

# Magma

Sebastian Pauli

Department of Mathematics and Statistics, UNCG

# Content

- Preliminaries
- Expressions
- Data Structures and Maps
- Control Structures
- Functions and Procedures
- Intrinsic and Packages
- Time and Profiling

# Magma

Magma is a computer algebra system for computations in discrete mathematics, algebra and geometry. A *Magma* is a set with a binary operation.

Goals:

- Correct and clearly defined realization of mathematical objects in a computer algebra system
- Efficient algorithms and implementations

Magma consists of a C-kernel and a library that is implemented in the Magma shell language. Magma consists of approximately 3 million lines of c-code and 360 000 lines of Magma code.

# Components

## Magma

- Interpreter/Shell

- C-Library

  - GMP, MPFR/MPC, ECM, ECPP, KANT, ...

- Datenbases

  - Groups, Lattices, Codes, Elliptic Curves,  
Modular Curves, ...

- Packages

  - many external contributions, see acknowledgements.

# Help

The Magma handbook is available online at

<http://magma.maths.usyd.edu.au/magma/handbook/>

Inline help and documentation:

- Command completion with tab key.
- Entering a command followed by `;` shows all signatures (types of parameters) and a short documentation.
- Entering `command:Maximal;` in addition shows the location of the source code for functions implemented in the Magma language.
- Entering `'+' , '/' , 'and' , '&+'` etc shows signatures and documentation.

# Example

```
> GaloisGroup:Maximal;  
Intrinsic 'GaloisGroup'
```

Signatures:

```
Defined in file: package/Ring/FldFin/Galois.m, line 6:  
(<FldFin> E, <FldFin> F) -> GrpPerm, SeqEnum
```

The Galois group of a finite extension of a finite field.

```
Defined in file: package/Ring/GaloisGrp/Galois.m, line 2176:  
(<RngUPolElt[RngOrd]> f) -> Grp, SeqEnum, GaloisData  
[  
  Ring, Prime, ShortOK, NextPrime  
]
```

Compute the Galois group of the splitting field of the monic and squarefree polynomial.

# Magma Language

## The Magma Language

- is imperative,
- is dynamically typed,
- is mathematically oriented,
- has functional components,
- is *call by value*.

# Identifiers and Expressions

Each object in Magma is made up of expressions, which in turn consist of expressions and identifiers.

There are three classes of **Identifiers**:

- Constants (true, false, 123, mod, ...),
- Variables,
- References (with prepended '~')



# Identifiers and Expressions

Each object in Magma is made up of expressions, which in turn consist of expressions and identifiers.

There are three classes of **Identifiers**:

- Constants (`true`, `false`, `123`, `mod`, ...),
- Variables,
- References (with prepended '~')

**Expressions** are composed of:

- Assignments,
- Operationens,
- Comparaisons,
- Constructors,
- Functions, Procedures and Intrinsicics

# Assigning and Deleting Values

## Example: Assignment

```
> x := 17; y := x^2+5;  
> x, y ;  
17 294  
> f := NextPrime; f(y);  
307
```

# Assigning and Deleting Values

## Example: Assignment

```
> x := 17; y := x^2+5;
> x, y ;
17 294
> f := NextPrime; f(y);
307
```

## Example: Assignment and Deletion

```
> assigned L;
false
> L := ["A": i in [1..2^20]]; #L;
1048576
> delete L;
> L;
User error: Identifier L has not been declared or assigned
```

# Assignment of Several Return Values

Some functions return more than one value. These can be assigned to different variables.

## Example

```
> XGCD(3,5);
```

```
1 2 -1
```

```
> g,c,d := XGCD(15,5); c*15+d*5;
```

```
3
```

# Assignment of Several Return Values

Some functions return more than one value. These can be assigned to different variables.

## Example

```
> XGCD(3,5);
```

```
1 2 -1
```

```
> g,c,d := XGCD(15,5); c*15+d*5;
```

```
3
```

Return values can be ignored by assigning them to ..

## Example:

```
> _,_,d := XGCD(3,5); d;
```

```
-1
```

# Assigning generators

The '.' operator is used to access generators of algebraic structures. In addition names can be assigned to these generators.

## Example

```
> Z := Integers();  
> Zx := PolynomialRing(Z); Zx;  
Univariate Polynomial Ring over Integer Ring  
> f := Zx.1^5+Zx.1^2+16; f;  
Zx.1^5+Zx.1^2+16  
> Assignnames(~Zx,["x"]); Zx;  
Univariate Polynomial Ring in x over Integer Ring  
> f, x^2+7;  
x^5+x^2+16 x^2+7
```

# Assigning generators

The '.' operator is used to access generators of algebraic structures. In addition names can be assigned to these generators.

The generators can be assigned when defining a structure.

## Example

```
> C:=ComplexField(4);  
> R<y,z>:=PolynomialRing(C,2); R;  
Polynomial ring of rank 2 over C  
Lexicographical Order  
variables:  y, z  
> (z*y^3+z+2*y+7)^2;  
y^6*z^2+2.000*y^3*z^2+14.00*y^3*z+z^2+14.00*z+49.00
```

# Operators

Magma has these infix-operators for

- Arithmetic: `*`, `+`, `-`, `/`, `^`, `div` (integral division), `mod`
- Logic: `and`, `or`, `xor`
- Lists: `cat` (concatination)
- Sets: `diff` (difference), `sdiff` (symmetric difference), `join` (union), `meet` (intersection),



# Operators

Magma has these infix-operators for

- Arithmetic: `*`, `+`, `-`, `/`, `^`, `div` (integral division), `mod`
- Logic: `and`, `or`, `xor`
- Lists: `cat` (concatination)
- Sets: `diff` (difference), `sdiff` (symmetric difference), `join` (union), `meet` (intersection),

Operators can be used in conjunction with `':='`.

## Example

```
> a := 10; a += 7; a;  
17
```

```
> B := [1,2,4]; B cat:=[89,12]; B;  
[1,2,4,89,12]
```

# Boolean Values and Comparissons

The boolean values are `true` and `false`.

- The usual boolean operations are `and`, `or`, `not` and `xor`.
- The evaluation of boolean expressions is terminates as soon as the result is known.

# Boolean Values and Comparissons

Comparisson operators return `true`, `false` or an error message:

- `eq` (equal), `ge` (greater than or equal to), `gt` (greater), `le` (less than or equal to), `lt` (less) and `ne` (not equal) result in an error message, if the values are not comparable.
- `cmpeq` (equal) returns `false`, if the values are not comparable.
- `cmpne` (not equal) returns `true`, if the values are not comparable.

# Boolean Values and Comparissons

Comparisson operators return true, false or an error message:

- eq (equal), ge (greater than or equal to), gt (greater), le (less than or equal to), lt (less) and ne (not equal) result in an error message, if the values are not comparable.
- cmpeq (equal) returns false, if the values are not comparable.
- cmpne (not equal) returns true, if the values are not comparable.

