note that, in principle, I/O operations may only partially complete before waiting becomes necessary. (For instance, network congestion may delay some network packets.) Thus some delays might still arise by using this approach with I/O operations that cannot partially complete (all writes; reads where the byte count is specified; reads of objects). In most cases this is rarely a serious issue, but a more robust approach is possible by using asynchronous I/O, as described in the next section.

`WaitForIO(R : parameters): SeqEnum -> SeqEnum`

`TimeLimit: RngIntElt`	`Default: Infinity()`

Given a sequence of I/O objects R, returns the sequence of those elements of R which are ready to have read operations performed upon them. If no elements of R are ready then this intrinsic will wait until at least one does become ready, or until the specified time limit has elapsed, whichever comes first. The time limit is measured in milliseconds.

Server sockets will be considered ready for reading when a connection attempt has been made and so a call to `WaitForConnection` will return without delay. Other channels will be considered ready for reading if either a pending read request has been fulfilled, or there are no pending read requests and some data is available for reading.

`WaitForIO(R, W : parameters): SeqEnum -> SeqEnum, SeqEnum`

`TimeLimit: RngIntElt`	`Default: Infinity()`

Given sequences of I/O objects R and W, returns two sequences: the sequence of those elements of R which are ready to have read operations performed upon them, and the sequence of those elements of W which are ready to have write operations performed upon them. If no elements of

R or W are ready then this intrinsic will wait until at least one does become ready, or until the specified time limit has elapsed, whichever comes first. The time limit is measured in milliseconds.

The condition for channels to be considered ready for reading are as described in the previous intrinsic. Channels will be considered ready for writing if all pending write requests have been fulfilled and it would be possible to write some data without delay.

Example: Simple Chat Server (ex-8b1b67)

We demonstrate the use of `WaitForIO` by writing an extremely simple "chat" server. This will accept connections and read messages from connected channels, forwarding those messages to all connected channels. A simple ID scheme based on port numbers is implemented in order to distinguish senders.

```
> chatserver := procedure(host, port)
>    server := Socket(: LocalHost := host, LocalPort := port);
>    chatters := {};
>    ids := AssociativeArray();
>    while true do
>       text := "";
>       ready := WaitForIO([server] cat Setseq(chatters));
>       for I in ready do
>          if I eq server then    // new chatter
>             C := WaitForConnection(server);
>             id := Sprintf("[%o]", r[2]) where _,r := SocketInformation(I);
>             text cat:= Sprintf("%o has joined\n", id);
>             ids[C] := id;
>             Include(~chatters, C);
>          else
>             ok, msg := ReadCheck(I);
>             if not ok or IsEof(msg) then    // error or EOF, assume they left
>                text cat:= Sprintf("%o has left\n", ids[I]);
>                Remove(~ids, I);
>                Exclude(~chatters, I);
>             else    // all is well; add message to text to send
>                text cat:= Sprintf("%o: %o", ids[I], msg);
>             end if;
>          end if;
>       end for;
>       if #text gt 0 then
>          for I in chatters do
>             Write(I, text);
>          end for;
>       end if;
>    end while;
> end procedure;
> chatserver("localhost", 7000);
```

After starting the chatserver; we test it (not shown here) using the UNIX `telnet` program to connect to the given host and port. As desired, messages sent by connected telnet instances are seen by all of the connected telnet instances.

Exercise: Modify the chat server to allow a client to change its reported ID by sending an appropriate message. Suggested syntax: `/name` *new-name*. Be careful with newlines.

Further exercise: Disallow changing ID to an already-used ID. If that is attempted, send an error message to the relevant client only.

3.8 Asynchronous I/O

One difficulty that arises when coordinating multiple channels is dealing with incomplete data. Due to network delays or other factors, data written by one side may arrive at the reader's side in several chunks with time delays between them. If we insist on reading all of the data then we will wait, and become unresponsive to the other channels in the meantime. If we try to manage partial data and reassemble it ourselves then the I/O handling becomes extremely complicated.

Magma's resolution of this difficulty is called *asynchronous I/O*. In asynchronous I/O, instead of performing I/O operations immediately they are instead queued to occur on the channel. At some later time they will complete, and this completion status can be waited for with `WaitForIO`. Thus it is possible to manage multiple channels while staying responsive, and using a simple interface.

Important: Currently Magma does not support truly asynchronous I/O; instead, asynchronous I/O operations are only guaranteed to advance during a call to `WaitForIO` with the associated channel in one of the argument lists. This meshes well with common paradigms, but it is important to be aware of it. In particular, queueing an asynchronous write without a following `WaitForIO` may never perform the write.

The asynchronous I/O intrinsics all start with `Async`, and are the counterparts of the corresponding synchronous intrinsics (without the `Async`). After an asynchronous read is performed, the data must be retrieved by using the corresponding synchronous read. An error will arise if the synchronous read operation is not compatible with the corresponding asynchronous read that was queued.

3.8.1 A Note on Files and Pipes

File objects are considered to be "fast" — it is expected that I/O on them will always complete immediately, without waiting. This is not always true (for instance, with network-mounted filesystems), but reflects an equivalent assumption in Linux. Consequently, although it is permitted to pass file objects to `WaitForIO` it is rarely useful, and asynchronous I/O operations will attempt to complete immediately.

Pipe objects are not properly handled yet, and attempting to perform asynchronous I/O on them will produce an error. When used with `WaitForIO`, pipes will incorrectly be treated as fast channels, which may cause undesired blocking. These issues will be fixed in an upcoming version of Magma.

3.8.2 Strings

`AsyncRead(I : parameters): IO`

```
Max: RngIntElt                    Default: 0
```

Queues an asynchronous read request of a string to I. If the parameter `Max` is set to a positive value then at most that many bytes will be read. Note that the resulting string may contain fewer than `Max` bytes, depending on the amount of data that becomes available.

`AsyncRead(I, n): IO, RngIntElt`

Queues an asynchronous read request of an n-byte string to I. Note that the resulting string may contain fewer than n bytes if the input has terminated.

`AsyncWrite(I, s): IO, MonStgElt`

Queues an asynchronous write request of the string s to I.

3.8.3 Byte Sequences

`AsyncReadBytes(I : parameters): IO`

```
Max: RngIntElt                    Default: 0
```

Queues an asynchronous read request of a byte sequence to I. If the parameter `Max` is set to a positive value then at most that many bytes will be read. Note that the resulting sequence may contain fewer than `Max` bytes, depending on the amount of data that becomes available.

AsyncReadBytes(I, n): IO, RngIntElt

Queues an asynchronous read request of an n-byte sequence to I. Note that the resulting sequence may contain fewer than n bytes if the input has terminated.

AsyncWriteBytes(I, S): IO, SeqEnum

Queues an asynchronous write request of the byte sequence S to I.

3.8.4 Objects

See Subsection *Objects* for more information about I/O with Magma objects and for descriptions of the corresponding synchronous functions.

AsyncReadObject(I): IO

Queues an asynchronous read request of an object to I.

AsyncWriteObject(I, x): IO, Any

Queues an asynchronous write request of x to I.

3.8.5 Versions

See Subsection *Versions* for an explanation of versions and for descriptions of the corresponding synchronous functions.

AsyncNotifyVersion(I): IO

Queues an asynchronous version notify request to I.

AsyncExpectVersion(I): IO

Queues an asynchronous read request of a version to I.

AsyncExchangeVersions(I): IO

Queues an asynchronous exchange of versions to I.

Example: Collatz Workers with Asynchronous I/O (ex-c7bac3)

We show an implementation of the manager/worker model that uses asynchronous I/O and object transmission, greatly simplifying the I/O loop.

The workload chosen will be computing hailstone sequences. Given an integer n, we iterate the Collatz map until we reach 1; the number of iterations taken and the maximum intermediate value will be returned to the manager. For the sake of simplicity we assume that 1 will always be reached.

The worker side of the computation involves a work function to actually do the work and a simple wrapper around it that interacts with the manager. The work function will take whatever object the manager sends, interpret it, do the computation, and return a single object as the result. Since we have multiple values to return, we combine them into a single tuple. A small random time delay is inserted in each computation for the purposes of this example so that the effects of multiple workers can be seen; real workloads would not include such a delay.

```
> collatz := function(n)
>    orign := n;
>    niters := 0;
>    maxval := n;
>    while n gt 1 do
>       n := IsEven(n) select n div 2 else 3*n + 1;
>       niters +:= 1;
>       maxval := Max(maxval, n);
>    end while;
>    // delay by 100, 200, or 300 milliseconds:
>    _ := WaitForIO([] : TimeLimit := 100*Random(1,3));
>    return <orign, niters, maxval>;
> end function;
```

The wrapper for the worker takes a host and port to contact the manager on, and is a simple work-processing loop thereafter. It uses synchronous I/O calls throughout, as the only I/O it could be waiting for involves the server.

```
> worker := procedure(host, port)
>    S := Socket(host, port);
>    ExchangeVersions(S);
>    while true do
>       ok, data := ReadObjectCheck(S);
>       if not ok then break; end if;
>       result := collatz(data);
>       WriteObject(S, result);
>    end while;
> end procedure;
```

First we create the communication channel and then immediately exchange versions with the server. This ensures that the program will work even if the manager and workers are using different versions of Magma. Then we loop on the work, despatching it and returning the results; using object I/O avoids any concerns about incomplete I/O operations. We use `ReadObjectCheck` in order to gracefully exit on EOF.

Now for the manager side of things. The manager will use asynchronous I/O calls in order to remain responsive. For the purposes of this example we will print output about the workers, so we start with a simple function to produce an identifier for each communication channel.

```
> channelid := func<I | t where _,t := SocketInformation(I)>;
```

The manager takes a sequence of work items as its argument (in this example, the work items are simply integers). It would also be reasonable for it to take the host and port information that it should listen on (since the workers will need to know this), but we instead let the system select those values and print them out.

```
> manager := procedure(workleft)
>    server := Socket();
>    host, port := Explode(SocketInformation(server));
>    printf "Manager listening on port %o of %o\n", port, host;
>    busy := {};
>    while #workleft gt 0 or #busy gt 0 do
>       ready := WaitForIO([server] cat Setseq(busy));
>       idle := [];          // workers awaiting work
>       for I in ready do
>          if I eq server then    // new worker has connected
>             W := WaitForConnection(I);
>             printf "New worker %o joined\n", channelid(W);
>             AsyncExchangeVersions(W);
>             Append(~idle, W);
>          else
>             result := ReadObject(I);
>             n, count, maxv := Explode(result);
>             printf "%o: %o reached 1 (max %o) after %o iterations\n",
>                 channelid(I), n, maxv, count;
>             Exclude(~busy, I);
>             Append(~idle, I);
>          end if;
>       end for;
>       for W in idle do
>          if #workleft eq 0 then break; end if;
>          data := workleft[1];
```

(continues on next page)

(continued from previous page)

```
>          Remove(~workleft, 1);
>          AsyncWriteObject(W, data);
>          AsyncReadObject(W);
>          Include(~busy, W);
>       end for;
>    end while;
> end procedure;
```

First the manager creates the server socket for workers to connect to; it prints out the associated information so that we know what arguments to give to start the workers with.

During the computation, the set `busy` will keep track of those channels associated with workers that currently have work assigned to them, and the sequence `idle` will track those workers which are waiting to have work assigned to them. The manager is done only when all work has been assigned, *and* all assigned work has been completed.

We wait for new workers to connect (on the server socket) and for completed work to be reported by the busy workers. A subtle point in the above is that we used the version of `WaitForIO` that only waits for read events. This is the typical case — the manager queues both the write and the read, and the worker will only send results after having read the data. Thus only the completion of the read event is of interest to the manager.

After a new worker joins then version exchange is queued immediately, since we will be sending and receiving objects. The worker is then added to the idle list.

When a worker has completed its task and the result is ready, we perform the synchronous `ReadObject` to get this result. We then mark the worker as idle (and no longer busy).

Finally, for each idle worker we attempt to (asynchronously) assign remaining work to it. We queue both the write and the following read, and mark the worker as busy.

We now demonstrate the code in action with two workers. First we start the manager in order to get the channel information for the workers to use.

```
> // manager
> manager([1..10]);
Manager listening on port 46119 of 0.0.0.0
```

Then we launch the workers (in this case, on the same machine).

```
> // worker 1
> worker("localhost", 46119);
```

```
> // worker 2
> worker("localhost", 46119);
```

And the manager displays the progress of the computation for us. Note that without the introduced delay the first worker would probably have performed all the computation before we could launch the second one.

```
New worker <"127.0.0.1", 43352> joined
<"127.0.0.1", 43352>: 1 reached 1 (max 1) after 0 iterations
New worker <"127.0.0.1", 43354> joined
<"127.0.0.1", 43354>: 3 reached 1 (max 16) after 7 iterations
<"127.0.0.1", 43352>: 2 reached 1 (max 2) after 1 iterations
<"127.0.0.1", 43352>: 5 reached 1 (max 16) after 5 iterations
<"127.0.0.1", 43354>: 4 reached 1 (max 4) after 2 iterations
<"127.0.0.1", 43354>: 7 reached 1 (max 52) after 16 iterations
<"127.0.0.1", 43352>: 6 reached 1 (max 16) after 8 iterations
```

(continues on next page)

(continued from previous page)

```
<"127.0.0.1", 43354>: 8 reached 1 (max 8) after 3 iterations
<"127.0.0.1", 43352>: 9 reached 1 (max 52) after 19 iterations
<"127.0.0.1", 43354>: 10 reached 1 (max 16) after 6 iterations
```

3.9 Loading Program Files

Load(F): MonStgElt

(Procedure.) Specify that the file with the name given by the string F (which may be any expression which evaluates to a string) is to be loaded immediately after this procedure call. Like the `load` directive below, the file F will then be read in, and the text will be treated as Magma input. Tilde expansion of file names is allowed. This procedure call may only be used in a simple statement at the top level.

Load(Q): [MonStgElt]

(Procedure.) Specify that the files with the names given by the strings in sequence Q are to be loaded in order immediately after this procedure call. As above, the files will be read in, and the text will be treated as Magma input. Tilde expansion of file names is allowed. The empty sequence is allowed for Q, in which case nothing is loaded. This procedure call may only be used in a simple statement at the top level.

load "filename";

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

A file name that is a valid identifier may also be given without quotes, as in `load myfile;`. A relative name, quoted or not, is looked for in the current directory, then in the directories listed in `MAGMA_PATH`, then in each `MAGMA_LIBRARIES` subdirectory of `MAGMA_LIBRARY_ROOT`. Unless `MAGMA_LIBRARY_ROOT` is already set, the `magma` start-up script sets it to the `libs` directory of the installation, described in `libs/README`. See `SetPath` and `SetLibraries` in *Set and Get*.

iload "filename";

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

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

3.10 Saving and Restoring Workspaces

save "filename";

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

restore "filename";

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

3.11 Logging a Session

SetLogFile(F): MonStgElt

```
Overwrite: BoolElt                    Default: false
```

Set the log file to be the file specified by the string F: all input and output will be sent to this log file as well as to the terminal. If a log file is already in use, it is closed and F is used instead. By using `SetLogFile(F: Overwrite := true)` the file F is emptied before input and output are written onto it. See also `HasOutputFile`.

UnsetLogFile()

Stop logging Magma's output.

SetEchoInput(b): BoolElt

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

3.12 Memory Usage

GetMemoryUsage() -> RngIntElt

Return the current memory usage of Magma (in bytes as an integer). This is the process data size, which does not include the executable code.

GetMaximumMemoryUsage() -> RngIntElt

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

ResetMaximumMemoryUsage()

Reset the value of the maximum memory usage of Magma to be the current memory usage of Magma (see `GetMaximumMemoryUsage`).

3.13 System Calls

Alarm(s): RngIntElt

A procedure which when used on UNIX systems, sends the signal SIGALRM to the Magma process after s seconds. This allows the user to specify that a Magma-process should self-destruct after a certain period.

ChangeDirectory(s): MonStgElt

Change to the directory specified by the string s. Tilde expansion is allowed.

GetCurrentDirectory()

Returns the current directory as a string.

Getpid()

Returns Magma's process ID (value of the Unix C system call `getpid()`).

Getuid()

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

Fork() -> RngIntElt

```
ExitSummary: BoolElt                  Default: false
```

Execute the Unix/Linux system call `fork()` and return the process ID of the child (if now the parent), or return 0 (if now the child). If the parameter `ExitSummary` is set to `true`, then the child process prints the usual exit summary (`"Total time: ..."`) when it exits; otherwise this is suppressed (the default).

`System(C): MonStgElt -> RngIntElt`

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

Example: Current Date from a System Call (ex-52e3aa)

The `System` intrinsic can be used to access the current date and time.

```
> System("date");
Tue Feb 28 13:02:03 EST 2017
```

`%! shell-command`

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

3.14 Creating Names

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

`Tempname(P): MonStgElt -> MonStgElt`

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

Chapter 4

Environment and Options

4.1 Introduction

This chapter describes the environmental features of Magma, together with options which can be specified at start-up on the command line, or within Magma by the `Set-` procedures. The history and line-editor features of Magma are also described.

4.2 Command Line Options

When starting up Magma, various command-line options can be supplied, and a list of files to be automatically loaded can also be specified. These files may be specified by simply listing their names as normal arguments (i.e., without a `-` option) following the Magma command. For each such file name, a search for the specified file is conducted, starting in the current directory, and in directories specified by the environment variable `MAGMA_PATH` after that if necessary. It is also possible to have a *startup file*, in which one would usually store personal settings of parameters and variables. The startup file is specified by the `MAGMA_STARTUP_FILE` environment variable which should be set in the user's `.cshrc` file or similar. This environment variable can be overridden by the `-s` option, or cancelled by the `-n` option. The files specified by the arguments to Magma are loaded *after* the startup file. Thus the startup file is not cancelled by giving extra file arguments, which is what is usually desired. The special argument `-` can be used to terminate option processing, so that later arguments are treated as ordinary file or script arguments even if they begin with `-`. This is needed when a filename or script argument starting with `-` should not be interpreted as a Magma option. Short command-line options that do not take an argument may be combined; for example, `-bn` is equivalent to `-b -n`.

Magma also allows one to set variables from the command line — if one of the arguments is of the form *var*`:=`*val*, where *var* is a valid identifier (consisting of letters, underscores, or non-initial digits) and there is no space between *var* and the `:=`, then the variable *var* is assigned within Magma to the *string* value *val* at the point where that argument is processed. (Functions like `StringToInteger` should be used to convert the value to an object of another type once inside Magma.)

`magma -b`

If the `-b` argument is given to Magma, the opening banner and all other introductory messages are suppressed. The final "total time" message is also suppressed. This is useful when sending the whole output of a Magma process to a file so that extra removing of unwanted output is not needed.

`magma -c filename`

If the `-c` argument is given to Magma, followed by a filename, the filename is assumed to refer to a package source file and the package is compiled and Magma then exits straight away. This option is rarely needed since packages are automatically compiled when attached.

magma -d

If the `-d` option is supplied to Magma, the licence for the current `magmapassfile` is dumped. That is, the expiry date and the valid hostids are displayed. Magma then exits.

magma -h

magma -?

magma --help

If the `-h`, `-?`, or `-help` option is supplied to Magma, a summary of the command-line syntax is displayed and Magma then exits.

magma -e arguments

magma -E arguments

If the `-e` or `-E` options are used, then the remaining arguments will be treated as Magma input, and Magma will exit after processing that input. This allows Magma to be used more easily from within scripts or pipelines, where (usually) only a single calculation is desired. The `-E` option is equivalent to `-b -e`.

magma -i filename arguments

magma -I filename arguments

Specifying the `-i` or `-I` options allows Magma to be used as the interpreter in the "shebang" line of a UNIX script. This can simplify the bundling of a common Magma calculation into a script. In such a case, the filename argument will be provided to Magma by the operating system and does not need to be specified. It can be retrieved from within Magma with the `GetScriptFilename` intrinsic. Arguments after the script filename are parsed as command-line options until a non-option argument or `-` is reached. The remaining arguments are *not* processed but are available (as strings) from the `GetScriptArguments` intrinsic. Use `-` before script arguments that begin with `-`. The `-I` option is equivalent to `-bn -i`.

magma -l limit

If the `-l` option is supplied to Magma, the memory limit is set to `limit` bytes at startup. This is equivalent to calling `SetMemoryLimit(limit)` after Magma has started; a limit of 0 means that no limit is imposed.

magma -n

If the `-n` option is supplied to Magma, any startup file specified by the environment variable `MAGMA_STARTUP_FILE` or by the `-s` option is cancelled.

magma -P

If the `-P` option is supplied to Magma, Magma reparses the package specification files of packages shipped with Magma at startup. This is useful if one of these shipped package files has been changed.

magma -r workspace

If the `-r` option is supplied to Magma, together with a workspace file, that workspace is automatically restored by Magma when it starts up.

magma -s filename

If the `-s` option is supplied to Magma, the given filename is used for the startup file for Magma. This overrides the variable of the environment variable `MAGMA_STARTUP_FILE` if it has been set. This option should not be used (as it was before), for automatically loading files since that can be done by just listing them as arguments to the Magma process.

magma -S integer

When starting up Magma, it is possible to specify a seed for the generation of pseudo-random numbers. (Pseudo-random quantities are used in several Magma algorithms, and may also be generated explicitly by some intrinsics.) The seed should be in the range 0 to $(2^{32} - 1)$ inclusive. If the seed is omitted, `-S` must be the last argument on the command line and Magma selects the seed itself. If the `-S` option is not used, Magma also selects the seed itself.

`magma -t N`

If the `-t` option is supplied to Magma at the command line, together with a positive number N, then the number of POSIX threads is set to N at startup (so this is equivalent to calling `SetNthreads(N);` after Magma has started).

`magma -V`

`magma --version`

If the `-V` argument is given to Magma, the version of Magma (without the initial 'V') is printed, and then Magma exits immediately. The option `-version` is equivalent. These options must not be used with any other command-line options.

Example: Startup with Files and Assignments (ex-10ce61)

By typing the command `magma file1 x:=abc file2` Magma would start up, read the user's startup file specified by `MAGMA_STARTUP_FILE` if existent, then read the file `file1`, then assign the variable `x` to the string value `"abc"`, then read the file `file2`, then give the prompt.

Example: Magma as a Command Line Calculator (ex-f69654)

In this example and the next one, the dollar symbol at the start of a line indicates the shell prompt, and does not form part of the input to the shell.

We can direct Magma to perform a specific computation from the command line using the `-e` or `-E` options. For instance: `$ magma -E 5^2 + 12^2 169` Magma runs, interprets the remaining arguments as input, performs the associated computations (printing output as appropriate), then exits.

Note that characters that are significant to the shell, such as brackets, will need to be protected in some fashion; the simplest approach is likely to enclose the entire command in single quotes. `$ magma -E 'Binomial(10, 4)*Binomial(5, 3)' 2100` Without the use of single quotes the shell will attempt to handle the brackets itself and raise a syntax error.

Example: Magma in a Shebang Script (ex-49332c)

Using Magma in the interpreter field of a UNIX script can simplify the automation of Magma computations. Suppose we put the following into a file named `prod`: `#!/usr/local/bin/magma -I print "3 * 4 =", 3*4;` After making `prod` executable, we run it, with unsurprising results: `$ prod 3 * 4 = 12` We can modify it as follows to get the values to multiply from the user-provided arguments: `#!/usr/local/bin/magma -I args := GetScriptArguments(); a := StringToInteger(args[1]); b := StringToInteger(args[2]); printf "\%o * \%o = \%o\n", a, b, a*b;` And now we can vary the arguments to suit our needs of the moment: `$ prod 5 7 5 * 7 = 35`

4.3 Environment Variables

This section lists some environment variables used by Magma. These variables are set by an appropriate operating system command and are used to define various search paths and other run-time options.

`MAGMA_STARTUP_FILE`

The name of the default start-up file. It can be overridden by the `magma -s` command.

`MAGMA_PATH`

Search path for files that are loaded (a colon separated list of directories). It need not include directories for the libraries, just personal directories. This path is searched before the library directories.

`MAGMA_MEMORY_LIMIT`

Limit on the size of the memory that may be used by a Magma-session (in bytes).

`MAGMA_LIBRARY_ROOT`

The root directory for the Magma libraries (by supplying an absolute path name). From within Magma `SetLibraryRoot` and `GetLibraryRoot` can be used to change and view the value.

MAGMA_LIBRARIES

Give a list of Magma libraries (as a colon separated list of sub-directories of the library root directory). From within Magma `SetLibraries` and `GetLibraries` can be used to change and view the value.

MAGMA_SYSTEM_SPEC

The Magma system spec file containing the system packages automatically attached at start-up.

MAGMA_USER_SPEC

The personal user spec file containing the user packages automatically attached at start-up.

MAGMA_HELP_DIR

The root directory for the Magma help files.

MAGMA_TEMP_DIR

Optional variable containing the directory Magma is to use for temporary files. If not specified, this defaults to `/tmp` (on Unix-like systems) or the system-wide temporary directory (on Windows systems).

MAGMA_NO_HISTORY

If this variable is set to anything at all, Magma keeps no command history: no file is read, written, or made, and no directory either. Unlike the `history` option (Section *The Options File*), which is read from a file inside the history directory, it is checked before that directory is made.

MAGMA_CONFIG_DIR

Optional variable naming a directory in which Magma keeps a persistent command history across sessions (Section *The History File*). The command lines typed in interactive sessions are saved there and reloaded in later sessions, and the directory may also hold an `options` file setting the history options (Section *The Options File*). It is created if it does not already exist.

If the variable is not set, Magma uses `~/.magma`, creating it if necessary, so a history is kept by default. To keep none, see `MAGMA_NO_HISTORY` above or the `history` option (Section *The Options File*).

MAGMA_HIST_PLAIN

If this variable is set, the session browser (Section *Saved Sessions and the Browser*) draws a plain, uncoloured display for this run, without editing the options file. Set to `dumb`, it also makes the browser emit no terminal escape sequences at all, as the `browser-plain dumb` option does. It can only make the display plainer than the `browser-plain` option asks for, never fancier.

4.4 Set and Get

The `Set-` procedures allow the user to attach values to certain internal variables which control system or global features. The `Get-` functions enable one to obtain the current values of these variables.

`SetAssertions(b): RngIntElt`

`GetAssertions() -> RngIntElt`

Controls the checking of assertions (see the `assert` statement and related statements in the chapter on the language). Default is `SetAssertions(1)`. The relevant values are 0 for no checking at all, 1 for normal checks, 2 for debug checks, and 3 for extremely stringent checking.

`SetAutoColumns(b): BoolElt`

`GetAutoColumns() -> BoolElt`

If enabled, the IO system will try to determine the number of columns in the window by using `ioctl()`; when a window change or a stop/cont occurs, the `Columns` variable (below) will be automatically updated. If disabled, the `Columns` variable will only be changed when explicitly done so by `SetColumns`. Default is `SetAutoColumns(true)`.

`SetAutoCompact(b): BoolElt`

`GetAutoCompact() -> BoolElt`

Control whether automatic compaction is performed. Normally the memory manager of Magma will compact all of its memory between each statement at the top level. This removes fragmentation and reduces excessive memory usage. In some very rare situations, the compactions may become very slow (one symptom is that an inordinate pause occurs between prompts when only a trivial operation or nothing is done). In such cases, turning the automatic compaction off may help (at the cost of possibly more use of memory). Default is `SetAutoCompact(true)`.

`SetBeep(b): BoolElt`

`GetBeep() -> BoolElt`

Controls 'beeps'. Default is `SetBeep(true)`.

`SetColumns(n): RngIntElt`

`GetColumns() -> RngIntElt`

Controls the number of columns used by the IO system. This affects the line editor and the output system. (As explained above, if AutoColumns is on, this variable will be automatically determined.) The number of columns will determine how words are wrapped. If set to 0, word wrap is not performed. The default value is `SetColumns(80)` (unless `SetAutoColumns(true)`).

`GetCurrentDirectory()`

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

`SetEchoInput(b): BoolElt`

`GetEchoInput()`

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

`GetEnvironmentValue(s): MonStgElt -> MonStgElt`

`GetEnv(s): MonStgElt -> MonStgElt`

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

`SetEnvironmentValue(s, v): MonStgElt, MonStgElt`

`SetEnv(s, v): MonStgElt, MonStgElt`

Sets the external environment variable s to have the value v, replacing any value it had before.

`UnsetEnvironmentValue(s): MonStgElt`

`UnsetEnv(s): MonStgElt`

Removes the external environment variable s. It is not an error if s is not set.

`SetExitSummary(b): BoolElt`

`GetExitSummary()`

Controls whether or not Magma should print the exit summary (`"Total time: ..."`) when it exits (`true` by default at startup).

`SetGPU(b): BoolElt`

`GetGPU() -> BoolElt`

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

`SetHistorySize(n): RngIntElt`

`GetHistorySize()`

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

`SetIgnorePrompt(b): BoolElt`

GetIgnorePrompt() -> BoolElt

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

SetIgnoreSpaces(b): BoolElt

GetIgnoreSpaces() -> BoolElt

Controls the option to ignore spaces when searching in the line editor. If the user moves up or down in the line editor using <Ctrl>-P or <Ctrl>-N (see the line editor key descriptions) and if the cursor is not at the beginning of the line, a search is made forwards or backwards, respectively, to the first line which starts with the same string as the string consisting of all the characters before the cursor. While doing the search, spaces are ignored if and only if this option is on (value `true`). Default is `SetIgnoreSpaces(true)`.

SetIndent(n): RngIntElt

GetIndent() -> RngIntElt

Controls the indentation level for formatting output. The default is `SetIndent(4)`.

