KASH

A User's Guide

KANT–Group
Technische Universität Berlin
Fachbereich 3 Mathematik
Straße des 17. Juni 136
10623 Berlin, Germany

February 3, 1999

TU Berlin
Fachbereich 3 Mathematik
Strasse des 17. Juni 136
10623 Berlin, Germany

Contents

Preface

This is the n-th release of KASH, the KANT V4 [1] shell.

KANT V4 is a program library for computations in algebraic number fields and (global) algebraic function fields. In the number field case, algebraic integers are considered to be elements of a specified order of an appropriate field $\mathcal{F}$. Algebraic numbers are presented by an integer and a denominator, usually chosen as a natural number. In the function field case, also orders of F are used, but now over different coefficient rings. The representation of algebraic functions is then done in an analogous way as for algebraic numbers. The available algorithms provide the user with the means to compute many invariants of $\mathcal{F}$. In the number field case it is possible to solve tasks like calculating the solutions of Diophantine equations related to $\mathcal{F}$. Further subfields of $\mathcal{F}$ can be generated and $\mathcal{F}$ 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.

KANT V4 was developed at the University of Düsseldorf from 1987 until 1993 and at the Technical University Berlin afterwards. During these years the performance of existing algorithms and their implementations grew dramatically. While calculations in number fields of degree 4 and up 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 V4, 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.

KANT V4 consists of a C–library of more than 1000 functions for doing arithmetic in number 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 computer algebra system MAGMA from which we adopt our storage management, arithmetic for (long) integers and arbitrary precision floating point numbers, arithmetic for finite fields, polynomial arithmetic and a variety

[1] The quasi-acronym KANT stands for Computational Algebraic Number Theory with a slight twist hinting at its German origin.

of other tools. Essentially, all of the public domain part of MAGMA is contained in KANT V4. In return, almost all KANT V4 routines are included in MAGMA.

To make KANT V4 easier to use we developed a shell called KASH. This shell is based on that of the group theory package GAP 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 **Factor** for the factorization of elements from a factorial monoid like rational integers in $\mathbb{Z}$, polynomials over a field, or ideals from a Dedekind ring.

The main features of the current release of KASH are

- computation of maximal orders in numbers fields,
- computation of class groups,
- computation of fundamental units in arbitrary orders,
- decomposition of ideals in number fields,
- arithmetic of ideals,
- arithmetic of relative extensions of number fields,
- computation of maximal orders of a relative extension
- computation of normal forms of modules in relative extensions,
- solution of norm equations in absolute and relative extensions,
- computation of subfields,
- computation of Galois groups up to degree 15,
- computation of automorphisms in normal number fields,
- computation of ray class fields,
- computation of the genus of an algebraic function field, handling of places, divisors and Riemann-Roch spaces,
- computation of maximal orders and ideal arithmetic in function fields,

- computation of reduced bases and fundamental units in global function fields,
- arithmetic in relative lattices,
- solving Thue and unit equations, integral points on Mordell curves,
- solving index form equations.

The development of KANT V4 as well as KASH is continued in view of providing the user with the most advanced tools for computations in algebraic number fields. Suggestions for important features to be included are welcome.

Acknowledgements

This program could not have been written without the support of Prof. J. Cannon, W. Bosma, S. Collins and A. Steele at the University of Sydney. Additionally, we would like to thank Prof. Dr. J. Neubüser at the RWTH Aachen, Germany for his permission to use and modify a large portion of the GAP source code. Special thanks also go to M. Schönert, a main contributor to the creation of GAP, for his kind support and aid. Finally, we would like to thank A. Weber at Cornell University, who developed the database for algebraic number fields and M. Klebel who did the (Ray) class fields for imaginary quadratic number fields.

Introduction

The following sections introduce you to the KASH system. A step by step introduction should give you an impression of how the KASH system works. After reading these sections the reader should know what kind of problems can be handled with KASH and how they can be handled.

KASH is based on the shell of GAP; however, GAP's functions have been replaced with KANT V4 functions. Although most functions listed in the GAP manual will not work in KASH, the GAP manual is still an invaluable reference for the general usage of the shell (i.e., the syntax of input and programming language, etc.). For a listing of the KANT V4 functions currently available in KASH, please refer to the reference manual.

The printed examples encourage you to try running KASH on your computer. This will support your feeling for KASH as a tool, which is the leading aim of this section. Do not believe any statement in this section so long as you cannot verify it with your own version of KASH. You will learn to distinguish between small deviations of the behavior of your personal KASH from the printed examples and serious nonsense.

In case you encounter serious nonsense it is highly recommended that you send a bug report to *kant@math.tu-berlin.de*.

Overall Organisation

This manual is organized into the following sections.

- **Interactive Use of KASH**

  This section describes how to start KASH and how to leave it. It provides a succinct discussion of the basic data types supported by KASH such as reals, matrices and polynomials. We also describe the main points of the KASH shell.

- **Algebraic Number Theory I-II**

  In these two sections we explain in great detail the central part of KASH. We establish the concepts necessary to understand and use orders, algebraic numbers and ideals in KASH. Such tasks as the computation of maximal order, unit and class group of an algebraic number field are discussed here. The first section deals with absolute extensions whereas relative extensions are discussed in the second section.

- **Algebraic function fields**

  In this section the basic concepts of KASH dealing with algebraic function fields are explained. It is shown in detail how to define an algebraic function field, how to compute it's genus, and how to work with elements and ideals in orders.

- **Lattices**

  This section gives an introduction on how to use lattices in KASH.

- **Programming**

  This section provides a brief description of the programming language. of KASH. You can skip this section if you are already familiar with the programming language of GAP.

Notice that most functions described in the following introduction allow alternative and more sophisticated calling sequences providing the user with additional options. A complete description of every function is contained in the reference manual.

1 Interactive Use of KASH

The first section is devoted to explaining the basic types supported by KASH and how to effectively use them. As the main points of the KASH shell are given in conjunction with the type specific information, special attention should be paid to the examples. To begin with we describe how to start KASH (you may have to ask your system administrator to install it correctly) and how to leave it.

1.1 Starting and Leaving KASH

If KASH is correctly installed, then you start KASH by simply typing `kash` at the prompt of your operating system. If you are successful in starting KASH, the KASH logo should appear, at which time a command or function call may be entered. To exit KASH type

```
quit;
```

at the prompt (the semicolon is necessary!). From now on, we will display the KASH prompt `kash>` to mean that the user is to type what comes after the `kash>` in the manual when the prompt appears on the screen.

1.2 First steps

A simple calculation with KASH 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*.

```
kash> (9 - 7) * (5 + 6);
22
kash>
```

Then KASH echoes the result 22 on the next line and shows with the prompt that it is ready for the next problem.

If you did omit the semicolon at the end of the line but have already typed *return*, then KASH 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.

```
kash> (9 - 7) * (5 + 6)
> ;
22
kash>
```

Whenever you see this partial prompt and you cannot decide what KASH is still waiting for, then you have to type semicolons until the normal prompt returns. In the following examples we will omit this prompt on the line after the result. Considering each example as a continuation of its predecessor this prompt occurs in the next example.

1.3 Online Help

For online help, `?` is a valuable tool. A single question mark followed by the name of a function (without the final semicolon!) causes the description of the function found in the reference manual to appear on the screen. If the function is not found, a list will appear of all functions beginning with that name. If a list of all functions beginning with a particular string is desired, use `??` followed by the string to match.

For example, to produce the manual page corresponding to the KASH function `Order`, use

```
kash> ?Order
```

and wait for the manual page to appear on the screen. The list of all functions beginning with Order can be obtained by

```
kash> ??Order
```

with the two question marks. Suppose that

```
kash> ?OrderSet
```

had been given. As there is no function with that name, but several which begin with that string, the `?` will behave like the `??` and produce a list for possibilities. If no match is found, the response `Help: no topic with this name was found` is displayed.

1.4 Line Editing

KASH allows you to edit the current input line with a number of editing commands. Those commands are accessible as **control keys**. You enter a control key by pressing the Ctrl key, and, while still holding the Ctrl key down, hitting another key. Below we denote control keys by `<Ctrl>-<key>`. The case of `<key>` does not matter, i.e., `<Ctrl>-A` and `<Ctrl>-a` are equivalent.

In the table below we list the most important commands for cursor movement, deleting text and yanking text.

Keystrokes	**Action**
`<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

The Tab key looks at the characters before the cursor, interprets them as the beginning of an identifier and tries to complete this identifier. If there is more than one possible completion, it completes to the longest common prefix of all those completions. If the characters to the left of the cursor are already the longest common prefix of all completions hitting Tab a second time will display all possible completions.

The next commands allow you to fetch previous lines. This history is limited to about 8000 characters.

Keystrokes	Action
<Ctrl>-L	insert last input line before current character
<Ctrl>-P	redisplay the last input line, another <Ctrl>-P will redisplay the line before that, etc. If the cursor is not in the first column only the lines starting with the string to the left of the cursor are taken.
<Ctrl>-N	like <Ctrl>-P but goes the other way round through the history.

1.5 Constants and Operators

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 KASH, arithmetical operators, comparison operators, and logical operators. You have already seen that it is possible to form the sum, the difference, and the product of two integer values. There are some more operators applicable to integers in KASH. Of course integers may be divided by each other, possibly resulting in noninteger rational values.

```
kash> 12345/25;
2469/5
```

Note that the numerator and denominator are divided by their greatest common divisor and that the result is uniquely represented as a division instruction.

We haven't met negative numbers yet. So consider the following self–explanatory examples.

```
kash> -3; 17 - 23;
-3
-6
```

The exponentiation operator is written as `^`. This operation in particular might lead to very large numbers. This is no problem for KASH as it can handle numbers of (almost) arbitrary size.

```
kash> 7^69;
205005145156954906122290109080958673914396262484637238056 07
```

KASH knows a precedence between operators that may be overridden by parentheses.

Besides these arithmetical operators there are comparison operators in KASH. 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 `<>`.

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`.

1.6 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 KASH is only used for the test of equality.

```
kash> a:= (9 - 7) * (5 + 6);
22
kash> a;
22
kash> a * (a + 1);
506
```

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.

```
kash> a:= 2;;
```

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 KASH distinguishes upper and lower case, `a` and `A` are different names. Keywords such as `quit` must not be used as names. The following letters are reserved for use as fixed objects and may not be used as variable names:

`C`	field of complex numbers
`e`	2.7182 ...
`pi`	$\pi = 3.14159 \ldots$
`Q`	field of rational numbers
`R`	field of real numbers
`Z`	ring of integers
`Zx`	polynomial algebra over `Z`

In order to determine the type of a named variable or constant, the function `TYPE` is used. For example, the command

```
kash> TYPE(e);
```

produces the result

```
"KANT real"
```

Whenever KASH returns a value by printing it on the next line, this value is assigned to the variable `last`. So if you computed

```
kash> (9 - 7) * (5 + 6);
22
```

and forgot to assign the value to the variable `a` for further use, you can still do it by the following assignment.

```
kash> a:= last;
22
```

Moreover there are variables `last2` and `last3`, guess their values.

1.7 Integers and rationals

KASH integers are entered as a sequence of digits optionally preceded by a `+` sign for positive integers or a `-` sign for negative integers. In KASH, 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 (see 1.8); automatic coercion is applied where necessary.

```
kash> 3+3;
6
kash> 3*3;
9
kash> 3^4;
81
kash> 11111^5;
169342410709747836551
```

KASH provides the following integer functions; refer to the reference manual for detailed descriptions and examples.

`Factor`	Returns the factorization of an integer.
`IntGcd`	Returns the greatest common divisor of two integers.
`IntIsPrime`	Checks whether an integer is a prime.
`IntLcm`	Returns the least common multiple of two integers.
`IntQuo`	Divides one integer by another and returns an integer quotient.
`NextPrime`	Returns the smallest prime larger than an integer.