## Example

```
> "true" eq true;  
Runtime error in 'eq': Bad argument types  
Argument types given: MonStgElt, BoolElt  
  
> "true" cmpeq true;  
false
```

# Direktives, Functions and Constructors

**Directives** offer functionality that is close to the system.

**Functions, Procedures and Intrinsic** have a fixed number of arguments. They either return values or take arguments that are references. There are user defined functions, procedures and intrinsic.

**Constructors** have an arbitrary number of arguments, they yield generic algebraic constructions.

# Directives

**Directives** offer functionality that is close to the system.

Magma has the following directives:

<code>print <i>expr</i>,...</code>	Output <i>expr</i> ,....
<code>printf <i>format</i>,...</code>	Formatted output.
<code>time <i>expr</i></code>	Prints the time needed to evaluate <i>expr</i> .
<code>vprint ...</code>	Output when a verbose flag is large enough
<code>vprintf ...</code>	Formatted output when a verbose flag is large enough
<code>vtime ...</code>	Execution time when a verbose flag is large enough
<code>load "<i>name</i>"</code>	Consider the content of file <i>name</i> as input.
<code>iload "<i>name</i>"</code>	interactive load
<code>freeze</code>	Do not reload a package after changes..
<code>declare...</code>	Declarations in packages
<code>assigned <i>variable</i></code>	true, if <i>variable</i> has a value
<code>delete <i>variable</i></code>	Delete <i>variable</i>
<code>forward <i>function</i></code>	forward deklaration of functions for recursion
<code>save "<i>name</i>"</code>	Save a Magma session
<code>restore "<i>name</i>"</code>	Restore a Magma session

# Functions, Procedures and Intrinsic

**Functions** have a fixed number of parameters and one or more return values

**Procedures** have a fixed number of parameters, which can be references, but no return values.

**Intrinsic** of the same name can have different input signatures, that is, they can be overloaded. Intrinsic either return values or have parameters which are references.

# Functions, Procedures and Intrinsic

**Functions** have a fixed number of parameters and one or more return values

**Procedures** have a fixed number of parameters, which can be references, but no return values.

**Intrinsic**s of the same name can have different input signatures, that is, they can be overloaded. Intrinsic either return values or have parameters which are references.

**Variable arguments** exist for functions, procedures and intrinsic.

Syntax: function-, procedure- or intrinsic-call

```
function(arg_1,...,arg_n : var_1 :=opt_1,... var_m :=opt_m)
```



# Functions, Procedures and Intrinsic

**Functions** have a fixed number of parameters and one or more return values

**Procedures** have a fixed number of parameters, which can be references, but no return values.

**Intrinsic**s of the same name can have different input signatures, that is, they can be overloaded. Intrinsic either return values or have parameters which are references.

**Variable arguments** exist for functions, procedures and intrinsic.

Syntax: function-, procedure- or intrinsic-call

```
function(arg_1,...,arg_n : var_1 := opt_1,... var_m := opt_m)
```

**Remark** Many intrinsic are available under different names, for example:

Factorization and Factorisation

ElementTosequence and Eltseq

# Constructors

**Constructors** have an arbitrary number of parameters and are meant for generic algebraic constructions.

<code>elt&lt;structure   arg_1, ..., arg_n&gt;</code>	Element
<code>ideal&lt;structure   arg_1, ..., arg_n&gt;</code>	Ideal
<code>ext&lt;structure   arg_1, ..., arg_n&gt;</code>	Extension
<code>sub&lt;structure   arg_1, ..., arg_n&gt;</code>	Substructure
<code>quo&lt;structure   arg_1, ..., arg_n&gt;</code>	Quotient
<code>recformat&lt;name_1 : type_1, ..., name_n : type_n&gt;</code>	
<code>rec&lt;recformat   name_1 := expr_1, ... &gt;</code>	Records
<code>map&lt;str_1 -&gt; str_2   x :-&gt; f(x, ...), y :-&gt; g(y, ...)&gt;</code>	Map
<code>hom&lt;str_1 -&gt; str_2   Images of generators of str_1&gt;</code>	Homomorphism
<code>func&lt;arg_1, ..., arg_n   expr&gt;</code>	function
<code>case&lt;expr   expr_1 : val_1, ..., default : val&gt;</code>	

# Constructors

## Example

```
> G<a,b,c> := FreeAbelianGroup(3); G;  
Abelian Group isomorphic to  $Z+Z+Z$  Defined on 3 generators (free)  
> S := sub<G| 6*a, 5*c>; S;  
Abelian Group isomorphic to  $Z+Z$  Defined on 2 generators in  
supergroup G:  
S.1 = 6*a S.2 = 5*c (free)  
> Q := quo<G|S>; Q;  
Abelian Group isomorphic to  $Z/30+Z$  Defined on 2 generators  
Relations: 30*Q.1 = 0
```

# Constructors

## Example

```
> G<a,b,c> := FreeAbelianGroup(3); G;
Abelian Group isomorphic to Z+Z+Z Defined on 3 generators (free)
> S := sub<G| 6*a, 5*c>; S;
Abelian Group isomorphic to Z+Z Defined on 2 generators in
supergroup G:
S.1 = 6*a S.2 = 5*c (free)
> Q := quo<G|S>; Q;
Abelian Group isomorphic to Z/30+Z Defined on 2 generators
Relations: 30*Q.1 = 0
> m := hom<G->G|a,c,b>; m;
Mapping from: GrpAb: G to GrpAb: G
> e := elt<G|1,2,3>; e;
a + 2*b + 3*c
> print "m(e) =",m(e);
m(e) = a + 3*b + 2*c
```

# Exercises

- 1 Write the commands from the constructor-example in a file (e.g. with the editor Notepad). Load this file using `load` and `iload`.
- 2 Construct a map with inverse from  $\mathbb{Z}$  to  $\mathbb{Z}$ , that maps  $x \in \mathbb{Z}$  to  $x + 5$ .
- 3 Generate an abelian group  $A$ , which is isomorphic to  $\mathbb{Z}/9\mathbb{Z} \times \mathbb{Z}/5\mathbb{Z}$ , such that the generators of the cyclic factors are called  $a$  and  $b$ . (Hint: `AbelianGroup`)
- 4 Let  $B$  be an abelian group isomorphic to  $\mathbb{Z}/3$ . Construct a homomorphism  $f$  of  $A$  nach  $B$ . Compute the kernel of  $f$ .
- 5 Construct the ring  $\mathbb{Z}/18\mathbb{Z}$ . (Hint: `Integers()` and `quo<>`)

# Solutions

- 1** load reads the lines, as if they had been entered at the Magma prompt.

iload waits for you to press <return> after every line.

