KASH
Introduction to KASH3

KANT–Group
Technische Universität Berlin
Institut für Mathematik, MA 8-1
Straße des 17. Juni 136
10623 Berlin, Germany
http://www.math.tu-berlin.de/~kant

January 31, 2006

Contents

Chapter 1

Preface

This is the current release of KANT, the KANT SHell. The quasi-acronym KANT stands for Computational Algebraic Number Theory with a slight twist hinting at its German origin.

1.1 Functionality

KANT is a program library for computations in algebraic number fields, algebraic function fields and local fields. In the number field case, algebraic integers are considered to be elements of a specified order of an appropriate field F. The available algorithms provide the user with the means to compute many invariants of F. It is possible to solve tasks like calculating the solutions of Diophantine equations related to F. Furthermore subfields of F can be generated and F can be embedded into an overfield. The potential of moving elements between different fields (orders) is a significant feature of our system. In the function field case, for example, genus computations and the construction of Riemann-Roch spaces are available.

1.2 History

KANT was developed at the University of Duesseldorf from 1987 until 1993 and at Technische Universitaet Berlin afterwards. During these years the performance of existing algorithms and their implementations grew dramatically. While calculations in number fields of degree 4 and greater were nearly impossible before 1970 and number fields of degree more than 10 were beyond reach until 1990, it is now possible to compute in number fields of degree well over 20, and – in special cases – even beyond 1000. This also characterizes one of the principles of KANT, namely to support computations in number fields of arbitrary degree rather than fixing the degree and pushing the size of the discriminant to the limit.

1.3 KANT

KANT consists of a C–library of thousands of functions for doing arithmetic in number fields, function fields, and local fields. Of course, the necessary auxiliaries from linear algebra over rings, especially lattices, are also included. The set of these functions is based on the core of the computer algebra system MAGMA from which we adopt our storage management, base arithmetic, arithmetic for finite fields, polynomial arithmetic and a variety of other tools.

1.4 Shell

To make KANT easier to use we developed a shell called KASH. This shell is based on that of the group theory package GAP3 and the handling is similar to that of MAPLE. We put great effort into enabling the user to handle the number theoretical objects in the very same way as one would do using pencil and paper. For example, there is just one command Factorization for the factorization of elements from a factorial monoid like rational integers in Z, polynomials over a field, or ideals from a Dedekind ring.

1.5 Copyright

Chapter 2

Getting Started

The following gives a short introduction to KASH3. We explain the basic types supported by KASH3 and how to effectively use them. As the main points of the KASH3 shell are given in conjunction with the type specific information, special attention should be paid to the examples.

2.1 Starting and Leaving

If KASH3 is correctly installed, then you start KASH3 by simply typing `kash3` at the prompt of your operating system. If you are successful in starting KASH3, the KASH3 banner should appear, at which time a command or function call may be entered. To exit KASH3 type `quit;` at the prompt (the semicolon is necessary!).

Example

```
kash% quit;
```

2.1.1 Command Line Options

```
kash3 [OPTIONS] [FILE]

-a <mem>  set maximal workspace memory usage to <mem>
          <mem> stands for byte-wise allocation
          <mem>k stands for kilobyte-wise allocation
          <mem>m stands for megabyte-wise allocation
-b        suppress banners
-c        suppress ANSI colours
-d        suppress using documentation dump file
-D        dump documentation to stdout
-e        suppress <CTRL>-D shortcut to exit KASH
-f        force line editing control sequences
-g        verbose gasman
```

```
-h        display this help
-l <dir>  use <dir> as library root path
-m <mem>  allocate <mem> bytes of memory at startup
-n        disable line editing control sequences
-p        start GAP package mode for output (deprecated)
-q        start in quiet mode
-x <cols>  use <cols> columns for output
-y <rows>  use <rows> rows for output
```

Run the Komputational Algebraic Number Theory software. If FILE is not omitted, load FILE after startup.

2.2 First Steps

A simple calculation with KASH3 is as easy as one can imagine. You type the problem just after the prompt, terminate it with a semicolon and then pass the problem to the program with the `return` key. For example, to multiply the difference between 9 and 7 by the sum of 5 and 6, that is to calculate `(9 - 7)*(5 + 6)`, you type exactly this last sequence of symbols followed by `;` and `return`. If you omitted the semicolon at the end of the line but had already typed `return`, then KASH3 has read everything you typed, but does not know that the command is complete. The program is waiting for further input and indicates this with a partial prompt. This little problem is solved by simply typing the missing semicolon on the next line of input. Then the result is printed and the normal prompt returns. Whenever you see this partial prompt and you cannot decide what KASH3 is still waiting for, then you have to type semicolons until the normal prompt returns.

Example

```
kash% (9 - 7) * (5 + 6);
22
Time: 0.000285 s
```

2.2.1 Line Editing

KASH3 allows you to edit the current input line with a number of editing commands. Those commands are accessible by control characters. The case of the characters does not matter. Below we list the most important commands for cursor movement, deleting text and yanking text.

<Ctrl>-A: move to beginning of line

<Ctrl>-B: move backward one character

<Ctrl>-D: delete character under cursor

<Ctrl>-E: move to end of line

<Ctrl>-F: move forward one character

<Ctrl>-H: delete previous character

<Ctrl>-K: delete from cursor to end of line

<Ctrl>-Y: insert (yank) what you've deleted

<Ctrl>-L: insert last input line before current character

<Ctrl>-P: redisplay the last input line, another.

<Ctrl>-N: like `<Ctrl>-P` but goes the other way round through the history.

You can also use the cursor up/down keys to scroll through the history.

The Tab key looks at the characters before the cursor, interprets them as the beginning of an identifier and tries to complete this identifier.

2.2.2 History

```
History()
```

Display the history of commands entered at the prompt

Example

```
kash% x_z3 := pAdicRing(3,30);
3-adic ring mod 3^30
Time: 0.000372 s
kash% x_R  := Extension(x_z3,4);
Unramified extension defined by the polynomial X^4 + 2*X^3 + 2
 over 3-adic ring mod 3^30
Time: 0.003282 s
kash% x_Ry := PolynomialRing(x_R);
Univariate Polynomial Ring over Unramified extension defined by the polynomial\
 X^4 + 2*X^3 + 2
 over pAdicRing(3, 30)
Time: 0.000305 s
kash% x_S  := Extension(x_R, x_Ry.1^2+3*x_Ry.1+3);
Totally ramified extension defined by the polynomial X^2 + 3*X + 3
 over Unramified extension defined by the polynomial X^4 + 2*X^3 + 2
 over 3-adic ring mod 3^30
Time: 0.000649 s
kash% History();
```

2.3 Inline Help

For inline help, `?` is a valuable tool. A single question mark followed by the name of a function or type or statement or keyword causes the description of the identifier found in the reference manual to appear on the screen. If a list of all functions beginning with a particular string is desired, use `?^` followed by the string to match. If no match is found, the response `No matches. Maybe try ?*<string>.` will be displayed. `?*` followed by something searches for all of its occurences in the documentation.

A query can also be started by ending a line with `?` and its modifier instead of starting it with `?`. In both cases the function `Help` with the entered line (without the ?) as a parameter.

2.3.1 Help

`?<func>`

Shows all possible signatures of a function.

`?<type>`

Gives information about a type.

`?<stmnt>`

Displays information on a statement.

`?<string>`

For some keywords help texts are available.

`?^<string>`

Lists all functions, operations, types, statements, and keywords beginning with string.

`?!<string>`

Lists all functions, operations, types, statements, and keywords which contain string.

`?*<string>`

Searches the complete documentation for string and returns all matching entries.

`?<elt-ord^rat>`

Each entry in the documentation is identified by a number. The corresponding entry is displayed.

`?(<type>)`

Finds all functions where <type> matches one type of the input signature.

`?-><type>`

Finds all functions where <type> matches the type of a return value.

Queries can be restricted to certain kinds of objects by appending

|CONSTANT or |FUNCTION or |KEYWORD or |OPERATION or |STATEMENT or |TYPE.

2.4 Operations

In an expression like `(9 - 7) * (5 + 6)` the constants `5`, `6`, `7`, and `9` are being composed by the operators `+`, `*` and `-` to result in a new value.

There are three kinds of operators in KASH3, arithmetical operators, comparison operators, and logical operators.KASH3 knows a precedence between operators that may be overridden by parentheses.

You have already seen that it is possible to form the sums, differences, and products. The remaining arithmetical operators are exponentiation `^` and `mod`.

A comparison result is a boolean value. Integers, rationals and real numbers are comparable via `=`, <, `<=`, `>=`, > and <>; algebraic elements, ideals, matrices and complex numbers can be compared via `=` and <>. Membership of an element in a structure can be tested eith `in`.

The boolean values `TRUE` and `FALSE` can be manipulated via logical operators, i.e., the unary operator `not` and the binary operators `and` and `or`.

Example

```
kash% 12345/25;
2469/5
Time: 0.00032 s

kash% 7^69;
205005145156954906122290109080958673914396262484637238056O7
Time: 0.000286 s
```

2.5 Variables and Assignments

Values may be assigned to variables. A variable enables you to refer to an object via a name. The assignment operator is `:=`. Do not confuse the assignment operator `:=` with the single equality sign `=` which in KASH3 is only used for the test of equality.

After an assignment, the assigned value is echoed on the next line. The printing of the value of a statement may be in every case prevented by typing a double semicolon.

After the assignment, the variable evaluates to that value if evaluated. Thus it is possible to refer to that value by the name of the variable in any situation.

A variable name may be sequences of letters and digits containing at least one letter. For example `abc` and `a1b2` are valid names. Since KASH3 distinguishes upper and lower case, `a` and `A` are different names. Keywords such as `quit` must not be used as names.

Example

```
kash% a:=32233;
32233
Time: 0.000257 s
kash%  A:=76;
76
Time: 0.000253 s
kash%  a+A;
32309
```

```
Time: 0.000256 s
```

2.5.1 last

```
last -> <any>
```

Whenever KASH3 returns a value by printing it on the next line, this value is assigned to the variable `last`. Moreover there are variables `last2` and `last3`, guess their values!

Example

```
kash% a:= (9 - 7) * (5 + 6);
22
Time: 0.000273 s
kash% a*(a+1);
506
Time: 0.000257 s
kash% a:=last;
506
Time: 0.000267 s
```

2.5.2 Constants

For convenience some constants are predefined in KASH. These constants cannot be overwritten. Some constants, namely `E` and `PI`. change their value with the global precision `Precision()`. By convention constants are written in capitals.

Enter `?*.|CONSTANT` to see a list of all constants.

2.6 Integers and Rationals

KASH3 integers are entered as a sequence of digits optionally preceded by a `+` sign for positive integers or a `-` sign for negative integers. In KASH3, the size of integers is only limited by the amount of available memory. The binary operations `+`, `-`, `*`, / allow combinations of arguments from the integers, the rationals, and real and complex fields; automatic coercion is applied where necessary.

Since integers are naturally embedded in the field of real numbers, all real functions are applicable to integers.

Rationals can be created by simply typing in the fraction using the symbol / to denote the division bar. The value is not converted to decimal form, however the reduced form of the fraction is found. Similarly all real functions are applicable to rationals.