SetLibraries(s): MonStgElt

GetLibraries() -> MonStgElt

Controls the Magma library directories via environment variable `MAGMA_LIBRARIES`. The procedure `SetLibraries` takes a string, which will be taken as the (colon-separated) list of sub-directories in the library root directory for the libraries; the function `GetLibraryRoot` returns the current value as a string. These directories will be searched when you try to `load` a file; note however that first the directories indicated by the current value of your path environment variable `MAGMA_PATH` will be searched. See `SetLibraryRoot` for the root directory.

SetLibraryRoot(s): MonStgElt

GetLibraryRoot() -> MonStgElt

Controls the root directory for the Magma libraries, via the environment variable `MAGMA_LIBRARY_ROOT`. The procedure `SetLibraryRoot` takes a string, which will be the absolute pathname for the root of the libraries; the function `GetLibraryRoot` returns the current value as a string. See also `SetLibraries`.

SetLineEditor(b): BoolElt

GetLineEditor() -> BoolElt

Controls the line editor. Default is `SetLineEditor(true)`.

SetLogFile(F): MonStgElt

```
Overwrite: BoolElt                    Default: false
```

UnsetLogFile()

Procedure. Set the log file to be the file specified by the string F: all input and output will be sent to this log file as well as to the terminal. If a log file is already in use, it is closed and F is used instead. The parameter `Overwrite` can be used to indicate that the file should be truncated before writing input and output on it; by default the file is appended.

SetMemoryLimit(n): RngIntElt

GetMemoryLimit() -> RngIntElt

Set the limit (in bytes) of the memory which the memory manager will allocate (no limit if 0). Default is `SetMemoryLimit(0)`.

SetNthreads(n): RngIntElt

`GetNthreads() -> RngIntElt`

Set the number of threads to be used in multi-threaded algorithms to be n, if POSIX threads are enabled in this version of Magma. Currently, this affects the coding theory minimum weight algorithm (`MinimumWeight(C: parameters): Code → RngIntElt`) and the F_4 Gröbner basis algorithm for medium-sized primes (`Groebner(I: parameters): RngMPol`).

`SetOutputFile(F): MonStgElt`

`Overwrite: BoolElt` Default: false

`UnsetOutputFile()`

Start/stop redirecting all Magma output to a file (specified by the string F). The parameter `Overwrite` can be used to indicate that the file should be truncated before writing output on it.

`SetPath(s): MonStgElt`

`GetPath() -> MonStgElt`

Controls the path by which the searching of files is done. The path consists of a colon separated list of directories which are searched in order (“.” implicitly assumed at the front). Tilde expansion is done on each directory. (May be overridden by the environment variable MAGMA_PATH.)

`SetPrintLevel(l): MonStgElt`

`GetPrintLevel() -> MonStgElt`

Controls the global printing level, which is one of `"Minimal"`, `"Magma"`, `"Maximal"`, `"Default"`. Default is `SetPrintLevel("Default")`.

`SetPrompt(s): MonStgElt`

`GetPrompt() -> MonStgElt`

Controls the terminal prompt (a string). Expansion of the following % escapes occurs:

: The character %

: The current history line number.

The parser ‘state’
when a new line is about to be read while the parser has only seen incomplete statements, the state consists of a stack of words like “if”, “while”, indicating the incomplete statements.

: Like %S except that only the topmost word is displayed.

Default is `SetPrompt("%S> ")`.

`SetQuitOnError(b): BoolElt`

Set whether Magma should quit on any error to b. If b is `true`, Magma will completely quit when any error (syntax, runtime, etc.) occurs. Default is `SetQuitOnError(false)`.

`SetRows(n): RngIntElt`

`GetRows() -> RngIntElt`

Controls the number of rows in a page used by the IO system. This affects the output system. If set to 0, paging is not performed. Otherwise a prompt is given after the given number of rows for a new page. The default value is `SetRows(0)`.

`GetScriptFilename() -> MonStgElt`

Returns a string giving the filename used to launch Magma in script mode. If Magma was not started in script mode this intrinsic will still succeed, and the returned string will contain text indicating that script mode was not in use.

`GetScriptArguments() -> SeqEnum`

Returns a sequence of strings that were given as arguments to a Magma running in script mode (i.e., with the `-i` or `-I` options). Will return an empty sequence if Magma was not started in script mode.

`SetSeed(s, c): RngIntElt`

`GetSeed() -> RngIntElt, RngIntElt`

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

`GetTempDir() -> MonStgElt`

Returns the directory Magma uses for storing temporary files. May be influenced on startup via the MAGMA_TEMP_DIR environment variable (see Section *Environment Variables*).

`SetTraceback(n): BoolElt`

`GetTraceback() -> BoolElt`

Controls whether Magma should produce a traceback of user function calls before each error message. The default value is `SetTraceback(true)`.

`GetVersion() -> RngIntElt, RngIntElt, RngIntElt`

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

`SetViMode(b): BoolElt`

`GetViMode() -> BoolElt`

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

`SetWarnIntrinsicOverride(b): BoolElt`

`GetWarnIntrinsicOverride() -> BoolElt`

Controls whether warnings are printed when a package intrinsic overrides a previous intrinsic. The default is `false` (no warning).

4.5 Verbose Levels

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

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

`SetVerbose(s, i): MonStgElt, RngIntElt`

`SetVerbose(s, b): MonStgElt, BoolElt`

Set verbose level for `s` to be level i or b. Here the argument `s` must be a string. The verbosity may have different levels. An integer i for the second argument selects the appropriate level. A second argument i of 0 or b of `false` means no verbosity. A boolean value for b of `true` for the second argument selects level 1. (See above for the valid values for the string s).

`GetVerbose(s): MonStgElt -> RngIntElt`

Return the value of verbose flag s as an integer. (See above for the valid values for the string s).

`IsVerbose(s): MonStgElt -> BoolElt`

Return the whether the value of verbose flag s is non-zero. (See above for the valid values for the string s).

`IsVerbose(s, l): MonStgElt, RngIntElt -> BoolElt`

Return the whether the value of verbose flag s is greater than or equal to l. (See above for the valid values for the string s).

`ListVerbose()`

List all verbose flags. That is, print each verbose flag and its maximal level.

`ClearVerbose()`

Clear all verbose flags. That is, set the level for all verbose flags to 0.

4.6 Other Information Procedures

The following procedures print information about the current state of Magma.

ShowIdentifiers()

(Procedure.) List all identifiers that have been assigned to.

ShowValues()

(Procedure.) List all identifiers that have been assigned to with their values.

Traceback()

(Procedure.) Display a traceback of the current Magma function invocations.

ListSignatures(C): Cat

```
Isa    : BoolElt                    Default: true
Search : MonStgElt                  Default: "Both"
ShowSrc: BoolElt                    Default: false
```

List all intrinsic functions, procedures, and operators having objects from category C among their arguments or return values. The parameter `Isa` may be set to `false` so that any categories which C inherit from are not considered. The parameter `Search`, with valid string values `Both`, `Arguments`, `ReturnValues`, may be used to specify whether the arguments, the return values, or both, are considered (default both). `ShowSrc` can be used to see where package intrinsics are defined. Use `ListCategories` for the names of the categories.

ListSignatures(F, C): Intrinsic, Cat

```
Isa    : BoolElt                    Default: true
Search : MonStgElt                  Default: "Both"
ShowSrc: BoolElt                    Default: false
```

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

ListCategories()

ListTypes()

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

4.7 The Magma Line Editor

Magma keeps the command lines of the current session in a history buffer — together with any lines it has remembered from earlier sessions (see Section *The History File*) — and the line editor is how they are recalled and edited.

Magma provides a line editor with both Emacs and VI style key bindings. To enable the VI style of key bindings, type

SetViMode(true): BoolElt

and type

SetViMode(false): BoolElt

to revert to the Emacs style of key bindings. By default ViMode is `false`; that is, the Emacs style is in effect. Many key bindings are the same in both Emacs and VI style. This is because some VI users like to be able to use some Emacs keys (like `<Ctrl>-P`) as well as the VI command keys. Thus key bindings in Emacs which are not used in VI insert mode can be made common to both.

4.7.1 Key Bindings (Emacs and VI mode)

`<Ctrl>-`*key* means hold down the Control key and press *key*.

`<Return>`

Accept the line and print a new line. This works in any mode.

`<Backspace>`

`<Delete>`

Delete the previous character.

`<Tab>`

Complete the word which the cursor is on or just after. If the word doesn't have a unique completion, it is first expanded up to the common prefix of all the possible completions. An immediately following Tab key will list all of the possible completions. Currently completion occurs for system functions and procedures, parameters, reserved words, and user identifiers.

`<Ctrl>-A`

Move to the beginning of the line ("alpha" = "beginning").

`<Ctrl>-B`

Move back a character ("back").

`<Ctrl>-C`

Abort the current line and start a new line.

`<Ctrl>-D`

On an empty line, send a EOF character (i.e., exit at the top level of the command interpreter). If at end of line, list the completions. Otherwise, delete the character under the cursor ("delete").

`<Ctrl>-E`

Move to the end of the line ("end").

`<Ctrl>-F`

Move forward a character ("forward").

`<Ctrl>-H`

Same as Backspace.

`<Ctrl>-I`

Same as Tab.

`<Ctrl>-J`

Same as Return.

`<Ctrl>-K`

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

`<Ctrl>-L`

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

`<Ctrl>-M`

Same as `<Return>`.

`<Ctrl>-N`

Go forward a line in the history buffer ("next"). If the cursor is not at the beginning of the line, go forward to the first following line which starts with the same string (ignoring spaces iff the ignore spaces option is on — see `SetIgnoreSpaces`) as the string consisting of all the characters before the cursor. Also, if `<Ctrl>-N` is typed initially at a new line and the last line entered was actually a recall of a preceding line, then the next line after that is entered into the current buffer. Thus to repeat a sequence of lines (with minor modifications perhaps to each), then one only needs to go back to the first line with `<Ctrl>-P` (see below), press `<Return>`, then successively press `<Ctrl>-N` followed by `<Return>` for each line.

`<Ctrl>-P`

Go back a line in the history buffer ("previous"). If the cursor is not at the beginning of the line, go back to the first preceding line which starts with the same string (ignoring spaces iff the ignore spaces option is on — see `SetIgnoreSpaces`) as the string consisting of all the characters before the cursor. For example, typing at a new line `x:=` and then `<Ctrl>-P` will go back to the last line which assigned `x` (if a line begins with, say, `x :=`, it will also be taken).

`<Ctrl>-U`

Clear the whole of the current line.

`<Ctrl>-V<char>`

Insert the following character literally.

`<Ctrl>-W`

Delete the previous word.

`<Ctrl>-X`

Same as `<Ctrl>-U`.

`<Ctrl>-Y`

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

`<Ctrl>-Z`

Stop Magma.

`<Ctrl>-_`

Undo the last change.

`<Ctrl>-\`

Immediately quit Magma.

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

4.7.2 Key Bindings in Emacs mode only

`<Meta>-`*key* means press the Meta key and then *key*. (At the moment, the Meta key is only the Esc key.)

`<Ctrl>-R`

Search backwards through the history for a line **containing** what the user types, a character at a time — the incremental reverse search of `readline` and `IPython`. The row becomes `(i-search)`pat':` followed by the most recent line that contains *pat*, and it is re-searched as each character is typed. Where `<Ctrl>-P` searches by **prefix**, this searches anywhere in the line.

Another `<Ctrl>-R` moves to the next older line that matches, and `<Ctrl>-N` to the next newer. Once the history buffer is exhausted the search goes on into the saved sessions, **if an archive is being kept** (Section *Saved Sessions and the Browser*); the prompt then reads `(i-search saved)`, and `(failing i-search)` when there is nothing left to find. With no archive the buffer is the whole of it — and it holds what was read from the history file (Section *The History File*), so there is nothing further to reach. `<Backspace>` shortens the pattern and `<Ctrl>-U` clears it.

`<Return>` puts the line that was found into the editor **without** running it, so that it can be edited first; a second `<Return>` runs it, as always. `<Ctrl>-G` or `<Esc>` abandons the search and puts back the line the user was typing — except that when the pattern has failed, the first `<Ctrl>-G` takes it back to the last part of it that did match, so that a typo costs one key rather than a `<Backspace>` for each character of it (as in `emacs` itself). Any other control character ends the search, keeps the line that was found, and then does what it normally does.

In emacs mode the session browser (Section *Saved Sessions and the Browser*) is `<Esc> r` (or `<Esc> R`), or a second `<Ctrl>-R` at an empty search prompt on an empty line. In vi mode `<Ctrl>-R` on an empty line opens the browser instead (or offers to start an archive), and this search is on `/` in command mode. All of these keys can be changed with the `search-key`, `search-key-vi`, `browser-key-emacs`, and `browser-key-vi` options (Section *The Options File*).

<Meta>-b

<Meta>-B

Move back a word ("Back").

<Meta>-f

<Meta>-F

Move forward a word ("Forward").

4.7.3 Key Bindings in VI mode only

In the VI mode, the line editor can also be in two modes: the insert mode and the command mode. When in the insert mode, any non-control character is inserted at the current cursor position. The command mode is then entered by typing the Esc key. In the command mode, various commands are given a *range* giving the extent to which they are performed. The following ranges are available:

0

Move to the beginning of the line.

$

Move to the end of the line.

<Ctrl>-space

Move to the first non-space character of the line.

%

Move to the matching bracket. (Bracket characters are (,), [,], {, }, <, and >.)

;

Repeat the last 'f', 'F', 't', or 'T' forwards.

,

Repeat the last 'f', 'F', 't', or 'T' backwards.

B

Move back a space-separated word ("Back").

b

Move back a word ("back").

E

Move forward to the end of the space-separated word ("End").

e

Move forward to the end of the word ("end").

F<char>

Move back to the first occurrence of **<char>**.

f<char>

Move forward to the first occurrence of **<char>**.

h

H

Move back a character (**<Ctrl>-H** = Backspace).

l

L

Move forward a character (**<Ctrl>-L** on some keyboards).

T<char>

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

t<char>
Move forward to just before the first occurrence of <char>.

w
Move forward a word ("word").

W
Move forward a space-separated word ("Word").

Any range may be preceded by a number to multiply to indicate how many times the operation is done. The VI-mode also provides the *yank-buffer*, which contains characters which are deleted or "yanked" – see below. The following keys are also available in command mode:

A
Move to the end of the line and change to insert mode ("Append").

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

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

c range
Delete all the characters to the specified range and change to insert mode ("change").

D
Delete all the characters to the end of line ("Delete").

d range
Delete all the characters to the specified range ("delete").

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

i
Change to insert mode ("insert").

j
Go forward a line in the history buffer (same as <Ctrl>-N).

k
Go back a line in the history buffer (same as <Ctrl>-P).

/
Start the incremental history search, as <Ctrl>-R does in emacs mode; the keys within the search are the same, except that <Ctrl>-P (rather than another /, which is part of the pattern) moves to the next older line that matches. If search-key-vi is set to a control key instead, that key does so, as <Ctrl>-R does in emacs mode. Unlike the / of readline's vi mode, the search is incremental. The key can be changed with the search-key-vi option (Section *The Options File*).

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

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

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

r<char>
Replace the character the cursor is over with <char>.

S

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

s

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

U

u

Undo the last change.

X

Delete the character to the left of the cursor.

x

Delete the character under the cursor.

Y

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

y range

Copy all characters from the cursor to the specified range into the yank-buffer ("yank").

4.7.4 Completion Preferences

When the user presses the `Tab` key for completion, the line editor offers all intrinsics matching the partial word typed so far. Because many intrinsics have more than one spelling (e.g., the British (`-ise`) and American (`-ize`) variants, or the `-our`/`-or` families), both spellings are normally offered. The following procedures let the user nominate a preferred spelling, so that the non-preferred duplicates are suppressed from the completion list. Usually, the user would simply prefer to use `SetCompletionPreferenceStyle("US");` or `SetCompletionPreferenceStyle("UK");` but the `SetCompletionPreference` procedure allows fine-grained behaviour to be specified.

`SetCompletionPreference(P, O): MonStgElt, MonStgElt`

Record a preference for the spelling given by the string `P` ('preferred') over that given by the string `O` ('other') when the line editor expands tab completions. Thereafter, whenever the completions for a partial word contain two entries that differ only by `P` versus `O` – at the end of the word or at the end of an internal subword (as delimited by a capital letter in a `CamelCase` name), the entry containing `O` is discarded and only the `P` form is offered.

As a special case, the pair `("ize", "ise")` (or its reverse `("ise", "ize")`) also installs the related suffix forms automatically, namely the pairs `izes`/`ises`, `ized`/`ised`, `izing`/`ising`, `izer`/`iser`, `izers`/`isers`, `izable`/`isable`, and `ization`/`isation`.

Several preferences may be in force at once; each call adds a new rule.

`ClearCompletionPreference()`

Remove all completion preferences previously set by `SetCompletionPreference`, restoring the default behaviour in which every registered spelling is offered as a completion.

`SetCompletionPreferenceStyle(S): MonStgElt`

A convenience procedure that calls `SetCompletionPreference` on a standard collection of British/American spelling pairs at once. If `S` is `"US"`, the American (US) spellings are preferred, so that `-ize`, `-or`, `-ors`, `-oring`, and `-er` are kept in preference to `-ise`, `-our`, `-ours`, `-ouring`, and `-re` respectively; if `S` is `"UK"`, the British spellings are preferred instead.

4.8 History Files

4.8.1 The History File

Since V2.30, Magma saves the lines the user types to a **history file**, so lines from earlier sessions can be recalled in later ones. This behaviour is **on by default**. Before V2.30 the history buffer, which the

line editor (Section *The Magma Line Editor*) scrolls through and the '%' commands of Section *History Commands* act on, was lost each time Magma exited. Everything in this section is new in V2.30.

How it works.

- At start-up, the last lines of the file are loaded into the history buffer, ready for the up arrow, the incremental search, and `%h`.
- Each line is appended to the file as soon as it is entered, before it is run. Nothing is left to do at exit, so a killed session loses nothing.
- The file is not created until the first real command is entered.
- The file is a rolling window: it keeps the last 1000 lines the user has typed, whatever sessions they came from, and once it is full the oldest lines drop off to make room. Nothing else is ever removed from it. Use the option `history-lines` (Section *The Options File*, below) to change the default 1000 to another value; 0 means no limit.
- The file is trimmed at start-up, not at exit, and only once it exceeds twice `history-lines`, so it is not rewritten on almost every start. Only one session trims at a time, locking `history.lock` (see below); a line typed in another session at the instant a trim finishes can be lost.
- Recall is also capped by the history buffer size, set by *`SetHistorySize`* (default 2000). If `history-lines` is larger, Magma says so at start-up, and whenever `SetHistorySize` makes the buffer smaller than it. The extra lines stay in the file but are not recalled.
- Sessions running at the same time interleave their lines in the file, in the order they were typed, as shells do. To keep each session's lines separately, see Section *Saved Sessions and the Browser*.

Recalling lines.

- The up arrow and the incremental search reach every line in the history buffer (normally the whole history file), from this session and earlier ones.
- Given no range, the '%' commands act only on the lines entered in this session; a line number reaches any line in the history buffer, and the word `all` takes in the whole buffer, which normally holds the whole history file.

What is not recorded.

- Only interactive sessions, with input from a terminal, use the history file. A run such as `magma < file` records nothing, reads no options file, creates no directory, and ignores `MAGMA_CONFIG_DIR` (Section *Environment Variables*).
- Input supplied to `read` and `readi` is not saved.
- Blank lines and a bare `quit;` are not saved, unless the option `remove-quit` is turned off (Section *The Options File*).
- Only the text of each line is saved, not the random number generator state, so the seed-setting forms `%s` and `%S` (Section *History Commands*) are not meaningful for lines from earlier sessions.

The `~/.magma` directory. Here are details of how the history is stored, for when the user wants to change the defaults (nothing needs to be understood or edited for the default behaviour).

- The first interactive session creates the directory `~/.magma`:

```
~/.magma/
    options            the user's settings
    state/
        history        the history file
        history.lock   used while trimming the file
```

  When Magma creates the directory, the file `options` (Section *The Options File*) starts as a copy of the default file (`libs/auxiliary/options` in the standard Magma distribution), with every line a comment initially, and can be edited by the user. After the initialization, Magma itself writes only under `state`.

- To use another directory, set the environment variable `MAGMA_CONFIG_DIR` (Section *Environment Variables*):

```
setenv MAGMA_CONFIG_DIR ~/.magma-history          # csh/tcsh
export MAGMA_CONFIG_DIR=~/.magma-history          # sh/bash
```

  It takes precedence over `~/.magma`, and is created if it does not exist.

- To keep no history at all, put `history no` in the options file, or set the environment variable `MAGMA_NO_HISTORY` to anything. The variable also stops Magma creating any directory.

- If `~/.magma` exists but is not a directory, Magma keeps no history and says so at the start of every session.

4.8.2 Saved Sessions and the Browser

Magma can also keep each session's lines in a file of its own (in the sub-directory `saved` of `state`), so that whole earlier sessions can be browsed, searched, and re-run with the browser described below. **This archive of earlier sessions is off by default.** To turn it on, do one of the following:

- Simply open the browser on an empty line (press `<Ctrl>-R` in vi mode, `<Esc> r` in emacs mode, or enter `%b` in either mode) and answer `y` when Magma asks `No saved sessions. Keep them from now on? (y/n)`. The lines already typed in the current session are kept as well, and the browser opens on them. Magma asks at most once a session, whatever the answer, and not at all if the options file sets `saved-sessions` explicitly.

- Set `saved-sessions yes` in the options file (Section *The Options File*).

- Create the directory: `mkdir -p ~/.magma/state/saved`.

Turning the archive off later does not delete it; Magma just stops adding to it.

The archive. The archive sits beside the history file under `state`:

```
~/.magma/
    options
    state/
        history
        history.lock
        saved/
            history20140317-101535
            history20140317-142208
            ...
```

- Each session has its own file, named by the date and time the session began, with a numeric suffix if needed to make the name unique.

- Each command line is appended to the session's file as well as to the history file.

- Like the history file, the session's file is not created until the first real command is entered, so a session in which nothing is typed leaves no file behind.

Opening the browser. The browser shows the archive, starting with the current session. It is available only in interactive sessions; with no archive kept, opening it offers to start one (see above).

- In vi mode, `<Ctrl>-R` on an empty line opens the browser, or, with no archive kept, asks whether to start one, opening the browser if the answer is yes. The incremental search is on `/` in command mode instead (see Section *The Magma Line Editor*).

- In emacs mode, `<Ctrl>-R` is the incremental history search (see Section *The Magma Line Editor*), so the browser is `<Esc> r`, or a second `<Ctrl>-R` at an empty search prompt, on an empty line.

- The keys may be changed with the options `browser-key-emacs` and `browser-key-vi` (Section *The Options File*). The `%b` command opens the browser whatever they are set to.

The display. Each session is shown as a short, fixed-size preview: a banner giving the session's date and time, its first few command lines, an ellipsis, and its last few lines.

- Each line is truncated to the screen width, so the display never wraps.
- Three lines are shown above the ellipsis and three below, unless the options `browser-lines-top` and `browser-lines-bottom` (Section *The Options File*) say otherwise. A session short enough to fit is shown whole, without an ellipsis.
- Tabs in the saved commands are expanded to the next tab stop, eight columns apart unless the option `browser-tab-width` says otherwise, so indentation is preserved in the preview.

Keys. The browser recognises the following keys:

```
k, up-arrow      move to an older session
j, down-arrow    move to a newer session
g                jump to the newest session
G                jump to the oldest session
Return, Enter, r re-enter and run the commands of the shown session
Space            load the commands into the history buffer, without running them
0-9, -           a count prefix for the next key (see below)
&                filter to the sessions whose commands match a pattern
/                search for the next matching session, toward older ones
?                search for the next matching session, toward newer ones
n                repeat the last search in the same direction
N                repeat the last search in the other direction
i                toggle case-insensitive matching
E                edit the shown session file in place, then redisplay it
e                edit a copy of the shown session and run the result
s                save the shown session (or a line range) to a named file
c                browse a different directory (empty returns to saved/)
d                delete the shown session file
q, Esc           leave the browser
```

Loading or running an older session's commands adds them to the current history, so they may then be recalled and edited like any others.

Finding a session. There are two ways of finding a session by what it contains, and both take the same kind of pattern. A session matches if any of its command lines does.

- `&` prompts for a pattern and **restricts** the browser to the matching sessions, leaving the others out of the list until an empty pattern clears the filter. The pattern in force is shown in the banner.
- `/` **searches** instead: the list is left as it is, and the browser moves to the next matching session, toward older ones; `?` does the same toward newer ones. `n` repeats the search in the same direction and `N` in the other, as does an empty pattern at the `/` or `?` prompt.
- A count before any of these keys asks for the N-th match rather than the first.
- The pattern is a regular expression, matched against each command line, unless the option `browser-search-literal` (Section *The Options File*) says it is to be taken literally. `<Tab>` at the prompt switches between the two for that pattern, and the prompt shows which is in force: `&re` or `&lit` for the filter, `/re` or `/lit` for a search. A literal pattern is the one to use for text full of brackets, as Magma code is.
- `i` toggles case-insensitive matching for both kinds of pattern; a `[ci]` tag in the banner shows when it is on. A case-insensitive regular expression is matched by folding both the pattern and the line to lower case, so a range such as `[A-Z]` becomes `[a-z]` and then matches any letter.

Counts. Typing digits (optionally preceded by `-`) before a key builds a count N, shown in the banner, that modifies the next key:

- `k`/`j` (and the arrow keys) move by N sessions.

- `g`/`G` jump to the N-th session: 1 is the newest, and a negative count counts from the oldest, so -1 is the oldest.
- `/`, `?`, `n`, and `N` go to the N-th match.
- `r`/`Return`, `Space`, `e`, and `s` act on only the last N commands of the session, or the first $|N|$ if N is negative, instead of all of them.
- `d` deletes N files: the shown one and the next $N-1$ older ones, or newer ones for a negative count.

Editing and deleting.

- `E` opens the shown session file in the user's editor; any changes are saved to the file permanently, and the browser redisplays it. The editor is the option `browser-editor`, else the `EDITOR` environment variable (as for the `%e` command, Section *History Commands*), else `/bin/ed`.
- `e` edits a temporary copy instead. If the user makes no real change, or leaves it blank, the edit is abandoned; otherwise the new contents are entered and run as if the user had accepted them.
- `d` asks for confirmation before deleting a session file, unless the option `browser-confirm-delete` is set to `no` (Section *The Options File*).

Saving and browsing other directories.

- `s` saves the shown session (or, with a count, just its last N or first $|N|$ command lines) to a file the user names, so that a session worth keeping can be moved out of the archive. Naming the shown session itself is refused: the copy would empty the file it was about to read.
- `c` browses such a directory: it prompts for one, and shows the files there in place of the `saved` archive. An empty answer returns to `saved`.
- The files in another directory need not be Magma's own saved sessions, and their names are arbitrary, so they are listed by modification time, newest first, and each is shown in the banner by that time followed by its name (see the option `browser-name-max`, Section *The Options File*). In the `saved` archive only the date is shown, so the name also shows that another directory is being browsed.
- All the other keys behave as usual there, except that `d` always asks for confirmation: those are the user's own files, not Magma's.
- When `s` or `c` prompts for a name on the bottom row, `<Tab>` completes it from the files (for `c`, the directories) that match what the user has typed, filling in as far as the possibilities agree and appending a `/` when the name is a directory. The prompt is also a small line editor with the usual keys, as in the main line editor. These are the left and right arrows, `<Ctrl>-A`/`<Ctrl>-E` to the start/end, `<Ctrl>-B`/`<Ctrl>-F` by a character, `<Ctrl>-D` to delete, `<Ctrl>-K` to the end, `<Ctrl>-W` the previous word, and `<Ctrl>-U` the whole line. `<Esc>` abandons the prompt and returns to the browser unchanged.

Appearance.

- By default the browser draws the session inside a coloured box (banner, contents, and key menu). The colour is blue unless the option `browser-color` (Section *The Options File*) says otherwise.
- The box can be turned off, for a plain, uncoloured display, with the option `browser-plain` (Section *The Options File*), or, for a single session and without editing the options file, by setting the environment variable `MAGMA_HIST_PLAIN` to any value (the variable can only make the display plainer, never fancier). It is also turned off automatically when the screen is very narrow.
- The setting `browser-plain dumb` goes further: the browser then emits no terminal escape sequences at all, neither colour nor the cursor movement it normally repaints itself with. Since it cannot move the cursor back up, each keystroke prints a fresh block below the last one and the screen scrolls, like a printed listing; everything else about the browser is unchanged. This is the setting for a terminal that does not understand the usual sequences, and it is chosen automatically when `TERM` is unset, empty, or `dumb`. Setting `MAGMA_HIST_PLAIN` to `dumb` asks for it for a single session.

Limiting the archive. The `saved` sub-directory grows without bound by default, one file per session for ever.

- The option `saved-max-files` (Section *The Options File*) keeps only the newest N sessions, and the option `saved-max-days` only those from the last N days. Either or both may be given; both default to zero, meaning no limit. **These options delete files, so nothing is discarded unless the user asks for it.**
- The limits are applied at start-up, and the current session's file is created afterwards, so the directory holds one more file than `saved-max-files` says until the next start trims it back. The point is to bound the growth, not to hit an exact count.
- Only files named as Magma itself names them are ever considered, so anything else the user keeps in that directory is left alone.
- The archive is the real record of the user's work; the `history` file is only the last `history-lines` lines of it. The `&` filter and the `/` and `?` searches read every file in the archive, so a very large archive makes them slower, which is the other reason to set a limit.

4.8.3 The Options File

The file `options` sets options for everything described above: the history file, the saved sessions, and the browser.