- 2** `m := map< Integers()->Integers() |  
     a:->a+5, b:->b-5>;`  
`m := map< Integers()->Integers() |  
     a:->func<x|x+5>(a), b:->func<y|y-5>(b)>;`

- 3** `A<a,b> := AbelianGroup([9,5]);`

- 4** `B<c> := AbelianGroup([3]);`  
`f := hom<A->B|c,0>;`  
`Kernel(f);`

- 5** `Q := quo<Integers()|18>;`

# Data Structures

**Sequences** `[1,3,6,2]`

**Strings** `"eins zwei drei"`

**Sets** `{1,2,3,6}`

**Tuple** `<1,"zwei">`

**Lists** `[* 1,"zwei",true *]`

**Records** `rec<recformat<a,b>|a:=1 b:="B">`

**Attribute** `RealField(20) 'StrLocalData`

# Sequences: Generation

Empty sequence with elements from a *structure*: `[structure|]`

Example: Empty sequence of fractions

```
> S := [Rationals()|]; ExtendedType(S);  
SeqEnum[FldRatElt]
```



# Sequences: Generation

Empty sequence with elements from a *structure*: `[structure]`  
sequence over the smallest structure that contains  $elt_1, \dots, elt_n$ :  
`[elt_1, \dots, elt_2]`

## Example

```
> S := [1/2, 17, 7787, 23/4]; ExtendedType(S);  
SeqEnum[FldRatElt]
```

# Sequences: Generation

Empty sequence with elements from a *structure*: `[structure]`

sequence over the smallest structure that contains  $elt_1, \dots, elt_n$ :

`[elt_1, ..., elt_2]`

sequence über *structure*, welche  $elt_1, \dots, elt_n$  enthält:

`[structure | elt_1, ..., elt_2]`

## Examples

```
> [Rationals() | 1/2, 17, 7787, 23/4];
```

```
[ 1/2, 17, 7787, 23/4 ]
```

```
> [ComplexField(5) | 1/2, 17, 7787, 23/4];
```

```
[ 0.50000, 17.000, 7787.0, 5.7500 ]
```

# Sequences: Generation

Empty sequence with elements from a *structure*: `[structure]`

sequence over the smallest structure that contains  $elt_1, \dots, elt_n$ :

`[elt_1, ..., elt_2]`

sequence über *structure*, welche  $elt_1, \dots, elt_n$  enthält:

`[structure | elt_1, ..., elt_2]`

The sequence of elements  $f(x)$  for all  $x$  from  $M$ : `[f(x) : x in M]`

## Example

```
> [x^2 : x in [1,2,3]];
[1,4,9]
```

## Sequences: Generation

Empty sequence with elements from a *structure*: `[structure]`

sequence over the smallest structure that contains  $elt_1, \dots, elt_n$ :

`[elt_1, ..., elt_2]`

sequence über *structure*, welche  $elt_1, \dots, elt_n$  enthält:

`[structure | elt_1, ..., elt_2]`

The sequence of elements  $f(x)$  for all  $x$  from  $M$ : `[f(x) : x in M]`

sequence of elements  $f(x)$  for all  $x$  from  $M$  for which  $P(x)$  is true:

`[f(x) : x in M | P(x)]`

sequence of integers from  $a$  to  $b$ : `[a..b]`

### Example

```
> [x : x in [1..20] | IsPrime(x)];
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

# Sequences: Generation

Empty sequence with elements from a *structure*:  $[structure]$

sequence over the smallest structure that contains  $elt_1, \dots, elt_n$ :

$[elt_1, \dots, elt_2]$

sequence über *structure*, welche  $elt_1, \dots, elt_n$  enthält:

$[structure \mid elt_1, \dots, elt_2]$

The sequence of elements  $f(x)$  for all  $x$  from  $M$ :  $[f(x) : x \text{ in } M]$

sequence of elements  $f(x)$  for all  $x$  from  $M$  for which  $P(x)$  is true:

$[f(x) : x \text{ in } M \mid P(x)]$

sequence of integers from  $a$  to  $b$ :  $[a..b]$

sequence of integers from  $a$  to  $b$  with difference  $d$ :  $[a..b \text{ by } d]$

# Sequences: Access and Assignment

## Self Reference

When defining a sequence one can refer to previous elements in the sequence with `Self`.

### Example: `Self`

```
> [ i gt 1 select i+Self(i-1) else 1: i in [1..10]];
[ 1, 3, 6, 10, 15, 21, 28, 36, 45, 55 ];
```

# Sequences: Access and Assignment

## Example: Access

```
> L := [21, 22, 23, 24];
```

```
> L[4];
```

```
1
```

```
> M := [L, [1,3]];
```

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

```
22
```

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

```
22
```

# Sequences: Access and Assignment

## Example: Access

```
> L := [21, 22, 23, 24];  
> L[4];  
1  
> M := [L, [1,3]];  
> M[1][2];  
22  
> M[1,2];  
22
```

## Example: Assignment

```
> L := [1/2, 3/4, 7];  
> L[3] := 18; L;  
[1/2, 3/4, 18]  
> L[5] := 19; L;  
[1/2, 3/4, 18, undef, 19];
```



# Sequences: Functionality

## Functions for Sequences

<code>seq_1 cat seq_2</code>	Concatenation of <code>seq_1</code> and <code>seq_2</code>
<code>elt in seq</code>	Test whether <code>elt</code> is in <code>seq</code>
<code>Append(seq,elt)</code>	Yields <code>seq</code> extended by <code>elt</code>
<code>Append(~seq,elt)</code>	Appends <code>elt</code> to <code>seq</code> .
<code>Max(seq), Min(seq)</code>	Minimum resp. maximum of the elements in <code>seq</code>
<code>#seq</code>	Number of elements in <code>seq</code>
<code>&amp;op seq</code>	Reduction of <code>seq</code> by <code>op</code>
<code>&amp;+ seq</code>	Sum of the elements of <code>seq</code>

# Sequences: Functionality

## Functions for Sequences

<code>seq_1 cat seq_2</code>	Concatenation of <code>seq_1</code> and <code>seq_2</code>
<code>elt in seq</code>	Test whether <code>elt</code> is in <code>seq</code>
<code>Append(seq,elt)</code>	Yields <code>seq</code> extended by <code>elt</code>
<code>Append(~seq,elt)</code>	Appends <code>elt</code> to <code>seq</code> .
<code>Max(seq), Min(seq)</code>	Minimum resp. maximum of the elements in <code>seq</code>
<code>#seq</code>	Number of elements in <code>seq</code>
<code>&amp;op seq</code>	Reduction of <code>seq</code> by <code>op</code>
<code>&amp;+ seq</code>	Sum of the elements of <code>seq</code>

## Example

```
> L := [1,3,4,6]; Max(L);
6
> &+L;
14
```

# Sequences: Functionality

## Functions for Sequences

<code>seq_1 cat seq_2</code>	Concatenation of <code>seq_1</code> and <code>seq_2</code>
<code>elt in seq</code>	Test whether <code>elt</code> is in <code>seq</code>
<code>Append(seq,elt)</code>	Yields <code>seq</code> extended by <code>elt</code>
<code>Append(~seq,elt)</code>	Appends <code>elt</code> to <code>seq</code> .
<code>Max(seq), Min(seq)</code>	Minimum resp. maximum of the elements in <code>seq</code>
<code>#seq</code>	Number of elements in <code>seq</code>
<code>&amp;op seq</code>	Reduction of <code>seq</code> by <code>op</code>
<code>&amp;+ seq</code>	Sum of the elements of <code>seq</code>

## Further functions for sequences:

Sort, Rotate, Reverse, Random, Representative,  
 ChangeUniverse, Universe, Position, Insert, Exclude,  
 Remove, Permutesequence, ParallelSort

# Sequences: Application

Many objects can be turned into sequences.

## Example