Example

```
kash% 4/6;
2/3
Time: 0.000309 s
```

2.6.1 Factorization

```
Factorization(<elt-ord^rat> n [, optargs]) -> <seq()>, <elt-ord^rat>,
<seq()>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 3)
<elt-alg^boo> Proof
<elt-ord^rat> Bases
<elt-ord^rat> TrialDivisionLimit
<elt-ord^rat> PollardRhoLimit
<elt-ord^rat> ECMLimit
<elt-ord^rat> MPQSLimit
```

Attempt to find the prime factorization of the value of n; the factorization is returned, together with the appropriate unit and the sequence of composite factors which could not be factored.

Example

```
kash% Factorization(4352);
[ <2, 8>, <17, 1> ], extended by:
  ext1 := 1,
  ext2 := Unassign
Time: 0.000909 s
```

2.6.2 GCD

```
GCD(<elt-ord^rat> x, <elt-ord^rat> y) -> <elt-ord^rat>
```

The greatest common divisor of `x` and `y`.

Example

```
kash% GCD(25,5);
5
Time: 0.00028 s
```

2.6.3 LCM

```
LCM(<elt-ord^rat> x, <elt-ord^rat> y) -> <elt-ord^rat>
```

The least common multiple of the integers whose factorization tuples are A and B, represented as a factorization tuple.

2.6.4 Div

```
Div(<elt-ord^rat> x, <elt-ord^rat> y) -> <elt-ord^rat>
```

The Euclidean quotient of x by y.

2.6.5 mod

```
<elt-ord^rat> x mod <elt-ord^rat> y -> <elt-ord^rat>
```

A canonical representative of a (belonging to order O) in the quotient O/I.

2.6.6 IsPrime

```
IsPrime(<elt-ord^rat> n [, optargs]) -> <elt-alg^boo>
```

with optional arguments

```
<elt-alg^boo> Proof
```

True iff the integer n is prime.

2.6.7 NextPrime

```
NextPrime(<elt-ord^rat> n [, optargs]) -> <elt-ord^rat>
```

with optional arguments

```
<elt-alg^boo> Proof
```

The least prime number greater than n.

Example

```
kash% NextPrime(100);
101
Time: 0.00029 s
```

2.7 Reals and Complex

Real numbers can only be stored in the computer effectively in the form of approximations. KASH3 provides a number of facilities for calculating with such approximations to (at least) a given, but arbitrary, precision. Real numbers have a default precision of 20. One can change the precision to arbitrary `n` (See example below!).

KASH3 provides the following real functions; refer to the reference manual for detailed descriptions and examples.

In KASH3, complex numbers have the same precision as real numbers. This precision can be modified by calling the `Precision` function (see example). A complex number can be designated using the function Element, which requires two real arguments (See example).

Most real functions can be applied to complex numbers. Additionally, KASH3 provides the several complex functions.

Example

```
kash% Precision();
30
Time: 0.00071 s

kash% R;
Real field of precision 30
Time: 0.000272 s

kash% Precision(40);
40
Time: 0.00116699999999999999999999999947 s

kash% C;
Complex field of precision 30
Time: 0.000272 s

kash% z := 1+2*I;
1.00000000000000000000000000000 + 2.00000000000000000000000000000*I
Time: 0.000394 s
kash%   # a complex number,
kash% z*z;
-3.00000000000000000000000000000 + 4.00000000000000000000000000000*I
Time: 0.000365 s
```

2.7.1 Precision

```
Precision() -> <elt-ord^rat> prec
```

Returns the global precision for real and complex computations in the shell.

2.7.2 Cos

```
Cos(<elt-fld^rea> x) -> <elt-fld^rea>
```

The cosine of f.

2.7.3 Exp

```
Exp(<elt-fld^rea> x) -> <elt-fld^rea>
```

Exponential function (to the base e) of x.

Example

```
kash% Exp(2*PI);
535.4916555247647365030493295890
Time: 0.000484 s
```

2.7.4 Log

```
Log(<elt-fld^com> x) -> <elt-fld^com>
```

The logarithm of x (to the natural base e).

Example

```
kash% Log(3+4*I);
1.6094379124341003746007593332 + 0.92729521800161223242851246292*I
Time: 0.000801 s
kash% Log(I);
1.5707963267948966192313216916*I
Time: 0.000303 s
```

2.7.5 Sqrt

```
Sqrt(<elt-ord^rat> x) -> <elt-fld^com>
```

The square root of x.

2.7.6 Floor

```
Floor(<elt-fld^rat> x) -> <elt-ord^rat>
```

The floor of x: the largest integer less than or equal to x.

Example

```
kash% Floor(9/5);
1
Time: 0.000304 s
kash%  Floor(3/5);
0
Time: 0.000265 s
```

2.7.7 Argument

```
Argument(<elt-fld^com> e) -> <elt-fld^rea>
```

The argument in radians of a complex number e.

Example

```
kash% x_c := 3+I;
3.00000000000000000000000000000 + 1.00000000000000000000000000000*I
Time: 0.000359 s
kash% Argument(x_c);
0.321750554396642193401404614359
Time: 0.000439 s
```

2.7.8 Imaginary

```
Imaginary(<elt-fld^com> e) -> <elt-fld^rea>
```

The imaginary part of e.

Example

```
kash% Imaginary(I);
1.00000000000000000000000000000
Time: 0.000294 s
```

2.8 Lists

A `list` is a collection of objects separated by commas and enclosed in brackets.

Example

```
kash% primes:= [2, 3, 5, 7, 11, 13, 17, 19];
```

```
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
Time: 0.000258 s
kash% # a list containing 8 elements
kash% Append_(primes, [23, 29]);
SUCCESS
Time: 0.000272 s
kash%  # append two numbers to the list
kash% primes[5];
11
Time: 0.000268 s
kash%  # the 5th element in the list
kash% primes[9] := 77;
77
Time: 0.000267 s
kash%  # set the 9th list entry to 77

kash% L := [1,2,TRUE,3/4, X^2+2];
[ 1, 2, TRUE, 3/4, X^2 + 2 ]
Time: 0.000418 s
```

2.8.1 Add

`Add(<list> L, <any> a) -> <list>`

Add `a` to `L` by assigning `a` at the next position beyond the scope of `L`.

Note: This function returns the list created by the addition but does not affect `L`.

Example

```
kash% L:=[1,,3,4];
[ 1,, 3, 4 ]
Time: 0.000266 s
kash% Add(L,5);
[ 1,, 3, 4, 5 ]
Time: 0.00074 s
kash%  L;
[ 1,, 3, 4 ]
Time: 0.00026 s
```

2.8.2 Add_

`Add_(<list> L, <any> a)`

Add `a` to `L` by assigning `a` at the next position beyond the scope of `L`.

Note: This function works by side effect and returns VOID.

Example

```
kash% L:=[1,,3,4];
[ 1,, 3, 4 ]
Time: 0.000275 s
kash% Add_(L,5);
SUCCESS
Time: 0.00072 s
kash%  L;
[ 1,, 3, 4, 5 ]
Time: 0.000256 s
```

2.8.3 Append_

```
Append_(<list> L, <list> a)
```

Modify `L` by adding every element of the list `a` to `L`.

Note: `L` is modified by side-effect.

Note: Unassigned values in `a` have no effect whatsoever.

Example

```
kash% L:=[1,,3,4];
[ 1,, 3, 4 ]
Time: 0.000272 s
kash% Append_(L,[5,6]);
SUCCESS
Time: 0.000272 s
kash%  L;
[ 1,, 3, 4, 5, 6 ]
Time: 0.000258 s
```

2.8.4 Apply

```
Apply(<list> l, <func> f) -> <list> r
```

Return the list where every member `b` of `l` is replaced by `f` applied to `b`.

Example

```
kash% l:=[1,2,3,4];
[ 1, 2, 3, 4 ]
Time: 0.000278 s
kash% Apply(l,i->3*i);
```

```
[ 3, 6, 9, 12 ]
Time: 0.000743 s
kash%  l;
[ 1, 2, 3, 4 ]
Time: 0.000251 s
```

```
kash% l:=[1,2,3,4];
[ 1, 2, 3, 4 ]
Time: 0.000272 s
kash% Apply(l,IsEven);
[ FALSE, TRUE, FALSE, TRUE ]
Time: 0.000759 s
kash%  l;
[ 1, 2, 3, 4 ]
Time: 0.000259 s
```

2.9 Ranges

A range is a finite sequence of integers which is another special kind of list. A range is described by its minimum (the first entry), its second entry and its maximum, separated by a comma resp. two dots and enclosed in brackets. In the usual case of an ascending list of consecutive integers the second entry may be omitted.

Example

```
kash% L := [1..100];
[ 1 .. 100 ]
Time: 0.000261 s
```

```
kash% L := [2,4..100];
[ 2, 4 .. 100 ]
Time: 0.000267 s
```

2.10 Sequences

A sequence is a list containing elements from the same `universe`. Writing different types in a sequence is not allowed.

Example

```
kash% Sequence([1,5,7]);
[ 1, 5, 7 ]
Time: 0.000278 s
```

```
kash% Sequence([1,Q]);
Error, Sequence: cannot create a sequence from given elements
```

2.11 Tuples

A tuple is a list containing any elements. A tuple is an element of a Cartesian product. The types of the elements of the factors of this product may be specified. Once a tuple is created, insertions are possible only if the type of the new element is the same as the type of the element that will be replaced.

Example

```
kash% x_a:=Tuple([1,3.2,"text"]);
<1, 3.20000000000000000000000000000, text>
Time: 0.000289 s
kash% Type(x_a);
tup()
Time: 0.000264 s
kash% Parent(x_a);
Cartesian Product<Integer Ring, Real field of precision 30, String structure>
Time: 0.000289 s
```

2.12 Polynomials

At the moment, KASH3 can only handle univariate polynomials. The polynomial ring `ZX` over the integers and its indeterminate `X=ZX.1` are predefined constants. Use the `Evaluation` routine to calculate the value of a polynomial when the variable `X` is substituted by certain values. To create the polynomial algebra `S[x]` with coefficients from a designated ring `S`, the routine `PolynomialAlgebra` should be used. This routine requires one argument, namely the coefficient ring of the polynomial algebra.

Recall that `Q` is the predefined constant for the ring .

Note that KASH3 always declares the variable by `P.1`, if `P` is the name of your polynomial algebra.

Example

```
kash% f := X^3 + X + 1;
X^3 + X + 1
Time: 0.000389 s
kash%  # a polynomial over Z
kash% f+f;
2*X^3 + 2*X + 2
Time: 0.000284 s
```

```
kash% f*f;
X^6 + 2*X^4 + 2*X^3 + X^2 + 2*X + 1
Time: 0.000322 s
kash%  Evaluate(f,10);
1011
Time: 0.000308 s

kash% Qx := PolynomialAlgebra(Q);
Univariate Polynomial Ring over Rational Field
Time: 0.000292 s
kash%  # Univariate Polynomial Ring over Rational Field
kash% Qx.1^5+7/3;
Qx.1^5 + 7/3
Time: 0.000436 s
```

2.13 Matrices

To construct a matrix first we need a ring from which are the coefficients. For example Q,R and C are predefined for the rational, real and complex numbers. We can also build matrices over function or number fields. Then we need also the number of rows and columns and finitely a list consisting of the entries.We can compute the kernel N of a matrix M (as a matrix which islinear map in the kernel of M) or the Nullspace of N.