Since integers are naturally embedded in the field of real numbers, all real functions are applicable to integers (see 1.8).

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.

```
kash> 4/6
2/3
```

All real functions are applicable to rationals (see 1.8).

1.8 Real and complex numbers

Real numbers can only be stored in the computer effectively in the form of approximations. KASH 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. To reset this precision to `n`, the call

```
kash> Prec(n);
```

is used.

The following self–explanatory examples show how to create real numbers in KASH and how to do arithmetic.

```
kash> 123.45;
123.45
kash> 34.5+20.2;
54.7
kash> 34.5/17.1;
2.0175438596491228
kash> 12.0E+30;
1.2e31
```

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

`Abs`	Returns the absolute value of a real number.
`Ceil`	Finds the smallest integer larger than or equal to a real number.
`Cos`	Evaluates the cosine of an angle in radians.
`Exp`	Computes the exponential of a real number.
`Floor`	Finds the largest integer less than or equal to a real number.
`Log`	Evaluates the natural logarithm of a real number.
`Sin`	Evaluates the sine of an angle in radians.
`Sqrt`	Computes the square root of a real number.

`Tan` Evaluates the tangent of an angle in radians.

In KASH, complex numbers have the same precision as real numbers. This precision can be modified by calling the `Prec` function (see above). A complex number can be designated using the function `Comp`, which requires two real arguments.

```
kash> z := Comp(1,2);
1 + 2*i
kash> z+1;
2 + 2*i
kash> z*z;
-3 + 4*i
```

Most real functions can be applied to complex numbers. Additionally, KASH provides the following complex functions.

`Arg` Returns the argument of a complex number.

`Im` Returns the imaginary part of a complex number.

`Re` Returns the real part of a complex number.

1.9 Polynomials

At the moment, KASH can only handle univariate polynomials. These polynomials can be defined using the routine `Poly` which takes two arguments. The first one is the polynomial algebra the polynomial comes from; the second argument holds a list of the coefficients (a list is a collection of objects separated by commas and enclosed in brackets; see 1.11).

For example, to create the polynomial $f(x) = x^3 + x + 1 \in \mathbb{Z}[x]$ in KASH, you can do it by the assignment

```
kash> f := Poly(Zx,[1,0,1,1]);
x^3 + x + 1
```

Let's look closely at this example. The first argument `Zx` is the predefined constant for the polynomial algebra $\mathbb{Z}[x]$. In the second argument we pass the list

$[1, 0, 1, 1]$ which contains the coefficients of $f(x)$. The list begins with the coefficient of the highest exponent down to that of the constant, including all zeroes.

The following self-explanatory examples show how to do some artihmetic.

```
kash> f+f;
2*x^3 + 2*x + 2
kash> f*f;
x^6 + 2*x^4 + 2*x^3 + x^2 + 2*x + 1
kash> f^3;
x^9 + 3*x^7 + 3*x^6 + 3*x^5 + 6*x^4 + 4*x^3 + 3*x^2 + 3*x + 1
kash> 5*f;
5*x^3 + 5*x + 5
```

Use the `Eval` routine to calculate the value of f when the variable x is substituted by certain values.

```
kash> Eval(f,1);
3
kash> Eval(f,10);
1011
kash> Eval(f,f);
x^9 + 3*x^7 + 3*x^6 + 3*x^5 + 6*x^4 + 5*x^3 + 3*x^2 + 4*x + 3
```

Suppose now, that we want to enter the polynomial $g(x) = \frac{1}{2}x^2 + x + \frac{2}{3} \in \mathbb{Q}[x]$ in KASH. This is not as easy as entering a polynomial over the integers, because the polynomial algebra $\mathbb{Q}[x]$ is not provided by KASH. To create the polynomial algebra $S[x]$ with coefficients from a designated ring S, the routine `PolyAlg` should be used. This routine requires one argument, namely the coefficient ring of the polynomial algebra. Recalling that `Q` is the predefined constant for the ring $\mathbb{Q}$, the following assignment creates the polynomial algebra $\mathbb{Q}[x]$ in KASH.

```
kash> Qx := PolyAlg(Q);
Univariate Polynomial Algebra in x over Rational Field
```

Now we are ready to define $g(x)$.

```
kash> g := Poly(Qx,[1/2,1,2/3]);
1/2*x^2 + x + 2/3
```

There are many KASH functions dealing with polynomials. They are all explained in the reference manual. The names usually start with Poly. The most important ones are listed below.

`Factor`	Returns the factorization of a polynomial.
`PolyDeg`	Returns the degree of a polynomial.
`PolyDeriv`	Returns the derivative of a polynomial.
`PolyDisc`	Returns the discriminant of a polynomial.
`PolyGcd`	Returns the greatest common divisor of two polynomials.

1.10 Matrices

To create a matrix in KASH, the routine `Mat` should be used. It requires two arguments. The first one is a ring from which the matrix entries come; the second argument is a list containing the rows of the matrix, each row again being a list.

For example, to create the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \in \mathbb{Z}^{2\times 2},$$

the following assignment will do it

```
kash> A := Mat(Z,[[1,2],[3,4]]);
[1 2]
[3 4]
```

Recall that there are several predefined constants which can be used as a ring type (for example, `Z, Q, R, C, Zx`). Other objects, such as an order (see 2.1) or a polynomial algebra, may be used as a ring type as well. See **Mat** in the reference manual for more examples.

We can easily access an entry or a row.

```
kash> A[2][2];
4
kash> A[2];
[3 4]
kash> A[2][2]:=5;;A;
[1 2]
[3 5]
```

See **MatElt** in the reference manual for the complete bracket notation capabilities.

Let's do some simple arithmetic.

```
kash> A+A;
[2 4]
[6 8]

kash> A^2-5*A;
[2 0]
[0 2]
```

KASH provides several linear algebra routines which are all explained in the reference manual. Their names usually start with Mat. The most important ones are catalogued below.

`MatCharPoly`	Returns the characteristic polynomial of a matrix.
`MatDet`	Returns the determinant of a matrix.
`MatInv`	Returns the inverse of a matrix.
`MatTrace`	Returns the trace of a matrix.
`MatTrans`	Returns the transpose of a matrix.

1.11 Lists

A *list* is a collection of objects separated by commas and enclosed in brackets. Let's for example construct the list `primes` of the first 10 prime numbers.

```
kash> primes:= [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ]
```

The next two primes are 31 and 37. They may be appended to the existing list by the function `Append`, which takes the existing list as its first and another list as a second argument. The second argument is appended to the list `primes` and no value is returned.

```
kash> Append(primes, [31, 37]);
kash> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37 ]
```

You can also add single new elements to existing lists by the function `Add` which takes the existing list as its first argument and a new element as its second argument. The new element is added to the list `primes` and again no value is returned but the list `primes` is changed.

```
kash> Add(primes, 41);
kash> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 ]
```

Single elements of a list are referred to by their position in the list. To get the value of the seventh prime, that is the seventh entry in our list `primes`, you simply type

```
kash> primes[7];
17
```

and you will get the value of the seventh prime. This value can be handled like any other value, for example multiplied by 2 or assigned to a variable. On the other hand, this mechanism allows a value to be assigned to a position in a list. So the next prime 43 may be inserted in the list directly after the last occupied position of `primes`. This last occupied position is returned by the function `Length`.

```
kash> Length(primes);
13
kash> primes[14] := 43;
43
kash> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43 ]
```

Note that this operation has again changed the object `primes`. Not only the next position of a list is capable of taking a new value. If you know that 71 is the 20th prime, you can enter it right now in the 20th position of `primes` as well. This will result in a list with holes, which is, however, still a list, the list being of length 20.

```
kash> primes[20] := 71;
71
kash> Length(primes);
20
kash> primes;
[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,,,,,, 71 ]
```

The list itself, however, must exist before a value can be assigned to a position of the list. This list may be the empty list `[ ]`.

```
kash> L[1] := 2;
Error, Variable: 'L' must have a value
kash> L := [];
[  ]
kash> L[1] := 2;
2
```

Of course existing entries of a list can be changed by this mechanism, too.

In all of the above changes to the list `primes`, the list has been automatically resized. There is no need for you to tell KASH how big you want a list to be.

It is not necessary for the objects collected in a list to be of the same type.

```
kash> L := [1,2,true,3/4,Poly(Zx,[1,0,2])];
[ 1, 2, true, 3/4, x^2 + 2 ]
```

In the same way a list may be part of another list.

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

```
1
kash> L[1][2];
2
kash> L[2][1];
3
kash> L[2][2];
4
```

In the rest of this section we introduce some further data types. You should read this part when you start to write programs in KASH . If you are beginning to use KASH, jump to the second section **Algebraic Number Theory**, which describes the central part of KASH.

1.12 Sets

A set in KASH is a special kind of list. A set contains no holes, and its elements are sorted according to the KASH ordering of all its objects. Moreover, a set contains no object twice.

For any list there exists a corresponding set. This set is constructed by the function `Set` which takes the list as its argument and returns a set obtained from this list by ignoring holes and duplicates and by sorting the elements.

```
kash> L := [1,4,2,7,4,9];
[ 1, 4, 2, 7, 4, 9 ]
kash> S := Set(L);
[ 1, 2, 4, 7, 9 ]
```

Note that the original list `L` is not changed by the function `Set`. We have to make a new assignment to the variable `S` in order to make it a set.

The `in` operator is used to test whether an object is an element of a set. It returns a boolean value `true` or `false`.

```
kash> 4 in S;
true
kash> 5 in S;
false
```

The `in` operator may be applied to ordinary lists as well. It is, however, much faster to perform a membership test for sets since sets are always sorted and a binary search can be used instead of a linear search.

New elements may be added to a set by the function `SetAdd` which takes the set `S` as its first argument and an element as its second argument and adds the element to the set if it wasn't already there. Note that the object `S` is changed.

```
kash> SetAdd(S,5);
kash> SetAdd(S,10);
kash> S;
[ 1, 2, 4, 5, 7, 9, 10 ]
```

Sets can be intersected by the function `SetIntersect` and united by the function `SetUnite` which both take two sets as their arguments and change their first argument to be the result. The second argument of the functions `SetIntersect` and `SetUnite` may as well be ordinary lists.

```
kash> T := [3,8];
[ 3, 8 ]
kash> SetUnite(S,T);
kash> S; [ 1, 2, 3, 4, 5, 7, 8, 9, 10 ]
```

1.13 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.

For example, to create the list of all positive integers less than or equal to 100, you can use the following assignment.

```
kash> L := [1..100];
[ 1 .. 100 ]
kash> Length(L);
100
```

```
kash> L[1];
1
kash> L[50];
50
```

To get the list of all even integers between 2 and 100, simply type

```
kash> L := [2,4..100];
[ 2, 4 .. 100 ]
kash> L[1];
2
kash> L[2];
4
kash> Length(L);
50
```

1.14 Records

A record provides another way to build new data structures. Like a list, a record is a collection of other objects. In a record the elements are not indexed by numbers, but by names (identifiers). An entry in a record is called a *record component* (or sometimes also record field).

```
kash> date := rec(year:=1995, month:=10, day := 27);
rec(
  year := 1995,
  month := 10,
  day := 27 )
```

Initially, a record is defined as a list consisting of assignments to its record components, which are separated by commas Then the value of a record component is accessible by the record name and the record component name separated by one dot as the record component selector.

```
kash> date.year;
1995
kash> date.month;
10
```

Assignments to new record components are possible in the same way. The record is automatically resized to hold the new component.

```
kash> date.time := rec(hour:=15, minute := 40);
rec(
  hour := 15,
  minute := 40 )
kash> date;
rec(
  year := 1995,
  month := 10,
  day := 27,
  time := rec(
      hour := 15,
      minute := 40 ) )
```

Records are objects that may be changed. An assignment to a record component changes the original object.

```
kash> date.year := 1997;
1997
kash> date;
rec(
  year := 1997,
  month := 10,
  day := 27,
  time := rec(
      hour := 15,
      minute := 40 ) )
```

Sometimes it is interesting to know which components of a certain record are bound. This information is available from the function `RecFields` which takes a record as its argument and returns a list of all bound components of this record as a list of strings.

```
kash> RecFields(date);
[ "year", "month", "day", "time" ]
```

2 Algebraic Number Theory I : Absolute extensions

This section describes the central part of KASH. After learning how to do simple arithmetic in algebraic number fields using KASH, you will be able to compute the main invariants of algebraic number fields.