```
> Z := Integers(); Zx<x> := PolynomialRing(Z);  
> f := x^3+7*x^2+3;  
> L := Eltseq(f); L;  
[3,0,7,1]
```

# Sequences: Application

Many objects can be turned into sequences.

## Example

```
> Z := Integers(); Zx<x> := PolynomialRing(Z);  
> f := x^3+7*x^2+3;  
> L := Eltseq(f); L;  
[3,0,7,1]
```

Many objects can be generated using sequences.

## Example (continued)

```
> elt<Zx|L>;  
x^3+7*x^2+3;  
> Zx!L;  
x^3+7*x^2+3;
```

# Sets

In Magma there are several kinds of sets:

- Enumerated Sets  $\{\text{Integers}() \mid 1,2,5,6\}$
- Indexed Sets  $\{\text{@ Integers}() \mid z^3 : z \text{ in } [1..9] \text{@}\}$
- Formal Sets  $\{\text{! } x \text{ in Integers}() \mid \text{IsPrime}(x) \text{!}\}$
- Multisets  $\{\text{* } 7^{95}, 3^{102}, 6^{94} \text{*}\}$

# Sets: Generation

Empty set with elements form a structure *structure*:  $\{structure\}$

Set over the smallest structure, that contains  $elt_1, \dots, elt_n$ :

$$\{elt_1, \dots, elt_n\}$$

Set over *structure*, which contains  $elt_1, \dots, elt_n$ :

$$\{structure \mid elt_1, \dots, elt_n\}$$

Set of elements  $f(x)$  for all  $x$  in  $M$  for which  $P(x)$  is true:

$$\{f(x) : x \text{ in } M \mid P(x)\}$$

Set of integers between  $a$  and  $b$  with difference  $d$ :  $\{a..b \text{ by } d\}$

For a finite structure  $S$   $\text{Set}(S)$  is the set of elements of  $S$ .

## Example

```
> F5 := FiniteField(5); Set(F5);
{ 0, 1, 2, 3, 4}
```

# Sets: Functionality

<code>set_1 join set_2</code>	Union of <code>set_1</code> and <code>set_2</code>
<code>set_1 meet set_2</code>	Intersection of <code>set_1</code> and <code>set_2</code>
<code>set_1 diff set_2</code>	Difference of <code>set_1</code> and <code>set_2</code>
<code>set_1 sdiff set_2</code>	Symmetric Difference of <code>set_1</code> and <code>set_2</code>
<code>set_1 subset set_2</code>	Symmetric Difference of <code>set_1</code> and <code>set_2</code>
<code>forall</code>	For alle quantor
<code>exists</code>	Existence quantor



# Sets: Functionality

<code>set_1 join set_2</code>	Union of <code>set_1</code> and <code>set_2</code>
<code>set_1 meet set_2</code>	Intersection of <code>set_1</code> and <code>set_2</code>
<code>set_1 diff set_2</code>	Difference of <code>set_1</code> and <code>set_2</code>
<code>set_1 sdiff set_2</code>	Symmetric Difference of <code>set_1</code> and <code>set_2</code>
<code>set_1 subset set_2</code>	Symmetric Difference of <code>set_1</code> and <code>set_2</code>
<code>forall</code>	For alle quantor
<code>exists</code>	Existence quantor

**Fur-**

## ther functions for sets:

Random, Representative, ChangeUniverse, Universe,  
Include, Exclude, in, notin, notsubset,...

# Tuple

Tuple are elements of cartesian products. cartesian products can be generated from all kinds of structures.

## Generation of Tuples

$\langle elt_1, \dots, elt_n \rangle$

## Generation of cartesian products

`CartesianProduct(str_1, str_2)`

`CartesianProduct(str, exponent)`

`CartesianProduct([str_1, ..., str_n])`

## Example

```
> a := <3, "drei">; Parent(a);  
Cartesian Product<Integer Ring, String structure>  
> C := CartesianProduct(Integers(), Parent("hihi"));  
> a in C;  
true
```

# Records: `reformat`

Records are data structures whose entries can be accessed by identifiers (names).

Before a record can be defined, its format has to be established.

## Syntax

```
reformat<name_1 : type_1, ... , name_n : type_n>
```

The types can be omitted.

`names(reformat)` returns a sequence of the names as strings.

# Records: `rec`

## Syntax

```
rec<recformat | name_1 := expr_1, ... >
```

<code>Names(rec)</code>	The names of the components of <code>rec</code>
<code>Format(rec)</code>	The format of <code>rec</code>
<code>rec' name</code>	Access the component <code>name</code> of <code>rec</code>
<code>assigned rec' name</code>	is true, if the component <code>name</code> has a value
<code>delete rec' name</code>	Deletes the component <code>name</code>

# Records: rec

## Syntax

```
rec<recformat | name_1 := expr_1, ... >
```

Names( <i>rec</i> )	The names of the components of <i>rec</i>
Format( <i>rec</i> )	The format of <i>rec</i>
<i>rec</i> ' <i>name</i>	Access the component <i>name</i> of <i>rec</i>
assigned <i>rec</i> ' <i>name</i>	is true, if the component <i>name</i> has a value
delete <i>rec</i> ' <i>name</i>	Deletes the component <i>name</i>

## Example

```
> RF := recformat< a:RngIntElt, L:SeqEnum, note:MonStgElt, d>;
> r := rec< RF | a:=2, L:=[1,2,3], note:="wichtig">;
> r'note;
wichtig
> n := names(RF); r'(n[1]), r'"a", assigned r'a;
2 2 true
> r'd := 17/6; delete r'L; r;
rec<RF | a := 2, note := wichtig, d := 17/6 >
```

# Attribute

**Attributes** contain additional information for objects.

**Attributes** are defined for all objects of a category (of a type).

<code>GetAttributes(<i>category</i>)</code>	Sequence of attributes for <i>category</i>
<code>AddAttribute(<i>category</i>, "A")</code>	New attribute <i>A</i> for <i>category</i>
<code><i>obj</i>' A</code>	Access the attribute <i>A</i> of <i>obj</i>
assigned <code><i>obj</i>' A</code>	true, if <i>obj</i> ' <i>A</i> has a value
<code>delete <i>obj</i>' A</code>	deletes the attribute <i>A</i>

# Attribute

**Attributes** contain additional information for objects.

**Attributes** are defined for all objects of a category (of a type).

<code>GetAttributes(<i>category</i>)</code>	Sequence of attributes for <i>category</i>
<code>AddAttribute(<i>category</i>, "A")</code>	New attribute <i>A</i> for <i>category</i>
<code><i>obj</i>' A</code>	Access the attribute <i>A</i> of <i>obj</i>
<code>assigned <i>obj</i>' A</code>	true, if <i>obj</i> ' <i>A</i> has a value
<code>delete <i>obj</i>' A</code>	deletes the attribute <i>A</i>

## Example