Example

```
kash% k   := RationalFunctionField(FiniteField(5,2));
Univariate rational function field over GF(5^2)
Variables: $.1
Time: 0.001963 s
kash% P   := PolynomialAlgebra(k);
Univariate Polynomial Ring over Univariate rational function field over GF(5^2\
)
Time: 0.000266 s
kash% F   := FunctionField(P.1^3 + k.1^3 + k.1 + 1);
Algebraic function field defined over Univariate rational function field over \
GF(5^2) by
P.1^3 + $.1^3 + $.1 + 1
Time: 0.290514 s
kash% M   := Matrix(F, 2,2, [ F.1+k.1, 2, F.1+k.1,2]);
[F.1 + $.1 2]
[F.1 + $.1 2]
Time: 0.00229 s

kash% N   := Matrix(Q, 2,3, [1,2,3,5,0,1]);
[1 2 3]
```

```
[5 0 1]
Time: 0.001599 s
kash% KernelMatrix(N);
Matrix with 0 rows and 2 columns
Time: 0.000462 s
kash% w   := KernelMatrix(Transpose(N));
[ 1  7 -5]
Time: 0.000355 s
kash% N*Transpose(w);
[0]
[0]
Time: 0.000316 s
kash% v   := Transpose(Matrix(Q, 3,1, [1,2,3]));
[1 2 3]
Time: 0.00149 s
kash% Solution(N,v);
[1 0], extended by:
  ext1 := Vector space of degree 2, dimension 0 over Rational Field
Time: 0.000442 s


kash% T   := Matrix(Q, 2,2,[1,2,1,2]);
[1 2]
[1 2]
Time: 0.001573 s
kash% KernelMatrix(T);
[ 1 -1]
Time: 0.000528 s
```

2.13.1 SetEntry

```
SetEntry(<elt-mdl^mat> m, <elt-ord^rat> row, <elt-ord^rat> col, <elt-rng>
val) -> <elt-mdl^mat> m
```

Sets `m[row][col]` to `val`, if `val` is convertible to an element of the ring over which `m` is defined

Example

```
kash% m :=Matrix(Z,2,2,[1,2,3,4]);
[1 2]
[3 4]
Time: 0.001543 s
kash% SetEntry(m,2,2,10);
[ 1  2]
[ 3 10]
Time: 0.000299 s
```

2.13.2 HermiteForm

```
HermiteForm(<elt-alg^mat> X [, optargs]) -> <elt-alg^mat>, <elt-alg^mat>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
<string> Al ("LLL" or "Sort")
<elt-alg^boo> Optimize
<elt-alg^boo> Integral
```

The (row) hermite form E of the matrix X over a Euclidean Domain, together with an invertible transformation matrix T such that E = T * X.

Example

```
kash%  x_M:= Matrix(Z,3,3,[3,2,1,2,5,3,1,0,0]);
[3 2 1]
[2 5 3]
[1 0 0]
Time: 0.001553 s
kash% HermiteForm(x_M);
[1 0 0]
  [0 1 0]
  [0 0 1], extended by:
  ext1 := [ 0  0  1]
    [ 3 -1 -7]
    [-5  2 11]
Time: 0.000377 s
```

Chapter 3

Algebraic Structures

In this section we show how to generate matrices, modules or groups.

3.1 Linear Algebra

3.1.1 NullSpace

```
NullSpace(<elt-mdl^mat> X) -> <mdl^vec>
```

3.1.2 Basis

```
Basis(<mdl^vec> M) -> <seq()>
```

The basis of the algebraic field F (as elements of R if given).

Example

```
kash% x_RS := RSpace(Z,3);
Full RSpace of degree 3 over Integer Ring
Time: 0.000323 s
kash% Basis(x_RS);
[
(1 0 0),
(0 1 0),
(0 0 1)
]
Time: 0.000297 s
```

3.2 Abelian Groups

In KASH3 we can work with abelian and symmetric groups. We can for example generate abelian groups and compute the direct sum. For elements of a symmetric group we can make use of cycles.

Example

```
kash% G   := AbelianGroup([2,3,3]);
Abelian Group isomorphic to Z/3 + Z/6
Defined on 3 generators
Relations:
2*G.1 = 0
3*G.2 = 0
3*G.3 = 0
Time: 0.003693 s
kash% H   := AbelianGroup([5,3,3,7]);
Abelian Group isomorphic to Z/3 + Z/105
Defined on 4 generators
Relations:
5*H.1 = 0
3*H.2 = 0
3*H.3 = 0
7*H.4 = 0
Time: 0.004243 s
kash% GH  := DirectSum(H,G);
Abelian Group isomorphic to Z/3 + Z/3 + Z/3 + Z/210
  Defined on 4 generators
  Relations:
  3*GH.1 = 0
  3*GH.2 = 0
  3*GH.3 = 0
  210*GH.4 = 0, extended by:
  ext1 := Mapping from: grp^abl: H to grp^abl: GH,
  ext2 := Mapping from: grp^abl: G to grp^abl: GH,
  ext3 := Mapping from: grp^abl: GH to grp^abl: H,
  ext4 := Mapping from: grp^abl: GH to grp^abl: G
Time: 0.00064 s
kash% FG1 := FreeAbelianGroup(3);
Abelian Group isomorphic to Z + Z + Z
Defined on 3 generators (free)
Time: 0.000264 s
kash% FG2 := FreeAbelianGroup(5);
Abelian Group isomorphic to Z (5 copies)
Defined on 5 generators (free)
Time: 0.000254 s
```

```
kash% G1G2:= DirectSum(FG1,FG2);
Abelian Group isomorphic to Z (8 copies)
  Defined on 8 generators (free), extended by:
  ext1 := Mapping from: grp^abl: FG1 to grp^abl: G1G2,
  ext2 := Mapping from: grp^abl: FG2 to grp^abl: G1G2,
  ext3 := Mapping from: grp^abl: G1G2 to grp^abl: FG1,
  ext4 := Mapping from: grp^abl: G1G2 to grp^abl: FG2
Time: 0.000268 s
```

3.2.1 FreeAbelianGroup

```
FreeAbelianGroup(<elt-ord^rat> n) -> <grp^abl>
```

Example

```
kash%  FreeAbelianGroup(12);
Abelian Group isomorphic to Z (12 copies)
Defined on 12 generators (free)
Time: 0.000306 s
```

3.2.2 Order

```
Order(<elt-grp^abl> x) -> <elt-ord^rat>
```

The order of the group element x, or zero if it has infinite order.

Example

```
kash% x_A := AbelianGroup([2,3]);
Abelian Group isomorphic to Z/6
Defined on 2 generators
Relations:
2*x_A.1 = 0
3*x_A.2 = 0
Time: 0.000851 s
kash%  Order(x_A.1);
2
Time: 0.000329 s
```

3.3 Modules and Lattices

In KASH3 we can construct for example modules in number fields over orders. Lattices are represented as matrices. We can compute the Gram matrix and a LLL-reduced basis of a lattice.

Example

```
kash% o2 := MaximalOrder(X^2-2);
Maximal Equation Order with defining polynomial X^2 - 2 over Z
Time: 0.00085 s
kash% Po2:= PolynomialAlgebra(o2);
Univariate Polynomial Ring over o2
Time: 0.000268 s
kash% O  := EquationOrder(Po2.1^3-3);
Equation Order with defining polynomial X^3 + [-3, 0] over o2
Time: 0.021048 s
kash% M  := Module(O);
Module over Maximal Equation Order with defining polynomial X^2 - 2 over Z
   generated by:
  Principal Ideal of o2
  Generator:
  _EF.1 * ( _EF.1 0 0 )
  Principal Ideal of o2
  Generator:
  _EF.1 * ( 0 _EF.1 0 )
  Principal Ideal of o2
  Generator:
  _EF.1 * ( 0 0 _EF.1 )
   in echelon form:
  Principal Ideal of o2
  Generator:
  _EF.1 * ( _EF.1 0 0 )
  Principal Ideal of o2
  Generator:
  _EF.1 * ( 0 _EF.1 0 )
  Principal Ideal of o2
  Generator:
  _EF.1 * ( 0 0 _EF.1 ) , extended by:
  ext1 := Mapping from: mdl^ded: M to ord^num: O
    using
    [_EF.1 0 0]
    [0 _EF.1 0]
    [0 0 _EF.1]


Time: 0.000718 s
kash% N  := Matrix(Q,3,3,[1/2,3,2,3,0,1,2,9/2,2]);
[1/2   3   2]
[  3   0   1]
[  2 9/2   2]
Time: 0.001599 s
kash% G  := GramMatrix(N);
```

```
[ 53/4   7/2  37/2]
[  7/2    10     8]
[ 37/2     8 113/4]
Time: 0.000288 s
kash% LLL(N);
[ 3/2  3/2    0]
  [ 3/2 -3/2    1]
  [   1 -3/2   -2], extended by:
  ext1 := [-1  0  1]
    [ 1  1 -1]
    [-2  0  1],
  ext2 := 3
Time: 0.000364 s
```

3.3.1 Module

```
Module(<ord^num> O [, optargs]) -> <mdl^ded>, <map()>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
```

The relative order O as a module over its coefficient ring.

Example

```
kash% x_o := MaximalOrder(X^5 - 2);
Maximal Equation Order with defining polynomial X^5 - 2 over Z
Time: 0.002993 s
kash%  x_oy := PolynomialAlgebra(x_o);
Univariate Polynomial Ring over x_o
Time: 0.00027 s
kash% x_eo := EquationOrder(x_oy.1^2 + 7);
Equation Order with defining polynomial X^2 + [7, 0, 0, 0, 0] over x_o
Time: 0.008094 s
kash% Module(x_eo);
Module over Maximal Equation Order with defining polynomial X^5 - 2 over Z
   generated by:
  Principal Ideal of x_o
  Generator:
  _IJ.1 * ( _IJ.1 0 )
  Principal Ideal of x_o
  Generator:
  _IJ.1 * ( 0 _IJ.1 )
   in echelon form:
  Principal Ideal of x_o
  Generator:
  _IJ.1 * ( _IJ.1 0 )
  Principal Ideal of x_o
  Generator:
  _IJ.1 * ( 0 _IJ.1 ) , extended by:
  ext1 := Mapping from: mdl^ded: _MP to ord^num: x_eo
    using
    [_IJ.1 0]
    [0 _IJ.1]


Time: 0.00068 s
```

Chapter 4

Number Fields

We call `alpha in C` an algebraic integer if there exists a monic irreducible polynomial `f(x) in Z[x]` with `f(alpha) = 0`. An algebraic number field `F` is a finite extension of the field of rationals `Q`. There always exists an algebraic integer `rho in C` such that `F = Q(rho)`. The set of algebraic integers in `F` forms a ring which is denoted by `O = O_F`. An order `o` in `F` is a unital subring of `O` which, as a `Z-module`, is finitely generated and of rank `[F:Q]`. Of course, `O` is an order which we call the maximal order of `F` (see Orders of Number Fields for details). In KASH3, any computations in an algebraic number field `F` are performed with respect to a certain order in `F`.

Example

```
kash% f := X^5 + 4*X^4 - 56*X^2 -16*X + 192;
X^5 + 4*X^4 - 56*X^2 - 16*X + 192
Time: 0.000539 s
kash% # we want to do arithmetic in the field F = Q(rho),
kash% # where 'rho' is a root of irreducible polynomial f
kash% o := EquationOrder(f);
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over\
 Z
Time: 0.001345 s
```

4.1 Orders of Number Fields

KASH3 makes it easy to compute in arbitrary orders of number fields. Given the minimal polynomial `f` of an algebraic integer `rho` one obtains the equation order `Z[rho]` easily as `Z[x]/(rho)`. In order to compute a maximal order `O` of the number field `F=Q(rho)`, one has to compute an integral bases of `F`. Maximal orders are not given by polynomials but a transformation matrix, which transforms a power basis `(1,rho,...,rho^4)` to a basis `(w_1,...,w_5)` of the maximal order. The `MaximalOrder` function computes an integral basis `(w_1,...,w_5)`. Using the `Element` function one can enter algebraic numbers with respect to this basis.