- It is at the top of the history directory: `~/.magma/options`, unless `MAGMA_CONFIG_DIR` says otherwise.
- **It can be edited by the user.** When Magma makes `~/.magma` it puts the default options file there, with every line a comment showing the built-in value (except `saved-sessions` and `browser-editor`, whose defaults depend on the directory and on `EDITOR`), so that the settings are in front of the user and not only in this section. Uncomment whatever is to be changed.
- After that Magma never writes to it, not even to recreate it: if it is deleted, every option goes back to its built-in default.
- A directory named by `MAGMA_CONFIG_DIR` gets no such file: copy it from `libs/auxiliary/options` in the Magma distribution, or write one from scratch.

Format. Each line names one option and gives its value:

```
browser-lines-top       10
browser-lines-bottom    4
browser-color           green
browser-confirm-delete  no

# blank lines and lines beginning with '#' are ignored
browser-date-format     %Y-%m-%d %H:%M
```

- White space separates the name from the value. The value runs to the end of the line, so it may itself contain spaces, as `browser-date-format` generally will.
- Names are matched without regard to case. If an option is named more than once, the last line wins.
- Naming an option that does not exist, or giving a value that cannot be understood, draws a warning at start-up and otherwise has no effect. A value that is understood but out of range is quietly brought into range.
- A boolean (yes/no below) may be written as `yes`/`no`, `true`/`false`, `on`/`off`, or `1`/`0`.

History file options.

- `history` (yes/no, default `yes`): the off switch. `history no` stops the history being kept at all.
- `history-lines` (N, default 1000): how many lines the global `history` file keeps: the last N the user has typed, whatever sessions they came from, which is what is read into the history buffer when Magma starts. Zero means no limit. Keep it below the size of the history buffer itself

(*SetHistorySize*, 2000 lines by default), or the lines read at start-up fill the buffer before the user has typed anything, and each line the user then types pushes out one that was read.

- `history-print-lines` (N, default 100): how many lines a printing command shows when it is given `all` with no count of its own: `%h`, `%PA`, `%p all`, and the seed forms. Zero means no limit. It exists so that a full history is not emptied onto the screen by accident; the command says when it has shortened its output, and a count, or `0`, still asks for everything. It does not affect `%e` or `%E`.
- `remove-quit` (yes/no, default `yes`): whether a line consisting of just `quit;`, the command that ends the session, is dropped from the saved file. By default it is, since it merely repeats in every session and is no help on recall. Turn it off if the user types code in which a whole physical line is legitimately `quit;` (for instance inside a multi-line string literal) and wants the saved session to reproduce it exactly.

Saved session options.

- `saved-sessions` (yes/no, default `yes` if `state/saved` exists, otherwise `no`): keep the per-session archive (Section *Saved Sessions and the Browser*). Answering `y` in the browser writes nothing to this file: creating `state/saved` is itself what keeps the archive on in later sessions, just as an existing archive goes on being kept.
- `saved-max-files` (N, default 0): how many session files to keep; 0 keeps all.
- `saved-max-days` (N, default 0): how many days of sessions to keep; 0 keeps all.

Browser display options.

- `browser-lines-top` and `browser-lines-bottom` (N or N%, default 3 each): the lines shown above and below the ellipsis.
  - A count is either a plain number of lines or a percentage of the height of the terminal, written `N%`, so `browser-lines-top 25%` asks for a quarter of the screen.
  - A percentage is worked out afresh each time the display is drawn, so it follows the terminal if the window is resized while the browser is open.
  - At least one line is always shown above and below, and if the two counts together would not leave the whole block room on the screen, they are reduced until they do.
- `browser-color`, also spelt `browser-colour` (default `blue`): the box colour. One of `blue`, `cyan`, `green`, `red`, `magenta`, `yellow`, `white`, or `none` (which draws the box without colour).
- `browser-plain` (yes/no or `dumb`, default `no`): a plain, unboxed display; `dumb` makes it free of escape sequences as well.
- `browser-tab-width` (N, 1 to 32, default 8): the tab stop width.
- `browser-date-format` (a format, default `%a %b %e %H:%M:%S %Y`): the format of the date in the banner. It is passed to the C `strftime` function, and if it yields nothing, or is too long to fit, the file name is shown instead. The default omits `%Z` (the time zone), which is the same for every session the user is likely to browse and only takes up banner width; add it back if wanted.
- `browser-name-max` (N, default 32): applies only when browsing a directory other than `saved` (the `c` key), where the file name is shown in the banner as well as the date. It is the greatest number of characters of that name to show, or 0 for all of it (at the risk of pushing the `(n/m)` position off the edge of the box).

Browser key and matching options.

- `browser-key-emacs` (a key, default `esc-r`): opens the browser in emacs mode.
- `browser-key-vi` (a key, default `^R`): opens the browser in vi mode.
- `search-key` (a key, default `^R`): starts the incremental search in emacs mode. It must be a control key: an `esc-x` value is rejected with a warning.
- `search-key-vi` (a key, default `/`): starts the incremental search in vi mode. Besides the forms below, it may be `/` or `?`, which then works only in command mode; in insert mode it is typed as

usual. If it is set to the same key as `browser-key-vi`, a second press at an empty search prompt, on an empty line, opens the browser, as in emacs mode. An `esc-x` value is rejected with a warning.

- The four key options (`browser-key-emacs`, `browser-key-vi`, `search-key`, and `search-key-vi`) name a key in any of these forms, or `none` (or `off`) to leave the action unbound; the letter in a control or meta key is not case-sensitive.
  - `^X`, `C-x`, `ctrl-x`, or `control-x`: a control character.
  - `esc-x`, `M-x`, `meta-x`, or `alt-x`: the two-character sequence `<Esc>` then `x`.
  - A meta key (`esc-x` form) for `browser-key-vi` has no effect, since in vi mode `<Esc>` means "enter command mode" and so cannot begin a binding.
  - Keys that cannot reach Magma at all are rejected with a warning rather than bound to nothing: `^@`, which never arrives, and `^S`, `^Q`, and `^Z`, which the terminal driver takes for flow control and suspension. So are the keys the editor cannot do without: `^H` (`<Backspace>`), `^I` (`<Tab>`), `^J` and `^M` (`<Return>`), and `^[` (`<Esc>`).
  - `^O` and `^T` are accepted but work only on Linux, being the "discard" and "status" characters elsewhere, which is why neither is the default.
- `search-ignore-case` (yes/no, default `no`): whether the incremental search ignores case, in either mode.
- `browser-filter-ignore-case` (yes/no, default `no`): whether matching in the browser starts case-insensitive. It sets only the state the browser starts in; the `i` key still toggles case-insensitivity from there. It applies to regular expressions as well as to literal patterns.
- `browser-search-literal` (yes/no, default `no`): whether the `&`, `/`, and `?` prompts start out expecting a literal string rather than a regular expression. `<Tab>` at the prompt switches between the two whatever it says.
- `browser-confirm-delete` (yes/no, default `yes`): whether `d` asks before deleting.
- `browser-editor` (a command, default unset): the command the browser's `E` and `e` keys run. It may include arguments; the file name is appended to it. If it is not set, the `EDITOR` environment variable is used, and failing that `/bin/ed`, so this is worth setting only if the user wants those keys to use a different editor from `%e` and everything else.

4.9 History Commands

These are the oldest part of the history system, and most of what they once did is now done by the line editor (Section *The Magma Line Editor*) and, since V2.30, by the session browser (Section *Saved Sessions and the Browser*).

`%P` remains the way to print what has been typed in the current session, for pasting or saving.

All the commands here begin with the history character '%' and are typed straight at the prompt, followed by Enter.

Since V2.30 there are several more of them: `%b`, `%c`, `%E`, `%f`, `%h`, `%o`, and `%r`; a range may be given as `all`; and a command given no range acts on the current session (or, for `%e` and `%`, its last line).

Each of the lettered commands may also be written out as a word — the letter is simply an abbreviation of it:

```
%e  %edit      %E  %Edit       %p  %print     %P  %Print
%s  %seed      %S  %Seed       %r  %rerun     %b  %browser
%c  %commands  %f  %find       %o  %options   %h  %history
```

`%c` prints the list itself, so it need not be remembered.

Case means the same thing in the word as in the letter, so `%Edit` is to `%edit` what `%E` is to `%e`, and `%Rerun` is no more a command than `%R` is. A line number may follow either form, with or without a

space (`%p3` and `%print 3` are the same). Anything else after the `%` is an error — in earlier versions text following a command letter was ignored, so that `%rubbish` quietly reran the whole history.

There are two things a command can act on, and the difference is worth having clear: the lines entered in **this session**, which is what it acts on if given nothing, and **everything the history remembers**, which the user asks for with the word `all`.

This session. With no argument, `%p`, `%P`, `%s`, `%S`, `%E`, and `%r` act on the lines entered in this session. A number here is a **line number**, as it always has been, and the numbers are the ones `%p` prints:

```
%p            this session's lines, each preceded by its line number
%P            the same without the numbers
%p 12         line 12
%p 12 20      lines 12 to 20
%r            re-enter and rerun this session's lines
```

A number reaches any line the buffer holds, though, not only this session's: the lines read from the history file (Section *The History File*) when Magma starts are numbered too, and they come first — which is why the first line typed in a session may be number 1001.

Everything the history remembers. This is the whole history buffer, which normally holds the whole history file (Section *The History File*): by default the last thousand lines the user has typed, this session's and earlier sessions' alike. Add the word `all` and the command takes them all in. It may be abbreviated to `a`, with or without a space, and may run on to the command letter, as in `%pa`. A number after `all` is a **count**: the last N lines, not a line number:

```
%p all        every remembered line, numbered
%pa   %p a    the same -- 'all' abbreviates to 'a', with or without a space
%PA           every remembered line, unnumbered -- the one to paste
%p all 20     the last 20 remembered lines, numbered
%PA 20        the last 20, unnumbered
%h    %h 20   %h is exactly %P all, and takes the same count
%h 0          all of them, however many
```

A count is useful when there may be a thousand lines or more in the full history (a count larger than the history gives all of it). `all` may be written in any case; the case of the command letter still matters (`%PA` is `%P all`).

Given `all` and no count, a printing command stops at the last 100 lines and says so (to avoid dumping a long history onto the screen by default). A count asks for a different number and `0` asks for all of them; the option `history-print-lines` (Section *The Options File*) changes the 100, and setting it to zero turns the limit off. It applies only to the commands that **print**: `%e all` and `%E all` are never shortened.

`%p`

List the lines entered in this session. Each line is preceded by its history line number.

`%p all`

List every line the history buffer holds, including any recalled from earlier sessions, in `%p` format. `all` may be abbreviated to `a`.

`%p all N`

List only the **last** N of them.

`%pn`

List the history line n in %p format.

`%pn`$_1$ `n`$_2$

List the history lines in the range n_1 to n_2 in %p format.

`%P`

List the lines entered in this session. The initial numbers are *not* printed.

%P all

List every line the history buffer holds, in %P format. `all` may be abbreviated to `a`, so that `%PA` is the shortest way to see everything the history remembers. This is what `%h` does.

%Pn

List the history line n in %P format.

%Pn$_1$ n$_2$

List the history lines in the range n_1 to n_2 in %P format.

%h

List every line the history buffer holds — those entered in this session and any recalled from earlier ones — without the line numbers. It is exactly `%P all`. It takes no line number of its own, but it does take the same count that `all` does: `%h 20` is the last twenty remembered lines and `%h 0` is all of them. With no count it stops at `history-print-lines`, 100 by default, and says that it has. (In a *prompt string* '%h' is a different thing altogether and is unchanged: there it expands to the number of the line being typed.)

%s

List the lines entered in this session with an initial statement for each line to reset the random number seed to the value it was just before the line was executed. This is useful when one wishes to redo a computation using exactly the same seed as before but does not know what the seed was at the time.

%sn

Print the history line n in %s format.

%sn$_1$ n$_2$

Print the history lines in the range n_1 to n_2 in %s format. `all` may be given instead of a range, and takes in every line the buffer holds — though the seeds of lines recalled from an earlier session are not meaningful, as noted in Section *The History File*.

%S

As for %s except that the statement to set the seed is only printed if the seed has changed since the previous time it was printed. Also, it is not printed if it would appear in the middle of a statement (i.e., the last line did not end in a semicolon).

%Sn

Print the history line n in %S format.

%Sn$_1$ n$_2$

Print the history lines in the range n_1 to n_2 in %S format.

%

Reenter the last line into the input stream.

%n

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

%n$_1$ n$_2$

Reenter the history lines in the range n_1 to n_2 into the input stream.

%e

Edit the last line. The editor is taken to be the value of the `EDITOR` environment variable if it is set, otherwise `/bin/ed` is used. If after the editor has exited the file has not been changed then nothing is done. Otherwise the contents of the new file are reentered into the input stream.

%en

Edit the line specified by line number n.

%en$_1$ n$_2$

Edit the history lines in the range n_1 to n_2. `all` may be given instead of a range, and edits every line the buffer holds.

%E

Edit this session's lines; this is like %e but with no line number given it edits all the lines entered in this session rather than just the last one. A line number or range may also be supplied, exactly as for %e, or `all` to edit every line the buffer holds.

%! shell-command

Execute the given command in the Unix shell then return to Magma.

%r

Re-enter and rerun the lines entered in this session: they are all run again, in order, as if they had been typed afresh. A line number or range may be given instead, exactly as for `%p`.

%r is the one ranged command with no `all` form. It is also the only one that runs what it selects with nothing in between, and rerunning every remembered line is too drastic for a short form; give an explicit range instead.

%f pattern

List the history lines that contain *pattern*, each with its line number, so that they can then be re-entered with `%n`. The number is printed as a Magma comment, in the form `/* n */`, so that the listing can also be pasted straight back into a session. The pattern is a regular expression unless option `browser-search-literal` is set, and is matched case-insensitively if option `browser-filter-ignore-case` is (Section *The Options File*) — the same two options the browser's own searching obeys.

%o

Show the history options: each one, the value in force, the default it replaced if it has replaced one, and whether it came from the options file. The heading names the file that was read, or else says why none was — there may be no history directory, or no `options` file in the one there is, or the history may be turned off altogether. Use it to find out why an option seems to have had no effect.

%c

Print a list of the history commands, a line for each giving its letter, its word, and what it does, followed by the forms that have no letter (`%n`, `%`, and `%!`).

%b

Open the browser of saved sessions (Section *Saved Sessions and the Browser*). It is usually opened with a key — `<Esc> r` in emacs mode, `<Ctrl>-R` in vi mode — but those can be rebound or unbound with the `browser-key-emacs` and `browser-key-vi` options (Section *The Options File*), whereas this command always works and is the same in both editing modes.

4.10 The Magma Help System

Magma provides extensive online help facilities that can be accessed in different ways.

The easiest way to get some information about any Magma intrinsic is by typing: (Here we assume you to be interested in `FundamentalUnit`)

```
> FundamentalUnit;
```

Which now will list all signatures for this intrinsic (i.e. all known ways to use this function):

```
> FundamentalUnit;
Intrinsic 'FundamentalUnit'

Signatures:
```

(continues on next page)

(continued from previous page)

```
    (<FldQuad> K) -> FldQuadElt
    (<RngQuad> O) -> RngQuadElt

        The fundamental unit of K or O

    (<RngQuad> R) -> RngQuadElt

        Fundamental unit of the real quadratic order.
```

Next, to get more detailed information, try

```
> ?FundamentalUnit
```

But now several things could happen depending on the installation. Using the default, you get

```
============================================================
PATH: /magma/ring-field-algebra/quadratic/operation/\\
      class-group/FundamentalUnit
KIND: Intrinsic
============================================================
FundamentalUnit(K) : FldQuad -> FldQuadElt
FundamentalUnit(O) : RngQuad -> RngQuadElt

    A generator for the unit group of the order O or the
maximal order
    of the quadratic field K.
============================================================
```

Second, a WWW-browser could start on the part of the online help describing your function (or at least the index of the first character). Third, some arbitrary program could be called to provide you with the information.

If `SetVerbose("Help", true);` is set, Magma will show the exact command used and the return value obtained.

`SetHelpExternalBrowser(S, T): MonStgElt, MonStgElt`

`SetHelpExternalBrowser(S): MonStgElt`

Defines the external browser to be used if `SetHelpUseExternalBrowser(true)` is in effect. The string has to be a valid command taking exactly one argument (`%s`) which will we replaced by a URL. In case two strings are provided, the second defines a fall-back system. Typical use for this is to first try to use an already running browser and if this fails, start a new one.

`SetHelpUseExternalBrowser(b): BoolElt`

Tells Magma to actually use (or stop to use) the external browser. If both `SetHelpUseExternalSystem` and `SetHelpUseExternalBrowser` are set to `true`, the assignment made last will be effective.

`SetHelpExternalSystem(s): MonStgElt`

This will tell Magma to use a user defined external program to access the help. The string has to contain exactly one `%s` which will be replaced by the argument to `?`. The resulting string must be a valid command.

`SetHelpUseExternalSystem(b): BoolElt`

Tells Magma to actually use (or stop to use) the external help system. If both `SetHelpUseExternalSystem` and `SetHelpUseExternalBrowser` are set to `true`, the assignment made last will be effective.

GetHelpExternalBrowser() -> MonStgElt, MonStgElt

Returns the currently used command strings.

GetHelpExternalSystem() -> MonStgElt

Returns the currently used command string.

GetHelpUseExternal() -> BoolElt, BoolElt

The first value is the currently used value from **SetHelpUseExternalBrowser**, the second reflects SetHelpUseExternalSystem.

4.10.1 Internal Help Browser

Magma has a very powerful internal help-browser that can be entered with

```
> ??
```

Chapter 5

Parallelism

5.1 Introduction

This chapter provides basic instructions for using the parallelism facilities provided in the current release of Magma. This version of Magma provides some general tools for implementing parallelism as well as the explicit parallelisation of some key algorithms in the areas of linear algebra, commutative algebra, integral lattices, and linear codes.

5.1.1 Approaches to Parallelism

The fact that Moore's law no longer applies to the development of ever faster computers has greatly increased interest in exploiting multiple processors for computations that use large amounts of CPU time. In 2019 a project was commenced to develop tools for parallelising some classes of computations in Magma. Two approaches are being taken:-

(i) The parallelisation of key algorithms in the Magma kernel. This utilises multiple cores on a single compute node using POSIX threads, multiple machines by distributing tasks across a nework, and GPUs.

(ii) A framework for the manager/worker model of parallelism has been constructed which allows users to parallelise their Magma code when the underlying algorithm lends itself to being parallelised.

A key element of this project has been the development of tools that support asynchronous communication between copies of Magma running on different machines. As part of this work, the serialisation of Magma objects is being undertaken. This machinery is described in Chapter *Input and Output* on Input and Output.

In Magma, when parallelism is selected, the execution is parallelised for the following intrinsics, or for intrinsics which are called (directly or indirectly) in the listed algorithms:

(1) Multiplication of matrices over a finite field $\mathrm{GF}(p^k)$, where p is a prime such that $2 \leq p \leq 2^{23.5}$. As a consequence, other intrinsics such as `EchelonForm`, inverse (A^{-1}), `Rank`, `Determinant`, `Nullspace`, `RowSpace`, `Solution`, `IsConsistent`, and `MinimalPolynomial` (if the Keller-Gehrig algorithm is specified) are also parallelised.

(2) Construction of a Gröbner basis for a system of polynomials defined over a finite field $\mathrm{GF}(p^k)$, where either $k = 1$ and p is a prime such that $2 \leq p < 2^{63}$, or $2 < p^k \leq 2^{20}$. Includes optimised asynchronous version for the dense variant of the Faugere F_4 over $\mathrm{GF}(2)$.

(3) Construction of the affine variety (by calling `Variety(I)`) of an ideal of a multivariate polynomial ring defined over a small finite field K of size q (by exhaustive evaluation of k variables at a time at all q elements of K, thus yielding many smaller systems to solve in parallel). Point counting variant `VarietySize(I)` gives the size only without constructing all the solutions.

(4) Multiplication of high-degree univariate polynomials defined over prime finite fields or integer residue rings with large integer coefficients; multiplication of high-degree univariate polynomials defined over finite fields of characteristic 2 (Karatsuba and Cantor algorithms).

(5) Bivariate resultant of two univariate polynomials (this also speeds up factorization of polynomials in $\mathbb{Z}[x]$).

(6) Factorisation of large integers (via ECM or MPQS algorithms).

(7) LLL reduction of an integral lattice or basis matrix.

(8) Enumeration of vectors in an integral lattice having small norm (directly in intrinsics `ShortVectors` and `ShortestVectors`). As a consequence the lattice intrinsics `Minimum`, `KissingNumber`, `PackingRadius`, `HermiteNumber`, `CentreDensity`, `Density`, and `ThetaSeries` are also effectively parallelised amongst others.

(9) Enumeration of vectors in an integral lattice that are close to a given vector which need not be in the lattice (`CloseVectors`, etc.).

(10) Minimum weight of a linear error-correcting code defined over a finite field. As well as `MinimumWeight`, the related intrinsics `MinimumDistance` and `MinimumWord` are also parallelised.

(11) Weight distribution of a linear error-correcting code defined over a finite field. As well as `WeightDistribution` the related intrinsics `WeightEnumerator` and `DualWeightDistribution` are also parallelised.

(12) Class group of a number field or an order (intrinsic `ClassGroup`).

5.2 Magma Kernel Distributed Parallelism

5.2.1 Introduction

In this section we discuss the way in which POSIX threads and distributed computation are used.

5.2.2 POSIX Threads

Recall that a *process* is a program that has been read into memory together with the various resources that it needs in order to be executed. This includes such things as registers, a stack, and a program counter. A *thread* is an independent stream of instructions taken from the compiled program which can be run by the operating system.

In the context of achieving efficient parallelism, threads have some advantages over processes. For example, all threads belonging to a particular process share the memory of their parent process and so can communicate with each other much faster than is the case with processes, each of which has its own private memory. Also context switching between threads is less expensive that is the case with processes.

For generating threads on machines running the UNIX operating system a standard programming interface has been defined by IEEE. Threads produced using this standard are known as *POSIX threads* or *Pthreads*.

When exploiting multiple cores of a compute node, Pthreads is generally more efficient than running the same number of processes. Consequently, we have used Pthreads to parallelise the enumeration of certain types of vectors in a lattice and the enumeration of words in a linear code. This parallelisation is restricted to the cores of a single compute node. To exploit multiple compute nodes distributed computation is used. Pthreads are used independently on each of the nodes while the tasks and inter-node communication is handled by the distributed computation machinery.

`SetNthreads(n): RngIntElt`

Set the number of threads to be used in multi-threaded algorithms to be n, if POSIX threads are enabled in this version of Magma.

`GetNthreads() -> RngIntElt`

Return the number of POSIX threads currently being used in multi-threaded algorithms in Magma.

`magma -t N`

If the `-t` option is supplied to Magma on the command line, together with a positive number N, then the number of POSIX threads is set to N at startup; this is equivalent to calling `SetNthreads(N);` after Magma has started.

5.3 GPUs

Support for NVIDIA Tesla Graphical Processing Units (GPUs) is provided when the CUDA library is installed on the system and the Magma CUDA executable `magma.cuda.exe` is used (suitably renamed for the driving `magma` command line script if need be).

The GPU mode is automatically selected by default in the Magma CUDA executable when running on a computer with CUDA support; the procedure call `SetGPU(false);` disables the CUDA matrix algorithms in this executable, while `SetGPU(true);` enables them again. (For the standard non-CUDA executable, `SetGPU` is ignored since the GPU mode is irrelevant.)

Multiple GPU-based linear algebra is supported since V2.26.

`SetGPU(b): BoolElt`

`GetGPU() -> BoolElt`

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

`SetNGPUs(n): RngIntElt`

Set the number of GPUs to be used to n (uses devices 0 .. $n-1$ by default).

`GetNGPUs() -> RngIntElt`

Return the number of GPUs now used.

`SetGPUDevices(S): SeqEnum`

Set the specific used GPU devices to be the numbers in sequence S (while setting the number of devices to $\#S$); the entries of S must be distinct and ≥ 0.

`SetGPUDevice(k): RngIntElt`

Set the single specific GPU device to be used to be k; this is equivalent to calling `SetGPUDevices([k])` .

Being able to set specific devices provides finer-grained management when a machine has more than one GPU. For example, suppose that the machine has two GPUs (device numbers 0 and 1). One could just use `SetNGPUs(2)` for one job to use two GPUs, but one might prefer to run two separate jobs each using a single GPU. In that case the first job could specify `SetGPUDevice(0)` while the second job would specify `SetGPUDevice(1)`.

NB: The utility `nvtop` (downloadable from `https://github.com/Syllo/nvtop`) is very useful for seeing the GPU usage during a run. It continuously shows how much each GPU is being used both in time and memory usage.

5.4 Linear Algebra

Linear algebra can be parallelised within Magma for operations with large matrices over finite fields of small characteristic. These are two main parallelisation modes available: multi-threaded and GPU. A combination of these two modes (called 'hybrid') is also available but is still in an experimental phase.

Once parallelisation in one of the below modes is selected, the most critical base algorithm which is parallelised is matrix multiplication for sufficiently large matrices over the above fields (directly called by `A*B`). The most important other matrix algorithms which directly benefit from this are powering (`A^e`) and the computation of the echelon form (`EchelonForm(A)`) and the many operations which are closely derived from that, including inverse (`A^-1`), and the functions `Rank`, `Determinant`, `Nullspace`, `Rowspace`, `Solution`, `IsConsistent`, and `MinimalPolynomial(A: Al := "KellerGehrig")` (which selects the asymptotically-fast Keller-Gehrig algorithm for minimal polynomial computation).

Any other algorithm of Magma which depends on large matrix computations over small finite fields will also be automatically parallelised when any of the above modes of parallelisation is selected and any of

the above functions are implicitly called. For example, most of the important functions to compute with KG-modules, where K is a small finite field and G is a finite group, depend heavily on multiplication and echelonisation of matrices, and so automatically benefit from linear algebra parallelisation in large examples; in particular the Meataxe (called by `CompositionFactors`, etc.) and the computation of sub- and quotient modules are greatly sped up in large examples when linear algebra parallelisation is enabled.

5.4.1 Multi-threaded Linear Algebra

Multi-threaded linear algebra is supported in these cases:

(i) For matrices over $\mathbf{F}_p$, where $p = 2, 3, 5, 7$ (POSIX threads with split-word representation).

(ii) For matrices over $\mathbf{F}_p$, where p is prime and $10 < p < 2^{23.5}$ (POSIX threads with exact floating point doubles and parallel OpenBLAS library or Intel Math Kernel Library).

(iii) For matrices over $\mathbf{F}_{p^k}$, where $p < 2^{23.5}$ and $k > 1$ (parallel Karatsuba combined with one of the above, depending on the size of p).

This POSIX threads mode will be selected for the above types of finite field when the global number of threads is first set to a number greater than 1 before calling the functions below; as usual, one simply first calls the procedure:

```
SetNthreads(NT);
```

where `NT` is the number of threads desired. Note that since matrix operations are memory intensive, the resulting speedups compared with the serial algorithms depend strongly on the memory speed of the computer, and it is unlikely that the speedup obtained will be close to `NT`. Also, hyperthreading (setting `NT` to more than the number of physical cores) is usually not beneficial because it increases the contention between threads on memory access.

5.4.2 GPU-based Linear Algebra

GPU-based linear algebra (with NVIDIA CUDA) is supported in these cases:

(i) For matrices over $\mathbf{F}_p$, where $p = 2, 3, 5, 7$ (CUDA code developed within Magma).

(ii) For matrices over $\mathbf{F}_p$, where p is prime and $10 < p < 2^{23.5}$ (CUBLAS library or Intel Math Kernel Library).

(iii) For matrices over $\mathbf{F}_{p^k}$, where $p < 2^{23.5}$ and $k > 1$ (parallel Karatsuba combined with one of the above, depending on the size of p).

Since V2.26, multiple GPUs may also be selected for matrix multiplication over GF(p), where $p = 2, 3, 5, 7$. This can give a significant speedup for very large dimensions. See Section *GPUs* for information on how to select multiple GPUs.

5.4.3 Hybrid Linear Algebra

This combines multi-threaded and GPU operations at the same time (for matrices defined over the same types of finite fields as above). This mode is selected by calling `SetNthreads(NT);` while running the CUDA executable (and with the GPU mode enabled). This mode may be not be beneficial compared to the simple POSIX threads mode or the simple GPU mode (it depends strongly on the number of threads and the type of GPU(s)), so when running the CUDA executable by default, it may be better to keep the number of threads at 1.