```
> G := AbelianGroup([2,4,8]);
> AddAttribute(Type(G), "pGroup");
> GetAttributes(Type(G));
[ ActionType, BCI, BasicAlgebra, Basis, pGroup ]
> assigned G'pGroup;
false
> G'pGroup := 2; G'pGroup;
2
```

# Maps

**Maps** differ from functions in programming languages. **Maps** can have:

- Codomain and Domain,
- Inverses,
- Preimages,
- Kernel and Image,
- Composita



# Maps

**Maps** differ from functions in programming languages. **Maps** can have:

- Codomain and Domain,
- Inverses,
- Preimages,
- Kernel and Image,
- Composita

**Maps** can be defined by:

- a function,
- two functions (map and its inverse),
- a graph (a subset of  $\text{domain} \times \text{codomain}$ ),
- images of generators of algebraic structures.

# Maps

## Constructors

<code>map&lt;...   ...&gt;</code>	Map constructor
<code>pmap&lt;...   ...&gt;</code>	Partial map constructor
<code>hom&lt;...   ...&gt;</code>	Homomorphism constructor

# Maps

## Constructors

<code>map&lt;...   ...&gt;</code>	Map constructor
<code>pmap&lt;...   ...&gt;</code>	Partial map constructor
<code>hom&lt;...   ...&gt;</code>	Homomorphism constructor

## Operations

<code>map_1 * map_1</code>	Composition <i>map_1</i> and <i>map_1</i>
<code>map<sup>-1</sup></code>	Inverse of <i>map</i>
<code>map<sup>n</sup></code>	<i>n</i> -fold composition of <i>map</i> with itself
<code>map_1 eq map_1</code>	Comparison of <i>map_1</i> and <i>map_1</i> (as objects in memory)

# Maps

## Functions

Composition( $map_1, map_1$ )	Composition of $map_1$ and $map_1$
Components( $map$ )	Sequence of maps, from which $map$ is composed.
Domain( $map$ )	Domain of $map$
Codomain( $map$ )	Codomain of $map$
Inverse( $map$ )	Inverse of $map$
Image( $map$ )	Image of $map$
Kernel( $map$ )	Kernel of $map$

# Exercises

- 1 Generate the set `set`, that contains all prime numbers under 100. (Hint: `IsPrime`)
- 2 `Factorization` returns factors and multiplicities. Extract the list of prime factors.
- 3 Compute a sequence that contains the first 100 elements of the Lucas sequence  $(a_i)$  mit  $a_1 = 1$ ,  $a_2 = 3$  and for  $i > 2$   $a_i = a_{i-1} + a_{i-2}$  enthält. (Hint: `Self`)
- 4 Let `N := [1..9]`. Sort `N` alphabetically by the German name of the numbers. (Hint: `ParallelSort`, `eins`, `zwei`, `drei`, `vier`, `fuenf`, `sechs`, `sieben`, `acht`, `neun`)

# Solutions

```
1 [a : a in [1..100] | IsPrime(a)];
2 n := 4677834567836567826;
  [a[1] : a in Factorization(n)];
3 [ i gt 2 select Self(i-1)+Self(i-2) else i eq 2
  select 3 else 1 : i in [1..100]];
4 N :=[1..9];
  D := ["eins", "zwei", "drei", "vier", "fuenf",
  "sechs", "sieben", "acht", "neun"];
  ParallelSort(~D,~N);
```

# control structures

## conditions

- `if...then...elif...then...else...end if;`
- `...:=...select...else...;`
- `case...when...else...end case;`
- `error...;`
- `assert...;`
- `require...;`

## loops

- `for...in...do...end for;`
- `for...:=...to...by...do...end for;`
- `while...do...end while;`
- `repeat...until...;`

# conditions: if

Syntax: if

```
if boolean_expression_1 then
  expressions_1
elif boolean_expression_2 then
  expressions_2
else
  expressions_3
end if;
```

The else and elif branches can be omitted.

Example

```
if a gt b then c := 1; else c := -1; end if;
```



# conditions: select

## Syntax

```
c := boolean_expression select expression_1 else expression_2;
```

Set *c* to *expression\_1*, if *boolean\_expression* is true, otherwise set *c* to *expression\_2*.

## Example

```
c := a gt b select 1 else -1;
```

# conditions: case

## Syntax: case control structure

```
case expression:  
  when value_1: expression_1  
  ...  
  when value_n: expression_n  
end case;
```

## Example

```
case a > b: when true: c := 1; when false: c := -1; end case;
```

# conditions: case

## Syntax: case control structure

```
case expression:  
  when value_1: expression_1  
  ...  
  when value_n: expression_n  
end case;
```

## Example

```
case a gt b: when true: c := 1; when false: c := -1; end case;
```

## Syntax: case constructor

```
case<expression | value_1:expression_1, ..., default:val>
```

## Example

```
c := case<a gt b | true:1, default:-1>;
```

## conditions: assert and error

### Syntax: assert

```
assert boolean_expression;
```

An error is raised, if *boolean\_expression* is false.

SetAssertions(false) switches this behaviour off.

### Syntax: error

```
error expression_1, ..., expression_n;
```

*expression\_1, ..., expression\_n* are output as an error message

### Syntax: error

```
error if boolean_expression, expression_1, ..., expression_n;
```

If *boolean\_expression* is true, then the error message *expression\_1, ..., expression\_n* is printed.

# loops: for

## Syntax

```
for variable in sequence_or_set do  
  expressions  
end for;
```

## Example

```
sum := 0; for i in [1..1000000] do sum+:=i; end for;
```

## Syntax

```
for variable := expression_1 to expression_2 by expression_3 do  
  expressions  
end for;
```

## Example

```
sum:=0; for i := 1 to 1000000 do sum+:=i; end for;
```

# loops: for

## Syntax

```
for variable in sequence_or_set do
  expressions
  condition break variable
  expressions
end for;
```

## Example

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

# loops: while

## Syntax

```
while boolean_expression do  
  expressions  
end while;
```

## Example

```
i := 1; sum := 0;  
while i le 1000000 do  
  sum+:=i;  
  i+:=1;  
end while;
```

With `break` a while loop can be exited early.

# loops: repeat

## Syntax

```
repeat
  expressions
until boolean_expression;
```

## Example

```
i := 1; sum := 0;
repeat
  sum+=i;
  i+=1;
until i gt 1000000;
```

With `break` a `repeat...until` can be exited.



# Loops: Timing

**&+[...]**