Example

```
kash% f := X^5 + 4*X^4 - 56*X^2 -16*X + 192;
X^5 + 4*X^4 - 56*X^2 - 16*X + 192
Time: 0.000528 s
kash% o := EquationOrder(f);
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over\
 Z
Time: 0.001328 s
kash% O := MaximalOrder(o);
Maximal Order of o
Time: 0.040288 s
kash%  # maximal order of 'o'
kash% w1 := Element(O,[1,0,0,0,0]);
[1, 0, 0, 0, 0]
Time: 0.000612 s
kash% w2 := Element(O,[0,1,0,0,0]);
[0, 1, 0, 0, 0]
Time: 0.000322 s
kash% w3 := Element(O,[0,0,1,0,0]);
[0, 0, 1, 0, 0]
Time: 0.000299 s
kash% w4 := Element(O,[0,0,0,1,0]);
[0, 0, 0, 1, 0]
Time: 0.000297 s
kash% w5 := Element(O,[0,0,0,0,1]);
[0, 0, 0, 0, 1]
Time: 0.000307 s
```

4.1.1 EquationOrder

```
EquationOrder(<elt-alg^pol> f [, optargs]) -> <ord^num>
```

with optional arguments

```
<elt-alg^boo> Check
```

The equation order defined by the monic irreducible integer polynomial f.

Example

```
kash% x_k:=PolynomialAlgebra(Q);
Univariate Polynomial Ring over Rational Field
Time: 0.000293 s
kash%  x_f:=x_k.1^2 - 2;
x_k.1^2 - 2
```

```
Time: 0.000416 s
kash% x_o:= EquationOrder(x_f);
Equation Order with defining polynomial X^2 - 2 over Z
Time: 0.000489 s
```

4.1.2 Signature

```
Signature(<ord^num> O [, optargs]) -> <elt-ord^rat>, <elt-ord^rat>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
```

The signature (number of real embeddings and pairs of complex embeddings) of K.

4.1.3 MaximalOrder

```
MaximalOrder(<ord^num> O [, optargs]) -> <ord^num>
```

with optional arguments

```
<any> Discriminant
<seq()> Ramification
<string> Al ("Auto" or "Round2" or "Round4")
```

The maximal order of O.

Example

```
kash% x_o:=EquationOrder(X^4+7*X+3);
Equation Order with defining polynomial X^4 + 7*X + 3 over Z
Time: 0.000829 s
kash% MaximalOrder(x_o);
Maximal Equation Order with defining polynomial X^4 + 7*X + 3 over Z
Time: 0.003231 s
```

4.1.4 Discriminant

```
Discriminant(<ord^num> O) -> <any>
```

The discriminant of the order O in an algebraic field.

Example

```
kash% x_o := EquationOrder(X^3+3*X^2+3);
Equation Order with defining polynomial X^3 + 3*X^2 + 3 over Z
Time: 0.000837 s
kash%  Discriminant(x_o);
-567
Time: 0.000326 s
```

4.1.5 Element

```
Element(<any> R, <nof(any)> a) -> <elt-any>
```

The element of R specified by a

Example

```
kash% o := MaximalOrder(X^3-77);
Maximal Equation Order with defining polynomial X^3 - 77 over Z
Time: 0.005283 s
kash% Element(o,[2, 1, 0]);
[2, 1, 0]
Time: 0.000462 s
```

4.2 Ideals

In KASH3 an ideal is an object which is, like an algebraic number, defined over a certain order. There are many ways to create an ideal in KASH3. The most basic one is to use the function Ideal. The sum and the difference of two ideals are the smallest ideals which contain both operands. The product of two ideals is the ideal formed by all products of an element of the first ideal with an element of the second one. KASH3 can also handle fractional ideals (a fractional ideal is an integral ideal divided by a certain non–zero integer). This feature allows ideals to be inverted if the underlying order is the maximal one (remember that in a Dedekind ring the fractional ideals form a group under multiplication).

4.2.1 Ideal

```
Ideal(<any> R, <nof(any)> a) -> <elt-ids>
```

The ideal of R specified by a

Example

```
kash% o := MaximalOrder(X^3-77);
```

```
Maximal Equation Order with defining polynomial X^3 - 77 over Z
Time: 0.005271 s
kash% Ideal(o,5,[2, 1, 0]);
Ideal of o
Two element generators:
[5, 0, 0]
[2, 1, 0]
Time: 0.000509 s
```

4.2.2 Factorisation

```
Factorisation(<elt-ids^fra/ord^num> I) -> <seq()>
```

The factorization of I into prime ideals.

4.2.3 IsPrincipal

```
IsPrincipal(<elt-ids^fra/ord^num> I [, optargs]) -> <elt-alg^boo>,
<elt-fld^fra>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
```

True and a generator if the fractional ideal I is principal, false otherwise.

4.3 Class Groups

The task of computing the ideal class group is solved by invoking the `ClassGroup` function. As a result an abelian group `G` that the class group is isomorphic to, extended the by the map from `G` into the set of Ideals in `O`. Using the `ClassGroupCyclicFactorGenerators` routine one can obtain a list of ideals representing generators of the cyclic factors of ideal class group. Notice that you must first compute the class group before you can use the `ClassGroupCyclicFactorGenerators` routine.

The Minkowski bound is used to compute a class group in KASH3. This bound always guarantees correct results. However, when the field discriminant is `large`, the Minkowski bound causes very time consuming computations requiring a large amount of memory. You can pass a smaller bound to the `OrderClassGroup` function calling it with optional arguments.

Example

```
kash% f := X^5 + 4*X^4 - 56*X^2 -16*X + 192;
X^5 + 4*X^4 - 56*X^2 - 16*X + 192
Time: 0.000524 s
```

```
kash% o := EquationOrder(f);
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over\
 Z
Time: 0.001351 s
kash% O := MaximalOrder(o);
Maximal Order of o
Time: 0.040208 s
kash% Cl := ClassGroup(O);
Abelian Group isomorphic to Z/6
  Defined on 1 generator
  Relations:
  6*Cl.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: Cl to ids/ord^num: _AQ
Time: 0.33001 s
```

4.3.1 ClassGroup

```
ClassGroup(<ord^num> O [, optargs]) -> <grp^abl>, <map()>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
<string> Proof ("Bound" or "Current" or "Full" or "GRH" or "Subgroup")
<elt-ord^rat> Bound
<elt-alg^boo> Enum
<elt-fld^rea> FactorBasisBound
<elt-fld^rea> ProofBound
<elt-ord^rat> ExtraRelations
<string> Al ("Automatic" or "ReducedForms" or "Relations" or "Shanks")
```

The class group of the ring of integers O.

4.3.2 ConditionalClassGroup

```
ConditionalClassGroup(<ord^num> O [, optargs]) -> <grp^abl>, <map()>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
```

The class group of the ring of integers O, assuming GRH.

4.4 Unit Groups

KASH3 provides several functions dealing with the units of an order. In order to compute a system of fundamental units the `UnitGroup` function should be used. It returns the unit group as a finitely generated abelian group and a map from this abelian group to the order.The function `TorsionUnitGroup` returns the torsion subgroup of the unit group

Example

```
kash% f := X^5 + 4*X^4 - 56*X^2 -16*X + 192;
X^5 + 4*X^4 - 56*X^2 - 16*X + 192
Time: 0.000542 s
kash% o := EquationOrder(f);
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over\
 Z
Time: 0.00136 s
kash% O := MaximalOrder(o);
Maximal Order of o
Time: 0.04091 s
kash% U := UnitGroup(O);
Abelian Group isomorphic to Z/2 + Z + Z
  Defined on 3 generators
  Relations:
  2*U.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: U to ord^num: O
Time: 0.157717 s
kash% Apply(x->U.ext1(x),List(Generators(U)));
[ [1, -1, 0, 0, 0], [1, 1, 0, 0, 0], [-1, 0, 0, 0, 0] ]
Time: 0.001152 s
```

4.4.1 UnitGroup

```
UnitGroup(<ord^num> O [, optargs]) -> <grp^abl>, <map()>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
<string> Al ("Automatic" or "ClassGroup" or "ContFrac" or "Dirichlet" or "Mixed" or
"Relation" or "Short")
```

The unit group of an order or a number field.

Example

```
kash% x_o := MaximalOrder(X^4+X^2-1);
Maximal Equation Order with defining polynomial X^4 + X^2 - 1 over Z
Time: 0.002544 s
kash% UnitGroup(x_o);
Abelian Group isomorphic to Z/2 + Z + Z
```

```
  Defined on 3 generators
  Relations:
  2*_PX.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: _PX to ord^num: x_o
Time: 0.342136 s
```

4.4.2 TorsionUnitGroup

```
TorsionUnitGroup(<ord^num> O [, optargs]) -> <grp^abl>, <map()>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
```

The torsion part of the unit group of the number field or order.

Chapter 5

Global Function Fields

In this section the basic steps necessary for the creation of an algebraic function field and for doing simple operations are explained. The concepts are quite similar to the algebraic number field case, so you may also have a look at the first sections dealing with algebraic number fields.

5.1 Function Fields

In KASH3, creation of an algebraic function field begins with choosing a bivariate polynomial `f` over `k`, which is separable and monic in the second variable, such that `f(T,y) = 0`. For this there have to be defined the field `k`, the polynomial rings `k[T]` and `k[T][y]`, respectively (see example). It is afterwards possible to define an algebraic function field. We test first whether the bivariate polynomial is irreducible and separable in the second variable. Then as a first application, one can compute the genus of the function field by calling `Genus` function.

Example

```
kash% k := FiniteField(25);
Finite field of size 5^2
Time: 0.001845 s
kash% kT := RationalFunctionField(k);
Univariate rational function field over GF(5^2)
Variables: $.1
Time: 0.000409 s
kash% kTy := PolynomialAlgebra(kT);
Univariate Polynomial Ring over Univariate rational function field over GF(5^2\
)
Time: 0.000268 s
kash% T := kT.1;
kT.1
Time: 0.000331 s
kash% ;
```

```
kash%  y := kTy.1;
kTy.1
Time: 0.00033 s
kash% ;
kash% f := y^3 + T^4 + 1;
kTy.1^3 + kT.1^4 + 1
Time: 0.00048 s
kash% K := FunctionField(f);
Algebraic function field defined over Univariate rational function field over \
GF(5^2) by
kTy.1^3 + kT.1^4 + 1
Time: 0.00158 s
kash% Genus(K);
3
Time: 0.011949 s
```

5.1.1 Genus

```
Genus(<fld^fun> F) -> <elt-ord^rat>
```

The genus of the function field F.

5.2 Finite and Infinite Maximal Orders

According to their coefficient rings `k[T]` or `O at infinity` orders are called finite or infinite. By an equation order (or coordinate ring) over `k[T]` we mean the quotient ring `k[T][y] / f(T,y)k[T][y]`. Equation orders over `O at infinity` are defined analogously for suitable, field generating polynomials (See examples for different orders).

