Part I. Magma Language

`Magma` is a programming language, which is interpretted rather than compiled, like perl, python, or various shell scripts. Some native `magma` code, written as `packages`, is compiled at startup time.

1. The magma shell.
2. Magma categories.
3. Category handles.
4. Primitive structures.
5. Aggregate structures.
6. Coercion: element creation and transmutation.
7. Creating new structures from old.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. Infrastructure of advanced algorithms.

1. The magma shell

The most typical way to run `magma` is interactively via the `magma` shell. Every statement ends in a semicolon. Output not assigned to a variable, using `:=`, is printed to the standard output. `$1`, `$2`, and `$3` refer three previous objects sent to standard output.

```
weyl:~> magma
Magma V2.8-BETA    Wed Oct  4 2000 10:40:38 on weyl
Linked at:         Fri Sep 15 2000 18:21:23
Type ? for help.   Type <Ctrl>-D to quit.

Loading startup file "/home/kohel/.magma"

> 1;
1
> 2;
2
> $1; $2;
2 1
```

2. Magma categories

Every object in `magma` has a `Parent` structure or category to which it belongs. Generally it is necessary to define the parent category before initializing an element.

```
> ZZ := Integers();
> u := One(ZZ);
> u;
1
> Parent(u);
Integer Ring
```

```
> V := RSpace(ZZ,7);
> v := Zero(V);
> v;
(0 0 0 0 0 0 0)
> Parent(v);
Full RSpace of degree 7 over Integer Ring
```

3. Primitive structures

Certain categories, such as the `Integers()`, are predefined as system-wide global structures, and do not have to be constructed in order to create elements. Other examples are the `Rationals()`, strings and booleans.

```
> n := 2^127-1;
> n;
170141183460469231731687303715884105727
> t := 2/31;
> t;
2/31
> s := "Integer Ring";
> s;
Integer Ring
> true;
true
```

4. Category handles

Every object has a `Category` or `Type` name or handle.

```
> Parent(n);
Integer Ring
> Parent(n) eq s;
false
> Parent(n) eq ZZ;
true
> Category(n);
RngIntElt
> Category(s);
MonStgElt
> Category($1);
Cat
```

The category handle can be used for comparisons (with `eq`) of possibly imcompatible objects, and for type checking, permitting function overloading.

5. Aggregate structures

A. Sequences. A sequence is an indexed list of elements all of which have the same parent, called the `Universe` of the sequence. A common pitfall is to construct empty sequences without defining the universe.

```
> [];
[]
> Universe($1);

>> Universe($1);
              ^
Runtime error in 'Universe': Illegal null sequence
> [ ZZ | ];
[]
> Universe($1);
Integer Ring
```

If the universe is not explicitly defined, then objects will be `coerced` into a common structure, if possible.

```
> S := [ 1, 2/31, 17 ];
> S;
[ 1, 2/31, 17 ]
> Universe(S);
Rational Field
> S[3];
17
> Parent($1);
Rational Field
```

The syntax for sequence construction is:

```
[ Universe | Element : Loop | Predicate ]
```

As an example, we have the following sequence:

```
> FF<w> := FiniteField(3^6);
> [ FF | x : x in FiniteField(3^2) | Norm(x) eq 1 ];
[ 1, w^182, 2, w^546 ]
```

B. Sets. A set is an unordered collection of objects having the same parent, again, defined to be its `Universe`.

```
> { FiniteField(2^8) | 1, 2, 3, 4 };
{ 1, 0 }
> Random($1);
0
```

The syntax for set construction is analogous to that for sequences:

```
{ Universe | Element : Loop | Predicate }
```

The enumeration operator `#` applies to both `sequences` and `sets`.

```
> #[ x^2 : x in FiniteField(3^3) | x ne 0 ];
26
> #{ x^2 : x in FiniteField(3^3) | x ne 0 };
13
```

C. Indexed sets. An indexed set is a collection of objects indexed by the positive integers. An element is assigned the next available index at its first occurence.

```
> S := {@ 4, 3, 7 @};
> S;
{@ 4, 3, 7 @}
> T := {@ 1, 1, 11 @};
> S join T; /* Union operator. */
{@ 4, 3, 7, 1, 11 @}
> $1[4];
1
> #$2;
5
```

D. Tuples. A `tuple` is analogous to a `sequence`, but unlike sets and sequences, the parent structure – the set-theoretic product of the parents of the entries – stores the parent of each component.

```
> <>;
<>
> Parent($1);
Cartesian Product<>
> <1,2/1>;
<1, 2>
> Parent($1);
Cartesian Product<Integer Ring, Rational Field>
```

The parent structure of a `tuple` is more important than in the case of sequences or sets.

```
> C := CartesianProduct(Integers(),RationalField());
> t := C!<1,1>;
> Parent(t[2]);
Rational Field
```

E. Vectors and matrices. Since there is a unique global free module of rank n over a ring R, the following shorthand constructors have now been provided in V2.7.