```
> time sum := &+[1..1000000];  
Time: 0.670
```

**for**

```
> sum := 0;  
> time for i in [1..1000000] do sum+=i; end for;  
Time: 1.640
```

**repeat**

```
> i := 1; sum := 0;  
> time repeat sum+=i; i+=1; until i gt 1000000;  
Time: 2.770
```

# where...is

## Syntax

*expression\_1* where variable is *expression\_2*;

*expression\_1* where variable := *expression\_2*;

## Example

```
> x + y where x is 5 where y is 6;
```

```
11
```

```
> x + y where x is y where y is 6;
```

```
12
```

```
> { a: i in [1..10] | IsPrime(a) where a is 3*i+1 };  
{ 7, 13, 19, 31 }
```

# Exercises

- 1 Sum the squares of the elements of a sequence in four different ways (three using loops).
- 2 Find all tuples  $\langle x, y \rangle$  mit  $x^3 y^2 \equiv 1 \pmod{y + 1}$  and  $1 \leq x \leq 100$  and  $1 \leq y \leq 100$ .

# Solutions

```
1 L := [1..100];
  s:=&+[ a^2:a in L]; s;
  s:=0; for a in L do s+:=a^2; end for; s;
  s:=0;i:=1;while i le #L do s+:=L[i]^2;i+:=1; end while;s;
  s:=0; i:=1; repeat s+:=L[i]^2; i+:=1; until i gt #L; s;

2 L := [];
  for x in [1..100] do
    for y in [1..100] do
      if (x^3*y^2) mod (y+1) eq 1 then
        Append(~L,<x,y>);
      end if;
    end for;
  end for; // or short, without a for-loop:

L := [ <x,y> : x,y in [1..100] |
      (x^3*y^2) mod (y+1) eq 1 ];
```

# functions

## Syntax

```
function f(arg_1, ..., arg_n)  
  expressions  
  return expression  
end function;
```

The arguments *arg\_1*, ..., *arg\_n* are passed as values, they can be changed locally in the function, which does not have any global effects.

# functions

## Syntax

```
function f(arg_1, ..., arg_n)  
  expressions  
  return expression  
end function;
```

Global variables keep the value, which they had at the time of definition of the function.

## Example

```
> function f(a) return 2*a; end function;  
> f(2);  
4
```

# functions

## Syntax

```
function f(arg_1, ..., arg_n)  
  expressions  
  return expression  
end function;
```

Global variables keep the value, which they had at the time of definition of the function.

## Example

```
> b := 2; function f(a) return b*a; end function;  
> f(2);  
4  
> b := 7; f(2);  
4
```

# functions

## Syntax

```
function f(arg_1, ..., arg_n:var_1:=expr_1, ..., var_m:=expr_m)  
  expressions  
  return expression  
end function;
```

Variable arguments  $arg_1, \dots, arg_n$  are identified by their names. The default values for  $var_1$  to  $var_m$  are given in the definition of the function.



# functions

## Syntax

```
function f(arg_1, ..., arg_n:var_1:=expr_1, ..., var_m:=expr_m)
  expressions
  return expression
end function;
```

Variable arguments  $arg_1, \dots, arg_n$  are identified by their names. The default values for  $var_1$  to  $var_m$  are given in the definition of the function.

## Example

```
> function f(a:b:=2) return b*a; end function;
> f(2);
4
> f(2:b := 3);
6
```

# Procedures

## Syntax

```
procedure f(arg_1, ..., arg_n:var_1:=expr_1, ... var_m:=expr_m)
  expressions
end procedure;
```

Procedures do not return anything. Each of the arguments *arg\_1*, ..., *arg\_n* can either be a reference or a value. References are prepended by `~`.

## Example

```
> procedure g(~a,b) a+:=b; end procedure;
> c := 5; g(~c,7); c;
12
```

# Recursion

A function can be called from the body of the function itself by `$$`.  
Alternatively it can be declared with the directive `forward`.

## Example

```
function sum(i)
  return i eq 1 select 1 else i+$(i-1);
end function;
```

# Recursion

A function can be called from the body of the function itself by `$$`.  
Alternatively it can be declared with the directive `forward`.

## Example

```
forward sum;
function sum(i)
  return i eq 1 select 1 else i+sum(i-1);
end function;

> sum(10000);
50005000
> sum(1000000);
Segmentation fault
```

# Comments

In a line everthing following `//` is treated as a comment.

Comments over several lines begin with `/*` and end with `*/`.

Intrinsic must have a comment in braces.

# Exercise

Write a function that sorts strings alphabetically and ignores capitalization.

# Solution

```
function to_upper(s)
  return &cat[a ge "a" and a le "z"
            select CodeToString(StringToCode(a)-32) else a:
            a in Eltseq(s)];
end function;
```

```
function cmp_alpha(s,t)
  s := to_upper(s); t := to_upper(t);
  if s eq t then return 0;
  elif s lt t then return -1;
  else return 1; end if;
end function;
```

```
function sort_alpha(L)
  return Sort(L,cmp_alpha);
end function;
```

# Intrinsics

**Intrinsics** are implemented in two ways:

- in the Magma C-library and
- in the Magma shell language.

**Intrinsics**, which are implemented in the shell language, are distributed in source code.

*Intrinsic*:Maximal; shows all signatures of *Intrinsic* and, if possible, the path to the file with the source code.



# Intrinsics

**Intrinsics** are implemented in two ways:

- in the Magma C-library and
- in the Magma shell language.

**Intrinsics**, which are implemented in the shell language, are distributed in source code.

*Intrinsic*:Maximal; shows all signatures of *Intrinsic* and, if possible, the path to the file with the source code.

## Example

```
> Flat:Maximal;
Intrinsic 'Flat', Signatures:
(<Tup> T) -> Tup
  Returns the flattened version of T.
Defined in file: package/Ring/RngLoc/class_field.m:
(<SeqEnum> S) -> SeqEnum
  Returns the flattend version of S.
```

# Structures

Each element is in a structure.

`Parent` returns the structure, in which an element lives.

`Universe` returns the structure in which the components of a sequence or set live.

# Structures

Each element is in a structure.

Parent returns the structure, in which an element lives.

Universe returns the structure in which the components of a sequence or set live.

## Examples

```
> Z := Integers(); Z, Parent(Z);  
Integer Ring  
Power Structure of RngInt  
> Parent(123);  
Integer Ring  
> Parent([1,2,3]), Universe([1,2,3]);  
Set of sequences over Integer Ring Integer Ring  
> Zx<x> := PolynomialRing(Z); Parent(Zx), Parent(x);  
Power Structure of RngUPol  
Univariate Polynomial Ring in x over Integer Ring
```

# Structures

Each element is in a structure.

`Parent` returns the structure, in which an element lives.

`Universe` returns the structure in which the components of a sequence or set live.

`CoveringStructure` is the smallest structure that contains tow structure.

# Structures

Each element is in a structure.

Parent returns the structure, in which an element lives.

Universe returns the structure in which the components of a sequence or set live.

CoveringStructure is the smallest structure that contains two structures.

## Examples