One can define elements of orders by calling `Element`. Since the orders have bases, it is enough to specify coefficients of linear combinations of the basis elements (see example). Afterwards one can perform the operations with these elements as usual.

Usually one wants to work with the maximal orders since only these are Dedekind rings. For convenience there is a function which expects the defining polynomial and which first checks for irreducibility and separability and defines then the algebraic function field `F` and the maximal orders `o` and `oi` (see example below).

Example

```
kash% ff := FiniteField(5);
Finite field of size 5
Time: 0.000322 s
kash% fx := FunctionField(ff);
Univariate rational function field over GF(5)
Variables: $.1
```

```
Time: 0.001577 s
kash% fxy := PolynomialAlgebra(fx);
Univariate Polynomial Ring over Univariate rational function field over GF(5)
Time: 0.00029 s
kash% F := FunctionField(fxy.1^3+fx.1^4+1);
Algebraic function field defined over Univariate rational function field over \
GF(5) by
fxy.1^3 + $.1^4 + 1
Time: 0.002666 s
kash% o:=MaximalOrderFinite(F);
Maximal Equation Order of F over Univariate Polynomial Ring in $.1 over GF(5)
Time: 0.001341 s
kash% oi:=MaximalOrderInfinite(F);
Maximal Order of F over Valuation ring of Univariate rational function field o\
ver GF(5) with generator 1/$.1
Time: 0.008775 s
kash% a:=Element(o,[0,1,0]);
[ 0, 1, 0 ]
Time: 0.000356 s
kash%  b:=Element(oi,[0,1/fx.1,1/fx.1^2+1]);
[ 0, 1/$.1, ($.1^2 + 1)/$.1^2 ]
Time: 0.000581 s
kash% a^3+fx.1^4+1;
0
Time: 0.000797 s
kash% a+b;
($.1^2 + 1)/$.1^5*F.1^2 + ($.1^3 + 1)/$.1^3*F.1
Time: 0.000825 s
kash% Coerce(o,a);
[ 0, 1, 0 ]
Time: 0.000315 s
```

5.2.1 MaximalOrderFinite

```
MaximalOrderFinite(<fld^fun> F) -> <ord^fun>
```

The `finite` maximal order of the function field F.

Example

```
kash% x_QX:=PolynomialAlgebra(Q);
Univariate Polynomial Ring over Rational Field
Time: 0.000281 s
kash% x_QY:=PolynomialAlgebra(x_QX);
Univariate Polynomial Ring over Univariate Polynomial Ring over Rational Field
Time: 0.000283 s
```

```
kash% x_f:=x_QY.1^2+2;
x_QY.1^2 + 2
Time: 0.000479 s
kash% x_F:=FunctionField(x_f);
Algebraic function field defined over Univariate rational function field over \
Rational Field by
x_QY.1^2 + 2
Time: 0.001165 s
kash% MaximalOrderFinite(x_F);
Maximal Equation Order of x_F over Univariate Polynomial Ring over Rational Fi\
eld
Time: 0.000619 s
```

5.2.2 MaximalOrderInfinite

```
MaximalOrderInfinite(<fld^fun> F) -> <ord^fun>
```

The `infinite` maximal order of the function field F.

Example

```
kash% x_QX:=PolynomialAlgebra(Q);
Univariate Polynomial Ring over Rational Field
Time: 0.000272 s
kash% x_QY:=PolynomialAlgebra(x_QX);
Univariate Polynomial Ring over Univariate Polynomial Ring over Rational Field
Time: 0.000286 s
kash% x_f:=x_QY.1^2+2;
x_QY.1^2 + 2
Time: 0.000461 s
kash% x_F:=FunctionField(x_f);
Algebraic function field defined over Univariate rational function field over \
Rational Field by
x_QY.1^2 + 2
Time: 0.001158 s
kash% MaximalOrderInfinite(x_F);
Maximal Equation Order of x_F over Valuation ring of Univariate rational funct\
ion field over Rational Field with generator 1/$.1
Time: 0.001096 s
```

5.3 Ideals and Divisors

There are two representations for ideals, first by two generating elements and second by a module basis over the coefficient ring of the order. Multiplicative arithmetic is supported and you may also take the sum of two ideals, which is the same as to compute the gcd of these ideals.

5.3.1 Places

```
Places(<fld^fun> F) -> <pls/fld^fun>
```

The set of places of F.

5.3.2 Places

```
Places(<fld^fun> F, <elt-ord^rat> m) -> <seq()>
```

The sequence of places of degree m of the global function field F.

5.3.3 Ideals

```
Ideals(<elt-dvs/fld^fun> D [, optargs]) -> <elt-ids^int/ord^fun>,
<elt-ids^int/ord^fun>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
```

The ideals corresponding to the divisor D.

Example

```
kash%  x_F:= HermitianFunctionField(5);
Algebraic function field defined over Univariate rational function field over \
GF(5^2) by
y^5 + y + 4*x^6
Time: 0.002536 s
kash% ;
kash%  x_O:= MaximalOrderFinite(x_F);
Maximal Equation Order of x_F over Univariate Polynomial Ring in x over GF(5^2\
)
Time: 0.001453 s
kash% ;
kash% x_p:= PrimitiveElement(x_O);
[ 0, 1, 0, 0, 0 ]
Time: 0.00033 s
kash% ;
kash%  x_D:= Divisor(x_p);
-6*(1/x, 1/x^5*x_F.1^4) + 6*(x, x_F.1 + x)
```

```
Time: 0.085252 s
kash% ;
kash% Ideals(x_D);
Ideal of x_O
  Basis:
  [     x^6        0        0        0        0]
  [       0        1        0        0        0]
  [       0        0        1        0        0]
  [       0        0        0        1        0]
  [       0        0        0        0        1], extended by:
  ext1 := Fractional ideal of _OW
    Basis:
    [1/x   0   0   0   0]
    [  0   1   0   0   0]
    [  0   0 1/x   0   0]
    [  0   0   0 1/x   0]
    [  0   0   0   0 1/x]
    Denominator: 1/x^2
Time: 0.007945 s
```

5.3.4 Divisors

```
Divisors(<elt-ids^int/ord^num> x) -> <seq()>
```

All divisors of x, which must be an integral ideal of a maximal order.

Example

```
kash%  x_o := MaximalOrder(X^3 -299*X -2403);
Maximal Order of _AB
Time: 0.002178 s
kash%  x_a := 3*x_o+Coerce(x_o,[-5,-5,7])*x_o;
Ideal of x_o
Two element generators:
[3, 0, 0]
[-5, -5, 7]
Time: 0.000883 s
kash%  Divisors(x_a);
[
Principal Ideal of x_o
Generator:
[1, 0, 0],
Prime Ideal of x_o
Two element generators:
[3, 0, 0]
[-5, -5, 7]
]
Time: 0.010482 s
```

Chapter 6

Programming Language

KASH3 uses the GAP3 shell as a user interface. The programming language of GAP3 is an imperative language with some functional and some object oriented features. In KASH3 additional features like Methods, Maps, and Extendable Objects are available. The following describes the imperative control structures of the GAP3/KASH3 programming language.

By convention the names of KASH3 functions, which change the arguments (in the GAP3/KASH3 programming language all complex data structures are passed to functions by reference), end in _.

6.1 if

```
if <elt-alg^boo> then <statements1> ;
{ elif <elt-alg^boo> then <statements2> }
[ else <statements3> ]
fi;
```

The `if` statement allows one to execute statements depending on the value of some boolean expression. The execution is done as follows: First the boolean expression following the `if` is evaluated. If it evaluates to `true` the statement sequence <statements1> after the first `then` is executed, and the execution of the `if` statement is complete. Otherwise the boolean expressions following `elif` are evaluated in turn. There may be any number of `elif` parts, possibly none at all. If the `if` expression and all, if any, `elif` expressions evaluate to `false` and there is an `else` part, which is optional, its statement sequence <statements3> is executed and the execution of the `if` statement is complete. If there is no `else` part, the `if` statement is complete without executing any statement sequence. Since the `if` statement is terminated by the `fi` keyword there is no question where an `else` part belongs.

6.2 while

```
while <elt-alg^boo> do <statements>  od;
```

The `while` loop executes the <statements> while the condition evaluates to `true`. First the boolean expression is evaluated. If it evaluates to `false` execution of the `while` loop terminates and the statement immediately following the `while` loop is executed next. Otherwise if it evaluates to `true` the <statements> are executed and the whole process begins again.

6.3 repeat

```
repeat <statements>  until <elt-alg^boo>;
```

The `repeat` loop executes the statement sequence <statements> until the condition evaluates to `true`. First <statements> are executed. Then the bollean expression is evaluated. If it evaluates to `true` the `repeat` loop terminates and the statement immediately following the `repeat` loop is executed next. Otherwise if it evaluates to `false` the whole process begins again with the execution of the <statements>.

6.4 for

```
for <variable> in <list> do <statements> od;
```

The `for` loop executes the <statements> for every element of <list>. The statement sequence <variable> is first executed with <variable> bound to the first element of <list>, then with <variable> bound to the second element of <list> and so on. <variable> must be a simple variable, it must not be a list element selection or a record component selection.

6.5 function

```
function ( [ <arg-ident> {, <arg-ident>} ] )
    [ local   <loc-ident> {, <loc-ident>} ; ]
    <statements>
end;
```

A function is in fact a literal and not a statement. Such a function literal can be assigned to a variable or to a list element or a record component.

Because for each of the formal arguments <arg-ident> and for each of the formal locals <loc-ident> a new variable is allocated when the function is called, it is possible that a function calls itself. This is usually called recursion.

When a function <fun1> definition is evaluated inside another function <fun2>, KASH binds all the identifiers inside the function <fun1> that are identifiers of an argument or a local of <fun2> to the corresponding variable. This set of bindings is called the environment of the function <fun1>. When <fun1> is called, its body is executed in this environment.

In KASH (by convention) the names of function which return nothing and act destructively on the input end in _.

6.6 return

```
return [<expr>];
```

`return;` terminates the call of the innermost function that is currently executing, and control returns to the calling function. An error is signalled if no function is currently executing. No value is returned by the function.

`return <expr>;` terminates the call of the innermost function that is currently executing, and returns the value of the expression <expr>. Control returns to the calling function. An error is signalled if no function is currently executing.

Some functions return several values. In this case the functions return a the first return value as an extended object with the additional components `ext1`, `ext2`, and so on.

6.7 arg

```
function(arg) ... end;
```

Collapse arbitrary many arguments to a list of arguments and pass it to arg.

When used as (only) argument in a function's declaration this so defined function accepts any number of arguments. Arguments passed to the function at call-time are gathered into a list whose value becomes `arg` in the function body.

Note: This will not work on abbreviated function declarations, like `arg->arg[1];`.

Note: Also, the function keyword will not collect rests of arguments into the arg argument, that is `function(some, arg) return TRUE; end;` is a valid function which takes exactly two arguments, `arg` has no no special meaning here (yet).

Example

```
kash% x_f := function(arg) return arg;
%  end;
function ( arg ) ... end
Time: 0.000252 s
kash% x_f(1,2.3,"hi mom",[12,3]);
[ 1, 2.300000000000000000000000000000, "hi mom", [ 12, 3 ] ]
Time: 0.000267 s
kash% x_f(6);
[ 6 ]
Time: 0.000263 s
```

Chapter 7

Inside KASH3

KASH3 contains several advanced features that facilitate extending its functionality. Some of these come from the GAP3 programming language, in particular the overloading of operators using records. Extendable objects are also build on records. They allow storing additional information in objects on the shell level. Together with the type system this also allows inheritance of functionality to objects of new types. The type system also makes it possible to overload functions `InstallMethods`) . Overloading and the type system are tied in with the help system.