Example: Threaded Matrix Multiplication `(ex-e8f312)`

The following input demonstrates simple use of POSIX threads when multiplying matrices (on an Intel 4-core i7-7700 CPU @ 3.60GHz):

```
> SetNthreads(1);
> X := Random(MatrixRing(GF(5), 10000));
> time P := X*X; // single thread, normal CPU time
```

(continues on next page)

(continued from previous page)

```
Time: 5.040
> time E,T := EchelonForm(X);
Time: 6.940
> SetNthreads(4);
> time P2 := X*X; // multi-threads, so real time shown by [r]
Time: 1.760[r]
> time E,T := EchelonForm(X);
Time: 2.660[r]
> assert P2 eq P;
```

Example: Matrix Multiplication on a GPU (ex-f549f7)

The following input demonstrates simple use of one GPU when multiplying matrices (NVIDIA V100 GPU; available with CUDA executable only):

```
> SetGPU(false);
> X := Random(MatrixRing(GF(5), 10000));
> time P := X*X; // no GPU
Time: 5.040
> time E,T := EchelonForm(X);
Time: 6.940
> SetGPU(true);
> time P2 := X*X; // 1 GPU
Time: 1.320
> time E,T := EchelonForm(X);
Time: 2.920
> assert P2 eq P;
```

Example: Matrix Multiplication on Two GPUs (ex-9775af)

The following input tests very large matrix multiplication over GF(2) to see speedups when two GPUs are available (CUDA executable only with GPU selected by default; hardware used here: dual NVIDIA V100 GPUs).

```
> K := GF(2);
> for i in [16 .. 19] do
>    n := 2^i;
>    printf "Dim: 2^%o = %o\n", i, n;
>    A := Random(MatrixRing(K, n));
>    B := Random(MatrixRing(K, n));
>    IndentPush();
>    "1 GPU:";
>    t0 := Cputime(); SetNGPUs(1); time P1 := A*B; t0 := Cputime(t0);
>    "2 GPUs:";
>    t1 := Cputime(); SetNGPUs(2); time P2 := A*B; t1 := Cputime(t1);
>    printf "Speedup: %.3o\n", t0/t1;
>    assert P1 eq P2;
>    IndentPop();
> end for;
Dim: 2^16 = 65536
    1 GPU:
    Time: 2.680
    2 GPUs:
    Time: 1.550
    Speedup: 1.729
Dim: 2^17 = 131072
    1 GPU:
    Time: 18.420
```

(continues on next page)

(continued from previous page)

```
    2 GPUs:
    Time: 11.170
    Speedup: 1.649
Dim: 2^18 = 262144
    1 GPU:
    Time: 134.380
    2 GPUs:
    Time: 75.840
    Speedup: 1.772
Dim: 2^19 = 524288
    1 GPU:
    Time: 1085.930
    2 GPUs:
    Time: 571.670
    Speedup: 1.900
```

Example: Modular Representations of a Simple Group `(ex-c94a83)`

In this example we compute absolutely irreducible modular representations of a finite simple group over the finite field $\mathbf{F}_{17}$. The server used has dual Intel 3.2GHz Gold 6146 CPUs with 24 cores total, and an NVIDIA V100 GPU, showing speedups when multiple cores or a GPU is used.

```
> G := PermutationGroup(ATLASGroup("O8m2"));
> K := GF(17);
> L := AbsolutelyIrreducibleModules(G, K);
> [Dimension(M): M in L];
[ 1, 34, 51, 83, 204, 204, 357, 476, 476, 595, 714, 714, 1020,
1071, 1071, 1071, 1071, 1190, 1261, 1428, 2142, 2142, 2142, 2142,
2176, 2295, 2295, 2295, 2835, 2856, 2856, 3264, 4284, 4760, 5355 ]
```

With one thread only:

```
> SetNthreads(1); SetGPU(false);
> time L := AbsolutelyIrreducibleModules(G, K);
Time: 6503.600
```

With 24 threads:

```
> SetNthreads(24); SetGPU(false);
> time L := AbsolutelyIrreducibleModules(G, K);
Time: 2432.150[r]
```

With one thread and a GPU:

```
> SetNthreads(1); SetGPU(true);
> time L := AbsolutelyIrreducibleModules(G, K);
Time: 2584.590
```

5.5 Gröbner Bases

The computation of Gröbner bases of multivariate polynomial ideals by the Faugere F_4 algorithm can be parallelised when the base ring is a finite field. These are the versions of the algorithm for which parallelisation is currently supported:

(i) *Standard (non-dense) F_4*: For ideals defined over $\mathbf{F}_p$ with p prime and $2 < p < 2^{23.5}$ (POSIX threads for the specialist mixed sparse/dense linear algebra).

(ii) *General Dense F_4*: For ideals with dense input polynomials defined over $\mathbf{F}_{p^k}$ with $p < 2^{23.5}$ and $k \geq 1$ (POSIX threads and GPUs, mostly mapping to the parallel linear algebra (above).

(iii) *Special HFE Dense F_4*: For ideals with dense input polynomials defined over $\mathbf{F}_2$, in the 'HFE' mode (where the hash table is not used in the symbolic reduction); the whole reduction phase is completely parallelised with POSIX threads, which is faster than the distinct calls to matrix multiplication in the general dense algorithm and significant memory saving is also achieved (POSIX threads and GPUs) [new in V2.26]).

For each of the above cases, parallelisation with POSIX threads is simply selected by calling the following procedure (before calling the functions below):

```
SetNthreads(NT);
```

where `NT` is the number of threads desired. Note that (as for linear algebra) the speedup will not necessarily be close to the number of threads, and hyperthreading may not be beneficial.

For the dense versions (second and third cases above), the GPU mode is automatically selected by default in the CUDA executable when running on a computer with CUDA support; this can be disabled by first calling `SetGPU(false);`. A hybrid mode (POSIX threads and GPU combined) can be selected by also setting the number of threads while running the CUDA executable; this is more likely to be beneficial in the Special HFE Dense version.

Once parallelisation is selected by one of the above methods, the parallel F_4 algorithm can be called by the procedure `Groebner` or the function `GroebnerBasis` and the other functions which take a multivariate polynomial ideal defined over a finite field and implicitly create a Gröbner basis first (such as `Dimension` and `Variety`).

The General Dense algorithm can be selected by passing the parameter `Dense` to the `Groebner` or `GroebnerBasis` functions, but this will also be automatically selected if the input polynomials are considered dense, while passing `Dense := false` will force the Standard (non-dense) algorithm to be used.

The Special HFE dense algorithm is selected by passing the parameters `Dense` and `HFE` to the `Groebner` or `GroebnerBasis` functions. (As before, `Dense` may be omitted if the input is already dense.)

Example: Solving MinRank Problems `(ex-cdcd67)`

In this example we solve (n, k, r)-MinRank problems by first using the General Dense F_4 algorithm to compute a `grevlex` Gröbner basis of the relevant ideal (based on the minors of a symbolic matrix with linear forms) and then using a block Wiedemann algorithm to compute the variety of the ideal (see [Ste15] for more details). After choosing a parameter r, a hard instance can be found with by taking $n = r + 3$ and $k = (n - r)^2$. The runs are on a server with dual Intel 3.2GHz Gold 6146 CPUs with 24 cores total, and an NVIDIA V100 GPU.

```
> K := GF(PreviousPrime(2^16));
> r := 6; n := r + 3; k := (n - r)^2;
> n, k, r;
9 9 6
> time B := MinRankSystem(K, n, k, r);
Time: 160.030
> I := Ideal(B);
> I: Minimal;
Ideal of Polynomial ring of rank 9 over GF(65521)

> // Single thread
> SetNthreads(1);
> // DynamicStrategy handles large reduction to zero better:
> time Groebner(I: PairsLimit := 2000, Dense, DynamicStrategy);
Time: 2511.040
> QuotientDimension(I);
41580
```

(continues on next page)

(continued from previous page)

```
> time V := Variety(I: Al := "Wiedemann");
Time: 2449.620
> V;
[ <40578, 37019, 22371, 45628, 40625, 2374, 42277, 33586, 51176> ]
> // Check solution is correct:
> time { Evaluate(f, V[1]): f in B };
{ 0 }
Time: 1.320

> // 24 threads
> SetNthreads(24);
> I := Ideal(B); // Recreate ideal
> time Groebner(I: PairsLimit := 2000, Dense, DynamicStrategy);
Time: 548.030[r]
> time V := Variety(I: Al := "Wiedemann");
Time: 539.280[r]
> V;
[ <40578, 37019, 22371, 45628, 40625, 2374, 42277, 33586, 51176> ]

> // GPU
> SetNthreads(1); SetGPU(true);
> I := Ideal(B); // Recreate ideal
> time Groebner(I: PairsLimit := 2000, Dense, DynamicStrategy);
Time: 477.280
> time V := Variety(I: Al := "Wiedemann");
Time: 520.450
> V;
[ <40578, 37019, 22371, 45628, 40625, 2374, 42277, 33586, 51176> ]
```

Taking $r = 7$ gives a much bigger system where the parallel speedup is greater, especially for the Wiedemann stage which completely depends on several large matrix multiplications.

```
> K := GF(PreviousPrime(2^16));
> r := 7; n := r + 3; k := (n - r)^2;
> n, k, r;
10 9 7
> time B := MinRankSystem(K, n, k, r);
Time: 305.080
> I := Ideal(B);
> I: Minimal;
Ideal of Polynomial ring of rank 9 over GF(65521)
> #B;
2025
> {* Length(f): f in B *};
{* 22283^^5, 22284^^83, 22285^^490, 22286^^1447 *}
```

With one thread only:

```
> SetNthreads(1);
> time Groebner(I: PairsLimit := 5000, Dense, DynamicStrategy);
Time: 16108.420
> QuotientDimension(I);
108900
> time V := Variety(I: Al := "Wiedemann");
Time: 35348.420
> V;
```

(continues on next page)

(continued from previous page)

```
[ <38565, 61856, 56127, 27828, 31304, 30257, 38178, 28576, 24230> ]
> // Check solution is correct:
> time { Evaluate(f, V[1]): f in B };
{ 0 }
Time: 4.220
```

With 24 threads:

```
> SetNthreads(24);
> I := Ideal(B); // Recreate ideal
> time Groebner(I: PairsLimit := 5000, Dense, DynamicStrategy);
Time: 2582.500[r]
> time V := Variety(I: Al := "Wiedemann");
Time: 5317.040[r]
> V;
[ <38565, 61856, 56127, 27828, 31304, 30257, 38178, 28576, 24230> ]
```

With one thread and a GPU:

```
> SetNthreads(1); SetGPU(true);
> I := Ideal(B); // Recreate ideal
> time Groebner(I: PairsLimit := 5000, Dense, DynamicStrategy);
Time: 2127.830
> time V := Variety(I: Al := "Wiedemann");
Time: 4316.960
> V;
[ <38565, 61856, 56127, 27828, 31304, 30257, 38178, 28576, 24230> ]
```

Example: Random Dense Quadratic Systems `(ex-eea1ea)`

In this example we solve some over-determined n-variable polynomial systems generated by m random dense quadratic polynomials, where the constant terms are adjusted so there is at least one solution over the original field. The hardest system in Table 1 of [MS17] has $n = 27, m = 2 \cdot n = 54$.

```
> function RandomDenseSystem(K, n, m, d)
>     P<[x]> := PolynomialRing(K, n, "grevlex");
>     B := [
>         &+[Random(K)*s: s in MonomialsOfDegree(P, k), k in [0 .. d]]:
>             i in [1 .. m]
>     ];
>     sol := [Random(K): j in [1 .. n]];
>     return [f - Evaluate(f, sol): f in B];
> end function;
> n := 27; B := RandomDenseSystem(GF(31), n, 2*n, 2);

> // Single thread only
> time G := GroebnerBasis(B, 6: Dense, PairsLimit := 5000);
Time: 2799.290
> G;
[
    x[1] + 12, x[2] + 19, x[3] + 8, x[4] + 10, x[5] + 25, ...  x[27] + 21
]
> Variety(Ideal(G));
[ <19, 12, 23, 21, 6, 27, 19, 14, 18, 17, 10, 14, 29, 25, 12, 0, 12,
13, 30, 13, 7, 17, 12, 30, 29, 22, 10> ]

> // 24 threads
```

(continues on next page)

(continued from previous page)

```
> SetNthreads(24);
> time G := GroebnerBasis(B, 6: Dense, PairsLimit := 5000);
Time: 815.190[r]

> // GPU
> SetNthreads(1); SetGPU(true);
> time G := GroebnerBasis(B, 6: Dense, PairsLimit := 5000);
Time: 519.140
```

Taking $n = 32$ gives a much harder system where the parallel speedup is greater.

```
> n := 32; B := RandomDenseSystem(GF(31), n, 2*n, 2);

> // Single thread only
> time G := GroebnerBasis(B, 6: Dense, PairsLimit := 10000);
Time: 156326.220
> Basis(I);
[
    x[1] + 12, x[2] + 19, x[3] + 8, x[4] + 10, x[5] + 25, ...  x[27] + 21
]
> Variety(Ideal(G));
[ <19, 12, 23, 21, 6, 27, 19, 14, 18, 17, 10, 14, 29, 25, 12, 0, 12,
13, 30, 13, 7, 17, 12, 30, 29, 22, 10> ]

> // 24 threads
> SetNthreads(24);
> I := Ideal(B); // Recreate ideal
> time G := GroebnerBasis(B, 6: Dense, PairsLimit := 10000);
Time: 21921.790[r]

> // GPU
> SetNthreads(1); SetGPU(true);
> I := Ideal(B); // Recreate ideal
> time G := GroebnerBasis(B, 6: Dense, PairsLimit := 10000);
Time: 13335.090
```

5.6 Polynomial System Solving

A fully parallel distributed algorithm has been developed for V2.28 for the solving of a multivariate polynomial system over a moderately small finite field $K = \mathbf{F}_q$. The algorithm has several applications, including algebraic attacks on multivariate polynomial cryptosystems defined over finite fields (including the important case of $\mathbf{F}_2$, and also some types of computations in the arithmetic geometry of schemes over finite fields, where the point counting on schemes often arises.

The algorithm is an extension of the existing algorithm to compute the variety of an ideal: it is given a small integer parameter e and works by evaluating the input polynomials at the last e variables to all possible constants in K (so there are q^e cases enumerated) and then computing the variety of each simplified ideal and combining all the results.

As usual, one first uses `SetNthreads` and optionally `StartWorkers` to select the number of cores/workers. Then the algorithm itself is called by creating an ideal I of a multivariate polynomial ring over a small finite field K and then by calling one of the following functions.

`Variety(I): RngMPol -> [ ModTupFldElt ]`

```
Eval: RngIntElt                    Default:
```

Assuming the parameter `Eval` is set to a small positive integer e, return the variety of I using the algorithm where e variables are set to all elements of the base finite field K.

Note that a Gröbner basis (GB) for I typically **should not** be first created (explicitly or implicitly), since that may be extremely expensive to compute in hard examples, and the main point of the evaluation-based algorithm is to avoid the computation of the GB of the original ideal with all variables.

`VarietySize(I): RngMPol -> RngIntElt`

```
Eval: RngIntElt                    Default:
```

Assuming the parameter `Eval` is set to a small positive integer e, return the size of the variety of I using the algorithm where e variables are set to all elements of the base finite field K.

5.7 Polynomial Rings

A parallel polynomial Schoenhage–Strassen FFT algorithm for very large univariate polynomial multiplication has been developed for V2.26. This applies to univariate polynomials with coefficients in the integer ring, an integer residue class ring, or a prime finite field. Currently non-trivial speedups are achieved when the polynomials have large degree and large coefficients.

`SetParallelFFT(b): BoolElt`

Set whether the parallel polynomial Schoenhage–Strassen FFT algorithm should be used for univariate polynomial multiplication when a non-trivial number of threads is selected (by `SetNthreads`).

Example: Polynomial Multiplication (ex-0a7349)

This example shows the speedups obtained on a 16-core 3.1GHz Intel E5-2687W CPU when multiplying high-degree polynomials with large integer coefficients, with the standard single-thread serial algorithm compared with the parallel Schoenhage–Strassen FFT algorithm running with 16 threads.

```
> SetParallelFFT(true);
> Z := IntegerRing();
> P<z> := PolynomialRing(Z);
> range := [12 .. 18]; SCALE := 8; Time := Realtime;
> for b in range do
>     n := 2^b;
>     nb := Round(n / SCALE); R := IntegerRing(2^nb);
>     F := P![Z!Random(R): i in [1 .. n]];
>     G := P![Z!Random(R): i in [1 .. n]];
>     SetNthreads(1); T1 := Time(); p1 := F*G; T1 := Time(T1);
>     SetNthreads(16); T16 := Time(); p2 := F*G; T16 := Time(T16);
>     assert p1 eq p2;
>     printf
>        "n: %o; #bits: %o, 1T time: %.2o, 16T time: %.1o, speedup: %.1o\n",
>         n, nb, T1, T16, T1/T16;
> end for;
Deg:   4096; #bits:   512, 1T time:   0.06, 16T time:  0.01, speedup:  6.0
Deg:   8192; #bits:  1024, 1T time:   0.30, 16T time:  0.06, speedup:  9.0
Deg:  16384; #bits:  2048, 1T time:   1.10, 16T time:  0.10, speedup:  8.8
Deg:  32768; #bits:  4096, 1T time:   5.20, 16T time:  0.60, speedup:  8.9
Deg:  65536; #bits:  8192, 1T time:  25.40, 16T time:  2.70, speedup:  9.3
Deg: 131072; #bits: 16384, 1T time: 127.30, 16T time: 12.90, speedup:  9.9
Deg: 262144; #bits: 32768, 1T time: 622.90, 16T time: 60.50, speedup: 10.3
```

5.8 Distributed Computation for Internal Algorithms

The term *distributed computation* refers to a computation which is performed on a number of nodes connected by a network. Magma uses the *manager/worker* model for its distributed computations. In this model, one of the nodes performing such a computation, the *manager*, breaks the computation up into small chunks, each of which it sends to one of the other nodes for execution. The later nodes are referred to as *workers*.

5.8.1 Automatic Worker Startup

The following procedures (and the manual worker startup method in the next subsection) apply to the internal distributed parallel algorithms supported by Magma (currently lattice vector enumeration, code min weight/distribution, and integer factorisation). Suppose that the manager is a machine called *node1* and the worker nodes are *node2*, *node3*, etc. Since V2.27, one can start up worker nodes by simply calling the following procedure from within the master job, before starting up a computation which will use distributed computation.

`StartWorkers(H, T): MonStgElt, RngIntElt`

```
Magma: MonStgElt                    Default:
```

Automatically start up a Magma worker on the remote host H (given by a string) with T threads to run on H. By default this procedure starts a *non-login* shell (via *ssh*) on the remote host H and assumes that the command to start Magma is called `magma` on H and is already in the path when that shell starts (determined by the PATH environment variable).

For the Z shell, for example, if the PATH variable is set and exported in the file `.zshenv` to include the directory where 'magma' is, then the proper `magma` command will be automatically found and used, even in a non-login shell. Please do something similar for other shells to ensure 'magma' is in the path with a non-login shell by default.

Alternatively, if `magma` is NOT in the path by default on H, then the `Magma` parameter must be set to the full path name of the Magma command to be used on H (see example below).

`SetWorkerPort(P): RngIntElt`

Set the port to be used for communicating with worker jobs to be P. It is not necessary to choose a port when using `StartWorkers` (an unused port is automatically chosen by default). But one can fix a specific port P by calling this procedure before calling `StartWorkers`.

Whether the port is automatically chosen by Magma or specified by the user, the port is always automatically passed on to the remote workers when they are started up via `StartWorkers`, so they know which port to use to connect back to the job running on the master host.

Example: Starting Workers Automatically (ex-16fbfc)

Here are simple examples which use `StartWorkers` to be run on the manager node1, assuming that the remote machines are node2, node3, etc. (with 16 threads each).

When Magma is in the shell path on these nodes by default:

```
> StartWorkers("node2", 16); // start worker with 16 threads on node2
> StartWorkers("node3", 16); // start worker with 16 threads on node3
```

When the full path of Magma is `/usr/local/bin/magma` (and **not** in the path for a non-login shell):

```
> StartWorkers("node2", 16: Magma := "/usr/local/bin/magma");
> StartWorkers("node3", 16: Magma := "/usr/local/bin/magma");
```

Now one could do a lattice/code operation or integer factorisation and the remote worker machines will automatically run (with the given number of threads when needed).

5.8.2 Manual Worker Startup

Suppose again that the manager is a machine called *node1* and the worker nodes are *node2*, *node3*, etc. One can manually start up worker jobs and communicate with them with a specific port. Since V2.27, it is preferable to use the automatic startup method explained above, but it may sometimes be useful to have more manual control.

The manual setting-up procedure involves first choosing a port number `PORT` which is not already used on the network (for example a number in the range 4000 to 6000 is typically free). The input to each worker consists of something similar to the following statements:

```
NUM_THREADS := 32;       // number of threads to run on worker
PORT := 4000;            // chosen port number
SetNthreads(NUM_THREADS);
Worker("node1", PORT);
```

Alternatively, for each worker, this information can be provided on the Magma command line (with no input file needed):

```
magma -m node1:4000 -t 32
```

The worker jobs can be started up both before and after the manager is started. The manager can successively run multiple kinds of distributed jobs and each worker will contribute to each job and then wait for another job until the manager job finally exits. Also, for the currently supported code and lattice enumeration cases, killing and restarting worker nodes should be robust; that is, if a worker job is deliberately killed or accidentally exits then the manager will notice and the worker's task will be reassigned to another worker (and thus completed at some point). Further, a new worker job with the same port can be started to connect to the same manager job and undertake new work on the same job.

The input for the manager job only needs to set the port number when commencing work:

```
SetWorkerPort(PORT);
```

where PORT is the same port number chosen for the worker jobs. After this, for the code and lattice enumeration algorithms, the input to the master job is simply what one would do for a normal enumeration job (see sections *Integral Lattices* and *Linear Codes* below for details).

Example: Setting the Worker Port (ex-91843e)

For determining the minimum weight of a linear code C, a typical manager input might be:

```
PORT := 4000;           // or other free port number (matching worker)
    SetWorkerPort(PORT);
    SetVerbose("Code", 1);  // for verbose information
    time MinimumWeight(C);
```

The distributed computation can be augmented with the use of threads in the manager process, simply by using `SetNthreads` prior to the start of the computation. For instance, for determining the theta series of a lattice L, a typical manager input might be:

```
NUM_THREADS := 32;      // number of threads to run on manager
    PORT := 4000;           // or other free port number (matching worker)
    SetNthreads(NUM_THREADS);
    SetWorkerPort(PORT);
    SetVerbose("Enum", 1);   // for verbose information
    time ThetaSeries(L, bound);
```

5.9 Integer Factorisation

Parallel versions of the ECM and MPQS integer factorisation algorithm are available since V2.27. One can simply use `SetNthreads(t);` to select t threads on the local node, and optionally use `StartWorkers` to include workers on other nodes. Then integer factorisation will automatically use the parallel versions when the integers are large enough for the ECM or MPQS algorithms to be selected.

Example: Distributed Integer Factorisation (ex-1169a4)

The following code factors 10 random 80-digit integers using 32 threads on the local manager node, and 32 threads on the worker node2.

```
> SetNthreads(32);
> StartWorkers("node2", 32); // start worker with 32 threads on node2
> SetVerbose("Factorization", 1);
> SetVerbose("MPQS",1);
> for i := 1 to 10 do
>     n := Random(10^80);
>     time Factorisation(n);
> end for;
```

5.10 Ideal Class Group of Algebraic Number Fields

Since V2.28, a parallel version of the main algorithm to compute the (ideal) class group of an algebraic number field is available.

The main feature of the new algorithm is that the gathering of relations is fully parallelised: each worker does an independent random walk with short products of random ideals and searches for smooth relations in terms of the factor basis (the set of prime ideals with norm up to some bound). When enough relations are gathered, the master process uses linear algebra operations on the large integer relation matrix to compute the class group.

The relation matrix is now also represented internally by the sparse matrix type, which allows much larger factor bases than could be handled previously, because the memory usage is much less than before, and this is particularly important when there are many workers each of which handles its own local relation matrix.

Although the relation gathering generally dominates the computation, the linear algebra phase at the end can be very non-trivial too, as it involves integer matrix algorithms operating on the relation matrix, which can be huge. (For some inputs, the relation gathering may be relatively much quicker than usual.)

As usual, one first uses `SetNthreads` and optionally `StartWorkers` to select the number of cores/workers. Then the `ClassGroup` function can be called on a number field or order.

5.11 Integral Lattices

5.11.1 Lattice Vector Enumeration

Introduction

Two types of lattice vector enumeration are provided:-

(a) Enumeration of all vectors in a lattice L having norm less than or equal to n.

(b) Enumeration of all vectors in a lattice L whose distance from a non-zero vector w is equal or less than n. Note that the vector w does not have to be in L.

The enumeration of vectors in a lattice is usually very expensive both in terms of CPU time and memory. The kernel of Magma contains code for POSIX threads parallelisation for both short and close vector enumerations. Further, kernel code is provided for distributed computation on top of the POSIX threads. Enumeration lends itself to very efficient parallelisation and the speedup factor achieved is close to the number of cores being used.

Enumeration of Short Vectors

The following intrinsics for enumerating short vectors in a lattice can be executed in parallel:-

`ShortVectors(L, B): Lat, RngIntElt -> SeqEnum`

A sequence containing all vectors in the lattice L having norm less than or equal to the integer B.

`ShortVectors(L, A, B): Lat, RngIntElt, RngIntElt -> LatElt`

A sequence containing all vectors in the lattice L having norm equal of greater then the integer A and less than or equal to the integer B.

`ShortestVectors(L): Lat -> SeqEnum`

A sequence containing all vectors in the lattice L having minimum non-zero norm.

If the vectors of minimal norm are required then the intrinsic `ShortestVectors(L)` should be used as it is faster. The above intrinsics all include a parameter `Max` which terminates the enumeration and returns the vectors found once `Max` vectors satisfying the conditions have been enumerated.

The reader should consult Chapter Lattices on Lattices for a full description of these functions. In particular, each of them has several parameters giving the user some control over the enumeration. The progress of an enumeration can be tracked using `SetVerbose("Enum", 2)` .

Example: Enumerating Vectors of Small Norm `(ex-25bf05)`

We enumerate all non-zero vectors having norm 6 or less in lattice number 6 of rank 64 in the Magma version of the Nebe-Sloane database of lattices using 32 threads on a 16 core machine.

```
> SetNthreads(32);
> L := Lattice(LatticeDatabase(), 64, 6);
> time sv6 := ShortVectors(L, 6);
> #sv6;
1305600
```

So the 1,305,600 non-zero vectors having norm 6 or less in L were enumerated in 63,278 seconds.

A Single Shortest Vector

For some applications it suffices to find a single vector satisfying the norm condition. There at least three ways of doing this efficiently.

(i) If a non-zero vector of minimum norm is sought when the minimum is known, then an application of strong lattice reduction, for example using LLL or BKZ reduction, may produce such a vector much more cheaply than enumeration.

(ii) If a non-zero vector of minimum norm is sought when an upper bound on the mimimum is known, then the intrinsic `Minimum`, which determines the minimum non-zero norm of a lattice, can be used. It has a parameter `NormBound` which allows the user to supply an upper bound on the size of the minimum norm. The intrinsic `Minimum` returns the minimum together with a vector having minimum norm. When the minimum in known this could provide a relatively fast method for obtaining a vector of minimum norm.

(iii) The intrinsics listed below perform a normal short vctor enumeration but terminate as soon as one non-zero vector satisfying the norm condition has been found. It is equivalent to using the `ShortVectors` or `ShortestVectors` intrinsics with the parameter `Max` set to 1.

`ShortVector(L, B): Lat, RngIntElt -> LatElt`

A single non-zero vector of the lattice L having norm at most B, or the zero vector if there is no such vector.

`ShortVector(L, A, B): Lat, RngIntElt, RngIntElt -> LatElt`

A single non-zero vector v of the lattice L having norm such that $A \geq \text{Norm}(v) \leq B$, where A and B are integers.

`ShortestVector(L): Lat -> LatElt`

A single non-zero vector of the lattice L having minimum norm.

Example: Finding a Single Shortest Vector (ex-236335)

The theta series for lattice number 1 of rank 105 in the Nebe-Sloane lattice database shows that there are vectors of norm 40. We find one such vector using 32 threads on a 16 core machine.

```
> SetNthreads(32);
> L := Lattice(LatticeDatabase(), 105, 1);
> sv40 := ShortVector(L, 40, 40);
> sv40;
(1 -15 9 -10 0 3 -6 4 -2 6 2 0 3 -3 1 -3 -1 4 4 -5 7 10 3 -2 5 -2 7
    -5 -1 1 7 -2 -2 -3 -3 -1 0 1 -1 2 -2 1 1 2 0 0 3 0 0 -1 1 1 1 0
     0 1 1 2 2 2 -1 0 0 0 1 2 0 0 -1 1 0 -1 -1 2 -1 -3 1 -2 1 -2 -1
     0 0 -1 1 -2 -2 1 0 1 -2 1 2 1 0 0 0 0 1 -1 0 0 -2 1 0)
> Norm(sv40);
40
```

The computation took 901 seconds real time.

Enumeration of Close Vectors

The intrinsics for enumerating close vectors are similar to those for enumerating short vectors. Again the parameter `Max` allows early termination.

The following functions which enumerate close vectors in a lattice can be executed in parallel:-

`CloseVectors(L, w, B): Lat, ModTupRngElt, RngElt -> SeqEnum`

A sequence containing all vectors v in the lattice L close to the vector w such that $\text{Norm}(v-w) \leq B$. The vector w need not belong to the lattice L.

`CloseVectors(L, w, A, B): Lat, ModTupRngElt, RngElt, RngElt -> SeqEnum`

A sequence containing all vectors v in the lattice L close to the vector w such that $A \leq \text{Norm}(v - w) \leq B$. The vector w need not belong to the lattice L.

`ClosestVectors(L, w): Lat, ModTupRngElt -> SeqEnum`

A sequence containing all vectors v in the lattice L such that $\text{Norm}(v, w)$ is a minimum. The vector w need not belong to the lattice L.

Example: Enumerating Close Vectors in a Database Lattice (ex-8c952d)

Lattice number 1 of rank 105 in the Nebe-Sloane lattice database is created and an arbitrary vector w is found. We then enumerate the vectors of norm 40 using 32 hyperthreads on a machine with 16 cores.

```
> SetNthreads(32);
> L := Lattice(LatticeDatabase(), 105, 1);
> B := Basis(L);
> w := 3* B[23] + 7*B[37];
> time cv40 := CloseVectors(L, w, 40, 40);
Time: 1820.750[r]
> #cv40;
478548
```

So there are 478,548 vectors at distance 40 from the vector w. The real time for the computation was 1,854 seconds while the total CPU time was 55,825 seconds.