2.1 Arithmetic

We call $\alpha \in \mathbb{C}$ an algebraic integer if there exists a monic irreducible polynomial $f(x) \in \mathbb{Z}[x]$ with $f(\alpha) = 0$. An algebraic number field $\mathcal{F}$ is a finite extension of the field of rationals $\mathbb{Q}$. There always exists an algebraic integer $\rho \in \mathbb{C}$ such that $\mathcal{F} = \mathbb{Q}(\rho)$. The set of algebraic integers in $\mathcal{F}$ forms a ring which is denoted by $\mathcal{O} = \mathcal{O}_{\mathcal{F}}$. An order $\mathfrak{o}$ in $\mathcal{F}$ is a unital subring of $\mathcal{O}$ which, as a $\mathbb{Z}$–module, is finitely generated and of rank $[\mathcal{F} : \mathbb{Q}]$. Of course, $\mathcal{O}$ is an order which we call the maximal order of $\mathcal{F}$.

In KASH, any computations in an algebraic number field $\mathcal{F}$ are performed with respect to a certain order in $\mathcal{F}$. Suppose that we want do some arithmetic in the field $\mathcal{F} = \mathbb{Q}(\rho)$, where ρ is a zero of the polynomial

$$f(x) = x^5 + 4x^4 - 56x^2 - 16x + 192 \in \mathbb{Z}[x]$$

(you can easily check that $f(x)$ is irreducible using the KASH command `Factor`). We will create the equation order $\mathbb{Z}[\rho]$ in $\mathcal{F}$ using the `Order` function. There are currently four ways in KASH to call this routine. Here, we use the simplest one, which takes only one argument, namely the minimal polynomial of ρ.

```
kash> f := Poly(Zx,[1,4,0,-56,-16,192]);
x^5 + 4*x^4 - 56*x^2 - 16*x + 192
kash> o := Order(f);
Generating polynomial:  x^5 + 4*x^4 - 56*x^2 - 16*x + 192
```

The computation of the discriminant and signature of an order are both basic tasks (by the signature of an order we mean of course the signature of the algebraic number field the order comes from).

```
kash> OrderDisc(o);
1364202618880
kash> OrderSig(o);
[ 1, 2 ]
```

The `OrderSig` routine returns a list whose first entry is the number of real embeddings of $\mathcal{F}$ in $\mathbb{C}$. The second entry equals the number of the complex embeddings. Remember, that KASH has Tab completion (see 1.4). Rather than typing `OrderDisc`, type `Or` and press Tab, then type `Di` and press Tab again.

To do some arithmetic in $\mathbb{Z}[\rho]$ we need to enter elements from $\mathbb{Z}[\rho]$ in KASH. Any element $\alpha \in \mathbb{Z}[\rho]$ has a unique representation

$$\alpha = a_1 + a_2\rho + a_3\rho^2 + a_4\rho^3 + a_5\rho^4$$

with $a_1, \ldots, a_5 \in \mathbb{Z}$. We can create α by invoking the KASH routine `Elt` which requires two arguments. The first one is the order $\mathbb{Z}[\rho]$. In the second argument the representation of α from above is passed as a list. For example, the following commands creates ρ, $\alpha = 2 + 3\rho + 4\rho^2 - 1\rho^3 + 6\rho^4$ and $\beta = -2\rho^3 + \rho^4$.

```
kash> rho := Elt(o,[0,1,0,0,0]);
[0, 1, 0, 0, 0]
kash> alpha := Elt(o,[2,3,4,-1,6]);
[2, 3, 4, -1, 6]
kash> beta := Elt(o,[0,0,0,-2,1]);
[0, 0, 0, -2, 1]
```

Now we are ready to do some simple calculations.

```
kash> rho+1;
[1, 1, 0, 0, 0]
kash> rho+rho^2;
[0, 1, 1, 0, 0]
kash> alpha-beta;
[2, 3, 4, 1, 5]
kash> alpha*beta;
[54720, -34128, -6392, 6876, -840]
```

So far, we have only entered elements from $\mathbb{Z}[\rho]$. However, the `Elt` function enables you also to create an element $\delta \in \mathcal{F}$ which is not contained in $\mathbb{Z}[\rho]$. Having a representation

$$\delta = \frac{d_1 + d_2\rho + d_3\rho^2 + d_4\rho^3 + d_5\rho^4}{d}$$

with $d_1, \ldots, d_5, d \in \mathbb{Z}, d \neq 0$, you can enter δ in KASH by calling the `Elt` routine and passing in as its second argument a list containing $d_1, \ldots, d_5$ followed by a division bar `/` and the denominator d. For example, the assignment below produces $\delta = \frac{1}{2}\rho + \frac{3}{2}\rho^2 \in \mathcal{F}$.

```
kash> delta := Elt(o,[0,1,3,0,0]/2);
[0, 1, 3, 0, 0] / 2
```

This assignment is tantamount to

```
kash> delta := Elt(o,[0,1/2,3/2,0,0]);
[0, 1, 3, 0, 0] / 2
```

There are many KASH functions dealing with algebraic numbers. Their names usually start with Elt. Look at the reference manual for a complete list. Here, we present the functions `EltNorm` and `EltTrace` which compute norms and traces of algebraic numbers.

```
kash> EltNorm(rho);
-192
kash> EltTrace(rho);
-4
kash> EltNorm(alpha);
16256331150880
kash> EltTrace(alpha);
-3498
```

2.2 Maximal orders

We are now going to compute the maximal order $\mathcal{O}$ of $\mathcal{F}$. This means we compute an integral basis $\omega_1, \ldots, \omega_5$ of $\mathcal{F}$.

```
kash> O := OrderMaximal(o);
   F[1]
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^5 + 4*x^4 - 56*x^2 - 16*x + 192
Discriminant: 1301005
```

Let's look closely at this example. So far, we have only been concerned with an equation order, namely $\mathbb{Z}[\rho]$. When KASH prints any equation order over $\mathbb{Z}$, it displays the corresponding polynomial. However, the maximal order $\mathcal{O}$ of $\mathcal{F}$ returned by the `OrderMaximal` function is not given as an equation order, but by a transformation matrix $T \in \mathbb{Q}^{5\times 5}$ such that

$$(w_1, \ldots, w_5) = (1, \rho, \ldots, \rho^4) \cdot T.$$

Instead of printing the transformation matrix, KASH displays a graph as above. The vertices `Q`, `F[1]` and `F[2]` in this graph are algebraic number fields which are the quotient fields of pairwise distinct orders (orders are supposed to be distinct in KASH when their bases differ). The vertex at the top of the graph corresponds to the order which is displayed. Therefore, `F[1]` in the above graph corresponds to the maximal order `O` returned by the `OrderMaximal` function. From the description at the bottom of the graph we get the information that the field `F[2]` is the quotient field of the equation order corresponding to the polynomial $x^5 + 4x^4 - 56x^2 - 16x + 192$. This means that `F[2]` represents the quotient field of $\mathbb{Z}[\rho]$. Of course, the vertex named `Q` stands for $\mathbb{Q}$. Notice, that vertices connected by a vertical edge `|` represent the same number field, whereas a slanted edge indicates that the field at the bottom of this edge is properly contained in the field at the other end.

The `OrderMaximal` function computes an integral basis $\omega_1, \ldots, \omega_5$ of $\mathcal{F}$. We are now going to enter the numbers $\omega_1, \ldots, \omega_5$ in KASH. So far, we have entered algebraic numbers whose representations with respect to the power basis $1, \rho, \ldots, \rho^4$ were known. By using the equation order $\mathbb{Z}[\rho]$ as the first argument we told the `Elt` function that the coefficients in the second argument are given with respect to this basis. To enter a number given as $a_1\omega_1 + \cdots + a_5\omega_5$ with $a_1, \ldots, a_5 \in \mathbb{Q}$, we again use the `Elt` function by passing the maximal order as its first argument. Therefore, it's quite easy to enter $\omega_1, \ldots, \omega_5$ in KASH.

```
kash> w1 := Elt(O,[1,0,0,0,0]);
1
kash> w2 := Elt(O,[0,1,0,0,0]);
[0, 1, 0, 0, 0]
kash> w3 := Elt(O,[0,0,1,0,0]);
[0, 0, 1, 0, 0]
kash> w4 := Elt(O,[0,0,0,1,0]);
[0, 0, 0, 1, 0]
kash> w5 := Elt(O,[0,0,0,0,1]);
[0, 0, 0, 0, 1]
```

After entering $\omega_1, \ldots, \omega_5$ in KASH we will compute the representations of $\omega_1, \ldots, \omega_5$ with respect to the power basis $1, \rho, \ldots, \rho^4$. To do so, the `EltMove` function should be used. When you call this routine, passing an algebraic number γ and an order $\mathfrak{o}$ as arguments, it returns a copy of γ represented with respect to the basis of $\mathfrak{o}$. Let's look at the following lines that should clarify how `EltMove` works.

```
kash> EltMove(w1,o);
1
kash> EltMove(w2,o);
[0, 1, 0, 0, 0] / 2
kash> EltMove(w3,o);
[0, 0, 1, 0, 0] / 4
kash> EltMove(w4,o);
[0, 0, 0, 1, 0] / 8
kash> EltMove(w5,o);
[0, 0, 0, 0, 1] / 16
```

The results returned by `EltMove` are copies of the numbers $\omega_1, \ldots, \omega_5$. Each copy is given with respect to the power basis $1, \rho, \ldots, \rho^4$ of $\mathbb{Z}[\rho]$. Therefore, we have[2]

$$\omega_1 = 1, \quad \omega_2 = \frac{1}{2}\rho, \quad \omega_3 = \frac{1}{4}\rho^2, \quad \omega_3 = \frac{1}{8}\rho^3, \quad \omega_5 = \frac{1}{16}\rho^4 .$$

Earlier, the algebraic number ρ was defined with respect to the power basis of $\mathbb{Z}[\rho]$. Using `EltMove` you can easily compute the representation of ρ with respect to $\omega_1, \ldots, \omega_5$. We have $\rho = 2\omega_2$.

```
kash> EltMove(rho,0);
[0, 2, 0, 0, 0]
```

Let's now discuss the concept of moving algebraic numbers more closely. In KASH, every order has a fixed basis. The basis of an equation order corresponding to a certain polynomial $g(x)$ is always a power basis built by powers of a root of $g(x)$. Two orders are considered as distinct in KASH if their bases differ. When you enter an algebraic number γ in KASH by calling the `Elt` routine, γ is defined with respect to the fixed basis of the order which is passed in the first argument of `Elt`, say `ord1`. The `Elt` routine returns an object, say `gamma1`, which represents the algebraic number γ. The object `gamma1` depends on the order `ord1` which was used to define `gamma1`. For example, printing the object `gamma1` always involves `ord1`. Assume now, that `ord2` is another order in KASH. By invoking the `EltMove` function we can get the representation of γ with respect to the basis of `ord2`. In fact, the `EltMove` routine creates a new object, say `gamma2`, which also represents the algebraic number γ. Of course, the new object `gamma2` depends on `ord2`.