7.1 Types

```
<simple type>
<aggtype>([<type>{, <type>}])
<elt|str>[-<level1type>^<level2type>{/<level1type>^<level2type>}]
```

In KASH there are three different kinds of types.

SIMPLE TYPES stand for themselves.

(TYPED) AGGREGATE TYPES can carry a further specification in parenthesis, they are written as <aggtype>([<type>, <type>]).

COMPOSED TYPES are composed from several atoms. The first atom is either `elt` or `str` followed by a `-`. It is followed by a LEVEL 1 atom, a `^`, and a LEVEL 2 atom. This may be followed by / and further <`level1type`>`^`<`level2type`> constructions. In these constructions `str-` can be left out, likewise `any` as a LEVEL 2 atom can be omitted. LEVEL 1 atoms describe the general algebraic structure (field, algebra, group, ...) and LEVEL 2 atoms give a more detailed description (finite, polynomial, abelian, ...)

The function `ShowTypes` displays the atoms used in the construction of aggregate and composed types. `?*.|TYPE` shows all (documented) types in KASH.

7.1.1 Type

```
Type(<any> obj) -> <type>
```

Returns the type of `obj`. If `obj` is extended, i.e., it is a record with a `base` component, then the type of the base component is returned. If `obj` is a record with a `type` component `obj.type` is returned.

Example

```
kash% Type(3);
elt-ord^rat
Time: 0.000263 s

kash% Type(true);
elt-alg^boo
Time: 0.000265 s

kash% Type(2/3);
elt-fld^rat
Time: 0.000319 s

kash% Type(Sequence([1.2,2]));
seq(elt-fld^rea)
Time: 0.000318 s
```

7.1.2 ShowTypes

```
ShowTypes()
```

Display a list of all type atoms ordered by level.

Example

```
kash% ShowTypes();

SIMPLE TYPES (LEVEL 0):
any  void  unknown  record  name  file  type  char  list  set  dry  alist  str\
ing  func  thue  state  newtgon  newtface  method

(TYPED) AGGREGATE TYPES:
map() seq() tup() nof()

ELEMENT OR STRUCTURE:
elt-  str-

TYPE ATOMS OF LEVEL 1:
any   unknown   alg   fld   ord   res   ids   mdl   dvs   pls   dif   rng   gr\
p
```

```
TYPE ATOMS OF LEVEL 2:
^any  ^unknown  ^fun  ^pad  ^ser  ^pow  ^loc  ^rat  ^rea  ^com  ^pol  ^mat  ^v\
ec  ^abl  ^ell  ^num  ^ded  ^pui  ^fra  ^int  ^boo  ^fin  ^inf  ^per  ^gap

Enter ?xyz for a description of the type atom xyz.
```

7.1.3 elt

The type of elements. To obtain the type of the elements of a structure of type `s`, one can write `elt-s`.

Example

```
kash% s := Type(ZX);
alg^pol/ord^rat
Time: 0.00028 s
kash%  elt-s;
elt-alg^pol/ord^rat
Time: 0.000258 s
```

7.1.4 ord^rat

The type of the ring of integers.

Note: This is predefined to `Z`.

Example

```
kash% x_z:=IntegerRing();
Integer Ring
Time: 0.00028 s
kash% Type(x_z);
ord^rat
Time: 0.000262 s
kash%  Type(Z);
ord^rat
Time: 0.000265 s
kash% Z=x_z;
TRUE
Time: 0.00028 s
```

7.1.5 elt-ord^rat

The type of integers.

Example

```
kash% Type(5);
elt-ord^rat
Time: 0.000274 s
```

7.1.6 elt-alg^pol

The type of polynomials.

Example

```
kash% x_p1:=ZX.1^2-1;
X^2 - 1
Time: 0.00038 s
kash% Type(x_p1);
elt-alg^pol/ord^rat
Time: 0.000257 s
```

7.1.7 NewType

```
NewType(<string> newtype, <elt-ord^rat> level) -> <type>
```

Install the new type `newtype` of level `level`.

if `level=0` then `newtype` can only be used as a simple type.

if `level=1` then `newtype` can only be used on the left of `^`.

if `level=2` then `newtype` can only be used on the right of `^`.

Example

```
kash% NewType("cox",2);
cox
Time: 0.000288 s
kash%  elt-grp^cox;
elt-grp^cox
Time: 0.00027 s
```

7.1.8 Is

```
Is(<type> t1, <type> t2)
```

TRUE if t2 matches more types than t1, FALSE otherwise.

Example

```
kash% Is(elt-ord^rat,elt);
TRUE
Time: 0.000338 s
kash%  Is(elt-ord^rat,elt-alg);
FALSE
Time: 0.000319 s
kash%  Is(elt-ord^rat,elt-any^rat);
TRUE
Time: 0.000325 s
kash%  Is(fld,rng);
TRUE
Time: 0.000266 s
kash%  Is(fld^rat,rng^rat);
TRUE
Time: 0.000298 s
```

7.2 Generic Functions

Some functions are implemented for a variety of structures. A few of them are not documented for all possible signatures. You find the most important of those listed below.

7.2.1 Generators

The generators of many of the structures can be accessed with the `.` operator. In most cases the respective elements are also printed in this representation.

Example

```
kash% x_a := FreeAbelianGroup(3);
Abelian Group isomorphic to Z + Z + Z
Defined on 3 generators (free)
Time: 0.000306 s
kash% Element(x_a,[1,2,3]);
x_a.1 + 2*x_a.2 + 3*x_a.3
Time: 0.000345 s
kash% x_a.2;
x_a.2
Time: 0.000316 s
```

7.2.2 Element

```
Element(<any> R, <nof(any)> a) -> <elt-any>
```

The element of `R` specified by `a`

Example

```
kash% o := MaximalOrder(X^3-77);
Maximal Equation Order with defining polynomial X^3 - 77 over Z
Time: 0.00529 s
kash% Element(o,[2, 1, 0]);
[2, 1, 0]
Time: 0.000453 s
```

7.2.3 Sequence

```
Sequence(<any> a) -> <seq()> se
```

Tries to convert `a` to a sequence. Some objects are converted to the list of their coefficients.

Example

```
kash% lis:=[1,2,3,4];
[ 1, 2, 3, 4 ]
Time: 0.00026 s
kash% Sequence(lis);
[ 1, 2, 3, 4 ]
Time: 0.000289 s
```

7.2.4 List

```
List(<any> a) -> <list> lis
```

Tries to convert `a` to a plain list. Some objects are converted to the list of their coefficients

Example

```
kash% se := Sequence([1,1,2]);
[ 1, 1, 2 ]
Time: 0.000274 s
kash% lis :=List(se);
[ 1, 1, 2 ]
Time: 0.000649 s
```

7.2.5 Ideal

```
Ideal(<any> R, <nof(any)> a) -> <elt-ids>
```

The ideal of `R` specified by `a`

Example

```
kash% o := MaximalOrder(X^3-77);
Maximal Equation Order with defining polynomial X^3 - 77 over Z
Time: 0.005274 s
kash% Ideal(o,5,[2, 1, 0]);
Ideal of o
Two element generators:
[5, 0, 0]
[2, 1, 0]
Time: 0.000518 s
```

7.2.6 Extension

```
Extension(<any> S, <nof(any)> a) -> <any>
```

The extension of `S` specified by `a`

Example

```
kash% K := Extension(Q,X^2+2);
Number Field with defining polynomial _WC.1^2 + 2 over the Rational Field
Time: 0.00076 s
```

7.2.7 Sub

```
Sub(<any> R, <nof(any)> a [, optargs]) -> <any> S, <map()> m
```

with optional arguments

```
<elt-ord^rat> Results (if Results:=2 also the map is returned)
```

The substructure `S` of `R` specified by `a` and the embedding `m` of `S` into `R` if requested.

Example

```
kash% a := FreeAbelianGroup(2);
Abelian Group isomorphic to Z + Z
```

```
Defined on 2 generators (free)
Time: 0.000294 s
kash% s := Sub(a,2*a.1,7*a.2);
Abelian Group isomorphic to Z + Z
  Defined on 2 generators in supergroup a:
  s.1 = 2*a.1
  s.2 = 7*a.2 (free), extended by:
  ext1 := Mapping from: grp^abl: s to grp^abl: a
Time: 0.00047 s

kash% b := Sub(Z,17,rec(Results:=2));
Ideal of Integer Ring generated by 17, extended by:
  ext1 := Mapping from: ord^rat: b to ord^rat: Z
Time: 0.000303 s
```

7.2.8 Quotient

```
Quotient(<any> R, <nof(any)> a [, optargs]) -> <any> Q, <map()> m
```

with optional arguments

```
<elt-ord^rat> Results (if Results:=2 also the map is returned)
```

The quotient of `R` and the substructure of `R` specified by `a` and the map `m` from `R` to `Q` if requested.

Example

```
kash% a := FreeAbelianGroup(3);
Abelian Group isomorphic to Z + Z + Z
Defined on 3 generators (free)
Time: 0.000286 s
kash% Quotient(a,3*a.1,4*a.2,2*a.3,rec(Results:=2));
Abelian Group isomorphic to Z/2 + Z/12
  Defined on 2 generators
  Relations:
  2*_GD.1 = 0
  12*_GD.2 = 0, extended by:
  ext1 := Mapping from: grp^abl: a to grp^abl: _GD
Time: 0.000907 s

kash% a := FreeAbelianGroup(2);
Abelian Group isomorphic to Z + Z
Defined on 2 generators (free)
Time: 0.000292 s
```

```
kash% s := Sub(a,2*a.1,7*a.2);
Abelian Group isomorphic to Z + Z
  Defined on 2 generators in supergroup a:
  s.1 = 2*a.1
  s.2 = 7*a.2 (free), extended by:
  ext1 := Mapping from: grp^abl: s to grp^abl: a
Time: 0.000479 s
kash%  q := Quotient(a,s);
Abelian Group isomorphic to Z/14
  Defined on 1 generator
  Relations:
  14*q.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: a to grp^abl: q
Time: 0.00064 s

kash% Quotient(pAdicRing(3,20),3^6);
Quotient of the 3-adic ring modulo the ideal generated by 3^6
Time: 0.000432 s
```

7.3 Records and Extended Objects

In KASH3 records serve multiple purposes. First of all they allow the construction of complex structures. These data structures can easily be assigned new types by simply adding a `type` component. They can also be used to extend existing data structures: if a record contains a `base` component, then it is treated like its `base` component by most functions. Furthermore records can be used to overload the operations `+`, `-`, `*`, /, >, `=`, `mod` by including in an `operations` component.

7.3.1 record

Records are next to lists and sequences the most important way to collect objects.A record is a collection of components. Each component has a unique name, which is an identifier that distinguishes this component, and a value, which is an object of arbitrary type. You can access and change the elements of a record using its name. Records usually contain elements of various types, i.e., they are usually not homogeneous like lists or sequences.

The record components `base`, `operations`, `size`, and `type` have special meaning.

7.3.2 rec

```
rec(<nof(any)>) -> <record>

rec({<name> field := <any> a[,]})
```

Record literals are written by writing down the components in order between `rec(` and `)` , and separating them by commas `,`. Each component consists of the name, the assignment operator `:=`, and the value. The empty record, i.e., the record with no components, is written as `rec()`.