Example: Close Vectors in a Sum of Leech Lattices (ex-4b7a6d)

A lattice D of dimension 120 is constructed by forming the direct sum of five copies on the 24-dimensional Leech lattice. A vector w is constructed and the non-zero vectors closest to it are enumerated. The computation was run on a manager machine node1 with 16 cores/32 hyperthreads together with 4 worker machines each with 16 cores/32 hyperthreads.

The following were executed on the master machine node1:-

```
> SetNthreads(32);
> StartWorkers("node2", 32); // start worker with 32 threads on node2
> StartWorkers("node3", 32); // start worker with 32 threads on node3
> StartWorkers("node4", 32); // start worker with 32 threads on node4
> StartWorkers("node5", 32); // start worker with 32 threads on node5
> L := Lattice(LatticeDatabase(),"Leech");
> D := DirectSum([L, L, L, L, L]);
> w := (D.9+D.24+D.31+D.44+D.67+D.60+D.75+D.82+D.97)/2;
> time cv, m := ClosestVectors(D, w);
Time: 45008.000[r]
> #cv;
10616832
```

The enumeration found $10,616,832 = 2 \times 48^4$ lattice vectors each of distance 9 from the vector w. The real time for the computation was 45,008 seconds while the total CPU time was 1,377,986 seconds.

A Single Closest Vector

`CloseVector(L, w, B): Lat, ModTupRngElt, RngElt -> LatElt`

A single non-zero vector w of the lattice L such that $\text{Norm}(v - w) \leq B$, or the zero vector if there is no such vector.

`CloseVector(L, w, A, B): Lat, ModTupRngElt, RngElt, RngElt -> LatElt`

A single non-zero vector w of the lattice L such that $A \leq \text{Norm}(v - w) \leq B$ or the zero vector if there is no such vector.

`ClosestVector(L, w): Lat, ModTupRngElt -> LatElt`

A single non-zero vector of the lattice L that is closest to w.

Example: Closest Vector in a Random Lattice `(ex-0606a0)`

A vector closest to a random vector w is found in a random lattice L of rank 60. The computation was run using 32 hyperthreads on a machine with 16 cores.

```
> SetNthreads(32);
> d := 60;
> B:= Matrix(d, d, [Random([-99..99]) : i in [1..d^2]]);
> L := LatticeWithBasis(B);
> L:Minimal;
Lattice of rank 60 and degree 60
> w := Vector([Random([-99..99]) : i in [1..d]]);
> w;
(50 -85 -21 -40 4 -72 32 -90 51 -78 -93 -84 -12 60 90 35 55 91 -54 -87 -92 -82
    -92 -7 75 72 -4 55 22 -81 88 56 23 94 1 -91 -16 -8 38 -93 -26 -41 -75 57 49
    -6 -55 56 31 41 -34 -28 34 -22 -74 23 24 31 -80 -35)
> Norm(w);
210094

> time sv := ClosestVector(L, w);
Time: 90.270[r]
>  sv;
(68 -68 -130 -10 -68 -154 88 -76 122 -113 -156 -37 34 43 77 -17 -9 -119 -202 -79
    -117 -86 -77 -1 53 41 24 -64 34 -106 125 101 79 -11 143 -12 62 -9 -13 -131
    -47 -19 47 140 73 -39 -25 4 -94 1 -104 -45 -83 -52 -89 -18 -41 30 -86 66)
> Norm(sv);
400300
```

(continues on next page)

(continued from previous page)

```
> Norm(sv - w);
270010
```

The total CPU time was 1540 seconds while the real time was 90 seconds.

Intrinsics that Use Enumeration

If parallelism is turned on then the computation of each of the following lattice properties is parallelised to the extent that vector enumeration is used:

1. Minimum: `Minimum`
2. Kissing number: `KissingNumber`
3. Packing radius: `PackingRadius`
4. Hermite invariant: `HermiteNumber`
5. Centre density: `CentreDensity`
6. Density of the lattice packing: `Density`
7. Covering radius: `CoveringRadius`
8. Theta series: `ThetaSeries`

Computing each of the above items involves the enumeration of lattice vectors. In some, such as `PackingRadius` it is the dominant part of the computation, while in others, such as `KissingNumber` and `CoveringRadius`, it may be only a small part.

Example: Serial and Threaded Lattice Minimum (ex-2c4b94)

We compute the minimum of lattice number 6 of dimension 64 in the Magma version of the Nebe-Sloane database by firstly doing a serial computation and then using 32 POSIX threads on a single 16-core/32 hyperthreads machine.

Note that lattices loaded from the database often contain some properties of the lattice and so these need to be removed in order to avoid Magma using the data from the database instead of computing it.

```
> // Serial computation (1 thread)
> L := Lattice(LatticeDatabase(), 64, 6);
> time m := Minimum(L); m;
Time: 2566.750
8

> // Parallel computation (32 threads)
> SetNthreads(32);
> L := Lattice(LatticeDatabase(), 64, 6);
> time m := Minimum(L); m;
Time: 125.850
8
```

The multithread calculation is 20.4 times faster than the serial computation. For harder problems the ratio of serial time to multicore time approaches n, where n is the number of threads being used. When using a machine with hyperthreading there may be benefit in setting the number of threads to be used higher than the number of cores as in the case above.

Example: Lattice Minimum on Two Machines (ex-e3d7d1)

In this example we use two machines each with 16 cores and 32 hyperthreads to compute the minimum of another lattice of dimension 64 from the Nebe-Sloane database of lattices. So we have a manager and one worker each using 32 threads.

The manager input is:

```
> SetNthreads(32);
> StartWorkers("node2", 32); // start worker with 32 threads on node2
> L := Lattice(LatticeDatabase(), 64, 4);
> time m := Minimum(L);
```

This is a hard enumeration as the minimum turns out to be 12 and so all possible combinations of basis vectors that could produce a vector of norm less than 12 have to be considered. The serial time is 239,741 seconds while the real time for the parallel run is 5,647 seconds. The use of 32 POSIX threads on the manager reduces the runtime by a factor of 21. Using a second machine (a worker) cuts that time in half, giving us a speed up by factor of 42.45, which is what we expect. A small amount of data suggests that using two identical workers results in cutting the single machine using POSIX threads time by a factor close to 3.

Example: Kissing Number of a Lattice (ex-c126c1)

We determine the kissing number of the first lattice of dimension 48 in the Nebe-Sloane database. We use 32 threads on a 16-core machine.

```
> SetNthreads(32);
> L := Lattice(LatticeDatabase(), 48, 1 );
> kn := KissingNumber(L : NormBound := 8);
> kn;
9828000
```

The minimum of the lattice is 8 and we find that the kissing number is 9,828,000. The runtime was 87,940 seconds. As a comparison, computing the minimum took 3,295 seconds also using 32 threads.

Theta Series

If the first few terms of the theta series of a lattice can be computed cheaply then it provides some important information about the lattice. The intrinsic for computing the first n terms of the theta series for lattice L is `ThetaSeries` $(L,\ n)$. The direct computation of the terms is similar to an application of `ShortVectors` and so it benefits directly from the parallelisation of enumeration.

Example: Theta Series of a Lattice of Dimension 40 (ex-f705e4)

We compute 8 terms of the theta series for a lattice of dimension 40 using Pthreads on a single 16-core machine.

```
> SetNthreads(32);
> L := Lattice(LatticeDatabase(), 40, 10);
> time ts<t> := ThetaSeries(L, 8);
```

The real time for the computation was 1,936 seconds and the CPU time was 57,722 seconds. The computation produces the following terms of the theta series:-

$$1 + 39600\ t^4 + 1048576\ t^5 + 45916160\ t^6 + 817889280\ t^7 + 10378028080\ t^8 + O(t^9)$$

Example: Theta Series by Direct Enumeration (ex-31137d)

The coefficients of the theta series of the lattice of dimension 105 in the Nebe–Sloane database grow very slowly and so determining the first 46 terms can be achieved using direct enumeration:-

```
> SetNthreads(32);
> L := Lattice(LatticeDatabase(), 105, 1);
> time ts<t> := ThetaSeries(L, 46);
> ts;
```

The computation was done on a single 16-core machine using Pthreads. The real time for the computation was 16,944 seconds and the CPU time was 524,539 seconds. The computation produces

the following terms of the theta series:

$$1 + 1890\ t^{24} + 3780\ t^{28} + 15120\ t^{34} + 57960\ t^{36} + 45360\ t^{38} + 478548\ t^{40} + 635040\ t^{42} + 2381400\ t^{44} + 5276880\ t^{46} + O(t^{47})$$

The theta series of an integral lattice L is (the q-expansion of) a modular form whose weight is half the dimension of the lattice. The space of modular forms having even weight, level, and character is finite dimensional, which means that the theta series is uniquely characterised as an element of that space from a knowledge of finitely many of its coefficients. Techniques for identifying the modular form in that space are being developed and this is likely to be a useful method when calculating the theta series of many lattices. One benefit of this approach is that it yields the entire theta series in the sense that the user can request any number of terms given the modular form of the lattice.

Example: Theta Series via Modular Forms `(ex-e70551)`

In the case of a lattice of dimension 72 recently studied by G. Nebe and M. Watkins the only property that could be determined directly was its minimum. However, the modular form method does succeed in computing its theta series after a rather lengthy computation. Here we display the first 40 terms:-

$$1 + 6218175600t^8 + 15281788354560t^{10} + 9026867482214400t^{12} + 1989179450818560000t^{14} + 213006159759990870000t^{16} + 31440875176314109952 00t^{18} + 525100718690287495741440t^{20} + 14756609779472604266496000t^{22} + 310160311536865273422120000t^{24} + 5108106498702230684344320000t^{26} + 68347693198345293752360140800t^{28} + 764604200349778385836033966080t^{30} + 7318835829467289011348969122800t^{32} + 61087852878139533146733527040000t^{34} + 451627409037619180720939253760000t^{36} + 2996530109404572751301781710438400t^{38} + 13721646486555338 55t^{40} + O(t^{42})$$

This calculation made heavy use of the parallel enumeration code and the real time was 276,498 seconds on a 16 core machine running no other jobs.

5.12 Linear Codes

Important invariants of a linear code relate to the weights of code words and in particular, their minimum and distribution. The computation of these invariants involves some form of enumeration of code words and in many ways is similar to the enumeration of lattice vectors. Since V2.26-4 (May 2021), we have parallelised the computation of the minimum weight of a code word in a code and the distribution of weights.

In this section all codes will be linear codes over a finite field. The two main enumerations which have been parallelised are:-

(i) Determining the minimum weight of a code C: Intrinsic `MinimumWeight`(C)

(ii) Determining the weight distribution of a code: Intrinsic `MinimumDistance`(C)

Related algorithms that are also parallelised are:-

(i) `MinimumDistance` (C): Same as minimum weight

(ii) `MinimumWord` (C): Returns a word of minimum weight

Pthreads code is available in the Magma kernel for each type of enumeration as well as distributed parallelism of nodes which are running the Pthreads code. The progress of a code word enumeration can tracked using the statement `SetVerbose ("Code", 1);` .

5.12.1 Minimum Weight

Perhaps the most important invariant is the minimum weight of a code word as this determines the error-correction capability of the code. The serial algorithm used in Magma is based on an algorithm of A.E. Brouwer and K.H. Zimmermann with improvements by M. Grassl, A. Steel, and G. White. A version of the algorithm that uses both POSIX threads and/or distributed computation is available in Magma. As usual, one uses `SetNthreads` and optionally `StartWorkers`.

Example: Minimum Weight of a QR Code (ex-79c75f)

Let C be the binary quadratic residue code of length 191 and dimension 96. Quadratic residue codes can be constructed using the Magma intrinsic `QRCode`. Its minimum was found to be 27 on single relatively slow machine with 36 cores. The real time was 516,120 seconds and the CPU time was 36,667,875 seconds. Note that a total of 4,761,899,689,016,196 vectors were enumerated. The code for this calculation is:-

```
> SetNthreads(108);
> C := QRCode( GF(2), 191);
> time mw := MinimumWeight(C); mw;
```

Let C be the quadratic residue code $[97, 49]$ over GF(3). Using Pthreads on a single machine, its minimum was found to be 23 in 21,121 seconds real time. The code for this calculation is:-

```
> SetNthreads(32);
> C := QRCode( GF(3), 97);
> time mw := MinimumWeight(C); mw;
```

Let D be the binary code constructed from the lines of the dual of a Dempwolff plane. Let the code C be the hull of D, that is $C := D \cup \mathrm{Dual}(D)$. This is a $[273,\ 102]$ code over GF(2) and we find its minimum to be 24 using one manager machine and one worker machine each having 16 cores. The real time for the computation was 1391 seconds.

Example: Minimum Weight of a Best Known Code (ex-02eba6)

In this rather large example we use both POSIX threads and distributed computation. We determine the minimum weight of the Best Known Linear Code over GF(2) having block length 256 and dimension 56. We run 32 threads on each of a master and four worker 16-core machines.

The following job is started on the master machine node1:–

```
> SetNthreads(32);
> StartWorkers("node2", 32); // start worker with 32 threads on node2
> StartWorkers("node3", 32); // start worker with 32 threads on node3
> StartWorkers("node4", 32); // start worker with 32 threads on node4
> StartWorkers("node5", 32); // start worker with 32 threads on node5
> C := BestKnownLinearCode( GF(2), 256, 56 );
> ResetMinimumWeightBounds(C);
> d := MinimumWeight(C);
> d;
68
```

The runtime was 20,560 seconds. A second job that used only 32 threads on a single machine took 94,082 seconds. The speedup using five machines instead of one is a factor of 4.5 whereas 160 threads compared to 32 threads gives a factor of 5. The difference of 0.5 is explained by the fact that out of the five machines used for the 160 thread run, three of the machines are slower than the machine used in the 32 thread run. Consequently, if the minimum of a code is computed using the same number of threads on each of n identical machines (compute nodes) the runtime can expected to be a factor of n faster than if the computation is run on a single machine (compute node).

5.12.2 Weight Distribution

The weight distribution of a code C is found using the intrinsic `WeightDistribution` (C).

Related algorithms that are also parallelised:-

(i) `WeightEnumerator` (C) : Weight distribution presented in polynomial form

(ii) `DualWeightDistribution` (C): Weight distribution of the dual code of C

Example: Weight Distribution of a QR Code `(ex-92e5c9)`

We construct the weight distribution for a binary [97, 49] quadratic residue code using the following Magma statements. The computation was run on a machine with 16 cores using Pthreads parallelisation combined with distribution of the work over two machines (one manager and one worker).

Manager (running on node1):

```
> SetNthreads(32);
> StartWorkers("node2", 32); // start worker with 32 threads on node2
> SetVerbose("Code", 1);
> C := QRCode(GF(2), 97);
> time wd := WeightDistribution(C); wd;
```

The computation took 10,840 seconds real time and 345,508 seconds CPU time. The weight distribution is:-

```
[ <0, 1>, <15, 4656>, <16, 23862>, <17, 14841>, <18, 65960>, <19, 190120>,
 <20, 741468>, <21, 3375988>, <22, 11662504>, <23, 40893648>, <24, 126088748>,
<25, 360406410>, <26, 998048520>, <27, 2614461952>, <28, 6536154880>,
<29, 15589859400>, <30, 35337014640>, <31, 76345991760>, <32, 157463608005>,
<33, 310186322386>, <34, 583880136256>, <35, 1050952990464>,
<36, 1809974594688>,  <37, 2983811972296>, <38, 4711282061520>,
<39, 7127688680200>,  <40, 10335148586290>, <41, 14368570181112>,
<42, 19158093574816>, <43, 24503812839176>, <44, 30072861211716>,
<45, 35419208941404>, <46, 40039105759848>, <47, 43447332367896>,
<48, 45257637883225>,  <49, 45257637883225>,
<50, 43447332367896>, <51, 40039105759848>, <52, 35419208941404>,
<53, 30072861211716>, <54, 24503812839176>, <55, 19158093574816>,
<56, 14368570181112>, <57, 10335148586290>, <58, 7127688680200>,
<59, 4711282061520>, <60, 2983811972296>,  <61, 1809974594688>,
<62, 1050952990464>,  <63, 583880136256>, <64, 310186322386>,
<65, 157463608005>, <66, 76345991760>, <67, 35337014640>, <68, 15589859400>,
<69, 6536154880>, <70, 2614461952>, <71, 998048520>, <72, 360406410>,
<73, 126088748>, <74, 40893648>, <75, 11662504>, <76, 3375988>, <77, 741468>,
<78, 190120>, <79, 65960>, <80, 14841>, <81, 23862>, <82, 4656>, <97, 1> ]
```

5.13 User-Implemented Distributed Parallelism

5.13.1 Introduction

The low-level features necessary to perform distributed parallelism in Magma are sockets and I/O multiplexing; these have been in place for a considerable while. However, using them effectively was often challenging and fiddly, especially when trying to be robust in the face of network errors. Amongst other issues, communicating anything but the simplest data meant needing to take into account the possibilities of fragmentation of that data and managing the reassembly explicitly.

Previously two key features for simplifying distributed communication have been introduced: object transmission and asynchronous I/O. Together, these remove most or all of the complexity in the commu-

nication component of a distributed parallel computation. The range of objects that support transmission in this way continues to be expanded as part of our ongoing efforts.

The next step up is to simplify the process of implementing distributed parallel computations themselves. There are several ways this might be done; for the V2.26 release the concentration is on the manager/worker model, with the goal to make implementing such a system as easy as possible. Other models are under consideration, but are left for the future.

5.13.2 The Manager/Worker Model

In this model of distributed parallelism, there is one special process designated the *manager*, and one or more other processes designated *workers*. The manager has a list of data defining work that needs to be done. Workers contact the manager to register their availability to do work, then settle into a cycle of performing tasks received from the manager and reporting the results. The manager allocates tasks to workers and collates the results, then returns them to the user when the computation is finished. It also needs to manage the pool of workers as a whole, handling the addition and removal of workers from this pool as necessary.

We focus on a particular way of using this model. In this approach, a contact socket is set up for the manager, and a list defining work tasks is given to it. Then workers are started on various machines and told to contact the manager (using the known socket information). Once all of the work has been completed the workers exit and the manager reports the results.

This particular approach is well-suited to the situation where the entire computation is distributed, with a clearly defined set of work that is known up-front. For instance, if one were to want to compute some data for each of a large number of objects and then save the results to a file to later be integrated into a database — this would naturally be handled by setting up a manager with the list of those objects and workers that know how to compute that data for one of the objects.

We have found this approach to be a good fit for many of our needs, and the implementation of the manager/worker model to be a powerful tool for this. However, it is not always the best approach to take; consequently, we also show details of an implementation so that it can be adjusted to meet different circumstances.

5.13.3 Basic Usage

At the simplest level, the interface consists of two intrinsics: `DistributedManager` and `DistributedWorker`.

A user wishing to use distributed parallelism first creates a socket for the manager to use and then calls `DistributedManager`. Then workers are started on the desired machines, with each worker calling `DistributedWorker`. The manager will allocate tasks to each worker and collect the results. When all tasks are complete the `DistributedManager` returns the collated results, and each call to `DistributedWorker` exits.

`DistributedManager(socket, items : parameters): IO, SeqEnum`

`DistributedManager(socket, items : parameters): IO, List`

`DistributedManager(socket, items : parameters): IO, Process`

> The arguments for `DistributedManager` are a previously-created server socket for communication purposes and a sequence (or list) of work items. Each work item will be sent to a worker. The return value is the sequence of results computed by the workers, in the order of the corresponding work items.
>
> There is also a version that takes a `Process` for cases where it is not desirable to generate all inputs at once.
>
> The parameters are explained later in the sections on Alternative Results Handling (*Alternative Results Handling*) and Task Group Management (*Task Group Management*).

`DistributedWorker(host, port, work_function): MonStgElt, RngIntElt, UserProgram`

> The arguments to `DistributedWorker` are the host and port information for the manager's socket, and a function that takes a work item as input and returns the appropriate result.

5.13.4 Basic Example

For example purposes we will use a setup with four machines, named circle, triangle, square, and star. The manager will run on circle, while one worker will run on each of triangle, square, and star. We will use port 10000 on circle for communication.

The workload in question will be some very basic information about the Collatz map. This map acts on a positive integer n to produce another positive integer, as follows: If n is even, then the result is $n/2$; if n is odd, then the result is $3n + 1$. It is widely believed that repeated iteration of the Collatz map will always eventually produce the number 1, regardless of the original starting number, but a proof has yet to be found.

Workers will be given a number n and return a tuple containing the original value of n, the number of iterations of the Collatz map required to reach 1, and the largest value that was encountered in the process. The manager will compute this data for each of the first 10 integers.

The worker code is the following, which we put in a file `"collatz_worker.m"` to make it easier to spawn workers. We will use this worker code unchanged in the basic examples.

Example: Basic Worker for the Collatz Map (ex-8a18c1)

```
host := "circle";
port := 10000;

procedure sleep_ms(ms)          // "sleeps" for the given number of milliseconds
    _ := WaitForIO([] : TimeLimit := ms);
end procedure;

collatz := func<n | IsEven(n) select n div 2 else 3*n + 1>;
function collatz_info(n)
    k := n;
    niters := 0;
    maxval := k;
    while k gt 1 do
        k := collatz(k);
        niters +:= 1;
        maxval := Max(maxval, k);
    end while;
    sleep_ms(100*Random(1, 3));        // random delay for example purposes only
    return <n, niters, maxval>;
end function;

DistributedWorker(host, port, collatz_info);
quit;
```

The implementation is straightforward except for the call to `sleep_ms`; that line causes the work function to delay for one of 100ms, 200ms, or 300ms. This is done purely for example purposes as the workload is so small that the first worker would likely accept and compute the data for all tasks before the second worker could even be started. Obviously actual workloads should not have such a delay.

The manager session (from a Magma launched on circle) might look like this:

```
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10]);
```

Now this Magma is waiting for workers. We start them on the other machines with the following command:

```
magma collatz_worker.m
```

A short while after doing the above, the workers on each machine exit and the call to `DistributedManager` returns:

```
[ <1, 0, 1>, <2, 1, 2>, <3, 7, 16>, <4, 2, 4>, <5, 5, 16>,
<6, 8, 16>, <7, 16, 52>, <8, 3, 8>, <9, 19, 52>, <10, 6, 16> ]
```

5.13.5 Verbose Output

Two verbose flags are available for the manager's use. These provide a little information about the workers, and can be useful in order to track down issues, to reassure yourself that things are actually happening, or to log partial progress in case of a crash.

`SetVerbose("ManagerWorker", v): MonStgElt, RngIntElt`

Setting this verbose flag to 1 will cause a message to be printed each time a worker joins or leaves. This message includes the worker's local host and port.

Setting this verbose flag to 2 will additionally cause a message to be printed each time a worker is assigned a task or reports the results of an assigned task. The task data (and result) will also be shown.

`SetVerbose("ManagerWorkerTime", v): MonStgElt, RngIntElt`

Setting this verbose flag to 1 will add timestamps to the other verbose output.

Example: Verbose Manager Output (ex-40e491)

We run the basic example again, but with verbose output on.

```
> SetVerbose("ManagerWorker", 2);
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10]);
New worker joined from triangle:14342
Worker triangle:14342 assigned task 1
Worker triangle:14342 completed task 1 with result <1, 0, 1>
Group 1 complete, terminating 0 other workers
Worker triangle:14342 assigned task 2
New worker joined from square:12711
Worker square:12711 assigned task 3
Worker triangle:14342 completed task 2 with result <2, 1, 2>
Group 2 complete, terminating 0 other workers
Worker triangle:14342 assigned task 4
Worker triangle:14342 completed task 3 with result <3, 7, 16>
Group 3 complete, terminating 0 other workers
Worker square:12711 assigned task 5
Worker triangle:14342 completed task 4 with result <4, 2, 4>
Group 4 complete, terminating 0 other workers
Worker triangle:14342 assigned task 6
New worker joined from star:17338
Worker star:17338 assigned task 7
Worker star:17338 completed task 7 with result <7, 16, 52>
Group 7 complete, terminating 0 other workers
Worker star:17338 assigned task 8
Worker square:12711 completed task 5 with result <5, 5, 16>
Group 5 complete, terminating 0 other workers
Worker square:12711 assigned task 9
Worker triangle:14342 completed task 6 with result <6, 8, 16>
Group 6 complete, terminating 0 other workers
Worker triangle:14342 assigned task 10
Worker square:12711 completed task 9 with result <9, 19, 52>
Group 9 complete, terminating 0 other workers
```

(continues on next page)

(continued from previous page)

```
Worker star:17338 completed task 8 with result <8, 3, 8>
Group 8 complete, terminating 0 other workers
Worker triangle:14342 completed task 10 with result <10, 6, 16>
Group 10 complete, terminating 0 other workers
[ <1, 0, 1>, <2, 1, 2>, <3, 7, 16>, <4, 2, 4>, <5, 5, 16>,
<6, 8, 16>, <7, 16, 52>, <8, 3, 8>, <9, 19, 52>, <10, 6, 16> ]
```

Here workers are identified by their host and local port number. The "task n" components are not the task number, but the task data; in this example they happen to be the same. The meaning of the groups and worker termination messages is explained later in the section on group tasks.

5.13.6 Alternative Results Handling

The simple approach described above works for many cases, but there will be other situations where it does not suffice. One common example is when the results are incompatible elements that cannot be put into the same sequence. To handle such situations, `DistributedManager` has three parameters related to results that may be set.

```
initial_results: Any                              Default: [ ]
update_results : UserProgram                      Default:
process_results: UserProgram                      Default:
```

The parameter `initial_results` provides the initial value to use for the results of the routine. If not set, this defaults to the empty sequence. If it is instead set to the empty list (`[* *]`) then the results will be returned in a list rather than a sequence.

The parameter `update_results` is a procedure that modifies the results value to incorporate the result that has been computed for a particular input. It must have one of two forms:

- `procedure(~results, result, index)`

or

- `procedure(~results, result, index, taskdata)`

The arguments are:

- `results` is the current value of the results information, which is initially set to `initial_results`.
- `result` is the new result to incorporate into the results.
- `index` is the index of the item (in the items array passed to `DistributedManager`) that this result is for.
- `taskdata` (in the second form) is the specific task data used to compute this result. This will be original input item unless task groups are used (see later).

The parameter `process_results` is a procedure that modifies the results information after it has all been computed and before returning it to the user. It must have the form:

- `procedure(~results)`

This is completely equivalent to a similar call on the results after delivery. Its uses are thus limited, but it is occasionally convenient to do things this way.

These parameters are all optional, except that the `update_results` *must* be provided if the `initial_results` were set to something other than the empty sequence or the empty list.

Example: Collecting Results in a List `(ex-67d2bd)`

We run the basic example again, but with list results.

```
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10] : initial_results := [* *]);
```

(continues on next page)

(continued from previous page)

```
[* <1, 0, 1>, <2, 1, 2>, <3, 7, 16>, <4, 2, 4>, <5, 5, 16>,
<6, 8, 16>, <7, 16, 52>, <8, 3, 8>, <9, 19, 52>, <10, 6, 16> *]
```

Example: Tracking the Maximum Value (ex-022c8d)

Suppose that, instead of the results for all items, we only wanted the maximum number encountered (and the associated initial value that led to it). Then we could change our results item to be a tuple of this maximum value and input value, as follows:

```
> initial_results_maxval := < 0, 0 >;
> update_results_maxval := procedure(~results, result, index)
>     n, niters, maxval := Explode(result);
>     assert index eq n;
>     if results[1] lt maxval then
>         results := < maxval, n >;
>     end if;
> end procedure;
>
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10] :
>     initial_results := initial_results_maxval,
>     update_results := update_results_maxval
> );
< 52, 7 >
```

Of course, for such a small example we could have just examined the results at the end to compute this, but for more interesting workloads it may be undesirable to keep large outputs around when only a small amount of data is wanted.

Example: Average Number of Iterations (ex-96e14f)

This time, we will only care about the average number of iterations taken over all inputs. We thus accumulate the total number of iterations during the computation, then process the result at the end.

```
> initial_results_niters := < 0, 0 >;
> update_results_niters := procedure(~results, result, index)
>     n, niters, maxval := Explode(result);
>     assert index eq n;
>     results := < results[1] + niters, results[2] + 1 >;
> end procedure;
> process_results_niters := procedure(~results)
>     results := results[1] / results[2];
> end procedure;
>
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10] :
>     initial_results := initial_results_niters,
>     update_results := update_results_niters,
>     process_results := process_results_niters
> );
67/10
```

5.13.7 Task Groups

While the options already presented work well for simple situations, there are a few common patterns that arise sufficiently often that special handling has been provided for them.

One such pattern is based on trying several different approaches for an item, and taking whichever result finishes first. This is particularly useful when you have a set of heuristics about ways to solve the problem, but also want to cater for the cases where those heuristics are not correct. Indeed, it can be a powerful tool to aid in the development of such heuristics. A subcategory of this pattern is the ability to try a task with a small bound, and then retry it with a higher bound if unsuccessful.

Another common approach is to split the work for a particular input into several components, such as localisations at certain primes, and then combine all of the factors together to produce the desired result for that item. While this could be handled in the existing framework by making separate items for each local component, it can become less clear to do so.

In order to handle the above usage patterns, and others, the notion of task groups is introduced. With task groups, each input item can be split into multiple tasks that are all associated with the original item. The results of these subtasks can be combined, and once a satisfactory result is computed any remaining workers still assigned to the no-longer-necessary tasks of that group are caused to exit. (This requires a means of restarting such workers, which we describe below.)