1. Vectors.

```
> Vector([2,11,7]);
( 2 11 7)
```

2. Matrices.

```
> Matrix([
>    Vector([ (i+j) mod 3 : i in [1..3] ])
>       : j in [1..3] ]);
[2 0 1]
[0 1 2]
[1 2 0]
> $1 eq Matrix(3,3,[ (i+j) mod 3 : i, j in [1..3] ]);
true
```

6. Coercion: element creation and transmutation.

The coercion operator `!` is used to construct an element of a structure, or to map it into a structure, where a nature mapping exists.

```
> QQ := RationalField();
> QQ!17;
17
> P<x> := PolynomialRing(QQ);
> P![2,-3,1];
x^2 - 3*x + 2
```

Automatic coercion of objects occurs systematically throughout the `magma` language. Consider the following examples:

```
> f := hom< QQ -> QQ | x :-> x >;
> f(2);
2
> 17 eq 17/1;
true
```

7. Creating new structures from old

The construction of objects in `magma` is recursive, we can rational function fields over the integers, create an elliptic curve over the function field, and compute the function field of the curve.

```
> F1<u> := FunctionField(ZZ);
> F2<x> := FunctionField(F1);
> E := EllipticCurve([u+1,u,u,0,0]);
> E;
Elliptic Curve defined by y^2 + (u + 1)*x*y + u*y =
x^3 + u*x^2 over Rational function field of rank 1
over Integer Ring
Variables: u
> P := E![0,0,1];
> P;
(0 : 0 : 1)
> P in E;
true
```

Now we can do arithmetic in this curve – even though no one designed function fields to be used as base fields for elliptic curves.

```
> [ k*P : k in [1..4] ];
[ (0 : 0 : 1), (-u : u^2 : 1), (-u : 0 : 1),
  (0 : -u : 1) ]
> 5*P;
(0 : 1 : 0)
> $1 eq E!0;
true
> KE<y> := FunctionField(E);
> y^2;
((-u - 1)*x - u)*y + x^3 + u*x^2
```

N.B. The above example shows that $X_1(5)$ has genus 0. The elliptic curve `E` above is a universal curve over $X_1(5)$.

Exercise. Construct a point on an elliptic curve over its own function field, and recover the division polynomials for the multiplication-by-n maps on the curve.

8. Built-in operators

We've already seen the assignment `:=` and coercion `!` operators.

Eltseq. In many instances, the coercion operator `!` can accept a defining sequence for an object. In such circumstances, `!` and `ElementToSequence` (or the shorthand `Eltseq`) are inverses.

Arithmetic operations. The standard arithmetic operators `+, -, *, /, ^` are defined for many categories. Where they exist, the standard assignment versions also exist `+:=, -:=, *:=, /:=, ^:=`. **N.B.** In non-commutative rings, like matrix algebras, or non-abelian groups or semigroups, no left multiplication assignment operator presently exists.

Integral division and remainder. The operators `mod` and `div` are defined such that `n` equals `(n div m)*m + (n mod m)` and `n mod m` is a nonegative number at less than the absolute value of `m`.

Boolean operators. The unary operator `not` and the binary operators `and` and `or` operate on the booleans `true` and `false`.

Comparison operators. The operator `eq` tests for equality of objects in `magma`, returning a boolean, and for objects which have a ordering or partial ordering, the comparison operators are `le`, `lt`, `gt`, and `ge`.

Sequence and set operators. Strings and sequences are elements of free monoids for which `cat` or `*` serve as the binary operation.

Sets. Sets admit the operators `join` and `meet`, as well as boolean operators `subset` and `in`.

Recursion operators Any of the above binary operators, say `op`, which satifies an associative law gives rise to a recursive operator `&op` which applies to `sequences`. If the operation is also commutative, then a recursion operator applies to `sets`.

```
> s := &*[ "I", "n", "t", "e", "g", "e", "r" ];
> t := &*[ "R", "i", "n", "g" ];
> s cat " " cat t;
Integer Ring
```

N.B. There are no functions `Sum` or `Product` in `magma`, because the recursion operators `&+` and `&*` fill these voids. The recursion operators `&op` can be very useful, as demonstrated by this one line implementation of the `subset` operator.

```
> X := {1..100};
> Y := { a : a in X | IsOdd(a) };
> &and[ a in X : a in Y ];
true
> Y subset X;
true
```

Membership and enumeration operators. The operator `in` is overloaded as both an membership operator and an enumeration operator, as demonstrated in the above example.