When doing arithmetic with algebraic numbers in KASH, it is sometimes necessary to invoke the `EltMove` function, because operations like `+`,`-`, `*` and `/` require that the operands depends on the same order. For example, assume, that we want to compute $\omega_2 + \rho$ in KASH. You must either move the object `w1` to $\mathbb{Z}[\rho]$ or the object `rho` to $\mathcal{O}$.

```
kash> w2+rho;
Error, different orders
```

[2]To obtain these representations of $\omega_1, \ldots, \omega_5$ in terms of ρ you should use the `OrderBasis` routine in practice. Refer to the reference manual for a detailed description.

```
kash> EltMove(w2,o)+rho;
[0, 3, 0, 0, 0] / 2
kash> w2+EltMove(rho,O);
[0, 3, 0, 0, 0]
```

Notice that the first result (`[0, 3, 0, 0, 0] / 2`) is an object which depends on $\mathbb{Z}[\rho] =$ `o`, whereas the second one (`[0, 3, 0, 0, 0]`) depends on $\mathcal{O} =$ `O`.

2.3 Unit groups

KASH provides several functions dealing with the units of an order. They are all explained in the reference manual. Here, we discuss only the most important ones. To compute a system of fundamental units the `OrderUnitsFund` function should be used. This function taking only one argument, namely an order, returns a list whose entries are algebraic numbers forming a system of fundamental units.

```
kash> OrderUnitsFund(O);
[ [-1, 1, 0, 0, 0], [1, 0, -1, 0, 0] ]
```

Notice that the entries `[-1, 1, 0, 0, 0]` and `[1, 0, -1, 0, 0]` depend on the order $\mathcal{O}$. Therefore, $-1+\omega_1$ and $1-\omega_2$ forms a system of fundamental units in $\mathcal{O}$. To obtain the regulator of $\mathcal{O}$, just type

```
kash> OrderReg(O);
5.6855418836578346
```

A generator for the torsion subgroup can be computed by the `OrderTorsionUnit` routine. Obviously, $\mathcal{O}$ contains only the torsion units $+1$ and -1.

```
kash> OrderTorsionUnit(O);
-1
```

KASH enables you to compute fundamental units in arbitrary orders.

```
kash> OrderUnitsFund(o);
[ [-11518081, 26088, 4956996, 1680132, 236679],
  [493088471, 153109288, -254564324, -27045279, 33856238] ]
```

2.4 Ideals

In KASH 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 KASH. The most basic one is to use the function `Ideal`. For example, the ideal $\mathfrak{a} = \langle \omega_3, \omega_4 \rangle \subseteq \mathcal{O}$ can be created by the assignment

```
kash> a := Ideal(w3,w4);
<[0, 0, 1, 0, 0], [0, 0, 0, 1, 0]>
```

Earlier we learned that the object which represents an algebraic number in KASH always depends on an order. This concept also applies to ideals in KASH. Since `w3` and `w4` depend on $\mathcal{O}$, the ideal `a` is linked to $\mathcal{O}$ as well. Before doing some arithmetic with ideals, we will generate another ideal, namely the principal ideal $\mathfrak{b} = \langle 2\,\omega_2 \rangle \subseteq \mathcal{O}$. Again, we invoke the `Ideal` routine.

```
kash> b := Ideal(2*w2,2*w2);
<[0, 2, 0, 0, 0], [0, 2, 0, 0, 0]>
```

However, this assignment is tantamount to each of the following ones.

```
kash> b := Ideal(2*w2);
<[0, 2, 0, 0, 0]>
kash> b := 2*w2*O;
<[0, 2, 0, 0, 0]>
```

The sum and the difference of two ideals is the smallest ideal which contains 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.

```
kash> a^2;
<1296, [12, -8, -13, 7, 4]>
kash> a*b;
<[0, 72, 0, 0, 0], [0, 0, 0, 0, 2]>
```

```
kash> c := a+b;
<
[12  6  0  0  0]
[ 0  1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0  1  0]
[ 0  0  0  0  1]
>
```

Let's see what's going on here. $\mathfrak{a}^2$ is the ideal generated by 1296 and $12 - 8\,\omega_2 - 13\,\omega_3 + 7\,\omega_4 + 4\,\omega_5$. The product $\mathfrak{a} \cdot \mathfrak{b}$ is the ideal generated by $72\,\omega_2$ and $2\,\omega_5$. However, the result of $\mathfrak{c} = \mathfrak{a} + \mathfrak{b}$ is printed in a quite different way. As $\mathfrak{c}$ is a finitely generated $\mathbb{Z}$–module of rank 5, there is always a transformation matrix $T \in \mathbb{Z}^{5\times 5}$ such that

$$T \cdot (\omega_1, \ldots, \omega_5)$$

is a $\mathbb{Z}$–basis for $\mathfrak{c}$. Such a transformation matrix is printed as above because KASH has not computed another representation of $\mathfrak{c}$ yet. Therefore, a $\mathbb{Z}$–basis of $\mathfrak{c}$ is given by

$$\omega_1' = 12, \quad \omega_2' = 6 + \omega_2, \quad \omega_3' = \omega_3, \quad \omega_4' = \omega_4, \quad \omega_5' = \omega_5\,.$$

Recalling that $\mathcal{O}$ is a Dedekind ring, there are algebraic numbers $c_1, c_2 \in \mathcal{O}$ such that $\langle c_1, c_2 \rangle$ equals $\mathfrak{c}$. Such numbers can be computed by applying the `Ideal2EltAssure` function to the ideal $\mathfrak{c}$. The next time $\mathfrak{c}$ is printed, these numbers instead of the transformation matrix are displayed.

```
kash> Ideal2EltAssure(c);
kash> c;
<12, [-12, -2, -1, 1, -2]>
```

So far, we have only considered integral ideals. KASH 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 Dedeking ring the fractional ideals form a group under multiplication). In the following example the inverse of $\mathfrak{a}$ is computed.

```
kash> 1/a;
<
[36  0  0  0 29]
[ 0 36  0  0 29]
[ 0  0 36  0 12]
[ 0  0  0 36  8]
[ 0  0  0  0  1]
/36>

kash> a*last;
<1>
```

We have

$$\mathfrak{a}^{-1} = \frac{36 + 36\,\omega_2\,\mathbb{Z} + 36\,\omega_3\,Z + 36\,\omega_4\,\mathbb{Z} + (29 + 29\,\omega_2 + 12\,\omega_3 + 8\,\omega_4 + \omega_5)\,\mathbb{Z}}{36}\,.$$

There are many KASH functions dealing with ideals. Their names usually start with Ideal. Look at the reference manual for a complete list. Here, we will discuss the functions `Factor`, `IdealNorm` and `IdealIsPrincipal`.

In maximal orders which are Dedekind rings every ideal can uniqely be factorized in prime ideals. The `Factor` routine returns a list containing the prime ideal decomposition of a certain ideal. Here, we factor the principal ideal $\langle 5\rangle \subseteq \mathcal{O}$.

```
kash> L := Factor(5*O);
[ [ <5, [1, 2, 0, 0, 0]>, 2 ], [ <5, [4, 2, 0, 0, 0]>, 1 ],
  [ <5, [3, 6, 4, 0, 0]>, 1 ] ]
```

Let's look at the result `L` which is a list containing three sublists. Each sublist has two entries. The first one is a prime ideal, whereas the second entry is the corresponding exponent. Therefore, we have

$$\langle 5\rangle = \langle 5, 1 + 2\,\omega_2\rangle^2 \cdot \langle 5, 4 + 2\,\omega_2\rangle \cdot \langle 5, 3 + 6\,\omega_2 + 4\,\omega_3\rangle\,.$$

The factorization can easily be checked in KASH.

```
kash> p1 := L[1][1];;
kash> p2 := L[2][1];;
kash> p3 := L[3][1];;
```

```
kash> b := p1^2*p2*p3;
<625, [-180, 110, 420, 240, 80]>
kash> IdealIsPrincipal(b);
5
kash> b = 5*O;
true
```

The function `IdealIsPrincipal` returns the generator of an ideal $\mathfrak{a}$ if $\mathfrak{a}$ is a principal ideal and `false` otherwise.

Finally, we compute the ideal norm of each factor.

```
kash> IdealNorm(p1);
5
kash> IdealNorm(p2);
5
kash> IdealNorm(p3);
25
```

2.5 Class Groups

The last invariant of the field $\mathcal{F}$ we will compute here is the ideal class group $\mathrm{Cl}_{\mathcal{F}}$. This task can be solved by invoking the `OrderClassGroup` function. This routine takes only one argument, namely a maximal order, and returns a list. The first entry in this list is the class number, whereas the second entry is another list containing the orders of the cyclic factors of the class group. In our example, $\mathrm{Cl}_{\mathcal{F}}$ is the cyclic group of order 6.

```
kash> OrderClassGroup(O);
[ 6, [ 6 ] ]
```

Using the `OrderClassGroupCyclicFactors` routine we can obtain a list of ideals representing generators of the cyclic factors of $\mathrm{Cl}_{\mathcal{F}}$. Here, because $\mathrm{Cl}_{\mathcal{F}} \simeq C_6$, we get only one representative.

```
kash> OrderClassGroupCyclicFactors(O);
[ [ <2, [1, 1, 0, 0, 1]>, 6 ] ]
```

Notice that the `OrderClassGroupCyclicFactors` function actually returns a list containing sublists. Each sublist consists of a representative of an ideal class and its order.

When KASH computes a class group, it uses the Minkowski bound. 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 two arguments, the second one being the new bound. Refer to the reference manual for a detailed description.

3 Algebraic Number Theory II : Relative extensions

KASH provides considerable support for computations in relative extensions. If you know how to handle absolute extensions in KASH, you will be able to adapt to relative extensions in KASH with no trouble. The commands change little from absolute extensions to relative extensions — you will find that the command `EltTrace` again means the trace of an algebraic number.

3.1 Arithmetic

Suppose that we want to do some arithmetic in the relative extension $\mathcal{F}/\mathcal{E}$ with $\mathcal{F} = \mathcal{E}(\sqrt{165})$ and $\mathcal{E} = \mathbb{Q}(\rho)$, where ρ is a root of $x^4-99x^3+99x^2-99x+99 \in \mathbb{Z}[x]$. First, we compute the maximal order $\mathcal{O}_\mathcal{E}$ of $\mathcal{E}$ in KASH.

```
kash> oE := Order(Poly(Zx, [1, -99, 99, -99, 99]));
Generating polynomial:  x^4 - 99*x^3 + 99*x^2 - 99*x + 99

kash> OE := OrderMaximal(oE);
   F[1]
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^4 - 99*x^3 + 99*x^2 - 99*x + 99
Discriminant: -175877319123
```

We will create the relative equation order $\mathfrak{o}_\mathcal{F} = \mathcal{O}_\mathcal{E}[\sqrt{165}]$ by using the `Order` function. First we have to enter the polynomial $x^2 - 165 \in \mathcal{O}_\mathcal{E}[x]$ in KASH.

```
kash> OEx := PolyAlg(OE);
Univariate Polynomial Algebra in x over
   F[1]
```

```
    |
   F[2]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^4 - 99*x^3 + 99*x^2 - 99*x + 99
Discriminant: -175877319123

kash>  f := Poly(OEx,[1,0,165]);
x^2 + 165
```

Now we are ready to define $\mathfrak{o}_{\mathcal{F}}$.

```
kash> oF := Order(f);
      F[1]
        /
       /
   E1[1]
    |
   E1[2]
  /
 /
Q
F  [ 1]     x^2 + 165
E 1[ 1]     Given by transformation matrix
E 1[ 2]     x^4 - 99*x^3 + 99*x^2 - 99*x + 99
```

Notice, that the `Order` function works here in the same way as in the absolute case (2.1).