```
> Z := Integers(); Zx<x> := PolynomialRing(Z);
> CoveringStructure(Z,Zx);
Univariate Polynomial Ring in x over Integer Ring
> CoveringStructure(Z,Rationals());
Rational Field
```

# Structures

Each element is in a structure.

`Parent` returns the structure, in which an element lives.

`Universe` returns the structure in which the components of a sequence or set live.

`CoveringStructure` is the smallest structure that contains tow structure.

! coerces an element into a structure.

# Structures

Each element is in a structure.

Parent returns the structure, in which an element lives.

Universe returns the structure in which the components of a sequence or set live.

CoveringStructure is the smallest structure that contains two structures.

! coerces an element into a structure.

## Example

```
> Z := Integers(); Zx<x> := PolynomialRing(Z);
> f := Zx!2; Parent(f);
Univariate Polynomial Ring in x over Integer Ring
> g := Zx![1,2,3];
3*x^2 + 2*x + 1
```

# Type

`Type` returns the type of an object.

`ExtendedType` returns the extended type of an object.

`ElementType` returns the type of an element in a structure.



# Type

Type returns the type of an object.

ExtendedType returns the extended type of an object.

ElementType returns the type of an element in a structure.

## Example

```

> Z := Integers(); Type(Z), ExtendedType(Z), elementType(Z);
RngInt RngInt RngIntElt
> Type(Type(Z); ExtendedType(Type(Z)));
Cat Cat[RngInt]
> Type([1,2,3]), ExtendedType([1,2,3]);
SeqEnum SeqEnum[RngIntElt]
> Zx<x> := PolynomialRing(Z); Type(Zx), ExtendedType(Zx);
RngUPol RngUPol[RngInt];
> Type([x,x^+1]), ExtendedType([x,x^2+1]);
SeqEnum SeqEnum[RngUPolElt[RngInt]]

```

# Type

Type returns the type of an object.

ExtendedType returns the extended type of an object.

ElementType returns the type of an element in a structure.

ISA checks whether a category contains another category

# Type

Type returns the type of an object.

ExtendedType returns the extended type of an object.

ElementType returns the type of an element in a structure.

ISA checks whether a category contains another category

## Example

```
> Z := Integers(); Zx<x> := PolynomialRing(Z);
> Type(12), Type(x), Type(1/2);
RngIntElt RngUPolElt FldRatElt
> ISA(RngIntElt, RngUPolElt), ISA(RngIntElt, FldRatElt);
false false
> ISA(RngIntElt, RngElt), ISA(RngUPolElt, RngElt);
true true
> ISA(FldRatElt, RngElt), ISA(FldRatElt, FldElt);
true true
```

# Signatures

Intrinsics offer functionality for different types of objects under the same name. The input signature decides which routines are called.

## Signatur

```
Intrinsic(var_1::typin_1, ..., var_n::typin_n) -> typout_1, ..., typout_m
```

# Signatures

Intrinsics offer functionality for different types of objects under the same name. The input signature decides which routines are called.

## Signatur

*Intrinsic*(*var\_1*::*typin\_1*, ..., *var\_n*::*typin\_n*) -> *typout\_1*, ..., *typout\_m*

## Example

```
> Factorization;
Intrinsic 'Factorization', Signatures:
(<RngIntElt> n) -> RngIntEltFact, RngIntElt, SeqEnum
  Attempt to find the prime factorization of the value of n.
(<RngUPolElt> f) -> SeqEnum, RngElt
  Factorization of f into irreducible factors together
  with the appropriate unit.
(<RngOrdFracIdl> I) -> SeqEnum
  The factorization of I into prime ideals.
```

# Intrinsic

## Syntax: Intrinsic procedure

```
intrinsic name (arg_1::typin_1, ..., arg_n::typin_n:var_1:=expr_1, ...)  
{comment}  
  expressions  
end intrinsic;
```

In the same way as for procedures the arguments  $arg_1, \dots, arg_n$  can be marked as references by prepending  $\sim$ .

# Intrinsics

file doppel.m

```
intrinsic Double(~a::RngIntElt)
{double the integer a.}
  a*:=2;
end intrinsic;
```

# Intrinsic

## file doppel.m

```
intrinsic Double(~a::RngIntElt)
{double the integer a.}
  a*:=2;
end intrinsic;
```

## Example

```
> Attach("doppel.m");
> Doppel;
Intrinsic 'Doppel', Signatures:
(<RngIntElt>~a)
  double the integer a.
> a := 3; Doppel(~a); a;
6
```



# Intrinsic

## Syntax: Intrinsic function

```
intrinsic name (arg_1::typin_1, ..., arg_n::typin_n) -> type_1, ..., type_m  
{comment}  
  expressions  
  return espressions  
end intrinsic;
```

Variable arguments may be used. The types of return values is not checked.

# Intrinsics

file doppel.m

```
...  
intrinsic Doppel(a::RngIntElt) -> RngIntElt  
{Double a.}  
  return 2*a;  
end intrinsic;
```

# Intrinsic

## file doppel.m

```
...
intrinsic Doppel(a::RngIntElt) -> RngIntElt
{Double a.}
  return 2*a;
end intrinsic;
```

## Example

```
> Attach("doppel.m");
> Doppel;
Intrinsic 'Doppel', Signatures:
(<RngIntElt>~a)
  Double the integer a.
(<RngIntElt> a) -> RngIntElt
  The double of the integer a.
> Doppel(2);
4
```

# Packages

**Packages** are files that contain functions, procedures and intrinsics.

**Packages** are loaded with `Attach("packagename")`.

Only intrinsics in a package **package** are available to the user. Functions and procedures in *packagename* are invisible.

# Packages

**Packages** are files that contain functions, procedures and intrinsics.

**Packages** are loaded with `Attach("packagename")`.

Only intrinsics in a package **package** are available to the user. Functions and procedures in *packagename* are invisible.

These directives can only be used in **packages**:

- `freeze;`
- `declare attributes Category:atr_1,...,atr_n;`
- `declare verbose Keyword, n;`

**package** are reloaded automatically, when they are changed, unless the first line of the file is `freeze;`.

# Packages: attributes

```
file pGroup.m
```

```
declare attributes GrpAb: pGroup;  
intrinsic IspGroup(G::GrpAb) -> BoolElt, RngIntElt  
{true, p if G is a p-Group}  
ipp, p, _ := IsPrimePower(GCD(InvariantFactors(G)));  
if ipp then G'pGroup:=p; return true,p else return false; end if;  
end intrinsic;
```

# Packages: attributes

file pGroup.m

```
declare attributes GrpAb: pGroup;
intrinsic IspGroup(G::GrpAb) -> BoolElt, RngIntElt
{true, p if G is a p-Group}
ipp, p, _ := IsPrimePower(GCD(InvariantFactors(G)));
if ipp then G'pGroup:=p; return true,p else return false; end if;
end intrinsic;
```

## Example