9. Language syntax

A. Language conventions. Functions in `magma` are upper case and *should* refer to the noun which they return. For example, instead of the verb `Factor`, `magma` uses the noun form:

```
> Factorization(2^(2^7)+1);
[ <59649589127497217, 1>, <5704689200685129054721, 1> ]
```

Syntax bugs. One bug in this convention is the function `Evaluate`, which should be called `Evaluation`. Another bug is the function `LLL` which should be called `LLLReduction`, since it does not return even one of the `L`'s to which it refers.

B. Loops and flow control. The most commonly used flow control routines are `if`, `for`, and `while` loops.

```
if P in S then    while P in S do   for P in S do
   ...;              ...;              ...;
end if;           end while;        end for;
```

The `if` statement also permits `elif..then` and `else` clauses. Note the two distinct `in` operators in the `for`, `if`, and `while` routines.

10. Functions and procedures

Consider the file `function_X.m` with content:

```
function X(A,B)
   A +:= B;
   return A;
end function;
```

and the file `procedure_X.m` with content:

```
procedure X(~A,B)
   A +:= B;
end procedure;
```

Back in the `magma` shell we `load` and use these functions.

```
> load "function_X.m";
Loading "function_X.m"
> A := 2; B := 7;
> X(A,B);
9
```

But notice that the global variable `A` remains unchanged by the function.

```
> A;
2
```

In contrast the variable `A` is passed by reference, with `~A`, to the procedure `X` and can be changed.

```
> load "procedure_X.m";
Loading "procedure_X.m"
> X(~A,B);
> A;
9
```

`Magma functions` and `procedures` have no type checking of arguments, and overwrite any and all functions or `intrinsics` of the same name.

11. Packages and intrinsics.

`Intrinsics` include all functions or procedures with type checking and overloading which are built into the kernel of `magma` (written and compiled in C). It is possible to view the `signature` of any such function from the `magma` shell.

```
> ModularCurveX0;
Intrinsic 'ModularCurveX0'

Signatures:

(<RngIntElt> N) -> CrvMod

The modular curve X_0(N) of level N.
```

More and more `intrinsics` are being written in the `magma` language, as part of `packages` distributed with the system. All such `magma` code is in human readable form in the various subdirectories of `$MAGMA_ROOT/package/`, where `$MAGMA_ROOT` is the root directory where `magma` is installed.

Consider the file `intrinsic_X.m` with content:

```
intrinsic X(A::RngIntElt,B::RngIntElt) -> RngIntElt
   {Returns the sum of A and B.}
   A +:= B;
   return A;
end intrinsic;

intrinsic X(~A::RngIntElt,B::RngIntElt)
   {Assigns the sum of A and B to A.}
   A +:= B;
end intrinsic;
```

The file `intrinsic_X.m` constitutes an integer addition package. Two `intrinsics` are defined, one is a function `X` and the second a procedure `X`. We use the package by means of the `Attach` command.

```
> Attach("intrinsic_X.m");
> A := 2; B := 7;
> X(A,B);
9
> X(~A,B);
> A;
9
```

In a Unix shell, we also notice that magma has created two new files, an `intrinsic_X.dat` file and an `intrinsic_X.sig` file.

```
weyl ~> ls intrinsic*
intrinsic_X.dat  intrinsic_X.m  intrinsic_X.sig
```

The former is the compiled file, and the latter is a signature, which is checked at each carriage return in the `magma` shell, to see if the file has changed and needs to be recompiled.

```
> Attach("intrinsic_X.m");
> X;
Intrinsic 'X'

Signatures:

(<RngIntElt> A, <RngIntElt> B) -> RngIntElt

Returns the sum of A and B.

(<RngIntElt> ~A, <RngIntElt> B)

Assigns the sum of A and B to A.
```

12. Infrastructure of advanced algorithms.

Here is a select list of some of the underlying high performance algorithms built into `magma`, and some of their principle authors and developers. Numerous people have contributed code and provided assistence in development during academic visits to Sydney and the `magma` group.

Group theory. (John Cannon, Bill Unger, Volker Gebhardt).
Number theory. (Claus Fieker, KANT, Nicole Sutherland).
Galois groups. (Katharina Geissler).
Elliptic curves. (Geoff Bailey).
Function fields of curves. (Florian Hess).
Groebner bases. (Allan Steel).
Lattice basis reduction. (Allan Steel).
Integer factorization, polynomial factorization (Allan Steel).
Fast multiplication. (up to Dan Bernstein to bring Allan Steel up to speed in the Cult of Fast Multiplication.)
Discrete logarithms. (Scott Contini and Paulette Lieby).
Number field sieve. (Scott Contini).

Various other high-level applications are written in the `magma` language by postdocs in the `magma` group.

Algebraic geometry. (Gavin Brown).
Mucking around. (David Kohel).

Still more are contributed by outside collaborators.

Hyperelliptic curves. (Michael Stoll).
Modular symbols. (William Stein).