To do some arithmetic in $\mathfrak{o}_{\mathcal{F}}$ we need to enter some elements from $\mathfrak{o}_{\mathcal{F}}$ in KASH using the `Elt` function. For example, let us define

$$\begin{aligned} \alpha &= \underbrace{(2\rho + 4\rho^2 + 6\rho^3)}_{a_1} + \underbrace{(1 + 3\rho + 5\rho^2 + 7\rho^3)}_{a_2}\sqrt{165}\,, \\ \beta &= \underbrace{(-2\rho - 4\rho^2 - 6\rho^3)}_{b_1} + \underbrace{(-1 + 3\rho - 5\rho^2 + 7\rho^3)}_{b_2}\sqrt{165} \end{aligned}$$

Since α and β are elements of the relative order $\mathfrak{o}_{\mathcal{F}}$, their coefficients are not integral numbers but algebraic numbers.

```
kash> a1 := EltMove(Elt(oE,[0,2,4,6]),OE);
[0, 2, 12, 18]
kash> a2 := EltMove(Elt(oE,[1,3,5,7]),OE);
[1, 3, 15, 21]
kash> b1 := EltMove(Elt(oE,[0,-2,-4,-6]),OE);
[0, -2, -12, -18]
kash> b2 := EltMove(Elt(oE,[-1,3,-5,7]),OE);
[-1, 3, -15, 21]
```

After entering a_1, a_2, b_1, b_2 we create α, β.

```
kash> alpha := Elt(oF,[a1,a2]);
[[0, 2, 12, 18], [1, 3, 15, 21]]
kash> beta := Elt(oF,[b1,b2]);
[[0, -2, -12, -18], [-1, 3, -15, 21]]
```

Now we are ready to do some computations.

```
kash> alpha + beta;
[0, [0, 6, 0, 42]]
kash> alpha * beta;
[[7800956526, -7721357688, 23166485484, -23166485049],
 [592020, -586084, 1758216, -1758336]]
kash> EltTrace(alpha);
[0, 4, 24, 36]
```

As already mentioned, the command `EltTrace(alpha)` computes the relative trace $\mathrm{Tr}_{\mathcal{F}/\mathcal{E}}$ of α. When you need to get the absolute trace $\mathrm{Tr}_{\mathcal{F}/\mathbb{Q}}$ of α, the `EltTrace` requires a second argument.

```
kash> EltTrace(alpha,Z);
11371536
```

This is tantamount to

```
kash> EltTrace(EltTrace(alpha));
11371536
```

3.2 Relative ideals

KASH provides arithmetic of ideals in relative orders. It is important that the coefficient order $\mathcal{O}_\mathcal{E}$ is a maximal order otherwise it is not sure to be a Dedekind ring and serious theoretic troubles arises.

A short example[3]:

```
kash> OE := OrderMaximal(Order(Poly(Zx,[1,-10,-3,-2])));
Generating polynomial:  x^3 - 10*x^2 - 3*x - 2
Discriminant: -8180

kash> OF := OrderMaximal(Order(OE, 3, 3));
      F[1]
       |
      F[2]
        /
       /
   E1[1]
  /
 /
Q
F  [ 1]     Given by transformation matrix
F  [ 2]     x^3 - 3
E 1[ 1]     x^3 - 10*x^2 - 3*x - 2
Generating polynomial:  x^3 - 3
Coef. Ideals are: <1>   <1>     <1>
```

Define relative ideals in terms of 2 generators:

```
kash> I1 := Ideal(3, Elt(OF, [0, Elt(OE, [0, 1, 0]), -1]));
<3, [0, [0, 1, 0], -1]>
kash> I2 := Ideal(5, Elt(OF, [Elt(OE, [2, -1, 1]),
> Elt(OE, [-1, 2, 0]), 1]));
<5, [[2, -1, 1], [-1, 2, 0], 1]>
```

[3]Be aware of the fact that your results may look different because some algorithms of KASH are pseudopropabilistic and may get different representations of the same ideal.

Some arithmetic:

```
kash> I1+I2;
<{<1><1><1>
[1 0 0]
[0 1 0]
[0 0 1]
}
>

kash> I := I1*I2;
<{<15, [3, 3, 0]><5, [0, 1, 1]><1>
[1 0 9]
[0 1 2]
[0 0 1]
}
>

kash> IdealGenerators(I);
[ 15, [[12, 3, 0], 2, 1] ]
kash> I1^5;
<243, [[72, -90, -18], [-90, 27, -72], [-111, 15, -63]]>
kash> I1^-1;
<{<1><1><1 / 3>
[1 0 0]
[0 1 0]
[0 0 1]
}
>
```

Ideals are represented either in 2-element representation or basis representation analogue to absolute ideals. The 2-element representation (one element in case the ideal is principal) is very similar to the absolute case — just the defining elements are elements of a relative order.

The basis representation is different because there is no canonical analogon to the HNF of integral matrices. The ideal is represented as the following sum:

$$\xi_1\mathfrak{a}_1 + \cdots + \xi_n\mathfrak{a}_n$$

where n is the relative degree of the order, ξ_i are algebraic numbers in this order and $\mathfrak{a}_i$ are ideals of the coefficient order $\mathcal{O}_\mathcal{E}$.

Because the ξ_i are represented as vectors of elements of $\mathcal{O}_\mathcal{E}$ the ideal is viewed as a **module of degree** n **over the coefficient order** $\mathfrak{o}_\mathcal{F}$ which is viewed in more detail in the next subsection 3.3. The form the ideal is represented in is called a pseudomatrix over the order $\mathcal{O}_\mathcal{E}$.

We can compute the minimum and the norm of relative ideals:

```
kash> IdealMin(I1);
<
[3 0 0]
[0 3 0]
[0 0 3]
>

kash> IdealNorm(I1);
<3>
```

The minimum and the norm of a relative ideal are ideals over the coefficient order $\mathcal{O}_\mathcal{E}$. The minimum is defined to be the intersection of the ideal with the coefficient order of the ideal. The norm of the ideal is defined in terms of the pseudobasis (which is a certain pseudomatrix): it is the determinant of the matrix times the product of all coefficient ideals.

3.3 Modules over Dedekind rings

This subsection describes computations with pseudomatrices in relative orders. Let $\mathcal{O}_\mathcal{E}$ be a maximal order. A finitely generated module over this order (as a ring) can be viewed as a subset of $\mathcal{O}_\mathcal{E}^n$ which can be represented as a pseudomatrix which is a matrix M over $\mathcal{O}_\mathcal{E}$ and for each column i of the matrix a $\mathcal{O}_\mathcal{E}$-Ideal $\mathfrak{a}_i$. Say the matrix has m columns $\xi_1, \ldots, \xi_m$ I write the pseudomatrix as

$$\left[\begin{array}{c} \mathfrak{a}_1 \ \cdots \ \mathfrak{a}_m \\ \left(\begin{array}{c} \\ \xi_1 \ \cdots \ \xi_m \\ \\ \end{array} \right) \end{array} \right]$$

which represents the sum:

$$\sum_{i=1}^{m} \mathfrak{a}_i \xi_i \subset \mathcal{O}_\mathcal{E}^n.$$

There are algorithms to compute from a given pseudomatrix another pseudomatrix which generates the same module and whose matrix is a square matrix and is in diagonal form with just ones on the diagonal. This form is called the normal form of the module.

With the normal form of modules membership, equality, and inclusion of modules can be readily decided.

Modules over Dedekind domains are given in KASH like

```
kash> OE := OrderMaximal(Order(Poly(Zx,[1,-10,-3,-2])));
Generating polynomial:  x^3 - 10*x^2 - 3*x - 2
Discriminant: -8180

kash> M:=Module([Ideal(15, Elt(OE,[0, -3, -3])),
> Ideal(5, Elt(OE,[1, 1, 0])),1*OE],
> Mat(OE, [[1,0,9],[-3,1,2],[1,Elt(OE,[0, -1, 1]),1]]));
{<15, [0, -3, -3]><5, [1, 1, 0]><1>
[1 0 9]
[-3 1 2]
[1 [0, -1, 1] 1]
}
```

We compute a normal form with:

```
kash> ModuleNF(M);
{<6562770, [-40410, 2739987, -350613]><5, [1, 1, 0]><1>
[1 350016 9]
[0 1 2]
[0 0 1]
}
```

Another interesting form of a module is a representation where all but possibly one ideal is the 1-ideal. From the theory it is clear that it is not always possible to find a pseudomatrix with a square matrix and just trivial ideals. The ideal class of the remaining ideal is uniquely determined and is an interesting invariant of the module. It is called Steinitz class therefore I call the presentation Steinitz form.

```
kash> ModuleSteinitz(M);
{<1><1><15, [3, 6, 3]>
[[21, 36, 0] 9 -1]
[[-1393, -108, 0] 2 [502, -1524, 144] / 15]
[[21, 1366, -1330] 1 [-471, -853, 133] / 15]
}
```

The following gives the trivial module of a certain degree:

```
kash> ModuleId(OE,3);
{<1><1><1>
[1 0 0]
[0 1 0]
[0 0 1]
}
```

Modules can be multiplied with numbers and algebraic numbers from the coefficient order.

```
kash> N := 6562770*ModuleId(OE,3);
{<1><1><1>
[6562770 0 0]
[0 6562770 0]
[0 0 6562770]
}
```

Modules can be checked on equality and inclusion:

```
kash> N=M;
false
kash> N<M;
true
```

4 Algebraic 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.

4.1 Definition of an algebraic function field

An algebraic function field is a field extension F/k of transcendence degree 1, where k denotes a field of characteristic $p \geq 0$. The function field may be an extension of a finite field $\mathbb{F}_q$, $\mathbb{Q}$, or an order. There is always an element $T \in F$ such that $F/k(T)$ is a finite separable field extension. From this follows the existence of an $y \in F$ and an irreducible bivariate polynomial f over k, which is separable and monic in the second variable, such that $f(T, y) = 0$. So we can consider the quotient ring $k(T)[y]$ / $f(T, y)k(T)[y]$ as a representation of our given function field.

In KASH, creation of an algebraic function field begins with choosing a bivarate polynomial as above. For this there have to be defined the field k, the polynomial rings $k[T]$ and $k[T][y]$:

```
kash> k := FF(5,2);
Finite field of size 5^2
kash> kT := PolyAlg(k,T);
Univariate Polynomial Algebra in T over Finite field of size 5^2

kash> kTy := PolyAlg(kT);
Univariate Polynomial Algebra in y over Univariate Polynomial
Algebra in T over Finite field of size 5^2

kash> T := Poly(kT, [1, 0]);
T
kash> y := Poly(kTy, [1, 0]);
y
```

For convenience, there is one function defining these variables:

```
kash> AlffInit(FF(5,2));
"Defining global variables: k, w, kT, kTf, kTy, T, y, AlffGlobals"
kash> k;
Finite field of size 5^2
kash> y;
y
```

In two further arguments variable names can be specified:

```
kash> AlffInit(Q, "u", "v");
"Defining global variables: k, w, ku, kuf, kuv, u, v, AlffGlobals"
```

This function defines also the variable `w`, which is a generator of the multiplicative group of the constant field, if finite. Independently of this definition, in all outputs `w` is used to denote a generator of the multiplicative group a finite field.

It is now possible to define an algebraic function field. We test first whether the bivariate polynomial is irreducible and separable in the second variable. It does not have to be neccessarily monic:

```
kash> f := y^3+T^4+1;
y^3 + T^4 + 1
kash> AlffPolyIsIrrSep(f);
true
kash> F := Alff(f);
Global function field defined over Finite field of size 5^2 by
y^3 + T^4 + 1
```

As a first application, the genus of the function field is computed:

```
kash> AlffGenus(F);
3
```

4.2 Orders, ideals and elements

For further applications usually orders are needed. Let P_∞ denote the degree valuation of $k(T)$ and write $\mathcal{O}_\infty$ for its valuation ring. We define an order over $k[T]$ resp. $\mathcal{O}_\infty$ to be a ring extension of $k[T]$ resp. $\mathcal{O}_\infty$ whose quotient field equals F. Such orders are free $k[T]$- or O_∞-modules of degree $[F : k(T)]$. According to their coefficient rings $k[T]$ or O_∞ 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_∞ are defined analogously for suitable, field generating polynomials. Maximal orders (maximal for inclusion) coincide with the integral closures of $k[T]$ or O_∞ in F.