5.13.8 Task Group Management

There are three further parameters to `DistributedManager` that provide functionality for task groups.

```
group_tasks     : UserProgram                    Default:
update_group    : UserProgram                    Default:
incomplete_group: UserProgram                    Default:
```

The parameter `group_tasks` is a function that takes an input item and returns the initial data defining the associated task group. It must have the following form:

- `function(item) -> group_result, task_data`

The first return value is an initial value for the group result (in a similar way to how the parameter `initial_results` provides an initial value for the overall results). The second return value is a sequence, list, or other iterable aggregate of task data that will be passed to workers.

The parameter `update_group` is a function that will take the result of a task and use it to adjust the group result (in a similar manner to how `update_results` updates the overall results). It must have the following form:

- `function(item, task, task_result, group_result)`
- `-> group_result, task_data, done`

The arguments are:

- `item` is the original input item with which this task group is associated.
- `task` is the data for the specific task within that group that was just completed.
- `task_result` is the result of that computation.
- `group_result` is the current partial result for the task group.

The first return value is the new value for the group result that incorporates the result of this task. The second return value is a sequence, list, etc. of task data to be used for more tasks. The third return value is a boolean status indicating whether the computation for this group is now complete.

The parameter `incomplete_group` is a function used to handle groups which ran out of associated tasks without reaching a definitive conclusion. i.e., for which the third value returned by `update_group` was always `false`. It must be a function with the following form:

- `function(item, group_result) -> result`

The first argument is the original input item that the task group is associated with, while the second is the partial group result that was computed for it. It returns a result value to include in the results (and processed by the `update_results` routine, if one is given).

Note that `update_group` may produce further tasks to try; this allows first trying a simple approach, and then spawning more complicated (or time-consuming) approaches if that is unsuccessful.

All of these parameters are optional.

Example: Task Groups over Ranges of Inputs (ex-844942)

We will treat each input item as a shorthand expressing a range of one hundred values of interest (item n indicating the range $100n - 99$ .. $100n$). For a given range, we return the largest value encountered and the associated starting value, as in variation one of the basic example.

```
> group_tasks_hundred := function(n)
>     return < 0, <0, 0> >, [ 100*n - 99 .. 100*n ];
> end function;
>
> update_group_maxval := function(item, task, tresult, gresult)
>     n, niters, maxval := Explode(tresult);
>     gcount, gtuple := Explode(gresult);
>     if gtuple[1] lt maxval then
>         gtuple := < maxval, n >;
>     end if;
>     gcount +:= 1;
>     done := gcount eq 100;
>     gresult := done select gtuple else < gcount, gtuple >;
>     return gresult, [], done;
> end function;
>
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10] :
>     group_tasks := group_tasks_hundred,
>     update_group := update_group_maxval
> );
[ <9232, 27>, <9232, 103>, <13120, 255>, <13120, 383>, <39364, 447>,
<39364, 511>, <41524, 639>, <250504, 703>, <190996, 871>,
<250504, 937> ]
```

Note: If running this example, we recommend adjusting the millisecond delay in the worker function down by using a multiplier of 10 instead of 100.

A key point to note in this example is that our partial group result includes an extra value (`gcount` in the code) that is not in the final result. This is so we can tell when the computation is actually complete; since some tasks finish earlier than others, we cannot just check the task data in order to decide whether the group computation is complete or not. Of course, we then have to remove it from the returned group result *if* the computation is complete. This is done in the second last line of that function where we conditionally assemble the new group result.

We could have left the count in and removed it with a suitable `update_results` routine (or even with `process_results`), but in this instance it felt tidier to do it in the group update function.

Example: Tasks Generating Further Tasks (ex-a19be5)

The original example generated all tasks in the range at the beginning, and no new ones after that. We demonstrate another alternative by making each task in the group generate the next one, until the group is complete.

```
> group_tasks_hundred := function(n)
>     return < 0, 0 >, [ 100*n - 99 ];
```

(continues on next page)

(continued from previous page)

```
> end function;
>
> update_group_maxval := function(item, task, tresult, gresult)
>     n, niters, maxval := Explode(tresult);
>     assert task eq n;
>     if gresult[1] lt maxval then
>         gresult := < maxval, n >;
>     end if;
>     done := IsDivisibleBy(n, 100);
>     tasks := done select [] else [ task + 1 ];
>     return gresult, tasks, done;
> end function;
>
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10] :
>     group_tasks := group_tasks_hundred,
>     update_group := update_group_maxval
> );
[ <9232, 27>, <9232, 103>, <13120, 255>, <13120, 383>, <39364, 447>,
<39364, 511>, <41524, 639>, <250504, 703>, <190996, 871>,
<250504, 937> ]
```

Here there is only ever one task in the group at a time; when that task completes it either spawns the next one (by returning the increased task number in a sequence) or finishes if this was the last task in the group. In this scenario we can use the task data instead of needing a `gcount` value or similar, because we have guaranteed the order in which tasks within the group run.

However, note that by using this approach we have also greatly reduced the amount of parallelism available — there is at most one task associated with a group at any time, so if we had more workers than task groups some of the workers would be idle. Still, it is sometimes useful to do this; an example might be a search for "small" points on a curve where the nature of the two-dimensional search can make this better than an incremental approach.

5.13.9 Monitoring and Respawning Workers

The result of a group could become known while there are still workers processing (no longer needed) tasks in that group. When that happens, it would clearly be desirable if the manager could tell those workers to stop those tasks and work on new ones instead. Unfortunately, this is not straightforward; the worker might be deep in the throes of a long computation that cannot be interrupted, or at least not without causing the worker process to exit.

Instead, a less sophisticated approach is used: The manager will close its communication port with such workers, and when they finally try to report a result they will fail and so exit. Whatever spawned the worker originally can then detect this abnormal exit and spawn a new worker to replace it.

A variation of the `DistributedWorker` intrinsic is provided for these situations:

`MonitoredDistributedWorker(host, port, work_function): MonStgElt, RngIntElt, UserProgram`

This procedure behaves the same as `DistributedWorker`, except that it expects that it may be unexpectedly terminated and uses exit code statuses to communicate with the calling environment. Additionally, on non-Windows platforms, a monitored worker will notice when the communication channel closes and exit immediately, instead of having to finish whatever computation it is working on.

The special exit status of 5 is used to indicate that a worker should *not* be respawned. The recommended procedure for spawning monitored workers, then, is to have a loop that creates a worker and waits for it to exit. The loop terminates if the exit status is 5, otherwise it continues. Using a standard UNIX shell, the following code would do:

```
while :
do
    magma collatz_worker.m
    [ $? -eq 5 ] && break
done
```

Example: Stopping at a Large Enough Answer (ex-82a4ef)

In this example we shall not require the best answer (the largest possible value encountered in the range), but simply one that is "large enough". More precisely, if the upper limit of our range is $100n$ then we shall be satisfied with any answer that is at least $6000n$. If there is no such value, we will return false.

```
> group_tasks_hundred := function(n)
>     return < 0, 0 >, [ 100*n - 99 .. 100*n ];
> end function;
>
> update_group_maxval := function(item, task, tresult, gresult)
>     n, niters, maxval := Explode(tresult);
>     if maxval ge 6000*item then
>         return < maxval, n >, [], true;
>     else
>         return gresult, [], false;
>     end if;
> end function;
>
> incomplete_group_maxval := function(item, gresult)
>     return false;
> end function;
>
> socket := Socket(: LocalHost := "circle", LocalPort := 10000);
> DistributedManager(socket, [1..10] :
>     initial_results := [* *],
>     group_tasks := group_tasks_hundred,
>     update_group := update_group_maxval,
>     incomplete_group := incomplete_group_maxval
> );
[* <9232, 27>, false, false, false, <39364, 447>, <39364, 511>,
false, <250504, 703>, <190996, 871>, <250504, 937> *]
> delete socket;        // ensure that workers terminate
```

Note the use of the incomplete group handling here; in particular, how using it enables the group updating code to be quite simple. We could also have used this approach in the original advanced example and thereby avoided the need to track counts, but for expository purposes chose not to.

Also note the explicit deletion of the socket at the end — this ensures that the workers will terminate promptly. It would be possible to keep the socket around and re-use it for multiple runs instead, as long as the workers were doing the same kind of work.

5.13.10 Implementing Distributed Parallelism

We show a stand-alone implementation of the manager/worker model for a particular workload. This will demonstrate the usage of both the asynchronous I/O and object transmission facilities provided by Magma. For more complete details about these facilities, see Chapter *Input and Output*, particularly the section on asynchronous I/O (Section *Asynchronous I/O*). Of course, the `DistributedManager` and `DistributedWorker` intrinsics are more convenient to use (and more powerful) than the example shown here. However, this example will serve to illustrate the relevant features so that users can implement alternative models of distributed parallelism as needed.

The foundational concepts and the associated intrinsics are:

1. Protocol version negotiation: `ExchangeVersions`.
2. Object transmission: `ReadObject`, `WriteObject`, and similar. These are *synchronous* operations: The intrinsics return only after the corresponding I/O operation has completed.
3. Asynchronous I/O: `AsyncReadObject`, `AsyncWriteObject`, and similar. These operations are *asynchronous*: The intrinsics return immediately, while the corresponding I/O operation may occur later. NB: For those I/O operations to proceed, the associated channel must be waited for using `WaitForIO`.
4. Management of multiple I/O streams: `WaitForIO`.

`ExchangeVersions(I): IO`

This routine exchanges information about the version of the object transmission protocol understood by each side of the channel. This allows newer versions of Magma to still communicate correctly with older versions, using a commonly understood format. Each side of the communication channel should call this before performing any I/O on the channel.

`ReadObject(I): IO -> Any`

Reads Magma Object Format data from I and returns the corresponding Magma object.

`WriteObject(I, x): IO, Any`

Writes the Magma object x to I, using the Magma Object Format.

`AsyncReadObject(I): IO`

Queues an asynchronous read request of a Magma object to I.

`AsyncWriteObject(I, x): IO, Any`

Queues an asynchronous write request of x to I.

`WaitForIO(R : parameters): SeqEnum -> SeqEnum`

```
TimeLimit: RngIntElt                    Default: Infinity()
```

Given a sequence of I/O objects R, returns the sequence of those elements of R which are ready to have read operations performed upon them. If no elements of R are ready then this intrinsic will wait until at least one does become ready, or until the specified time limit has elapsed, whichever comes first. The time limit is measured in milliseconds.

Server sockets will be considered ready for reading when a connection attempt has been made (and so a call to `WaitForConnection` will return without delay). Other channels will be considered ready for reading if either a pending (asynchronous) read request has been fulfilled, or there are no pending read requests and some data is available for reading.

`WaitForIO(R, W : parameters): SeqEnum, SeqEnum -> SeqEnum, SeqEnum`

```
TimeLimit: RngIntElt                    Default: Infinity()
```

Given sequences of I/O objects R and W, returns two sequences: the sequence of those elements of R which are ready to have read operations performed upon them, and the sequence of those elements of W which are ready to have write operations performed upon them. If no elements of R or W are ready then this intrinsic will wait until at least one does become ready, or until the specified time limit has elapsed, whichever comes first. The time limit is measured in milliseconds.

The condition for channels to be considered ready for reading are as described in the previous intrinsic. A channel will be considered ready for writing if all pending (asynchronous) write requests have been fulfilled and it would be possible to write some data without delay.

Example: Manager and Worker for the Collatz Map `(ex-aa2ecb)`

As in earlier sections, the workload will be based on computing information about the Collatz map. Given an integer n, we iterate the Collatz map until we reach 1; the number of iterations taken

and the maximum intermediate value will be returned to the manager. For the sake of simplicity we assume that 1 will always be reached.

The worker side of the computation involves a work function to actually do the work and a simple wrapper around it that interacts with the manager. The work function will take whatever object the manager sends, interpret it, do the computation, and return a single object as the result. Since we have multiple values to return, we combine them into a single tuple. A small random time delay is inserted in each computation for the purposes of this example so that the effects of multiple workers can be seen; real workloads would not include such a delay.

```
> collatz := function(n)
>    orign := n;
>    niters := 0;
>    maxval := n;
>    while n gt 1 do
>       n := IsEven(n) select n div 2 else 3*n + 1;
>       niters +:= 1;
>       maxval := Max(maxval, n);
>    end while;
>    // delay by 100, 200, or 300 milliseconds:
>    _ := WaitForIO([] : TimeLimit := 100*Random(1,3));
>    return <orign, niters, maxval>;
> end function;
```

The wrapper for the worker takes a host and port to contact the manager on, and is a simple work-processing loop thereafter. It uses synchronous I/O calls throughout, as the only I/O it could be waiting for involves the server.

```
> worker := procedure(host, port)
>    S := Socket(host, port);
>    ExchangeVersions(S);
>    while true do
>       ok, data := ReadObjectCheck(S);
>       if not ok then break; end if;
>       result := collatz(data);
>       WriteObject(S, result);
>    end while;
> end procedure;
```

First we create the communication channel and then immediately exchange versions with the server. This ensures that the program will work even if the manager and workers are using different versions of Magma. Then we loop on the work, despatching it and returning the results; using object I/O avoids any concerns about incomplete I/O operations. We use `ReadObjectCheck` in order to gracefully exit on EOF, which we will receive when the server has no more work to allocate.

Now for the manager side of things. The manager will use asynchronous I/O calls in order to remain responsive. For the purposes of this example we will print output about the workers, so we start with a simple function to produce an identifier for each communication channel.

```
> channelid := func<I | t where _,t := SocketInformation(I)>;
```

The manager takes a sequence of work items as its argument (in this example, the work items are simply integers). It would also be reasonable for it to take the host and port information that it should listen on (since the workers will need to know this), but we instead let the system select those values and print them out.

```
> manager := procedure(workleft)
>    server := Socket();
>    host, port := Explode(SocketInformation(server));
```

(continues on next page)

(continued from previous page)

```
>    printf "Manager listening on port %o of %o\n", port, host;
>    busy := {};
>    while #workleft gt 0 or #busy gt 0 do
>       ready := WaitForIO([server] cat Setseq(busy));
>       idle := [ ];         // workers awaiting work
>       for I in ready do
>          if I eq server then    // new worker has connected
>             W := WaitForConnection(I);
>             printf "New worker %o joined\n", channelid(W);
>             AsyncExchangeVersions(W);
>             Append(~idle, W);
>          else
>             result := ReadObject(I);
>             n, count, maxv := Explode(result);
>             printf "%o: %o reached 1 (max %o) after %o iterations\n",
>                 channelid(I), n, maxv, count;
>             Exclude(~busy, I);
>             Append(~idle, I);
>          end if;
>       end for;
>       for W in idle do
>          if #workleft eq 0 then break; end if;
>          data := workleft[1];
>          Remove(~workleft, 1);
>          AsyncWriteObject(W, data);
>          AsyncReadObject(W);
>          Include(~busy, W);
>       end for;
>    end while;
> end procedure;
```

First the manager creates the server socket for workers to connect to; it prints out the associated information so that we know what arguments to give when we start the workers.

During the computation, the set `busy` will keep track of those channels associated with workers that currently have work assigned to them, and the sequence `idle` will track those workers which are waiting to have work assigned to them. The manager is done only when all work has been assigned, *and* all assigned work has been completed.

We wait for new workers to connect (on the server socket) and for completed work to be reported by the busy workers. A subtle point in the above is that we used the version of `WaitForIO` that only waits for read events. This is the typical case — the manager queues both the write and the read, and the worker will only send results after having read the data. Thus only the completion of the read event is of interest to the manager.

After a new worker joins then version exchange is queued immediately, since we will be sending and receiving objects. Note the use of the asynchronous `AsyncExchangeVersions` for this, in order to keep the manager responsive. The worker is then added to the idle list.

When a worker has completed its task and the result is ready, we perform the synchronous `ReadObject` to get this result. This is safe (that is, will not delay arbitrarily) since the manager earlier issued the corresponding asynchronous request `AsyncReadObject` and the channel becoming ready for reading indicates that this request has completed. We then mark the worker as idle (and no longer busy).

Finally, for each idle worker we attempt to (asynchronously) assign remaining work to it. We queue both the write and the following read, and mark the worker as busy.

We now demonstrate the code in action with two workers. First we start the manager in order to get the channel information for the workers to use.

```
> // manager
> manager([1..10]);
Manager listening on port 46119 of 0.0.0.0
```

Then we launch the workers (in this case, on the same machine).

```
> // worker 1
> worker("localhost", 46119);
```

```
> // worker 2
> worker("localhost", 46119);
```

And the manager displays the progress of the computation for us. Note that without the introduced delay the first worker would probably have performed all the computation before we could launch the second one.

```
New worker <"127.0.0.1", 43352> joined
<"127.0.0.1", 43352>: 1 reached 1 (max 1) after 0 iterations
New worker <"127.0.0.1", 43354> joined
<"127.0.0.1", 43354>: 3 reached 1 (max 16) after 7 iterations
<"127.0.0.1", 43352>: 2 reached 1 (max 2) after 1 iterations
<"127.0.0.1", 43352>: 5 reached 1 (max 16) after 5 iterations
<"127.0.0.1", 43354>: 4 reached 1 (max 4) after 2 iterations
<"127.0.0.1", 43354>: 7 reached 1 (max 52) after 16 iterations
<"127.0.0.1", 43352>: 6 reached 1 (max 16) after 8 iterations
<"127.0.0.1", 43354>: 8 reached 1 (max 8) after 3 iterations
<"127.0.0.1", 43352>: 9 reached 1 (max 52) after 19 iterations
<"127.0.0.1", 43354>: 10 reached 1 (max 16) after 6 iterations
```

5.13.11 A Note on Security

A manager cannot tell which user created the worker that is connecting to it. (Indeed, the very concept is not well-defined if the worker is running on another machine.) Thus it is possible for someone on another machine on the same network to connect to "your" manager, whether by accident or otherwise, and this cannot be directly distinguished from an intended connection from a worker under your control. Problems may then arise if this worker does not interact in the expected way with the manager.

If you are reasonably sure that no-one else with access to your machine or its network will act in such a way, then you can simply use the manager/worker functions as described above. If your machine is accessible to a wider network, however, then some kind of authentication process is probably desirable. A simple method is to require that client connections first send some data known only to the server, and have the server reject connections which do not do so.

Unfortunately, such behaviour is not currently integrated with the manager/worker routines described here, and so they cannot be used when authentication is a requirement. The code in Section *Implementing Distributed Parallelism* can be used as a base for re-implementing manager/worker functionality with authentication added.

A more robust and convenient version of authentication is being worked upon, and will appear in a later patch release.

5.14 Timing

When running statements within Magma in serial mode only, using the CPU time to measure the time taken between one or more statements is reasonable and accurate. In particular, one can: (1) use the `time` statement; (2) call the function `Cputime()` which returns the total CPU time (in seconds) taken by Magma since startup; (3) call the function `Cputime(T)` to give the CPU time (in seconds) taken since the variable `T` was last set to `Cputime()`.

However, when parallelism is used with multiple threads and/or distributed computation, the CPU time is not useful as a measure of the time taken, since this the value which the kernel yields for this typically includes the sum of the CPU time taken by all threads together with extra overhead in the kernel handling, and this number can be skewed even more if hyperthreading is present. So the real (wall-clock) time taken is usually not related to the CPU time divided by the total number of threads in any useful or accurate way. Also, in the distributed case, the CPU time can only refer to threads running on the master node, and not on worker nodes.

Consequently, the real (wall-clock) time taken by one or more statements in Magma is the best way to measure the time taken when parallelism is present. One way of obtaining this information is to call the function `Realtime(T)` to give the real time taken (in seconds) since `T` was last set to `Realtime()`. But it is easier just to use the `time` statement or the `Time` function, since these give useful timing information no matter whether serial or parallelism is present or not in any computation. These can be used as follows:

(i)
If `time` is placed before a statement and a multi-threaded/distributed algorithm is **not** used within the executed statement, then the CPU time alone is printed.

(ii)
If `time` is placed before a statement and a multi-threaded/distributed algorithm **is** used within the executed statement while the real time r taken is less than the CPU time c taken, then the time printed is r, and the tag `[r]` is appended (to indicate that the time printed is a real time); otherwise the CPU time c is printed. So one can know whether a multi-threaded/distributed algorithm has been used within the executed statement: that is the case if and only if the tag `[r]` is printed.

(iii)
The verbose time statement `vtime` has the same behaviour as `time` in the above two cases.

(iiv)
The function `Time` is used in a similar way to `Cputime` and `Realtime`, while selecting their behaviour depending on whether parallelism is used: after setting a variable such as T to `Time()`, then after a series of statements, one can print the value `Time(T)` (which is always a string) to give the lapsed time since T was assigned. Just as for the `time` statement above, the tag `[r]` is added if and only if the real time is used in the second call to `Time`. So the presence of that tag indicates that at least one multi-threaded/distributed algorithm has been used between the calls to `Time`, while the absence of the tag indicates that no parallelism was used and the CPU time is returned by the `Time` function.

Example: Serial and Parallel Timing (ex-0cea74)

A simple example of the use of `time` and `Time` is the following.

```
> SetNthreads(1); // SERIAL (NO PARALLELISM)
> X := Random(MatrixRing(GF(5), 10000));
> time P := X*X; // serial only, so prints CPU time (seconds)
Time: 4.060
> T := Time(); P := X*X; Time(T); // same as what time statement prints
4.060
```

```
> SetNthreads(8); // PARALLEL (8 POSIX THREADS)
> time P := X*X;  // parallel, so real time shown by [r] (seconds)
Time: 1.050[r]
> T := Time(); P := X*X; Time(T); // same as what time statement prints
1.050[r]
```

5.15 Bibliography

- [MS17] Rusydi H. Makarim and Marc Stevens. M4GB: An Efficient Gröbner-Basis Algorithm. In Michael A. Burr, Chee K. Yap, and Mohab Safey El Din, editors, *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*, 293–300. ACM, 2017. URL: https://doi.org/10.1145/3087604.3087638, doi:10.1145/3087604.3087638.
- [Ste15] Allan K. Steel. Direct solution of the (11, 9, 8)-MinRank problem by the block Wiedemann algorithm in Magma with a Tesla GPU. In Jean-Guillaume Dumas, Erich L. Kaltofen, and Clément Pernet, editors, *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015, Bath, United Kingdom, July 10-12, 2015*, 2–6. ACM, 2015. URL: https://doi.org/10.1145/2790282.2791392, doi:10.1145/2790282.2791392.

Chapter 6

Magma Semantics

6.1 Introduction

This chapter describes the semantics of Magma (how expressions are evaluated, how identifiers are treated, etc.) in a fairly informal way. Although some technical language is used (particularly in the opening few sections) the chapter should be easy and essential reading for the non-specialist. The chapter is descriptive in nature, describing how Magma works, with little attempt to justify why it works the way it does. As the chapter proceeds, it becomes more and more precise, so while early sections may gloss over or omit things for the sake of simplicity and learnability, full explanations are provided later. It is assumed that the reader is familiar with basic notions like a function, an operator, an identifier, a type and so on. And now for some buzzwords: Magma is an imperative, call by value, statically scoped, dynamically typed programming language, with an essentially functional subset. The remainder of the chapter explains what these terms mean, and why a user might want to know about such things.

6.2 Terminology

Some terminology will be useful. It is perhaps best to read this section only briefly, and to refer back to it when necessary. The term *expression* will be used to refer to a textual entity. The term *value* will be used to refer to a run-time value denoted by an expression. To understand the difference between an expression and a value consider the expressions `1+2` and `3`. The expressions are textually different but they denote the same value, namely the integer 3. A *function expression* is any expression of the form `function ... end function` or of the form `func< ... | ... >`. The former type of function expression will be said to be *in the statement form*, the latter *in the expression form*. A *function value* is the run-time value denoted by a function expression. As with integers, two function expressions can be textually different while denoting the same (i.e., extensionally equal) function value. To clearly distinguish function values from function expressions, the notation FUNC(… : …) will be used to describe function *values*. The *formal arguments* of a function in the statement form are the identifiers that appear between the brackets just after the `function` keyword, while for a function in the expression form they are the identifiers that appear before the `|`. The *arguments* to a function are the expressions between the brackets when a function is applied. The *body* of a function in the statement form is the statements after the formal arguments. The body of a function in the expression form is the expression after the `|` symbol. An identifier is said to occur *inside* a function expression when it is occurs textually anywhere in the body of a function.

6.3 Assignment

An assignment is an association of an identifier to a *value*. The statement,

```
> a := 6;
```

establishes an association between the identifier a and the value 6 (6 is said to be *the value of* a, or to

be *assigned to* a). A collection of such assignments is called a *context*. When a value V is assigned to an identifier I one of two things happens:

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

(2) if I has been previously assigned to, the value associated with I is changed to V. I is said to be *re-assigned*.

The ability to assign and re-assign to identifiers is why Magma is called an *imperative* language. One very important point about assignment is illustrated by the following example. Say we type,

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

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

```
> a := 0;
```

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

6.4 Uninitialized Identifiers

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

```
> a;
> b := a;
```

Magma can determine that execution of either line will cause an error since a has no assigned value. The user should be aware that the checks made for semantic well-formedness are necessarily not exhaustive! There is one important rule concerning uninitialized identifiers and assignment. Consider the line,

```
> a := a;
```

Now if a had been previously declared then this is re-assignment of a. If not then it is an error since a on the right hand side of the `:=` has no value. To catch this kind of error Magma checks the expression on the right hand side of the `:=` for semantic well formedness *before* it declares the identifiers on the left hand side of the `:=`. Put another way the identifiers on the left hand side are not considered to be declared in the right hand side, *unless* they were declared previously.

6.5 Evaluation in Magma

Evaluation is the process of computing (or constructing) a value from an expression. For example the value 3 can be computed from the expression `1+2`. Computing a value from an expression is also known as *evaluating an expression*. There are two aspects to evaluation, namely *when* and *how* it is performed. This section discusses these two aspects.

6.5.1 Call by Value Evaluation

Magma employs call by value evaluation. This means that the arguments to a function are evaluated before the function is applied to those arguments. Assume f is a function value. Say we type,

```
> r := f( 6+7, true or false );
```

Magma evaluates the two arguments to 13 and true respectively, *before* applying f. While knowing the exact point at which arguments are evaluated is not usually very important, there are cases where such knowledge is crucial. Say we type,

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

and we apply f as follows

```
> r := f( 4/0, false );
```

Magma treats this as an error since the $4/0$ is evaluated, and an error produced, *before* the function f is applied. By contrast some languages evaluate the arguments to a function only if those arguments are encountered when executing the function. This evaluation process is known as call by name evaluation. In the above example r would be set to the value 1 and the expression $4/0$ would never be evaluated because b is `false` and hence the argument n would never be encountered. Operators like $+$ and $*$ are treated as infix functions. So

```
> r := 6+7;
```

is treated as the function application,

```
> r := '+'(6,7);
```

Accordingly all arguments to an operator are evaluated before the operator is applied. There are three operators, 'select', 'and', and 'or' that are exceptions to this rule and are thus not treated as infix functions. These operators use call by name evaluation and only evaluate arguments as need be. For example if we type,

```
> false and (4/0 eq 6);
```

Magma will reply with the answer false since Magma knows that `false and X` for all X is false.

6.5.2 Magma's Evaluation Process

Let us examine more closely how Magma evaluates an expression as it will help later in understanding more complex examples, specifically those using functions and maps. To evaluate an expression Magma proceeds by a process of identifier substitution, followed by simplification to a canonical form. Specifically expression evaluation proceeds as follows,

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

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

The key point here is that the replacement step takes an expression and yields an unsimplified *value*! A small technical note: to avoid the problem of having objects that are part expressions, part values, all substitutions in step 1 are assumed to be done simultaneously for all identifiers in the expression. The examples in this chapter will however show the substitutions being done in sequence and will therefore be somewhat vague about what exactly these hybrid objects are! To clarify this process assume that we type,

```
> a := 6;
> b := 7;
```

producing the context `[ (a,6), (b,7) ]`. Now say we type,

```
> c := a+b;
```

This produces the context `[ (a,6), (b,7), (c,13) ]`. By following the process outlined above we can see how this context is calculated. The steps are,

(1) replace `a` in the expression `a+b` by its value in the current context giving `6+b`.

(2) replace `b` in `6+b` by its value in the current context giving `6+7`.

(3) simplify `6+7` to `13`

The result value of `13` is then assigned to `c` giving the previously stated context.

6.5.3 Function Expressions

Magma's evaluation process might appear to be an overly formal way of stating the obvious about calculating expression values. This formality is useful, however when it comes to function (and map) expressions. Functions in Magma are first class values, meaning that Magma treats function values just like it treats any other type of value (e.g., integer values). A function value may be passed as an argument to another function, may be returned as the result of a function, and may be assigned to an identifier in the same way that any other type of value is. Most importantly however function expressions are evaluated *exactly* as are all other expressions. The fact that Magma treats functions as first class values is why Magma is said to have an essentially functional subset. Take the preceding example. It was,

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

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

```
> d := func< n | a+b+c+n >;
```

Magma uses the same process to evaluate the function expression `func< n | a+b+c+n >` on the right hand side of the assignment to `d` as it does to evaluate expression `a+b` on the right hand side of the assignment to `c`. So evaluation of this function expression proceeds as follows.

(1) Replace `a` in the expression `func< n | a+b+c+n >` by its value in the current context giving `func< n | 6+b+c+n >`.

(2) Replace `b` in `func< n | 6+b+c+n >` by its value in the current context giving `func< n | 6+7+c+n >`.

(3) Replace *c* in `func< n | 6+7+c+n >` by its value in the current context giving `FUNC(n : 6+7+13+n)`

(4) Simplify the resultant *value* `FUNC(n : 6+7+13+n)` to the *value* `FUNC(n : 26+n)`.