Example

```
kash% rec( a := 1, b := '2' );
rec(
  a := 1,
  b := '2' )
Time: 0.000276 s

kash% rec( a := 1, b := rec( c := 2 ));
rec(
  a := 1,
  b := rec(
      c := 2 ) )
Time: 0.000272 s
```

7.3.3 base

A record `r` that contains a special `base` component is treated in the same way as the base component itself. These record are called extended objects The other components of `r` are considered as additional data, which is emphasized when printing. Many functions return extended objects, for instance normal forms of matrices are usually extended by the transformation matrices.

Example

```
kash% x_r := rec(base := 1.2*I+2, what:="a complex number");
2.0000000000000000000000000000 + 1.2000000000000000000000000000*I, extended \
by:
  what := "a complex number"
Time: 0.000409 s
kash% x_r+2;
4.0000000000000000000000000000 + 1.2000000000000000000000000000*I
Time: 0.000269 s

kash% XGCD(12655,342676);
1, extended by:
  ext1 := -151801,
  ext2 := 5606
Time: 0.000286 s

kash% x_m := Matrix(Z,2,[1,2,3,4]);
[1 2]
```

```
[3 4]
Time: 0.001542 s
kash% SmithForm(x_m);
[1 0]
  [0 2], extended by:
  ext1 := [ 1  0]
    [-1  1],
  ext2 := [-1  2]
    [ 1 -1]
Time: 0.000397 s
```

7.3.4 Extend

```
Extend(<any> a) -> <record>
```

Extends the object `a`. `a` becomes a record such that `a.base` is the original `a`. Most functions consider `a` as `a.base`. Most complex objects can be extended. Integers (<elt-ord^rat>) and booleans (<elt-alg^boo>) cannot be extended with this function. Extended objects of all types can be constructed directly.

7.3.5 Base

```
Base(<any> A) -> <any>
```

If `A` is a record with a `base` component then return `A.base`. Otherwise return `A`.

7.3.6 Eq

```
Eq(<any> A, <any> B) -> <elt-alg^boo>
```

Equal for extended types. Return `TRUE` if `Base(A)=Base(B)`, `FALSE` otherwise. This is useful, if one of the values of a comparison could be extended.

Example

```
kash% a := 2/3;
2/3
Time: 0.000295 s
kash%  Extend(a);
kash%  a=2/3;
FALSE
Time: 0.000272 s
kash%  Eq(a,2/3);
TRUE
Time: 0.000275 s
```

```
kash%  Base(a)=2/3;
TRUE
Time: 0.000257 s
```

7.3.7 operations

A record may contain a special `operations` component which in turn is a record that contains functions. These functions are called when this record is an operand of `+`, `-`, `*`, /, `^`, `mod`, `in`, `Print`, `=`, <. In the `operations` the names of the components of `+`, `-`, `*`, /, `^`, `=`, < have to be quoted, i.e., written as \\`+`, \\`-` and so on. The unary `-a` is computed as `0-a','a <= b` is `a < b or a = b`, `a > b` is `not a < b`, etc.

Example

```
kash% o := rec();
rec(
   )
Time: 0.000251 s
kash% o.\+:= function(a,b) return rec(n :=2*a.n+b.n, operations := o);
%  end;
function ( a, b ) ... end
Time: 0.00026 s
kash% r := rec(n:=3, operations:=o);
rec(
  n := 3,
  operations := rec(
      + := function ( a, b ) ... end ) )
Time: 0.000272 s
kash%  s:= rec(n:=4, operations := o);
rec(
  n := 4,
  operations := rec(
      + := function ( a, b ) ... end ) )
Time: 0.000261 s
kash% r+s;
rec(
  n := 10,
  operations := rec(
      + := function ( a, b ) ... end ) )
Time: 0.000254 s
kash%  s+r;
rec(
  n := 11,
  operations := rec(
      + := function ( a, b ) ... end ) )
Time: 0.000261 s
```

7.4 Optional Arguments

Many KASH3 functions take optional arguments. These are passed to a function by passing a record as a last argument to the function.

7.4.1 ExtractOptarg

```
ExtractOptarg(<list> arglist) -> <record>
```

Return arglist's last argument if it is an optional argument, FAILURE otherwise otherwise.

7.4.2 HasOptarg

```
HasOptarg(<list> arglist) -> <elt-alg^boo>
```

Return TRUE iff arglist's last argument is an optional argument record.

7.4.3 CheckArgs

```
CheckArgs(<list> arglist, <list> argnames, <list> defaults) -> <record>
```

Traverse through `arglist` and bind arguments to argument names in `argnames`. If some arguments are not provided bind them to values from `defaults`. Return the resulting record.

7.5 Maps

In KASH3 maps are functions with additional information, namely domain, codomain, and in some cases a map for computing preimages.

7.5.1 map

```
map()
map(<type>,<type>)
```

The type of maps. The type of the domain and the codomain may be specified.

Example

```
kash% x_o := MaximalOrder(X^5-5*X^3+7*X^2-15*X+16);
Maximal Equation Order with defining polynomial X^5 - 5*X^3 + 7*X^2 - 15*X + 1\
6 over Z
Time: 0.004152 s
kash% x_cl := ClassGroup(x_o);
```

```
Abelian Group isomorphic to Z/4
  Defined on 1 generator
  Relations:
  4*x_cl.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: x_cl to ids/ord^num: _EX
Time: 0.3485 s
kash% Type(x_cl.ext1);
map(grp^abl,ids/ord^num)
Time: 0.000297 s
```

7.5.2 Map

```
Map(<any> domain, <any> codomain, <func> phi) -> <map()>
```

Create a map with domain `domain` and codomain `codomain` from the function `phi`.

Example

```
kash% x_mult := function(a)
% # this function returns a map that multiplies by 'I*a'
% local phi;
% phi := function(b) return b*a*I;
%  end;
% return Map(R,C,phi);
% end;
function ( a ) ... end
Time: 0.000265 s
kash%
kash% x_f := x_mult(5);
Mapping from fld^rea: R to fld^com: C
Time: 0.000562 s
kash% x_f(2);
10.0000000000000000000000000000*I
Time: 0.000365 s
kash% Image(3.1,x_f);
15.5000000000000000000000000000*I
Time: 0.000335 s
kash% Domain(x_f);
Real field of precision 30
Time: 0.000261 s
kash% Codomain(x_f);
Complex field of precision 30
Time: 0.000281 s
```

7.5.3 Map

```
Map(<any> domain, <any> codomain, <func> phi, <func> psi) -> <map()>
```

Create a map with domain `domain` and codomain `codomain` from the function `phi`. The function `psi` is used for the computation of preimages.

Example

```
kash% x_add_with_inv := function(a)
% # this function returns a map that adds 'a'
% local phi, psi;
% phi := function(b) return b+a;
%  end;
% psi := function(c) return c-a;
%  end;
% return Map(Z,Z,phi,psi);
% end;
function ( a ) ... end
Time: 0.000253 s
kash%
kash% x_f := x_add_with_inv(5);
Mapping from ord^rat: Z to ord^rat: Z with inverse
Time: 0.000876 s
kash% x_f(2);
7
Time: 0.000269 s
kash% Image(3,x_f);
8
Time: 0.000267 s
kash% Preimage(8,x_f);
3
Time: 0.000282 s
kash% Preimage(x_f(900),x_f);
900
Time: 0.000292 s
```

7.5.4 Domain

```
Domain(<map()> f) -> <any>
```

The domain of f.

Example

```
kash% x_o := MaximalOrder(X^5-5*X^3+7*X^2-15*X+16);
```

```
Maximal Equation Order with defining polynomial X^5 - 5*X^3 + 7*X^2 - 15*X + 1\
6 over Z
Time: 0.004148 s
kash% x_cl := ClassGroup(x_o);
Abelian Group isomorphic to Z/4
  Defined on 1 generator
  Relations:
  4*x_cl.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: x_cl to ids/ord^num: _HT
Time: 0.418198 s
kash%  Domain(x_cl.ext1);
Abelian Group isomorphic to Z/4
Defined on 1 generator
Relations:
4*x_cl.1 = 0
Time: 0.000319 s
```

7.5.5 Codomain

```
Codomain(<map()> f) -> <any>
```

The codomain of f.

7.5.6 Image

```
Image(<any> a, <map()> f) -> <any>
```

The image of `a` under `f`.

7.5.7 Preimage

```
Preimage(<any> x, <map()> f) -> <any>
```

The preimage of x under f.

Example

```
kash% x_R := PolynomialAlgebra( IntegerRing() );
Univariate Polynomial Ring in X over Integer Ring
Time: 0.00028 s
kash% x_x := Generator(x_R, 1);
X
Time: 0.000279 s
kash% x_O := MaximalOrder( x_x^5-5*x_x^3+7*x_x^2-15*x_x+16 );
Maximal Equation Order with defining polynomial X^5 - 5*X^3 + 7*X^2 - 15*X + 1\
6 over Z
Time: 0.004142 s
kash% x_G := ClassGroup(x_O);
Abelian Group isomorphic to Z/4
  Defined on 1 generator
  Relations:
  4*x_G.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: x_G to ids/ord^num: _JB
Time: 1.229756 s
kash% x_f := x_G.ext1;
Mapping from: grp^abl: x_G to ids/ord^num: _JB
Time: 0.000282 s
kash% x_I := Factorization( 7*x_O )[1][1];
Prime Ideal of x_O
Two element generators:
[7, 0, 0, 0, 0]
[1, 1, 0, 0, 0]
Time: 0.002488 s
kash% x_g := Preimage(x_I, x_f);
3*x_G.1
Time: 0.001795 s
kash% x_J := x_f(x_g);
Ideal of x_O
Two element generators:
[8, 0, 0, 0, 0]
[1, 0, 7, 7, 4]
Time: 0.000399 s
kash% IsPrincipal(x_I / x_J);
TRUE, extended by:
  ext1 := -5/1*_RI.1 + 17/8*_RI.2 + 13/4*_RI.3 - 7/8*_RI.4 - 1/2*_RI.5
Time: 0.427467 s
```

7.5.8 HasPreimage

```
HasPreimage(<any> x, <map()> f [, optargs]) -> <elt-alg^boo>, <any>
```

with optional arguments

```
<elt-ord^rat> Results (1 <= Results <= 2)
```

True and the preimage of x under f iff it exists, false otherwise.

Example

```
kash% x_G:=AbelianGroup([2,3]);
Abelian Group isomorphic to Z/6
```

```
Defined on 2 generators
Relations:
2*x_G.1 = 0
3*x_G.2 = 0
Time: 0.000834 s
kash% x_H:=AbelianGroup(x_G);
Abelian Group isomorphic to Z/6
  Defined on 1 generator in supergroup x_G:
  x_H.1 = x_G.1 + x_G.2
  Relations:
  6*x_H.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: x_G to grp^abl: x_H
Time: 0.000536 s
kash% HasPreimage(x_G.1,x_H.ext1);
TRUE, extended by:
  ext1 := x_G.1
Time: 0.000568 s
```

7.5.9 Composition

```
Composition(<map()> phi, <map()> psi) -> <map()>
```

The composition `phi*psi` of the maps `phi` and `psi`.