We consider the computation of various orders:

```
kash> oe := AlffOrderEqFinite(F);
Finite order over Finite field of size 5^2 defined by
y^3 + T^4 + 1

kash> oie := AlffOrderEqInfty(F);
Infinite order over Finite field of size 5^2 defined by
y^3 + (T^4 + 1)/T^6

kash> o := AlffOrderMaxFinite(F);
Finite order over Finite field of size 5^2 defined by
y^3 + T^4 + 1

kash> oi := AlffOrderMaxInfty(F);
Infinite order over Finite field of size 5^2 defined by
y^3 + (T^4 + 1)/T^6
and transformation matrix
[ 1/T    0    0]
[   0  1/T    0]
[   0    0    1]
den: 1/T
```

The computation of these orders amounts in finding their module bases. For equation orders this causes no problem, a basis of powers of a root of the defining

polynomial is chosen. On the other hand the computation of the corresponding maximal orders is not an easy task. In the example we see that the finite equation order is already maximal, but the infinite equation order is not maximal. The matrix shown gives a transformation from the module basis of `oie` to `oi`. The bases itself are given internally in the orders.

Now we like to define elements of these orders. Since the orders have bases, it is enough to specify coefficients of linear combinations of the basis elements. This is done as follows:

```
kash> a := AlffElt(oe, [0, 1, 0]);
[ 0, 1, 0 ]
kash> b := AlffElt(oe, [0, T, T^2+1]);
[ 0, T, T^2 + 1 ]
kash> c := AlffElt(oie, [1, 1/T, 0]);
[ 1, 1/T, 0 ]
kash> d := AlffElt(o, [0, 1/T, 0]);
[ 0, 1, 0 ] / (T)
```

The last example looks strange since $1/T \notin k[T]$. This is a general concept of the representation of elements: They do not have to be elements of their orders in the mathematical sense, but they are simply expressed by a linear combination with coefficients in $k(T)$ with respect to the order basis. This actually means that elements are elements (in the mathematical sense) of the quotient field of their order, which equals F. Elements with no denominators are true elements of their order.

It is now possible to operate with the above defined elements. This includes for example algebraic operations such as addition, multiplication and their inverse operations, but also norm and trace computations as well as changing the representation with respect to different orders:

```
kash> a^3 + T^4 + 1;
0
kash> a+b;
[ 0, T + 1, T^2 + 1 ]
kash> b^2;
```

```
[ 3*T^7 + 3*T^5 + 3*T^3 + 3*T, 4*T^8 + 3*T^6 +
 3*T^4 + 3*T^2 + 4, T^2 ]
kash> 1/c;
[ T^9/(T^9 + 4*T^4 + 4), 4*T^8/(T^9 + 4*T^4 + 4),
 T^7/(T^9 + 4*T^4 + 4) ]
kash> AlffEltMove(a, oi);
[ 0, 1, 0 ] / (1/T^2)
kash> AlffEltMove(c, oe);
[ T^3, 1, 0 ] / (T^3)
kash> AlffEltNorm(a);
4*T^4 + 4
kash> AlffEltTrace(a);
0
```

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`.

```
kash> AlffInit(FF(5,2));;
kash> f := y^3 + T^4 + 1;
y^3 + T^4 + 1
kash> AlffOrders(f);
"Defining global variables: F, o, oi, one"
kash> o;
Finite order over Finite field of size 5^2 defined by
y^3 + T^4 + 1
```

As a last example we consider computations with ideals of the maximal orders. Similar as the case for elements ideals may be fractional. 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:

```
kash> a := AlffElt(o, [1,1,0]);;
```

```
kash> b := AlffElt(o, [T,T^2,1]);;
kash> I := T*(T+1)*o + a*b*o;
<
[   T^2 + T    3*T + 1    3*T + 4]
[         0          1          0]
[         0          0          1]
/ 1
>
kash> 1/I;
<
[   T^2 + T          0    2*T + 1]
[         0    T^2 + T    2*T + 4]
[         0          0          1]
/ T^2 + T
>
kash> l := AlffIdealFactor(I);
[ [ < T, [ 1, 1, 0 ]>, 1 ], [ < T + 1, [ 3, 1, 0 ]>, 1 ] ]
kash> p := l[1][1];
< T, [ 1, 1, 0 ]>
kash> AlffIdealValuation(p, I);
1
kash> AlffIdealValuation(p, I^-3);
-3
kash> AlffIdealIsPrime(I/p);
true
```

The ideal bases are given by transformation matrices with respect to the order bases. The second ideal is fractional as the denominator indicates. Afterwards the factorisation is computed. It returns a list of lists consisting of prime ideals and exponents. Then we compute some valuations and test for primality. The computation of factorisations and valuations is possible since maximal orders are Dedekind rings (the infinite maximal order is in fact semilocal).

5 Lattices

In this section we deal with lattices in KASH. First we recall the main definitions. Let $a_1, \ldots, a_k \in \mathbb{R}^n$ be k linearly independent vectors. Then the set

$$\Lambda = \mathbb{Z}\, a_1 + \cdots + \mathbb{Z}\, a_k$$

is called a lattice. Any system of vectors $b_1, \ldots, b_k$ is called a basis of Λ if $\Lambda = \mathbb{Z}\, b_1 + \cdots + \mathbb{Z}\, b_k$.

From now on, let $b_1, \ldots, b_k$ be a fixed basis of Λ. The positive definite matrix

$$G(b_1, \ldots, b_k) = \begin{pmatrix} b_1^t \cdot b_1 & \ldots & b_1^t \cdot b_k \\ \vdots & & \vdots \\ b_k^t \cdot b_1 & \ldots & b_k^t \cdot b_k \end{pmatrix}$$

is called the Gram matrix of $b_1, \ldots, b_k$. The square root of the determinant of $G(b_1, \ldots, b_k)$ is the discriminant $d(\Lambda)$ of the lattice Λ. $d(\Lambda)$ does not depend on the choice of the basis $b_1, \ldots, b_k$.

We denote by $[\Lambda]$ the set $\mathbb{R}\, b_1 + \cdots + \mathbb{R}\, b_k$. For given $x = x_1 b_1 + \cdots + x_k b_k \in [\Lambda]$, its length $\ell(x)$ is defined by

$$\ell(x) = \|x\|_2^2 = x^t \cdot x = (x_1, \ldots, x_k) \cdot G(b_1, \ldots, b_k) \cdot \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix}.$$

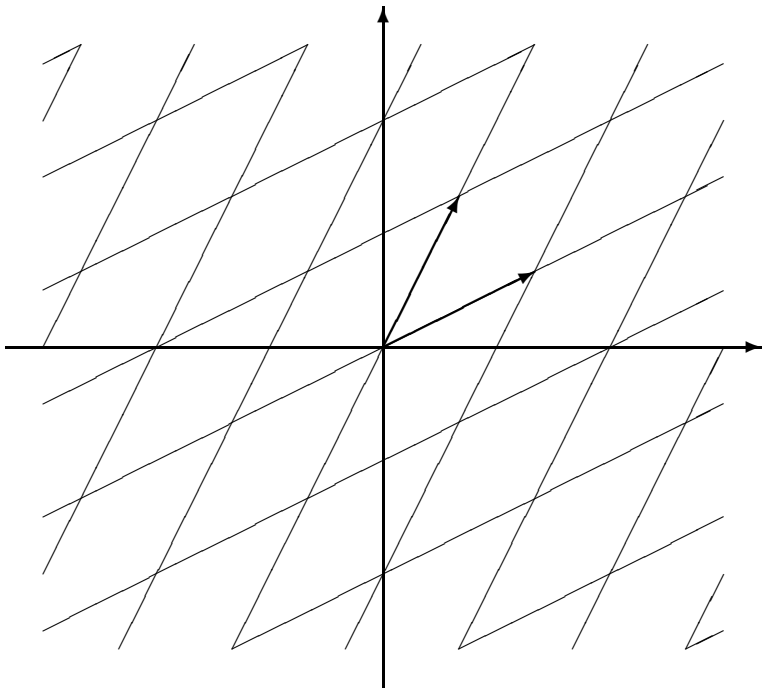
5.1 Defining lattices in KASH

To create a lattice in KASH, the `Lat` routine should be used. Suppose that we want to enter the lattice

$$\Delta = \mathbb{Z}\, g_1 + \mathbb{Z}\, g_2 = \mathbb{Z} \begin{pmatrix} 1 \\ 2 \end{pmatrix} + \mathbb{Z} \begin{pmatrix} 2 \\ 1 \end{pmatrix}.$$

First, we enter the matrix $G = (g_1|g_2) \in \mathbb{Z}^{2\times 2}$ in KASH.

```
kash> G := Mat(Z,[[1,2],[2,1]]);
[1 2]
[2 1]
```



$\Delta = \mathbb{Z}\, g_1 + \mathbb{Z}\, g_2$

The following assignment creates Δ in KASH.

```
kash> Delta := Lat(G);
Basis:
[1 2]
[2 1]
```

Let's look closely at this example. There are many different ways to call the `Lat` routine in KASH (refer to the reference manual for a detailed description). The way we used above is to pass a list containing a basis of the lattice to be entered[4]. The `Lat` returns a new object, named `Delta`, which represents the lattice Δ in KASH. Like orders, every lattice in KASH has a fixed basis (two lattices are considered to be distinct in KASH if their bases differ). Thus, computing the Gram matrix of a lattice in KASH using the `LatGram` function means computing the Gram matrix of the fixed basis assigned to a lattice.

[4]Here, `g1` and `g2` are given over $\mathbb{Z}$. Of course, `g1` and `g2` might also be defined over $\mathbb{Q}$ or $\mathbb{R}$ to create the lattice. However, some lattice algorithms are faster if all its basis vectors are defined as matrices over $\mathbb{Z}$.

```
kash> LatGram(Delta);
[5 4]
[4 5]
```

Invoking `LatDisc`, it is easy to calculate the discriminant of Δ.

```
kash> LatDisc(Delta);
3
```

5.2 Lattice Elements

In KASH, a lattice element of Λ is any element from $[\Lambda]$ (of course, mathematically this convention is slight incorrect because of $\Lambda \neq [\Lambda]$). To enter a lattice element in KASH, there are two different ways using the `LatElt` routine. Assume that $x \in [\Lambda]$ is given by $x = x_1 b_1 + \cdots + x_k b_k$. The first way to call `LatElt` is to pass the vector x itself, the second one is to pass the coefficients $x_1, \ldots, x_k$ in a list. Suppose, that we want to enter the lattice elements

$$a = \begin{pmatrix} 4 \\ 2 \end{pmatrix} = 2\, g_1, \quad b = \frac{3}{2} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{2}\, g_1 + \frac{1}{2}\, g_2.$$

The examples below demonstrate both ways of calling the `LatElt` routine.

```
kash> a := LatElt(Delta,Mat(Z,[[4,2]]));
[0 2]
kash> b := LatElt(Delta,Mat(R,[[3/2,3/2]]));
[0.5 0.5]
```

These assignments are tantamount to

```
kash> a := LatElt(Delta,[0,2]);
[ 0 2 ]
kash> b := LatElt(Delta,[1/2,1/2]);
[0.5 0.5]
```