Note again that the process starts with an expression and ends with a value, and that throughout the function expression is evaluated just like any other expression. A small technical point: function simplification may not in fact occur but the user is guaranteed that the simplification process will at least produce a function extensionally equal to the function in its canonical form. The resultant function value is now assigned to *d* just like any other type of value would be assigned to an identifier yielding the context `[ (a,6),(b,7), (c,8), (d,FUNC(n : 26+n)) ]`. As a final point note that changing the value of any of *a*, *b*, and *c*, does *not* change the value of *d*!

6.5.4 Function Values Assigned to Identifiers

Say we type the following,

```
> a := 1;
> b := func< n | a >;
> c := func< n | b(6) >;
```

The first line leaves a context of the form `[ (a,1) ]`. The second line leaves a context of the form `[ (a,1), (b,FUNC(n : 1)) ]`. The third line is evaluated as follows.

(1) Replace the value of `b` in the expression `func< n | b(6) >` by its value in the current context giving `FUNC(n : (FUNC(n : 1))(6))`.

(2) Simplify this value to `FUNC(n : 1)` since applying the function value `FUNC(n : 1)` to the argument 6 always yields 1.

The key point here is that identifiers whose assigned value is a function value (in this case b), are treated exactly like identifiers whose assigned value is any other type of value. Now look back at the example at the end of the previous section. One step in the series of replacements was not mentioned. Remember that `+` is treated as a shorthand for an infix function. So `a+b` is equivalent to `'+'(a,b)`. `+` is an identifier (assigned a function value), and so in the replacement part of the evaluation process there should have been an extra step, namely,

(4) Replace `+` in `func< n : 6+7+13+n >` by its value in the current context giving `FUNC(n : A( A( A(6,7), 13 ), n ))`.

(5) Simplify the resultant value to `FUNC(n : A( 26, n ))`. where `A` is the (function) value that is the addition function.

6.5.5 Recursion and Mutual Recursion

How do we write recursive functions? Function expressions have no names so how can a function expression apply *itself* to do recursion? It is tempting to say that the function expression could recurse by using the identifier that the corresponding function value is to be assigned to. But the function value may not be being assigned at all: it may simply be being passed as an actual argument to some other function value. Moreover even if the function value were being assigned to an identifier the function expression cannot use that identifier because the assignment rules say that the identifiers on the left hand side of the `:=` in an assignment statement are not considered declared on the right hand side, unless they were previously declared. The solution to the problem is to use the `$$` pseudo-identifier. `$$` is a placeholder for the function value denoted by the function expression inside which the `$$` occurs. An example serves to illustrate the use of `$$`. A recursive factorial function can be defined as follows,

```
> factorial := function(n)
>     if n eq 1 then
>         return 1;
>     else
>         return n * $$(n-1);
>     end if;
> end function;
```

Here `$$` is a placeholder for the function value that the function expression `function(n) if n eq ... end function` denotes (those worried that the denoted function value appears to be defined in terms of itself are referred to the fixed point semantics of recursive functions in any standard text on denotational semantics). A similar problem arises with mutual recursion where a function value f applies another function value g, and g likewise applies f. For example,

```
> f := function(...) ... a := g(...); ... end function;
> g := function(...) ... b := f(...); ... end function;
```

Again Magma's evaluation process appears to make this impossible, since to construct f Magma requires a value for g, but to construct g Magma requires a value for f. Again there is a solution. An identifier can be declared 'forward' to inform Magma that a function expression for the forward identifier will be supplied later. The functions f and g above can therefore be declared as follows,

```
> forward f, g;
> f := function(...) ... a := g(...); ... end function;
> g := function(...) ... b := f(...); ... end function;
```

(strictly speaking it is only necessary to declare g forward as the value of f will be known by the time the function expression `function(...) ... b := f(...); ... end function` is evaluated).

6.5.6 Function Application

It was previously stated that Magma employs call by value evaluation, meaning that the arguments to a function are evaluated before the function is applied. This subsection discusses how functions are applied once their arguments have been evaluated. Say we type,

```
> f := func< a, b | a+b >;
```

producing the context `[ (f,FUNC(a,b : a+b)) ]`. Now say we apply f by typing,

```
> r := f( 1+2, 6+7 ).
```

How is the value to be assigned to r calculated? If we follow the evaluation process we will reach the final step which will say something like,

```
    "simplify (FUNC(a, b : A(a,b)))(3,13) to its canonical form"
```

where as before A is the value that is the addition function. How is this simplification performed? How are function values applied to actual function arguments to yield result values? Not unsurprisingly the answer is via a process of substitution. The evaluation of a function application proceeds as follows,

(1) replace each formal argument in the function body by the corresponding actual argument.

(2) simplify the function body to its canonical form.

Exactly what it means to "simplify the function body ..." is intentionally left vague as the key point here is the process of replacing formal arguments by values in the body of the function.

6.5.7 The Initial Context

The only thing that remains to consider with the evaluation semantics, is how to get the ball rolling. Where do the initial values for things like the addition function come from? The answer is that when Magma starts up it does so with an initial context defined. This initial context has assignments of all the built-in Magma function values to the appropriate identifiers. The initial context contains for example the assignment of the addition function to the identifier `+`, the multiplication function to the identifier `*`, etc. If, for example, we start Magma and immediately type,

```
> 1+2;
```

then in evaluating the expression `1+2` Magma will replace `+` by its value in the initial context. Users interact with this initial context by typing statements at the top level (i.e., statements not inside any function or procedure). A user can change the initial context through re-assignment or expand it through new assignments.

6.6 Scope

Say we type the following,

```
> temp := 7;
> f := function(a,b)
>    temp := a * b;
>    return temp^2;
> end function;
```

If the evaluation process is now followed verbatim, the resultant context will look like `[ (temp,7), (f,FUNC(a,b :\ 7 := a*b; return 7^2;)) ]`, which is quite clearly not what was intended!

6.6.1 Local Declarations

What is needed in the previous example is some way of declaring that an identifier, in this case `temp`, is a 'new' identifier (i.e., distinct from other identifiers with the same name) whose use is confined to the enclosing function. Magma provides such a mechanism, called a local declaration. The previous example could be written,

```
> temp := 7;
> f := function(a,b)
>    local temp;
>    temp := a * b;
>    return temp^2;
> end function;
```

The identifier `temp` inside the body of f is said to be '(declared) local' to the enclosing function. Evaluation of these two assignments would result in the context being `[ (temp, 7), (f, FUNC(a,b :\ local temp := a*b; return local temp^2;)) ]` as intended. It is very important to remember that `temp` and `local temp` are *distinct*! Hence if we now type,

```
> r := f(3,4);
```

the resultant context would be `[ (temp,7), (f,FUNC(a,b :\ local temp := a*b; return local temp^2;)), (r,144) ]`. The assignment to `local temp` inside the body of f does *not* change the value of `temp` outside the function. The effect of an assignment to a local identifier is thus localized to the enclosing function.

6.6.2 The 'first use' Rule

It can become tedious to have to declare all the local variables used in a function body. Hence Magma adopts a convention whereby an identifier can be implicitly declared according to how it is first used in a function body. The convention is that if the first use of an identifier inside a function body is on the left hand side of a `:=`, then the identifier is considered to be local, and the function body is considered to have an implicit local declaration for this identifier at its beginning. There is in fact no need therefore to declare `temp` as local in the previous example as the first use of `temp` is on the left hand side of a `:=` and hence `temp` is implicitly declared local. It is very important to note that the term 'first use' refers to the first *textual* use of an identifier. Consider the following example,

```
> temp := 7;
> f := function(a,b)
>    if false then
>       temp := a * b;
>       return temp;
>    else
>       temp;
>       return 1;
>    end if;
> end function;
```

The first *textual* use of `temp` in this function body is in the line

```
> temp := a * b;
```

Hence `temp` is considered as a local inside the function body. It is not relevant that the `if false ...` condition will never be true and so the first time `temp` will be encountered when f is applied to some arguments is in the line

```
> temp;
```

'First use' means 'first textual use', modulo the rule about examining the right hand side of a `:=` before the left!

6.6.3 Identifier Classes

It is now necessary to be more precise about the treatment of identifiers in Magma. Every identifier in a Magma program is considered to belong to one of three possible classes, these being:

(a) the class of value identifiers

(b) the class of variable identifiers

(c) the class of reference identifiers

The class an identifier belongs to indicates how the identifier is used in a program. The class of value identifiers includes all identifiers that stand as placeholders for values, namely:

(a) all loop identifiers;

(b) the $$ pseudo-identifier;

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

Because value identifiers stand as placeholders for values to be substituted during the evaluation process, they are effectively constants, and hence they cannot be assigned to. Assigning to a value identifier would be akin to writing something like `7 := 8;`! The class of variable identifiers includes all those identifiers which are declared as local, either implicitly by the first use rule, or explicitly through a local declaration. Identifiers in this class may be assigned to. The class of reference identifiers will be discussed later.

6.6.4 The Evaluation Process Revisited

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

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

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

Strictly speaking the first step of this process should read,

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

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

6.6.5 The 'single use' Rule

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

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

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

6.7 Procedure Expressions

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

```
> X := f(Y);
```

where Y is a list of arguments possibly including the current value of X. At times however using re-assignment to change the value associated with an identifier can be both un-natural and inefficient. Take the problem of computing some reduced form of a matrix. We could write a function that looked something like this,

```
reduce :=
   function( m )
      local lm;
      ...
      lm := m;
      while not reduced do
         ...
         lm := some_reduction(m);
         ...
      end while;
   ...
   end function;
```

Note that the local `lm` is necessary since we cannot assign to the function's formal argument `m` since it stands for a value (and values cannot be assigned to). Note also that the function is inefficient in its space usage since at any given point in the program there are at least two different copies of the matrix (if the function was recursive then there would be more than two copies!). Finally the function is also un-natural. It is perhaps more natural to think of writing a program that takes a given matrix and *changes* that matrix into its reduced form (i.e., the original matrix is lost). To accommodate for this style of programming, Magma includes a mechanism, the *procedure expression* with its *reference arguments*, for changing an association of an identifier and a value *in place*. Before examining procedure expressions further, it is useful to look at a simple example of a procedure expression. Say we type:

```
> a := 5; b := 6;
```

giving the context `[ (a,5), (b,6) ]`. Say we now type the following:

```
> p := procedure( x, ~y ) y := x; end procedure;
```

This gives us a context that looks like `[ (a,5), (b,6), (p, PROC(x,~y : y := x;)) ]`, using a notation analogous to the `FUNC` notation. Say we now type the following *statement*,

```
> p(a, ~b);
```

This is known as a *call of the procedure* p (strictly it should be known as a call to the *procedure value* associated with the identifier p, since like functions, procedures in Magma are first class values!). Its effect is to *change* the current context to `[ (a,5), (b,5), (p, PROC(a,~b : b := a;)) ]`. a and x are called *actual* and *formal value arguments* respectively since they are not prefixed by a $\sim$, while b and y are called *actual* and *formal reference arguments* respectively because they are prefixed by a $\sim$. This example illustrates the defining attribute of procedures, namely that rather than returning a value, a procedure changes the context in which it is called. In this case the value of b was changed by the call to p. Observe however that *only* b was changed by the call to p as *only* b in the call, and its corresponding formal argument y in the definition, are reference arguments (i.e., prefixed with a $\sim$). A procedure may therefore only change that part of the context associated with its reference arguments! All other parts of the context are left unchanged. In this case a and p were left unchanged! Note that apart from reference

arguments (and the corresponding fact that procedures do not return values), procedures are exactly like functions. In particular:

a) procedures are first class values that can be assigned to identifiers, passed as arguments, returned from functions, etc.

b) procedure expressions are evaluated in the same way that function expressions are.

c) procedure value arguments (both formal and actual) behave exactly like function arguments (both formal and actual). Thus procedure value arguments obey the standard substitution semantics.

d) procedures employ the same notion of scope as functions.

e) procedure calling behaves like function application.

f) procedures may be declared 'forward' to allow for (mutual) recursion.

g) a procedure may be assigned to an identifier in the initial context.

The remainder of this section will thus restrict itself to looking at reference arguments, the point of difference between procedures and functions.

6.8 Reference Arguments

If we look at a context it consists of a set of pairs, each pair being a name (an identifier) and a value (that is said to be assigned to that identifier). When a function is applied actual arguments are substituted for formal arguments, and the body of the function is evaluated. The process of evaluating an actual argument yields a value and any associated names are ignored. Magma's evaluation semantics treats identifiers as 'indexes' into the context – when Magma wants the value of say x it searches through the context looking for a pair whose name component is x. The corresponding value component is then used as the value of x and the name part is simply ignored thereafter. When we call a procedure with a reference argument, however, the name components of the context become important. When, for example we pass x as an actual reference argument to a formal reference argument y in some procedure, Magma remembers the name x. Then if y is changed (e.g., by assignment) in the called procedure, Magma, knowing the name x, finds the appropriate pair in the calling context and updates it by changing its corresponding value component. To see how this works take the example in the previous section. It was,

```
> a := 5; b := 6;
> p := procedure( x, ~y ) y := x; end procedure;
> p(a, ~b);
```

In the call Magma remembers the name b. Then when y is assigned to in the body of p, Magma knows that y is really b in the calling context, and hence changes b in the calling context appropriately. This example shows that an alternate way of thinking of reference arguments is as synonyms for the same part of (or pair in) the calling context.

6.9 Dynamic Typing

Magma is a dynamically typed language. In practice this means that:

(a) there is no need to declare the type of identifiers (this is especially important for identifiers assigned function values!).

(b) type violations are only checked for when the code containing the type violation is actually executed.

To make these ideas clearer consider the following two functions,

```
> f := func< a, b | a+b >;
> g := func< a, b | a+true >;
```

First note that there are no declarations of the types of any of the identifiers. Second consider the use of `+` in the definition of function f. Which addition function is meant by the `+` in `a+b`? Integer addition? Matrix addition? Group addition? … Or in other words what is the type of the identifier `+` in function

f? Is it integer addition, matrix addition, etc.? The answer to this question is that `+` here denotes all possible addition function values (`+` is said to denote a *family* of function values), and Magma will automatically chose the appropriate function value to apply when it knows the type of a and b. Say we now type,

```
> f(1,2);
```

Magma now knows that a and b in f are both integers and thus `+` in f should be taken to mean the integer addition function. Hence it will produce the desired answer of 3. Finally consider the definition of the function g. It is clear X+true for all X is a type error, so it might be expected that Magma would raise an error as soon as the definition of g is typed in. Magma does not however raise an error at this point. Rather it is only when g is applied and the line `return a + true` is actually executed that an error is raised. In general the exact point at which type checking is done is not important. Sometimes however it is. Say we had typed the following definition for g,

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

Now because the `if false` condition will never be true, the line `return a+true` will *never* be executed, and hence the type violation of adding a to true will *never* be raised! One closing point: it should be clear now that where it was previously stated that the initial context "contains assignments of all the built-in Magma function values to the appropriate identifiers", in fact the initial context contains assignments of all the built-in Magma function *families* to the appropriate identifiers.

6.10 Traps for Young Players

This section describes the two most common sources of confusion encountered when using Magma's evaluation strategy.

6.10.1 Trap 1

We boot Magma. It begins with an initial context something like `[ ..., ('+',A), ('-',S), ... ]` where A is the (function) value that is the addition function, and S is the (function) value that is the subtraction function. Now say we type,

```
> '+' := '-';
> 1 + 2;
```

Magma will respond with the answer `-1`. To see why this is so consider the effect of each line on the current context. After the first line the current context will be `[ ..., ('+',S), ('-',S), ... ]`, where S is as before. The identifier `+` has been re-assigned. Its new value is the value of the identifier '-' in the current context, and the value of '-' is the (function) value that is the subtraction function. Hence in the second line when Magma replaces the identifier `+` with its value in the current context, the value that is substituted is therefore S, the subtraction function!

6.10.2 Trap 2

Say we type,

```
> f := func< n | n + 1 >;
> g := func< m | m + f(m) >;
```

After the first line the current context is `[ (f,FUNC( n : n+1)) ]`. After the second line the current context is `[ (f,FUNC( n : n+1)), (g,FUNC(m : m + FUNC(n : n+1)(m))) ]`. If we now type,

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

Magma will respond with the answer 13. Now say we decide that our definition of f is wrong. So we now type in a new definition for f as follows,

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

If we again type,

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

Magma will again reply with the answer 13! To see why this is so consider how the current context changes. After typing in the initial definitions of f and g the current context is `[ (f, FUNC(n : n+1)), (g, FUNC(m : m + FUNC(n : n+1)(m))) ]`. After typing in the second definition of f the current context is `[ (f, FUNC(n : n+2)), (g, FUNC(m : m + FUNC(n : n+1)(m)))]`. Remember that changing the *value* of one identifier, in this case f, does *not* change the value of any other identifiers, in this case g! In order to change the value of g to reflect the new value of f, g would have to be re-assigned.

6.11 Appendix A: Precedence

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

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

6.12 Appendix B: Reserved Words

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

```
_           else        join          require
adj         end         le            requirege
and         eq          load          requirerange
assert      error       local         restore
assert2     eval        lt            return
assert3     exists      mark_sweep    save
assigned    exit        meet          sdiff
break       false       mod           select
by          for         ne            subset
case        forall      not           then
cat         forward     notadj        time
catch       fprintf     notin         to
clear       freeze      notsubset     true
cmpeq       function    or            try
cmpne       ge          print         until
continue    gt          printf        vprint
declare     if          procedure     vprintf
default     iload       quit          vtime
delete      import      random        when
diff        in          read          where
div         intrinsic   readi         while
do          is          repeat        xor
elif
```

Chapter 7

The Magma Profiler

7.1 Introduction

One of the most important aspects of the development cycle is optimization. It is often the case that during the implementation of an algorithm, a programmer makes erroneous assumptions about its run-time behavior. These errors can lead to performance which differs in surprising ways from the expected output. The unfortunate tendency of programmers to optimize code before establishing run-time bottlenecks tends to exacerbate the problem.

Experienced programmers will thus often be heard repeating the famous mantra "Premature optimization is the root of all evil", coined by Sir Charles A. R. Hoare, the inventor of the Quick sort algorithm. Instead of optimizing during the initial implementation, it is generally better to perform an analysis of the run-time behaviour of the complete program, to determine what are the actual bottlenecks. In order to assist in this task, Magma provides a *profiler*, which gives the programmer a detailed breakdown of the time spent in a program. In this chapter, we provide an overview of how to use the profiler.

7.2 Profiler Basics

The Magma profiler records timing information for each function, procedure, map, and intrinsic call made by your program. When the profiler is switched on, upon the entry and exit to each such call the current system clock time is recorded. This information is then stored in a call graph, which can be viewed in various ways.

`SetProfile(b): BoolElt`

Turns profiling on (if b is `true`) or off (if b is `false`). Profiling information is stored cumulatively, which means that in the middle of a profiling run, the profiler can be switched off during sections for which profiling information is not wanted. At startup, the profiler is off. Turning the profiler on will slow down the execution of your program slightly.

`GetProfile() -> BoolElt`

Returns whether profiling is turned on.

`ProfileReset()`

Clear out all information currently recorded by the profiler. It is generally a good idea to do this after the call graph has been obtained, so that future profiling runs in the same Magma session begin with a clean slate.

`ProfileGraph() -> GrphDir`

Get the call graph based upon the information recorded up to this point by the profiler. This function will return an error if the profiler has not yet been turned on.

The call graph is a directed graph, with the nodes representing the functions that were called during the program's execution. There is an edge in the call graph from a function x to a function

y if y was called during the execution of x. Thus, recursive calls will result in cycles in the call graph.

Each node in the graph has an associated label, which is a record with the following fields:

(i) `Name`: the name of the function

(ii) `Time`: the total time spent in the function

(iii) `Count`: the number of times the function was called

Each edge $\langle x, y \rangle$ in the graph also has an associated label, which is a record with the following fields:

(i) `Time`: the total time spent in function y when it was called from function x

(ii) `Count`: the total number of times function y was called by function x

Example: Basic Profiling `(ex-cfc120)`

We illustrate the basic use of the profiler in the following example. The code we test is a simple implementation of the Fibonacci sequence; this can be replaced by any Magma code that needs to be profiled.

```
> function fibonacci(n)
>     if n eq 1 or n eq 2 then
>       return 1;
>     else
>       return fibonacci(n - 1) + fibonacci(n - 2);
>     end if;
> end function;
>
> SetProfile(true);
> time assert fibonacci(27) eq Fibonacci(27);
Time: 10.940
> SetProfile(false);
> G := ProfileGraph();
> G;
Digraph
Vertex  Neighbours

1       2 3 6 7 ;
2       2 3 4 5 ;
3       ;
4       ;
5       ;
6       ;
7       ;

> V := Vertices(G);
> Label(V!1);
rec<recformat<Name: Strings(), Time: RealField(), Count: IntegerRing()> |
    Name := <main>,
    Time := 10.93999999999999950262,
    Count := 1
    >
> Label(V!2);
rec<recformat<Name: Strings(), Time: RealField(), Count: IntegerRing()> |
    Name := fibonacci,
    Time := 10.93999999999999950262,
```

(continues on next page)

(continued from previous page)

```
    Count := 392835
    >
> E := Edges(G);
> Label(E![1,2]);
rec<recformat<Time: RealField(), Count: IntegerRing()> |
    Time := 10.93999999999999950262,
    Count := 1
    >
```

7.3 Exploring the Call Graph

7.3.1 Internal Reports

The above example demonstrates that while the call graph contains some useful information, it does not afford a particularly usable interface. The Magma profiler contains some profile report generators which can be used to study the call graph in a more intuitive way.

The reports are all tabular, and have a similar set of columns:

(i) `Index`: The numeric identifier for the function in the vertex list of the call graph.

(ii) `Name`: The name of the function. The function name will be followed by an asterisk if a recursive call was made through it.

(iii) `Time`: The time spent in the function; depending on the report, the meaning might vary slightly.

(iv) `Count`: The number of times the function was called; depending on the report, the meaning might vary slightly.

`ProfilePrintByTotalCount(G): GrphDir`

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

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

`ProfilePrintByTotalTime(G): GrphDir`

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

Print the list of functions in the call graph, sorted in descending order by the total time spent in them. Apart from the sort order, this function's behaviour is identical to that of `ProfilePrintByTotalCount`.

`ProfilePrintChildrenByCount(G, n): GrphDir, GrphVert`

`ProfilePrintChildrenByCount(G, n): GrphDir, RngIntElt`

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

Given a vertex n in the call graph G, print the list of functions called by the function n, sorted in descending order by the number of times they were called by n. The `Time` and `Count` fields of the report give the time spent during calls by the function n and the number of times the function was called by the function n. If `Percentage` is true, then the `Time` and `Count` fields represent their

values as percentages of the total value. If `Max` is set, then the report only displays the first `Max` entries.

`ProfilePrintChildrenByTime(G, n): GrphDir, GrphVert`

`ProfilePrintChildrenByTime(G, n): GrphDir, RngIntElt`

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

Given a vertex n in the call graph G, print the list of functions in the called by the function n, sorted in descending order by the time spent during calls by the function n. Apart from the sort order, this function's behaviour is identical to that of `ProfilePrintChildrenByCount`.

Example: Call Graph Profile Reports (ex-981ae1)

Continuing with the previous example, we examine the call graph using profile reports.

```
> ProfilePrintByTotalTime(G);
Index Name                                                    Time    Count
1     <main>                                                  10.940  1
2     fibonacci                                               10.940  392835
3     eq(<RngIntElt> x, <RngIntElt> y) -> BoolElt             1.210   710646
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.630   392834
5     +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.250   196417
6     Fibonacci(<RngIntElt> n) -> RngIntElt                   0.000   1
7     SetProfile(<BoolElt> v)                                 0.000   1
> ProfilePrintChildrenByTime(G, 2);
Function: fibonacci
Function Time: 10.940
Function Count: 392835

Index Name                                                    Time    Count
2     fibonacci (*)                                           182.430 392834
3     eq(<RngIntElt> x, <RngIntElt> y) -> BoolElt             1.210   710645
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.630   392834
5     +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.250   196417
* A recursive call is made through this child
```

7.3.2 HTML Reports

While the internal reports are useful for casual inspection of a profile run, for detailed examination a text-based interface has serious limitations. Magma's profiler also supports the generation of HTML reports of the profile run. The HTML report can be loaded up in any web browser. If Javascript is enabled, then the tables in the report can be dynamically sorted by any field, by clicking on the column heading you wish to perform a sort with. Clicking the column heading multiple times will alternate between ascending and descending sorts.

`ProfileHTMLOutput(G, prefix): GrphDir, MonStgElt`

Given a call graph G, an HTML report is generated using the file prefix $prefix$. The index file of the report will be "$prefix$.html", and exactly n additional files will be generated with the given filename $prefix$, where n is the number of functions in the call graph.

7.4 Recursion and the Profiler

Recursive calls can cause some difficulty with profiler results. The profiler takes care to ensure that double-counting does not occur, but this can lead to unintuitive results, as the following example shows.

Example: Profiling a Recursive Function (ex-89657b)

In the following example, `recursive` is a recursive function which simply stays in a loop for half a second, and then recurses if not in the base case. Thus, the total running time should be approximately $(n+1)/2$ seconds, where n is the parameter to the function.

```
> procedure delay(s)
>     t := Cputime();
>     repeat
>       _ := 1+1;
>     until Cputime(t) gt s;
> end procedure;
>
> procedure recursive(n)
>     if n ne 0 then
>       recursive(n - 1);
>     end if;
>
>     delay(0.5);
> end procedure;
>
> SetProfile(true);
> recursive(1);
> SetProfile(false);
> G := ProfileGraph();
```

Printing the profile results by total time yield no surprises:

```
> ProfilePrintByTotalTime(G);
Index Name                                                    Time    Count
1     <main>                                                  1.020   1
2     recursive                                               1.020   2
5     delay                                                   1.020   2
8     Cputime(<FldReElt> T) -> FldReElt                       0.130   14880
7     +(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.020   14880
9     gt(<FldReElt> x, <FldReElt> y) -> BoolElt               0.020   14880
3     ne(<RngIntElt> x, <RngIntElt> y) -> BoolElt             0.000   2
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.000   1
6     Cputime() -> FldReElt                                   0.000   2
10    SetProfile(<BoolElt> v)                                 0.000   1
```

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

```
> ProfilePrintChildrenByTime(G, 2 : Percentage);
Function: recursive
Function Time: 1.020
Function Count: 2

Index Name                                                    Time    Count
5     delay                                                   100.00  33.33
2     recursive (*)                                           50.00   16.67
3     ne(<RngIntElt> x, <RngIntElt> y) -> BoolElt             0.00    33.33
4     -(<RngIntElt> x, <RngIntElt> y) -> RngIntElt            0.00    16.67
* A recursive call is made through this child
```

At first glance, this doesn't appear to make sense, as the sum of the time column is 150%! The reason for this behavior is because some time is "double counted": the total time for the first call to `recursive` includes the time for the recursive call, which is also counted separately. In more

detail:

```
> V := Vertices(G);
> E := Edges(G);
> Label(V!1)`Name;
<main>
> Label(V!2)`Name;
recursive
> Label(E![1,2])`Time;
1.01999999999999795718
> Label(E![2,2])`Time;
0.51000000000000000888
> Label(V!2)`Time;
1.01999999999999795718
```

As can seen in the above, the total time for `recursive` is approximately one second, as expected. The double-counting of the recursive call can be seen in the values of `Time` for the edges `[1,2]` and `[2,2]`.

Chapter 8

Debugging Magma Code

8.1 Introduction

In order to facilitate the debugging of complex pieces of Magma code, Magma includes a debugger. *This debugger is very much a prototype, and can cause Magma to crash.*

`SetDebugOnError(f): BoolElt`

If f is `true`, then upon an error Magma will break into the debugger. The usage of the debugger is described in the next section.

8.2 Using the Debugger

When use of the debugger is enabled and an error occurs, Magma will break into the command-line debugger. The syntax of the debugger is modelled on the GNU GDB debugger for C programs, and supports the following commands (acceptable abbreviations for the commands are given in parentheses):

`backtrace (bt)`

Print out the stack of function and procedure calls, from the top level to the point at which the error occurred. Each line in this trace gives a single *frame*, which consists of the function/procedure that was called, as well as all local variable definitions for that function. Each frame is numbered so that it can be referenced in other debugger commands.

`frame (f)` n

Change the current frame to the frame numbered n (the list of frames can be obtained using the `backtrace` command). The current frame is used by other debugger commands, such as `print`, to determine the context within which expressions should be evaluated. The default current frame is the top-most frame.

`list (l) [n]`

Print a source code listing for the current context (the context is set by the `frame` command). If n is specified, then the `list` command will print n lines of source code; the default value is 10.

`print (p)` *expr*

Evaluate the expression *expr* in the current context (the context is set by the `frame` command). The `print` command has semantics identical to evaluating the expression `eval "expr"` at the current point in the program.

`identifiers (id)`

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

`help (h)`

Print brief help on usage.

quit (q)
Quit the debugger and return to the Magma session.

Example: Debugging a Harmonic Sum Function (ex-12f5cb)

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

```
> function f(n)
>   if n ge 0 then
>      return 1.0 / n + f(n - 1);
>   else
>      return 1.0 / n;
>   end if;
> end function;
>
> SetDebugOnError(true);
> f(3);

f(
    n: 3
)
f(
    n: 2
)
f(
    n: 1
)
f(
    n: 0
)
>>      return 1.0 / n + f(n - 1);
                     ^
Runtime error in '/': Division by zero

debug> p n
0
debug> p 1.0 / (n + 1)
1.0000000000000000000000000000
debug> bt
#0 *f(
    n: 0
) at <main>:1
#1  f(
    n: 1
) at <main>:1
#2  f(
    n: 2
) at <main>:1
#3  f(
    n: 3
) at <main>:1
debug> f 1
debug> p n
1
debug> p 1.0 / n
1.0000000000000000000000000000
```

References

[MS17] Rusydi H. Makarim and Marc Stevens. M4GB: An Efficient Gröbner-Basis Algorithm. In Michael A. Burr, Chee K. Yap, and Mohab Safey El Din, editors, *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation, ISSAC 2017, Kaiserslautern, Germany, July 25-28, 2017*, 293–300. ACM, 2017. URL: https://doi.org/10.1145/3087604.3087638, doi:10.1145/3087604.3087638.