```
> Attach("pGroup.m");
> A := AbelianGroup([5,25]); A'pGroup;
Attribute pGroup for this structure is valid but not assigned
> IspGroup(A);
true 5
> A'pGroup;
5
```

## Packages: verbose

Verbose flags are a tool for controlling the output of additional information.

<code>declare verbose <i>Flag</i>, <i>n</i></code>	Verbose <i>flag</i> between 0 and <i>n</i>
<code>SetVerbose("Flag", <i>i</i>)</code>	Set verbose <i>flag</i> to <i>i</i>
<code>vprint <i>flag</i>, <i>i</i>: <i>expr_1</i>,...</code>	If verbose <i>flag</i> greater than <i>i</i> , print <i>expr_1</i> ,...
<code>vtime <i>Flag</i>, <i>i</i>: <i>expression</i></code>	If verbose <i>flag</i> is greater than or equal <i>i</i> , print the time needed to evaluate <i>expression</i> .



## Packages: verbose

Verbose flags are a tool for controlling the output of assitional information.

<code>declare verbose <i>Flag</i>, <i>n</i></code>	Verbose <i>flag</i> between 0 and <i>n</i>
<code>SetVerbose("Flag", <i>i</i>)</code>	Set verbose <i>flag</i> to <i>i</i>
<code>vprint <i>flag</i>, <i>i</i>: expr_1,...</code>	If verbose <i>flag</i> greater than <i>i</i> , print <i>expr_1,...</i>
<code>vtime <i>Flag</i>, <i>i</i>: expression</code>	If verbose <i>flag</i> is greater than or equal <i>i</i> , print the time needed to evaluate <i>expression</i> .

file it.m

```
declare verbose PrintIt, 2;
intrinsic f(a) -> .
{one vprint}
  vprint PrintIt, 1: "it";
  return a;
end intrinsic;
```

## Packages: verbose

Verbose flags are a tool for controlling the output of additional information.

<code>declare verbose <i>Flag</i>, <i>n</i></code>	Verbose <i>flag</i> between 0 and <i>n</i>
<code>SetVerbose("Flag", <i>i</i>)</code>	Set verbose <i>flag</i> to <i>i</i>
<code>vprint <i>flag</i>, <i>i</i>: expr_1,...</code>	If verbose <i>flag</i> greater than <i>i</i> , print <i>expr_1,...</i>
<code>vtime <i>Flag</i>, <i>i</i>: expression</code>	If verbose <i>flag</i> is greater than or equal <i>i</i> , print the time needed to evaluate <i>expression</i> .

### Example

```
> Attach("it.m"); f(2);
2
> SetVerbose("PrintIt",2); f(2);
it
2
```

# Exercise

- 1 Find the source code of the intrinsic

```
ClassGroup(<FldFunG> F) -> GrpAb, Map, Map
```

- 2 Write a intrinsic procedure, which negates all elements of a sequence.
- 3 Overload the addition for strings, such that it concatenates strings (Hint: '+' )
- 4 Write an intrinsic that multiplies integers and strings, such that  $n*s$  concatenates  $n$ -times  $s$ .

# Solutions

- 1** `> ClassGroup:Maximal;`  
Defined in file: `.../package/Ring/FldFun/classgroup.m`
- 2** `intrinsic Negate(L::SeqEnum) -> SeqEnum`  
{Negate all entries of L}  
    `return [-a:a in L];`  
`end intrinsic;`
- 3** `intrinsic '+' (a::MonStgElt,b::MonStgElt) -> MonStgElt`  
{Concatenate a and b}  
    `return a cat b;`  
`end intrinsic;`
- 4** `intrinsic '*' (n::RngIntElt,s::MonStgElt) -> MonStgElt`  
{n times s}  
    `return &cat[ s:i in [1..n]];`  
`end intrinsic;`

# Timing

The directive `time` returns the time it takes to evaluate an expression in CPU time.

## Syntax

```
time Expression
```

## Example

```
> time  
Factorization(672654564564345647235347534678565567412651237);  
[ <6580031, 1>, <30393726024835583, 1>,  
<3363413086414897650469, 1> ]  
Time: 1.530
```

# Timing

<code>Cputime()</code>	CPU time since the start of Magma in seconds
<code>Cputime(<i>time</i>)</code>	CPU time since <i>time</i>
<code>Realtime()</code>	Seconds since 1.1.1970 00:00:00 GMT
<code>Realtime(<i>time</i>)</code>	Seconds sincetime

## Example

```
> c := Cputime(); r := Realtime();
> time
Factorization(672654564564345647235347534678565567412651237);
[ <6580031, 1>, <30393726024835583, 1>,
<3363413086414897650469, 1> ]
Time: 1.500
> Cputime(c), Realtime(r);
1.510 12.700
```

# Profiler

<code>SetProfile(true);</code>	Begin Profiling
<code>SetProfile(false);</code>	End Profiling
<code>p:=ProfileGraph();</code>	Save the Profiling information
<code>ProfilePrintByTotalTime(p);</code>	output sorted by time
<code>ProfilePrintByTotalCount(p);</code>	output sorted by count of calls

## Example

```
> SetProfile(true);
> sum := &+[1..1000000];
> SetProfile(false);
> p:=ProfileGraph(); ProfilePrintByTotalTime(p);
```

Index	name	Time	Count
1	<main>	0.320	1
3	&+(<SeqEnum> S) -> .	0.320	1
2	Constructor	0.000	1

function-, procedure- and Intrinsic- calls are counted.

# Profiler

Profile: `sum:=&+[1..1000000];`

Index	Name	Time	Count
1	<main>	0.320	1
3	&+(<SeqEnum> S) -> .	0.320	1
2	Constructor	0.000	1

Profile: `sum:=0; for i in [1..1000000] do sum+=i; end for;`

Index	Name	Time	Count
3	+=(<RngIntElt> x, <RngIntElt> y)	0.450	1000000
1	<main>	0.450	1
2	Constructor	0.000	1

Profile: `i:=1; sum:=0; repeat sum+=i; i+=1; until i gt 1000000;`

Index	Name	Time	Count
1	<main>	1.490	1
2	+=(<RngIntElt> x, <RngIntElt> y)	0.980	2000000
3	gt(<RngIntElt>, <RngIntElt>)->BoolElt	0.510	1000000



# Exercise

Analyze the following commands with the Magma profiler:

```
> q2 := pAdicField(2,100);  
> q2y<y> := PolynomialRing(q2);  
> Factorization((y^32+16)*(y^32+32*y^8+16)+2^10);
```

Set the verbose flag RoundFour to different values.

# Solution

```
SetProfile(true);  
q2 := pAdicField(2,100);  
q2y<y> := PolynomialRing(q2);  
Factorization((y^32+16)*(y^32+32*y^8+16)+2^10);  
SetProfile(false);  
p:=ProfileGraph(); ProfilePrintByTotalTime(p);  
  
SetVerbose("RoundFour",3);  
Factorization((y^32+16)*(y^32+32*y^8+16)+2^10);
```