Let's look more closely at these examples. In the first example we pass the vectors a and b themselves. Notice that KASH automatically computes the representations of a and b with respect to the fixed basis g_1, g_2 of Δ. Such a representation is printed whenever KASH prints a lattice element. Let's do some simple arithmetic.

```
kash> a+b;
[0.5 2.5]
kash> a-b;
[-0.5  1.5]
kash> 5*a;
[ 0 10 ]
```

The length of a lattice element can be computed by invoking the `LatEltLength` routine.

```
kash> LatEltLength(a);
20
kash> LatEltLength(b);
4.5
```

To obtain the vector which corresponds to a given lattice element, the `LatEltVec` routine should be used.

```
kash> LatEltVec(a);
[4]
[2]
kash> LatEltVec(b);
[1.5]
[1.5]
```

Notice that the vectors returned by `LatEltVec` are given as columns. However, the `Lat` and `LatElt` routines both do not distinguish between vectors given as columns or as rows. Thus, the following assignment again creates a.

```
kash> a := LatElt(Delta,Mat(Z,[[4],[2]]));
[0 2]
```

5.3 Shortest vectors

Let L, U be any real numbers with $0 \leq L < U$ and let y denote any lattice element from $[\Lambda]$. We define $S \subseteq [\Lambda]$ by setting

$$S = S(L, U) = \{x \in [\Lambda] \,|\, L \leq \ell(x) \leq U\}. \tag{1}$$

Using the KASH function `LatShortestElt`, we can determine a lattice element $z \in \Lambda$ such that

$$\ell(z-y) = \min\{\ell(x-y) \,|\, x \in \Lambda, x-y \in S\}. \tag{2.1}$$

Before invoking the `LatShortestElt` routine, you have to initialize the bounds L, U and the lattice element y. To set the bounds L, U and the reference vector y in KASH, the functions `LatEnumLowerBound`, `LatEnumUpperBound` and `LatEnumRefVec` should be used. Each of these routines requires two arguments. The first argument is always the lattice, whereas the second one either holds the bound or the reference vector. If L (respectively U) is not initialized, KASH uses the default bound 0 (respectively $+\infty$). When y is not initialized, the `LatShortestElt` routine computes $z \in \Lambda \cap S$ such that

$$0 < \ell(z) = \min\{\ell(x) \,|\, x \in \Lambda \cap S, x \neq 0\}. \tag{2.2}$$

The following examples will demonstrate the use of the `LatShortestElt` function (refer to reference manual for a detailed description of the `LatShortestElt` routine).

Assume that we want to determine a non-zero element $z \in \Delta$ such that z has minimal length. To do this just enter

```
kash> LatShortestElt(Delta);
[ [ 1 -1 ] ]
```

Notice that `LatShortestElt` actually returns a list containing lattice elements. Here, we are interested only in the first entry.

```
kash> LatEltVec(last[1]);
[-1]
[ 1]
kash> LatEltLength(last2[1]);
2
```

Suppose now that we want to compute an element $z \in \Lambda$ which is closest to the vector $y = \frac{3}{2}\binom{1}{1}$.

```
kash> y := LatElt(Delta,Mat(R,[[3/2,3/2]]));
[0.5 0.5]
kash> LatEnumRefVec(Delta,y);;
kash> LatShortestElt(Delta);
[ [ 1 0 ] ]
```

5.4 Enumerating lattices

Let $L, U \in \mathbb{R}$ and $y \in [\Lambda]$ as in 5.3. Earlier, we saw how to compute a shortest lattice element $z \in \Delta$ element satisfying (2.1), respectively (2.2). Now we are going to enumerate all $z \in \Lambda$ with $z - y \in S$ (y initialized), respectively all non–zero $z \in \Lambda \cap S$ (y not initiliazed). After setting L, U and y appropriately, we use the `LatEnum` function in conjunction with a `while` loop (see 6.1). The following example should clarify the method. We compute all $z \in \Delta$ with $0 < \ell(z) \leq 10$.

```
kash> LatEnumReset(Delta);
kash> LatEnumUpperBound(Delta,10);
10
kash> while LatEnum(Delta) do
> Print(LatEnumElt(Delta));
> od;
[ 1 -2 ]
[ 2 -2 ]
[ 0 -1 ]
[ 1 -1 ]
[ 2 -1 ]
[ -1 0 ]
```

Let's look closely at this example. The `LatEnumReset` routine resets the bounds L, U to their defaults and undefines the reference vector. Here, we call the routine `LatEnumReset` to avoid unexpected results due to your prior definitions. The `LatEnum` successively computes all the elements we want. Each time `LatEnum` is invoked, this routine tries to find another element which meets the conditions formed by the bounds L, U and the reference vector y. If `LatEnum` is successful, it returns `true` and the element found by `LatEnum` can be accessed using the `LatEnumElt` function. When `LatEnum` returns `false`, this indicates that there

are no more elements satisfying the conditions. In this case, the enumeration is finished. As you may have noticed, `LatEnum` did not return all $z \in \Lambda$ with $0 < \ell(z) \leq 10$. Because $\ell(z) = \ell(-z)$, it will only return half the number of elements if the reference vector is not initialized.

Finally, we compute all $z \in \Delta$ with $2 < \ell(z - y) < 5$ where $y = \frac{3}{2}\binom{1}{1}$.

```
kash> LatEnumReset(Delta);
kash> LatEnumLowerBound(Delta,2);
2
kash> LatEnumUpperBound(Delta,5);
5
kash> y := LatElt(Delta,Mat(R,[[3/2,3/2]]));
[0.5 0.5]
kash> LatEnumRefVec(Delta,y);
[0.5 0.5]
kash> while LatEnum(Delta) do
> Print(LatEnumElt(Delta));
> od;
[ 2 -1 ]
[ 0 0 ]
[ 1 1 ]
[ -1 2 ]
```

Notice that any changes to L, U or y reset the enumeration process of `LatEnum`.

6 The Programming Language

We have already seen many examples of two key elements of KASH's programming language, the expression statement (a command or expression) and the assignment statement. This section discusses the elements of more elaborate programming. It assumes that the reader is already familiar with another conventional programming language. It is not intended to be an introduction to computer programming !

The approach in this section is informal, and uses extensive examples to illustrate syntax and features of the KASH programming language. Refer to the reference manual for a detailed description of the programming language.

6.1 While

`while` *condition* `do` *statements* `od;`

The `while` loop executes the statement sequence *statements* while the condition, an expression that is either `true` or `false`, evaluates to `true`. In the following example all primes between 2 and 20 are computed.

```
kash> L := [];
[  ]
kash> p := 2;
2
kash> while p < 20 do Add(L,p); p := NextPrime(p); od;
kash> L;
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

You can display intermediate results in the repetition by invoking the `Print` function, which takes a variable number of objects to be printed. The output of `Print` looks exactly like the displayed representation of these objects by the main loop. Notice that no space or newline is printed between the objects. The example below demonstrates the use of `Print`.

```
kash> p := 2;
2
kash> while p < 20 do Print(p,"\n"); p := NextPrime(p); od;
```

```
2
3
5
7
11
13
17
19
```

6.2 Repeat

`Repeat` *statements* `until` *condition*`;`

The `repeat` loop executes the statement sequence *statements* until the condition evaluates to *true*. The difference between the `while` loop and the `repeat until` loop is that the statements in the `repeat until` loop are executed at least once, while the *statements* in the `while` loop are not executed at all if the *condition* is `false` at the first iteration. In the below example we again compute all primes between 2 and 20.

```
kash> L := [];
[  ]
kash> p := 2;
2
kash> repeat Add(L,p); p := NextPrime(p); until p > 20;
kash> L;
[ 2, 3, 5, 7, 11, 13, 17, 19 ]
```

6.3 For

`for` *var* `in` *list* `do` *statements* `od;`

The effect of the `for` loop is to execute the *statements* for every element of the *list*. The `for` loop can be used in conjunction with any kind of list. You can, for instance, loop over a range to compute 15! in the following way.

```
kash> f := 1;
```

```
1
kash> for i in [1..7] do f := f*i; od;
kash> f;
5040
```

6.4 If

`if` *condition1* `then` *statements1*
`{ elif` *condition2* `then` *statements2* `} [ else` *statements3* `] fi;`

The `if` statement allows one to execute statements depending on the values of some conditions. The execution is done as follows.

First the *condition1* 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 *condition2* following the `elif` are evaluated in turn. There may be any number of `elif` parts, possibly none at all. As soon as an expression evaluates to `true`, the corresponding statement sequence *statements2* is executed and execution of the `if` statement is complete.

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.

In the example below we first compute all fields $\mathbb{Q}(\sqrt{-p})$ with a rational prime $p \in [2, 20]$, putting them in a list named `L`. Then we print those fields from `L` having a non-trivial class group.

```
kash> L := [];
[  ]
kash> p := 2;
2
kash> # Compute fields
kash> while p < 20 do
> Add(L,OrderMaximal(Order(Poly(Zx,[1,0,p]))));
> p := NextPrime(p);
```

```
> od;
kash> # Print all fields having a non-trivial class group
kash> for o in L do
> if OrderClassGroup(o)[1] > 1 then Print(o,"\n"); fi;
> od;
Generating polynomial:  x^2 + 5
Discriminant: -20
class number 2
class group structure C2
cyclic factors of the class group:
<2, [1, 1]>

Generating polynomial:  x^2 + 13
Discriminant: -52
class number 2
class group structure C2
cyclic factors of the class group:
<2, [1, 1]>

Generating polynomial:  x^2 + 17
Discriminant: -68
class number 4
class group structure C4
cyclic factors of the class group:
<6, [5, 1]>
```

6.5 Writing Functions

You have already seen how to use the functions of the KASH library, i.e., how to apply them to arguments. This section will show you how to write your own functions.

Writing a function that prints 'hello, world.' on the screen is a simple exercise in KASH.

```
kash> hello := function()
> Print("hello, world.\n");
```

```
> end;
function (  ) ... end
```

This function when called will only execute the `Print` statement in the second line. This will print the string 'hello, world.' on the screen followed by a newline character `\n` that causes the KASH prompt to appear on the next line rather than immediately following the printed characters.

```
kash> hello();
hello, world.
```

The function definition has the following syntax.

`function(`*arguments*`)` *statements* `end`

A function definition starts with the keyword `function` followed by the formal parameter list *arguments* enclosed in parentheses. The formal parameter list may be empty, as in the example. Several parameters are separated by commas. Note that there must be **no** semicolon behind the closing parenthesis. The function definition is terminated by the keyword `end`.

A KASH function is an expression like integers, sums and lists. It therefore may be assigned to a variable. The terminating semicolon in the example does not belong to the function definition but terminates the assignment of the function to the name `hello`. Unlike in the case of integers, sums, and lists, the value of the function `hello` is echoed in the abbreviated fashion `function ( ) ... end`. This shows the most interesting part of a function: its formal parameter list (which is empty in this example). The complete value of `hello` is returned if you use the `Print` function.

```
kash> Print(hello);
function (  )
    Print( "hello, world.\n" );
```

The `hello` is however not a typical example, as no value is returned; instead only a string is printed.

A more useful function is given in the following example. We define a function `sign` which shall determine the sign of a number.

```
kash> sign := function(x)
> if x < 0 then
>   return -1;
> elif x > 0 then
>   return 1;
> else
>   return 0;
> fi;
> end;
function ( x ) ... end
kash> sign(-10); sign(0); sign(10);
-1
0
1
```

A function `gcd` that computes the greatest common divisor of two integers by Euclid's algorithm will need a variable in addition to the formal arguments.