Example

```
kash% x_add_with_inv := function(a)
% # this function returns a map that adds 'a'
% local phi, psi;
% phi := function(b) return b+a;
%  end;
% psi := function(c) return c-a;
%  end;
% return Map(Z,Z,phi,psi);
% end;
function ( a ) ... end
Time: 0.000258 s
kash%
kash% x_f := x_add_with_inv(5);
Mapping from ord^rat: Z to ord^rat: Z with inverse
Time: 0.000874 s
kash% x_f(2);
7
Time: 0.000272 s
kash% x_Z7 := Quotient(Z,7);
Residue class ring of integers modulo 7, extended by:
  ext1 := Mapping from: ord^rat: Z to res^rat: x_Z7
Time: 0.000287 s
kash% x_g := Composition(x_Z7.ext1,x_f);
Mapping from ord^rat: Z to res^rat:  with inverse
Time: 0.000603 s
kash% x_g(1);
6
Time: 0.000295 s
kash% Preimage(Coerce(x_Z7,8),x_g);
-4
Time: 0.000337 s
```

7.6 Documentation

```
rec(
kind:= <string>,
name:= <string>,
sin := <list(list(type,string))>,
opt := <list(list(type,string,string))>,
sou := <list(list(type,string))>,
syntax := <string>,
short := <string>,
ex := <list(string)>,
see := <list(string)>
author := <list(string)>
)
```

Documentation is stored in the global list `__DOC`. Its elements are records of the form given above.

`kind` must be "CONSTANT" or "FUNCTION" or "KEYWORD" or "OPERATION" or "STATEMENT" or "TYPE".

`name` is the name of the object being documented.

`sin` is the input signature as a list of lists containing the type and a name for the parameter, mandatory for "FUNCTION" and "OPERATION".

`opt` is a list of optional arguments containing types and names comments.

`sou` is the output signature (return values) if the name of the first return value is r the other return values are called r.ext1, r.ext2 and so on mandatory for "CONSTANT" and "FUNCTION" and "OPERATION".

A `syntax` entry can be added if `name`, `sin`, and `sou` do not describe the syntax sufficiently.

`short` contains a description of the purpose of the object.

`ex` is a list of examples, one example per string. By convention all variables in example start with `x_`.

`see` contains a list of references to related things as hash strings.

7.6.1 InstallDocumentation

```
InstallDocumentation(<record> r [, optargs]) -> <elt-alg^boo> success
```

with optional arguments

```
<any> Fail: defaults to FAILURE (Determines what to return in case of
failure.)
<elt-alg^boo> ForceAdd (Default: FALSE, indicate you want to add or replace existing
documentation.)
```

Add documentation given by `r` to global documentation hash table.

The documentation in `r` is blown up and checked. Then it is tried to be added with a dry operation (DryReplaceOrAdd) and thus overwrites existing documentation iff the hash value computed by `r` is already in the global dry, and is appended otherwise.

7.6.2 CheckDocumentation

```
CheckDocumentation(<record> r)
```

Return true iff documentation in record `r` tends to be correct.

7.6.3 MergeDocumentation

```
MergeDocumentation(<record> r [, optargs])
```

with optional arguments

```
<any> Fail: defaults to FAILURE (Determines what to return in case of
failure.)
<any> Success (Default: TRUE, indicate what to return in case of success)
<any> Add (Default: FALSE, add ‘r’ to documentation dry in either case)
```

Merge documentation given (even partially) by `r` to global documentation hash table.

7.6.4 DocHash

```
DocHash(<string> s) -> <string> hash
```

Return the hash value used to identify a function specified by `s`. The string `s` must be of the form ”functionname(typearg1,typearg2,...)” or ”type” or ”keyword”.

Example

```
kash% x_s := DocHash("GCD(elt-ord^rat,elt-ord^rat)");
"9430ce"
Time: 0.000352 s
kash% );
Syntax error: expression expected
);
^


kash% x_s := DocHash("record");
"275a70"
Time: 0.000363 s

kash% x_s := DocHash("operations");
"e8c39c"
Time: 0.000365 s
```

7.7 Methods

Methods allow overloading of functions. A method is installed by calling `InstallMethod` with a documentation record and a function. The documentation record specifies the name and the signature under which the given function will be called.

7.7.1 InstallMethod

```
InstallMethod(<record> docrec, <func> body [, optargs])
```

with optional arguments

```
<elt-ord^rat> Position (Default: 1, negative numbers are ...)
```

Install the function `body` to be executed when called with arguments as specified by `docrec`. sin.

Note: `docrec` has to be a fully qualified documentation record, `MergeDocumentation` is called automatically.

Chapter 8

Outside KASH3

KASH3 supports sophisticated access function to the world outside of KASH. We will discuss input/output functions to access the file system and the system environment as well as functions to access the QaoS databases.

Furthermore, for historical reasons we provide a so called GAP compatibility mode.

8.1 Files

Basically KASH is aware of both writing to files and reading from files. While the former may be used to output arbitrary data formats, the latter is restricted to files with valid KASH syntax.

Indeed, the primary intention for output to files is logging. In constrast, reading files (thusly, evaluating their contents) is primarily suitable for storing user function definitions or even variable bindings permanently.

8.1.1 Read

```
Read(<string> filename)
```

Read a file `filename` containing KASH commands.

The file `filename` must both existing and readable.KASH looks first in the given path, then in the current directory and finally in $LIBNAME../src.

8.1.2 ReadLib

```
ReadLib(<string> lib)
```

Same as `Read`, but here the file should be in the KASH lib directory and it must have the extension .g .

8.1.3 LogTo

```
LogTo(<string> filename)
```

`LogTo` instructs KASH to echo all input from the standard input files, `*stdin*` and `*errin*` and all output to the standard output files, `*stdout*` and `*errout*`, to the file with the name `filename`. The file is created if it does not exist, otherwise it is truncated.

8.1.4 LogTo

```
LogTo()
```

`LogTo` called with no argument closes the current logfile again, so that input from `*stdin*` and `*errin*` and output to `*stdout*` and `*errout*` will no longer be echoed to a file.

8.1.5 PrintTo

```
PrintTo(<string> filename, <nof()>)
```

`PrintTo` prints all following arguments to the file with the name `filename`. It opens the file with the name `filename`. If the file does not exist it is created, otherwise it is truncated. If you do not want to truncate the file use `AppendTo`. After opening the file `PrintTo` evaluates its arguments in turn and prints the values to `filename`. Finally it closes the file again. During evaluation of the arguments `filename` is the current output file. This means that output printed with `Print` during the evaluation, for example to inform the user about the progress, also goes to `filename`. To make this feature more useful `PrintTo` will silently ignore if one of the arguments is a procedure call, i.e., does not return a value.

8.1.6 AppendTo

```
AppendTo(<string> filename, <nof()>)
```

`AppendTo` appends all following arguments to the file with the name `filename`. `AppendTo` works like `PrintTo` except that it does not truncate the file if it exists.

8.2 System

KASH offers access to the system environment, read the underlying shell, from within a running session.

Note: Having bound the Exec() and Pipe() functions _IS A SECURITY RISK_! Everybody who has access to your KASH session is also able to run commands in the name of your uid.

You may consider unbinding the function definitions globally.

8.2.1 Exec

```
Exec(<string> S)
```

`Exec` passes the string `S` to the command interpreter of the operating system.

8.2.2 Pipe

```
Pipe(<string> C, <string> S) -> <string>
```

Given a shell command line C and an input string S, create a pipe to the command C, send S into the standard input of C, and return the output of C as a string.

8.2.3 EvalString

```
EvalString(<string> s) -> <any>
```

`EvalString` tries to evaluate `s` using the KASH interpreter. It returns the value of the first statement in `s`

Example

```
kash% EvalString("1+2");
3
Time: 0.000296 s
```

8.3 Database

KASH can read various algebraic objects from the QaoS databases in Berlin. You can establish criteria and query for meeting objects.

Note: The system KASH runs on must have access to the WWW and you must have cURL installed and configured properly.

8.3.1 QaosNumberField

```
QaosNumberField(<string> query [, optargs]) -> <list> L
```

with optional arguments

```
<elt-ord^rat> Limit: defaults to 25 (Determines how many fields may be
retrieved maximally)
<elt-ord^rat> Offset: defaults to 0 (Determines an offset of fields)
<string> Action: defaults to query (Determines which action to perform on the query string
Possible values are ‘query’ and ‘count’.)
<list> ColGroups: defaults to [ "cgall" ] (Determines which information to return.
This is a list of column group specifiers.)
```

Searches the Algebraic Objects Database in Berlin. The query string equals the keyword search method in the web surface.

See `http://www.math.tu-berlin.de/cgi-bin/kant/qaos/query.scm?type=anf&action=Help` for more information about the syntax and keywords.

Note: You must have `curl` (see http://curl.haxx.se) installed and properly configured in order to use QaoS from within KASH.

8.3.2 QaosTransitiveGroup

```
QaosTransitiveGroup(<string> query [, optargs]) -> <list> L
```

with optional arguments

```
<elt-ord^rat> Limit: defaults to 25 (Determines how many groups may be
retrieved maximally)
<elt-ord^rat> Offset: defaults to 0 (Determines an offset of groups)
<string> Action: defaults to query (Determines which action to perform on the query string
Possible values are ‘query’ and ‘count’.)
<list> ColGroups: defaults to [ "cgall" ] (Determines which information to return.
This is a list of column group specifiers.)
```

Searches the Algebraic Objects Database in Berlin. The query string equals the keyword search method in the web surface.

See `http://www.math.tu-berlin.de/cgi-bin/kant/qaos/query.scm?type=trnsg&action=Help` for more information about the syntax and keywords.

Note: You must have `curl` (see http://curl.haxx.se) installed and properly configured in order to use QaoS from within KASH.

8.3.3 QaosResult

```
QaosResult(<list> collection) -> <list> L
```

Return the actual list of objects in `collection`.

8.4 GAP compatibility mode

KASH can operate in a so called GAP compatibilty mode. This provides additional group theoretic functions and objects as in GAP3.

You can enter the GAP emulation mode with the command `GAP();`. Leaving the GAP mode is not possible until the end of your session.

Note: Also, since GAP does not follow our type system, many of the usual GAP functions might not succeed or even work!

8.4.1 Creating GAP groups/objects

The main constructor function for GAP objects (read groups) is `Group()`. There are other several built-in `‘templates’` for groups which we assume are known from GAP.

Example

```
kash% GAP();

GAP-Emulation-Mode
not all functions available, and NO DOCUMENTATION

GAP initialized
kash% Group((1,2),(3,4));
Group( (1,2), (3,4) )
Time: 0.38609 s
kash% CyclicGroup(4);
Group( (1,2,3,4) )
Time: 0.010139 s
```

8.4.2 Accessing GAP groups/objects

Some of the usual GAP accessor functions are available.

`Generators`, `Elements`, `Normalizer`, `Centralizer`, `Stabilizer`, `Centre`.

8.4.3 Properties of GAP groups/objects

Some of the usual GAP predicates are available.

`IsAbelian`, `IsSolvable`, `IsCyclic`, ...

8.4.4 Converting GAP groups/objects to KASH3

Unfortuneately, (automated) conversion of arbitrary groups is not possible.

For the special case of Abelian groups, use the usual constructors, such as `AbelianGroup`, or `FreeAbelianGroup`.

This might change in the future.

8.4.5 Known failures in GAP compat mode

Some of the GAP functions are known to fail due to different type systems. This is a (incomplete) list thereof:

ElementaryAbelianGroup(<elt-ord^rat>)