[Mar95] G. Marsaglia. DIEHARD: a battery of tests of randomness. http://stat.fsu.edu/pub/diehard/, 1995.

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

[Ste15] Allan K. Steel. Direct solution of the (11, 9, 8)-MinRank problem by the block Wiedemann algorithm in Magma with a Tesla GPU. In Jean-Guillaume Dumas, Erich L. Kaltofen, and Clément Pernet, editors, *Proceedings of the 2015 International Workshop on Parallel Symbolic Computation, PASCO 2015, Bath, United Kingdom, July 10-12, 2015*, 2–6. ACM, 2015. URL: https://doi.org/10.1145/2790282.2791392, doi:10.1145/2790282.2791392.

Index

Symbols

"abc" (*literal*), 50
(literal), 92
, (*literal*), 92
/ (*literal*), 93
/* */ (*literal*), 16
// (*literal*), 16
<Backspace> (*literal*), 90
<Ctrl>-A (*literal*), 90
<Ctrl>-B (*literal*), 90
<Ctrl>-C (*literal*), 1, 90
<Ctrl>-D (*literal*), 1, 90
<Ctrl>-E (*literal*), 90
<Ctrl>-F (*literal*), 90
<Ctrl>-H (*literal*), 90
<Ctrl>-I (*literal*), 90
<Ctrl>-J (*literal*), 90
<Ctrl>-K (*literal*), 90
<Ctrl>-L (*literal*), 90
<Ctrl>-M (*literal*), 90
<Ctrl>-N (*literal*), 90
<Ctrl>-P (*literal*), 90
<Ctrl>-R (*literal*), 91
<Ctrl>-U (*literal*), 91
<Ctrl>-V<char> (*literal*), 91
<Ctrl>-W (*literal*), 91
<Ctrl>-X (*literal*), 91
<Ctrl>-Y (*literal*), 91
<Ctrl>-Z (*literal*), 91
<Ctrl>-_ (*literal*), 91
<Ctrl>-\ (*literal*), 1, 91
<Ctrl>-space (*literal*), 92
<Delete> (*literal*), 90
<Meta>-B (*literal*), 92
<Meta>-F (*literal*), 92
<Meta>-b (*literal*), 92
<Meta>-f (*literal*), 92
<Return> (*literal*), 90
<Tab> (*literal*), 90
B (*operation*), 5
T (*operation*), 19
s (*operation*), 51
$ (*literal*), 92
% (*literal*), 92, 103
%! shell-command (*literal*), 80, 104
%E (*literal*), 104
%P (*literal*), 102
%P all (*literal*), 102
%Pn (*literal*), 103
%Pn$_1$ n$_2$ (*literal*), 103
%S (*literal*), 103
%Sn (*literal*), 103
%Sn$_1$ n$_2$ (*literal*), 103
%b (*literal*), 104
%c (*literal*), 104
%e (*literal*), 103
%en (*literal*), 103
%en$_1$ n$_2$ (*literal*), 103
%f pattern (*literal*), 104
%h (*literal*), 103
%n (*literal*), 103
%n$_1$ n$_2$ (*literal*), 103
%o (*literal*), 104
%p (*literal*), 102
%p all (*literal*), 102
%p all N (*literal*), 102
%pn (*literal*), 102
%pn$_1$ n$_2$ (*literal*), 102
%r (*literal*), 104
%s (*literal*), 103
%sn (*literal*), 103
%sn$_1$ n$_2$ (*literal*), 103
&* s (*literal*), 50
&cat s (*literal*), 50
_ := expression (*literal*), 2
\ (*literal*), 16
0 (*literal*), 92

A

A (*literal*), 93
a (*literal*), 93
A Simple Conditional Statement (*example*), 9
AddAttribute(C, F) (*function*), 39
Alarm(s) (*procedure*), 79
Alphabetic Shifts with External Commands (*example*), 61
Argument Checking with require (*example*), 35
assert Boolean expression (*literal*), 12
assert2 Boolean expression (*literal*), 12
assert3 Boolean expression (*literal*), 12
assigned S`` N (*operation*), 39
assigned S`fieldname (*operation*), 39
assigned x (*operation*), 2
AssignNames(~S, [s$_1$, ... s$_n$]) (*procedure*), 4
AsyncExchangeVersions(I) (*procedure*), 75
AsyncExpectVersion(I) (*procedure*), 75
AsyncNotifyVersion(I) (*procedure*), 75
AsyncRead(I : parameters) (*procedure*), 74

AsyncRead(I, n) (*procedure*), 74
AsyncReadBytes(I : parameters) (*procedure*), 74
AsyncReadBytes(I, n) (*procedure*), 75
AsyncReadObject(I) (*procedure*), 75, 138
AsyncWrite(I, s) (*procedure*), 74
AsyncWriteBytes(I, S) (*procedure*), 75
AsyncWriteObject(I, x) (*procedure*), 75, 138
AtEof(I) (*function*), 66
Attach(F) (*procedure*), 34
AttachSpec(S) (*procedure*), 37
Automatic Printing (*example*), 58
Average Number of Iterations (*example*), 133

B

B (*literal*), 92
b (*literal*), 92
BaseType(T) (*function*), 19
Basic Profiling (*example*), 158
Basic Use of Eval (*example*), 15
Basic Worker for the Collatz Map (*example*), 130
BinaryString(s) (*function*), 50
Boolean expression select $expr_1$ else $expr_2$ (*literal*), 10
Booleans() (*function*), 5
break (*literal*), 14
break identifier (*literal*), 14
Breaking Out of Nested Loops (*example*), 15
BString(s) (*function*), 50
Building Objects with Eval (*example*), 16

C

C (*literal*), 93
c range (*literal*), 93
Call Graph Profile Reports (*example*), 160
case expr : when $expr_i$: statements end case (*literal*), 10
Case Statement on a Sign (*example*), 10
case< expr | expr_left,1 : expr_right,1, ..., expr_left,n : expr_right,n, default : expr_def> (*literal*), 11
Category(x) (*function*), 19
ChangeDirectory(s) (*procedure*), 79
Checking Whether a Name Is Intrinsic (*example*), 24
ClearCompletionPreference() (*procedure*), 94
ClearPrevious() (*procedure*), 57
ClearVerbose() (*procedure*), 88
ClockCycles() (*function*), 17
Clone(X) (*function*), 43
Close Vectors in a Sum of Leech Lattices (*example*), 122
Closest Vector in a Random Lattice (*example*), 123
ClosestVector(L, w) (*function*), 123
ClosestVectors(L, w) (*function*), 122
CloseVector(L, w, A, B) (*function*), 123
CloseVector(L, w, B) (*function*), 123
CloseVectors(L, w, A, B) (*function*), 122
CloseVectors(L, w, B) (*function*), 122
CodeToString(n) (*function*), 51
Collatz Iteration with a While Loop (*example*), 13
Collatz Steps with a Repeat Loop (*example*), 13
Collatz Workers with Asynchronous I/O (*example*), 75
Collecting Results in a List (*example*), 132
Comments and Line Continuation (*example*), 16
Concatenating and Quoting Strings (*example*), 52
continue (*literal*), 14
continue identifier (*literal*), 14
Counting Lines in a File (*example*), 67
CoveringStructure(S, T) (*function*), 21
Cputime() (*function*), 17
Cputime(t) (*function*), 17
Current Date from a System Call (*example*), 80
Current Time from a Pipe (*example*), 63

D

D (*literal*), 93
d range (*literal*), 93
Debugging a Harmonic Sum Function (*example*), 164
declare attributes C: F_1,...,F_n (*literal*), 39
declare type T (*literal*), 42
declare type T: P_1, ..., P_n (*literal*), 42
declare type T[E] (*literal*), 42
declare type T[E]: P_1, ..., P_n (*literal*), 43
declare verbose F, m (*literal*), 40
Defining an Intrinsic for GCDs (*example*), 32
delete S`` N (*literal*), 40
delete S`fieldname (*literal*), 40
delete x (*operation*), 5
Detach(F) (*procedure*), 34
DetachSpec(S) (*procedure*), 37
Distributed Integer Factorisation (*example*), 120
DistributedManager(socket, items : parameters) (*procedure*), 129
DistributedWorker(host, port, work_function) (*procedure*), 129

E

E (*literal*), 92
e (*literal*), 92
E<x_1, x_2, ...x_n> := expression (*literal*), 3
E<[x]> := expression (*literal*), 3
e`Object (*attribute*), 11
e`Position (*attribute*), 11
e`Traceback (*attribute*), 11
e`Type (*attribute*), 11
ElementToSequence(s) (*function*), 50
ElementType(S) (*function*), 21

Eltseq(s) (*function*), 50
End of File Marker (*example*), 66
Enumerating Close Vectors in a Database Lattice (*example*), 122
Enumerating Vectors of Small Norm (*example*), 121
Eof() (*function*), 66
error expression, ..., expression (*literal*), 11
error if Boolean expression, expression, ..., expression (*literal*), 11
Error Objects and Try-Catch (*example*), 12
Error(x) (*function*), 11
eval expression (*literal*), 15
ExchangeVersions(I) (*procedure*), 70, 138
Exchanging Objects between Processes (*example*), 70
ExistsCoveringStructure(S, T) (*function*), 21
ExpectVersion(I) (*procedure*), 70
expression$_1$ where identifier := expression$_2$ (*literal*), 7
expression$_1$ where identifier is expression$_2$ (*literal*), 7
Extended Type Accessors (*example*), 19
ExtendedCategory(x) (*function*), 19
ExtendedType(x) (*function*), 19

F

f := func< x$_1$, ..., x$_n$: parameters | expression > (*literal*), 26
f := func< x$_1$, ..., x$_n$, ... : parameters | expression > (*literal*), 26
f := function(x$_1$, ..., x$_n$: parameters) statements end function (*literal*), 25
F<char> (*literal*), 92
f<char> (*literal*), 92
false (*literal*), 5
Finding a Single Shortest Vector (*example*), 122
Flush(I) (*procedure*), 72
for i := expr$_1$ to expr$_2$ by expr$_3$ do (*literal*), 13
for i := expr$_1$ to expr$_2$ do (*literal*), 13
for i -> x in S do statements end for (*literal*), 14
for random i -> x in S do statements end for (*literal*), 14
for random x in S do statements end for (*literal*), 14
for x in S do statements end for (*literal*), 13
Fork() (*function*), 79
forward f (*literal*), 29
fprintf file, format, expression, ..., expression (*literal*), 56
freeze (*literal*), 34
function f(x$_1$, ..., x$_n$: parameters) statements end function (*literal*), 25
Function Parameters (*example*), 27
Further Uses of printf (*example*), 56

G

GetAssertions() (*function*), 84
GetAttributes(C) (*function*), 40
GetAutoColumns() (*function*), 84
GetAutoCompact() (*function*), 84
GetBeep() (*function*), 85
Getc(I) (*function*), 67
GetColumns() (*function*), 85
GetCurrentDirectory() (*procedure*), 79, 85
GetEchoInput() (*procedure*), 85
GetEnv(s) (*function*), 85
GetEnvironmentValue(s) (*function*), 85
GetExitSummary() (*procedure*), 85
GetGPU() (*function*), 85, 109
GetHelpExternalBrowser() (*function*), 105
GetHelpExternalSystem() (*function*), 106
GetHelpUseExternal() (*function*), 106
GetHistorySize() (*procedure*), 85
GetIgnorePrompt() (*function*), 85
GetIgnoreSpaces() (*function*), 86
GetIndent() (*function*), 86
GetLibraries() (*function*), 86
GetLibraryRoot() (*function*), 86
GetLineEditor() (*function*), 86
GetMaximumMemoryUsage() (*function*), 79
GetMemoryLimit() (*function*), 86
GetMemoryUsage() (*function*), 79
GetMOTPMaxVersion() (*function*), 69
GetMOTPVersion() (*function*), 69
GetNGPUs() (*function*), 109
GetNthreads() (*function*), 86, 108
GetPath() (*function*), 87
Getpid() (*procedure*), 79
GetPreviousSize() (*function*), 58
GetPrintLevel() (*function*), 87
GetProfile() (*function*), 157
GetPrompt() (*function*), 87
GetRows() (*function*), 87
Gets(I) (*function*), 67
GetScriptArguments() (*function*), 87
GetScriptFilename() (*function*), 87
GetSeed() (*function*), 22, 87
GetShowRealTime() (*function*), 18
GetTempDir() (*function*), 88
GetTraceback() (*function*), 88
Getuid() (*procedure*), 79
GetVerbose(s) (*function*), 88
GetVersion() (*function*), 88
GetViMode() (*function*), 88
GetWarnIntrinsicOverride() (*function*), 88

H

H (*literal*), 92
h (*literal*), 92
HasOutputFile() (*function*), 60

I

I (*literal*), 93
i (*literal*), 93
Identifiers and Intrinsic Names (*example*), 2
if Boolean expression then statements end if (*literal*), 9
if Boolean expression then statements$_1$ else statements$_2$ end if (*literal*), 9
if Boolean expression$_1$ then statements$_1$ elif Boolean expression$_2$ then statements$_2$ else statements$_3$ end if (*literal*), 9
iload "filename" (*literal*), 78
import "filename": ident_list (*literal*), 34
Importing Constants from a Package (*example*), 34
IndentPop() (*procedure*), 59
IndentPop(C) (*procedure*), 59
IndentPush() (*procedure*), 59
IndentPush(C) (*procedure*), 59
Index(s, t) (*function*), 51
Inline Conditional Expressions (*example*), 10
IntegerToString(n) (*function*), 51
IntegerToString(n, b) (*function*), 51
Interactive User Attributes (*example*), 41
Intrinsic Lookup for Overloaded Signatures (*example*), 32
intrinsic name(arguments) [-> return-values] { comment-text } statements end intrinsic (*literal*), 31
IOType(I) (*function*), 62
ISA(T, U) (*function*), 19
IsCoercible(S, x) (*function*), 7
IsEof(S) (*function*), 66
IsIntrinsic(S) (*function*), 24
IsServerSocket(S) (*function*), 64
IsVerbose(s) (*function*), 88
IsVerbose(s, l) (*function*), 88
Iterating over the Booleans (*example*), 7

J

j (*literal*), 93

K

k (*literal*), 93
Kissing Number of a Lattice (*example*), 125

L

L (*literal*), 92
l (*literal*), 92
Lattice Minimum on Two Machines (*example*), 124
ListAttributes(C) (*procedure*), 40
ListCategories() (*procedure*), 89
ListSignatures(C) (*procedure*), 89
ListSignatures(F, C) (*procedure*), 89
ListTypes() (*procedure*), 89
ListVerbose() (*procedure*), 88
load "filename" (*literal*), 78
Load(F) (*procedure*), 78
Load(Q) (*procedure*), 78
Local Bindings with Where (*example*), 8

M

magma -? (*literal*), 82
magma --help (*literal*), 82
magma --version (*literal*), 83
magma -b (*literal*), 81
magma -c filename (*literal*), 81
magma -d (*literal*), 81
magma -E arguments (*literal*), 82
magma -e arguments (*literal*), 82
magma -h (*literal*), 82
magma -I filename arguments (*literal*), 82
magma -i filename arguments (*literal*), 82
magma -l limit (*literal*), 82
magma -n (*literal*), 82
magma -P (*literal*), 82
magma -r workspace (*literal*), 82
magma -s filename (*literal*), 82
magma -S integer (*literal*), 82
magma -t N (*literal*), 82, 108
magma -V (*literal*), 83
Magma as a Command Line Calculator (*example*), 83
Magma in a Shebang Script (*example*), 83
MAGMA_CONFIG_DIR (*literal*), 84
MAGMA_HELP_DIR (*literal*), 84
MAGMA_HIST_PLAIN (*literal*), 84
MAGMA_LIBRARIES (*literal*), 83
MAGMA_LIBRARY_ROOT (*literal*), 83
MAGMA_MEMORY_LIMIT (*literal*), 83
MAGMA_NO_HISTORY (*literal*), 84
MAGMA_PATH (*literal*), 83
MAGMA_STARTUP_FILE (*literal*), 83
MAGMA_SYSTEM_SPEC (*literal*), 84
MAGMA_TEMP_DIR (*literal*), 84
MAGMA_USER_SPEC (*literal*), 84
MakeType(S) (*function*), 21
Manager and Worker for the Collatz Map (*example*), 138
Matrix Multiplication on a GPU (*example*), 111
Matrix Multiplication on Two GPUs (*example*), 111
Minimum Weight of a Best Known Code (*example*), 127
Minimum Weight of a QR Code (*example*), 127
Modifying Nested Sequence Entries (*example*), 3
Modular Representations of a Simple Group (*example*), 112
MonitoredDistributedWorker(host, port, work_function) (*procedure*), 136
Multiple Return Values (*example*), 2
Mutation Assignment (*example*), 4

Mutual Recursion with Forward Declarations (*example*), 29

N

Naming Generators with a Sequence (*example*), 3
New(T) (*function*), 43
not x (*operation*), 6
NotifyVersion(I) (*procedure*), 69
Nresults() (*function*), 35
Number of Results Requested (*example*), 36
NumberOfResults() (*function*), 35

O

Open(name, mode) (*function*), 62
OpenTest(name, mode) (*function*), 63

P

P (*literal*), 93
p (*literal*), 93
p := proc< x_1, ..., x_n : parameters | expression > (*literal*), 28
p := proc< x_1, ..., x_n, ... : parameters | expression > (*literal*), 28
p := procedure(x_1, ..., x_n : parameters) statements end procedure (*literal*), 28
Package Specification File (*example*), 37
Package User Attributes (*example*), 41
Pipe(cmd, input) (*function*), 61
Pipe(cmds, inputs) (*function*), 61
Polynomial Multiplication (*example*), 117
POpen(cmd, mode) (*function*), 63
Position(s, t) (*function*), 51
print expression (*literal*), 55
print expression : parameters (*literal*), 55
print expression, ..., expression (*literal*), 55
printf format, expression, ..., expression (*literal*), 55
PrintFile(F, x) (*procedure*), 59
PrintFile(F, x, L) (*procedure*), 59
PrintFileMagma(F, x) (*procedure*), 59
Printing to a Sorting Pipe with fprintf (*example*), 56
Printing to a String with Sprintf (*example*), 60
procedure p(x_1, ..., x_n : parameters) statements end procedure (*literal*), 28
Procedures (*example*), 28
ProfileGraph() (*function*), 157
ProfileHTMLOutput(G, prefix) (*procedure*), 160
ProfilePrintByTotalCount(G) (*procedure*), 159
ProfilePrintByTotalTime(G) (*procedure*), 159
ProfilePrintChildrenByCount(G, n) (*procedure*), 159
ProfilePrintChildrenByTime(G, n) (*procedure*), 160
ProfileReset() (*procedure*), 157
Profiling a Recursive Function (*example*), 160
Put(I, s) (*procedure*), 67
Puts(I, s) (*procedure*), 67

Q

quit (*literal*), 1

R

R (*literal*), 93
r<char> (*literal*), 93
Random Dense Quadratic Systems (*example*), 115
Random(a, b) (*function*), 23
Random(B) (*function*), 5
Random(b) (*function*), 23
Random(S) (*function*), 22
read identifier (*literal*), 62
read identifier, prompt (*literal*), 62
Read(I : parameters) (*function*), 66
Read(I, n) (*function*), 67
Read(name) (*function*), 60
ReadBinary(name) (*function*), 60
ReadBytes(I : parameters) (*function*), 68
ReadBytes(I, n) (*function*), 68
ReadBytesCheck(I : parameters) (*function*), 68
ReadBytesCheck(I, n) (*function*), 68
ReadCheck(I : parameters) (*function*), 67
ReadCheck(I, n) (*function*), 67
readi identifier (*literal*), 62
readi identifier, prompt (*literal*), 62
Reading a Complete File (*example*), 60
Reading Output through a Pipe (*example*), 61
ReadObject(I) (*function*), 69, 138
ReadObjectCheck(I) (*function*), 69
Realtime() (*function*), 17
Realtime(t) (*function*), 17
Recursive Functions (*example*), 26
Regexp(R, S) (*function*), 54
Regular Expression Matching (*example*), 54
Renaming Group Generators (*example*), 4
repeat statements until Boolean expression (*literal*), 13
require condition: print_args (*literal*), 35
requirege v, L (*literal*), 35
requirerange v, L, U (*literal*), 35
ResetMaximumMemoryUsage() (*procedure*), 79
restore "filename" (*literal*), 78
Restoring a Random State (*example*), 23
Returning Undefined Values (*example*), 27
Rewind(I) (*procedure*), 71

S

S ! x (*operation*), 7
S (*literal*), 93
s (*literal*), 94
s * t (*operation*), 50
s *:= t (*operation*), 50

s ^ n (*operation*), 50
s cat t (*operation*), 50
s cat:= t (*operation*), 50
s eq t (*operation*), 52
s ge t (*operation*), 52
s gt t (*operation*), 52
s in t (*operation*), 52
s le t (*operation*), 52
s lt t (*operation*), 52
s ne t (*operation*), 52
s notin t (*operation*), 52
S`` N (*literal*), 39
S`` N := expression (*literal*), 40
S`fieldname (*literal*), 39
S`fieldname := expression (*literal*), 40
save "filename" (*literal*), 78
Seek(I, offset) (*procedure*), 71
Seek(I, offset, origin) (*procedure*), 71
Semantics of Equality Operators (*example*), 6
Sending a Message over a Socket (*example*), 65
Serial and Parallel Timing (*example*), 142
Serial and Threaded Lattice Minimum (*example*), 124
SetAssertions(b) (*procedure*), 84
SetAutoColumns(b) (*procedure*), 84
SetAutoCompact(b) (*procedure*), 84
SetBeep(b) (*procedure*), 85
SetColumns(n) (*procedure*), 85
SetCompletionPreference(P, O) (*procedure*), 94
SetCompletionPreferenceStyle(S) (*procedure*), 94
SetDebugOnError(f) (*procedure*), 163
SetEchoInput(b) (*procedure*), 79, 85
SetEnv(s, v) (*procedure*), 85
SetEnvironmentValue(s, v) (*procedure*), 85
SetExitSummary(b) (*procedure*), 85
SetGPU(b) (*procedure*), 85, 109
SetGPUDevice(k) (*procedure*), 109
SetGPUDevices(S) (*procedure*), 109
SetHelpExternalBrowser(S) (*procedure*), 105
SetHelpExternalBrowser(S, T) (*procedure*), 105
SetHelpExternalSystem(s) (*procedure*), 105
SetHelpUseExternalBrowser(b) (*procedure*), 105
SetHelpUseExternalSystem(b) (*procedure*), 105
SetHistorySize(n) (*procedure*), 85
SetIgnorePrompt(b) (*procedure*), 85
SetIgnoreSpaces(b) (*procedure*), 86
SetIndent(n) (*procedure*), 86
SetLibraries(s) (*procedure*), 86
SetLibraryRoot(s) (*procedure*), 86
SetLineEditor(b) (*procedure*), 86
SetLogFile(F) (*procedure*), 79, 86
SetMemoryLimit(n) (*procedure*), 86
SetMOTPVersion(I, v) (*procedure*), 69
SetMOTPVersion(v) (*procedure*), 69
SetNGPUs(n) (*procedure*), 109
SetNthreads(n) (*procedure*), 86, 108
SetOutputFile(F) (*procedure*), 60, 87
SetParallelFFT(b) (*procedure*), 117
SetPath(s) (*procedure*), 87
SetPreviousSize(n) (*procedure*), 57
SetPrintLevel(l) (*procedure*), 87
SetProfile(b) (*procedure*), 157
SetPrompt(s) (*procedure*), 87
SetQuitOnError(b) (*procedure*), 87
SetRows(n) (*procedure*), 87
SetSeed(s) (*procedure*), 22
SetSeed(s, c) (*procedure*), 22, 87
SetShowRealTime(v) (*procedure*), 18
Setting the Worker Port (*example*), 119
SetTraceback(n) (*procedure*), 88
SetVerbose("ManagerWorker", v) (*procedure*), 131
SetVerbose("ManagerWorkerTime", v) (*procedure*), 131
SetVerbose(s, b) (*procedure*), 88
SetVerbose(s, i) (*procedure*), 88
SetViMode(b) (*procedure*), 88
SetViMode(false) (*procedure*), 89
SetViMode(true) (*procedure*), 89
SetWarnIntrinsicOverride(b) (*procedure*), 88
SetWorkerPort(P) (*procedure*), 118
ShortestVector(L) (*function*), 121
ShortestVectors(L) (*function*), 121
ShortVector(L, A, B) (*function*), 121
ShortVector(L, B) (*function*), 121
ShortVectors(L, A, B) (*function*), 121
ShortVectors(L, B) (*function*), 121
ShowIdentifiers() (*procedure*), 89
ShowPrevious() (*procedure*), 57
ShowPrevious(i) (*procedure*), 57
ShowValues() (*procedure*), 89
Simple Chat Server (*example*), 73
Simple Uses of printf (*example*), 55
Socket(: parameters) (*function*), 64
Socket(H, P : parameters) (*function*), 64
SocketInformation(S) (*function*), 64
Solving MinRank Problems (*example*), 113
Split(S) (*function*), 53
Split(S, D) (*function*), 53
Splitting Strings (*example*), 54
Sprint(x) (*function*), 60
Sprint(x, L) (*function*), 60
Sprintf(F, ...) (*function*), 60
Starting Workers Automatically (*example*), 118
Startup with Files and Assignments (*example*), 83
StartWorkers(H, T) (*procedure*), 118
Stopping at a Large Enough Answer (*example*), 137
StringToCode(s) (*function*), 51
StringToInteger(s) (*function*), 51
StringToInteger(s, b) (*function*), 51
StringToIntegerSequence(s) (*function*), 51
Substring(s, n, k) (*function*), 50

System Attributes (*example*), 40
System(C) (*function*), 80
s[i] (*literal*), 50

T

T<char> (*literal*), 92
t<char> (*literal*), 92
Task Groups over Ranges of Inputs (*example*), 135
Tasks Generating Further Tasks (*example*), 135
Tell(I) (*function*), 71
Tempname(P) (*function*), 80
Theta Series by Direct Enumeration (*example*), 125
Theta Series of a Lattice of Dimension 40 (*example*), 125
Theta Series via Modular Forms (*example*), 126
Threaded Matrix Multiplication (*example*), 110
time statement (*literal*), 17
Time() (*function*), 17
Time(T) (*function*), 17
Timing a Factorisation (*example*), 18
Timing Multithreaded Computations (*example*), 18
Traceback() (*procedure*), 89
Tracking the Maximum Value (*example*), 133
true (*literal*), 5
try statements catch e statements end try (*literal*), 12
Type and Structure Functions (*example*), 21
Type(x) (*function*), 19
T[i] (*literal*), 19

U

U (*literal*), 94
u (*literal*), 94
Ungetc(I, c) (*procedure*), 67
UnsetEnv(s) (*procedure*), 85
UnsetEnvironmentValue(s) (*procedure*), 85
UnsetLogFile() (*procedure*), 79, 86
UnsetOutputFile() (*procedure*), 60, 87
User Startup Specification Files (*example*), 38
User-Defined Type for Direct Products (*example*), 46
User-Defined Type for Rationals (*example*), 45

V

Variadic Functions (*example*), 27
Variety(I) (*function*), 116
VarietySize(I) (*function*), 117
Verbose Manager Output (*example*), 131
vprint flag, n: expression, ..., expression (*literal*), 57
vprint flag: expression, ..., expression (*literal*), 57
vprintf flag, n: format, expression, ..., expression (*literal*), 57
vprintf flag: format, expression, ..., expression (*literal*), 57
vtime flag, n: statement: (*literal*), 17
vtime flag: statement (*literal*), 17

W

W (*literal*), 93
w (*literal*), 93
WaitForConnection(S) (*function*), 64
WaitForIO(R : parameters) (*function*), 72, 138
WaitForIO(R, W : parameters) (*function*), 72, 138
Weight Distribution of a QR Code (*example*), 128
while Boolean expression do statements end while (*literal*), 13
Write(F, x) (*procedure*), 59
Write(F, x, L) (*procedure*), 59
Write(I, s) (*procedure*), 67
WriteBinary(F, s) (*procedure*), 59
WriteBytes(I, S) (*procedure*), 68
WriteBytesCheck(I, S) (*function*), 68
WriteCheck(I, s) (*function*), 67
WriteObject(I, x) (*procedure*), 69, 138
WriteObjectCheck(I, x) (*function*), 69

X

X (*literal*), 94
x (*literal*), 94
x := expression (*literal*), 2
x and y (*operation*), 5
x cmpeq y (*operation*), 6
x cmpne y (*operation*), 6
x eq y (*operation*), 6
x ne y (*operation*), 6
x o:= expression (*literal*), 4
x or y (*operation*), 5
x xor y (*operation*), 6
x[$expr_1$,$expr_2$,...,$expr_n$] := expression (*literal*), 3
x[$expr_1$][$expr_2$]...[$expr_n$] := expression (*literal*), 3
x_1, x_2, ..., x_n := expression (*literal*), 2

Y

Y (*literal*), 94
y range (*literal*), 94