```
kash> gcd := function(a,b)
> local c;
> while b <> 0
> do
>   c := b;
>   b := a mod b;
>   a := c;
> od;
> return c;
> end;
function ( a, b ) ... end
kash> gcd(66,78);
6
```

The additional variable `c` is declared as a local variable in the `local` statement of the function definition. The `local` statement, if present, must be the first statement of a function definition. When several local variables are declared in only one `local` statement, they are separated by commas.

The variable `c` is indeed a local variable that is local to the function `gcd`. If you try to use the value of `c` in the main loop, you will see that `c` has no assigned value unless you have already assigned a value to the variable `c` in the main loop. In this case the local nature of `c` in the function `gcd` prevents the value of the `c` in the main loop from being overwritten.

We say that in a given scope an identifier identifies a unique variable. A scope is a lexical part of a program text. There is the global scope that encloses the entire program text, and there are local scopes that range from the `function` keyword, denoting the beginning of a function definition, to the corresponding `end` keyword. A local scope introduces new variables whose identifiers are given in the formal argument list and the local declaration of the function. The usage of an identifier in a program text refers to the variable in the innermost scope that has this identifier as its name.

A Installation

KASH is currently available in binary form for several architectures. For installation, simply ftp an appropriate file from

ftp.math.tu-berlin.de (/pub/algebra/Kant/Kash/),

which is compatible with your system and uncompress it. Then set the UNIX environment variable `KASH` to the `lib` subdirectory in your KASH directory. For example, type `setenv KASH /usr/local/kash/lib`.

For more information about KASH and KANT V4 via WWW, our address is

http://www.math.tu-berlin.de/algebra/

B Customizing KASH

For variables used in more than one session, it may be useful to set up the file `.kashrc` (respectively `kashrc` when using MS DOS or OS/2) in your home directory. When KASH is executed, these variables are automatically initialized.

C PVM, KASH and KANT V4

In this section we assume that pvm release 3.3 or higher is already installed on your system resp. network. In all examples given below we assume that `donald` and `primus` are HP-workstations and `teltow` and `buckow` are SUN's.

There are two distinct modes for running pvm and KASH together: By simply instructing KASH to use pvm

```
PvmUse(true);
```

several of the underlying KANT V4 functions, such as `OrderMaximal` (the different p-maximal overorders will be computed on different machins), `OrderClassGroup` (creation of the factorbasis and computation of suitable "relations" use pvm), `OrderUnitsIndep` (same as classgroup), `OrderUnitsFund` (computation of p-maximl overorders) and `OrderNormEquation`, are enabled to use pvm. This mode will be called KANT V4-pvm.

The second mode is the KASH-pvm mode, in which pvm-applications can be written using the KASH language. This is especially useful when computing a large series of examples or just experimenting with parallel algorithms.

C.1 Installing KANT V4-pvm and KASH-pvm

First retrieve KASH binaries for each architecture you intend to use pvm on. Unpack each binary and place it in the appropriate directory for the architecture (In our example: place the HP-binaries in `~/pvm3/bin/HPPA` and the SUN-binaries into `~/pvm3/bin/SUN4`). Currently, sun (SUN4), IBM (RS6K), PA-Risc (HPPA), PC-Linux (LINUX) and Silicon Graphics (SGI5) are supported. In each directory, create hardlinks hardlinks from `kash` to `kash_slave.x`, `kant_slave.x` and `pvm_watch.x`.

```
cd ~/pvm3/bin/HPPA
ln kash kash_slave.x
ln kash kant_slave.x
ln kash pvm_watch.x
```

and

```
cd ~/pvm3/bin/SUN4
ln kash kash_slave.x
ln kash kant_slave.x
ln kash pvm_watch.x
```

In each direktory you should now have three files.

C.2 KANT V4-pvm

To start pvm simply type (in KASH)

```
kash> PvmUse(true);
kash> Exec("pvm");
pvmd already running.
pvm> add teltow donald buckow
3 successful
                    HOST     DTID
                  teltow    80000
                  donald    c0000
                  buckow   100000
pvm> conf
4 hosts, 1 data format
                    HOST     DTID     ARCH   SPEED
                  primus    40000     HPPA    1000
                  teltow    80000     SUN4    1000
                  donald    c0000     HPPA    1000
                  buckow   100000     SUN4    1000
pvm> quit

pvmd still running.
kash>
```

Now all KANT V4-functions that support pvm will use it. For example

```
OrderMaximal(Order(Z, 2, 3^4*5^4*7^4*11^4));
```

will use all three slaves (`teltow`, `buckow` and `donald`), the fourth (`primus`) is the master. Unless especially requested (`PvmUseMastersHost(true);` no slave will run on the master. A watch (see C.4) is available which provides information on what is currently happening. To exit the pvm-shell and stop pvm after all computations are finished, enter `halt`. Be careful not to do this while running KASH, as it will cause KASH to cease as well.

In this mode, there is a built–in security system which takes care of networking problems such as workstations being shut-down or processes being killed. In every such case, the current task will be redistributed, so that there is no need to worry about restarting such processes.

C.3 KASH-pvm

It is now easy to write parallel programs in KASH! Before giving the details needed for using pvm, a short example will be discussed:

Be sure that the slave will be able to find all init-files; we recommend using the `KASH` environment variable to achieve this. Next start KASH.

After the banner appears, enter

```
kash> PvmInit();
true
kash> PvmStartSlave(["donald", "primus", "buckow", "teltow"]);
4
kash> o := OrderMaximal(Order(Z, 4, 2));
Generating polynomial:  x^4 - 2
Discriminant: -2048

kash> PvmSendAll("PvmGetEval();\n");
kash> PvmRead();
Print from donald : OK, enjoy it...
Print from primus : OK, enjoy it...
Print from buckow : OK, enjoy it...
Print from teltow : OK, enjoy it...
kash> p := 2;;
kash> for i in [1..12] do
```

```
> PvmSendNext(Factor(p*o)); p := NextPrime(p); od;
kash> l:=[];;
kash> for i in [1..12] do l[i] := PvmGet()[2][2]; od;
kash> PvmGet();
false
kash> l[1][1][1] * l[2][1][1];

[6 0 2 4]
[0 3 1 1]
[0 0 1 0]
[0 0 0 1]

kash> PvmExit();
```

What has just happened? In order to initialize all of the `Pvm...` stuff, `PvmInit` must be entered at the beginning of each session. Since our (sample) pvm consists of four machines, four slaves were started. A maximal of 3 jobs per machine is the default value. This means that only 3 jobs can be waiting or active on any one slave simultaneously, but, of course, you will be able to run as many jobs as you like — just wait for the first job to finish before sending the fourth. After creating a maximal order using KASH, a slave was started on all machines with the function `PvmGetEval();`, which is defined in the library. This function simply waits for data and sends it back. The main point is that evaluation of data is only performed at the slave! The next step in the example sends the order to all slaves. (This could be omitted and sent at each task.)

In the next step, `read();` is entered; this is an abbreviation for receive all data and print it. If nothing seems to happen, wait a bit and try again. On some systems it may take up to 2 minutes to start the slave. If nothing has happened for 5 minutes, you are in trouble (see C.5).

Now we can finally begin to use the slave(s): The next two entries ask them to perform some factorizations. Remember that all evaluations take place on the slave, so even `p*o` is computed on the slave.

To collect the results, we have used the function `PvmGet();`, which returns either `false` if no data has arrived or a list containing valid data if the first entry is 1. The data format received is `[ 1, [ 1, [ ... ] ] ]`, the first 1 meaning that

the slave provided an answer, which is contained in the second argument of that list. The second 1 simply shows that the slave received valid data. The second argument of that list `...` is the evaluated data sent from the master. In the above example, the message `false` after the function call `PvmGet();` means that there is no more data to receive from the slave.

Use `PvmExit();` to kill all slaves. Note however, that this will not stop the pvm daemon. To stop pvm use (after `PvmExit()`)

```
Exec("pvm");
pvmd already running.
pvm> halt
```

For a more sophisticated example, enter:

```
kash> Read("PvmClassgroup.kash");
```

Now you should have a new function `PvmClassgroup(n, b)` which computes the classgroup's of all monic irrdeducible polynomials of degree n and coefficients bounded by b.

All files in the src directory starting with `Pvm` may be used to illustrate some features of KASH-pvm.

C.4 pvm-watch

To monitor some of the activities of the pvm you may use the watch. After starting pvm (`PvmUse(true);` or `PvmInit();`) you may use `PvmUseWatch(true)`. Notification of all data sent by the master using `PvmSendNext`, `PvmSendAll` and `PvmSendLast`, and, to some extent, the response of the slave is now provided.

C.5 Trouble shooting

In addition to the watch (see C.4) (all) data normally printed on the screen of the slave will go into a file named `/tmp/pvml.uid` with uid being your user id. This file always resides on the `/tmp`-directory of the machine running the master. A lot of information can be obtained by simultanously reading this file.

D Printlevel

Here is a (uncomplete) list of printlevels that are useful to find out what KASH is doing during computations. Please have a look at the `PRINTLEVEL` function for details.

Function	**Printlevel**
EltApproximation	ORDER_ELT_APPROX
EltCon	ORDER_ELT_CON
EltIsInIdeal	ORDER_ELT_IS_IN_IDEAL
EltListAbsDisc	REL_EXT_DISC
EltMove	ORDER_MOVE
EltNorm	ORDER_ELT_NORM
EltRoot	ORDER_ELT_ROOT
EltSimplify	ORDER_ELT_SIMPLIFY
Factor	ORDER_PRIME_FACTORIZE
Ideal	ORDER_IDEAL
Ideal2EltNormalAssure	ORDER_IDEAL_2_NORMAL_ASSURE
Ideal2EltNormalAssure	ORDER_IDEAL_2_NORMAL_CHECK
IdealBasis	ORDER_IDEAL_2_Z
IdealCollection	ORDER_MODULE
IdealGenerators, Ideal2EltAssure	ORDER_IDEAL_2_ASSURE
IdealIsIntegral	ORDER_IDEAL_IS_INTEGRAL
IdealPrimeElt	ORDER_IDEAL_PRIME_ELT
ideal operation $*$	ORDER_IDEAL_MULT
ideal operation /	ORDER_IDEAL_INVERT_INTEGRAL
ideal operation /	ORDER_IDEAL_ORDER_ELT_DIV
Lat	LAT_CREATE
Order	ORDER_MULT_TABLE_CREATE
OrderAbs	ORDER_ABS
OrderClassGroup	CLASS_GROUP_CHECK
OrderClassGroup	ORDER_CLASS_GROUP_CALC
OrderDisc	ORDER_DISC_CALC
OrderIsSubfield	ORDER_IS_SUBFIELD
OrderKextDisc	K_EXT_DISC

Function	**Printlevel**
OrderKextGenAbs, OrderKextGenRel	K_EXT_GEN
OrderMaximal	RND2_PP
OrderMaximal	ROUND2
OrderMerge	MERGE
OrderMergeUnit	ORDER_MERGE_UNIT
OrderMinkowski	ORDER_MINKOWSKI_BOUND
OrderNormEquation	ORDER_LAT_NORM_EQUATION
OrderNormEquation	RELNEQ
OrderPrec, Order	ORDER_BASIS_REAL_CREATE
OrderReg	ORDER_REG_CALC
OrderRegLowBound	ORDER_REG_LBOUND_CALC
OrderRelNF	REL_EXT_NF
OrderSubfield	SEARCH_FIELD
OrderTorsionUnit	ORDER_TORSION_SUBGROUP_CALC
OrderUnitsFund	ORDER_MERGE_UNIT, ORDER_UNITS_CAP_SUBORDER, ORDER_UNITS_FUND_CALC, ORDER_UNITS_FUND_CHECK, ORDER_UNITS_INDEP_CALC, ORDER_UNITS_LLL_REDUCE, ORDER_UNITS_LOGS_CALC
OrderUnitsIndep	ORDER_MERGE_UNIT, ORDER_UNITS_INDEP_CALC, ORDER_UNITS_LLL_REDUCE, ORDER_UNITS_LOGS_CALC
OrderUnitsPMaximal	ORDER_UNITS_P_MAXIMAL, ORDER_UNITS_P_MAXIMAL_CHECK