User's Guide

to

PARI / GP

(version 2.3.5)

C. Batut, K. Belabas, D. Bernardi, H. Cohen, M. Olivier

Laboratoire A2X, U.M.R. 9936 du C.N.R.S.
Université Bordeaux I, 351 Cours de la Libération
33405 TALENCE Cedex, FRANCE
e-mail: pari@math.u-bordeaux.fr

Home Page:
http://pari.math.u-bordeaux.fr/

Table of Contents

Chapter 1:
Overview of the PARI system

1.1 Introduction.

PARI/GP is a specialized computer algebra system, primarily aimed at number theorists, but can be used by anybody whose primary need is speed.

Although quite an amount of symbolic manipulation is possible, PARI does badly compared to systems like Axiom, Macsyma, Maple, Mathematica or Reduce on such tasks (e.g. multivariate polynomials, formal integration, etc...). On the other hand, the three main advantages of the system are its speed, the possibility of using directly data types which are familiar to mathematicians, and its extensive algebraic number theory module which has no equivalent in the above-mentioned systems.

PARI is used in three different ways:

1) as a library, which can be called from an upper-level language application, for instance written in ANSI C or C++;

2) as a sophisticated programmable calculator, named `gp`, whose language `GP` contains most of the control instructions of a standard language like C;

3) the compiler GP2C translates GP code to C, and loads it into the `gp` interpreter. A typical script compiled by GP2C runs 3 to 10 times faster. The generated C code can be edited and optimized by hand. It may also be used as a tutorial to library programming.

The present Chapter 1 gives an overview of the PARI/GP system; GP2C is distributed separately and comes with its own manual. Chapter 2 describes the `GP` programming language and the `gp` calculator. Chapter 3 describes all routines available in the calculator. Programming in library mode is explained in Chapters 4 and 5 in a separate booklet (*User's Guide to the PARI library*, `libpari.dvi`).

Important note. A tutorial for `gp` is provided in the standard distribution (*A tutorial for PARI/GP*, `tutorial.dvi`) and you should read this first. You can then start over and read the more boring stuff which lies ahead. You can have a quick idea of what is available by looking at the `gp` reference card (`refcard.dvi` or `refcard.ps`). In case of need, you can refer to the complete function description in Chapter 3.

How to get the latest version? Everything can be found on PARI's home page:

`http://pari.math.u-bordeaux.fr/`

From that point you may access all sources, some binaries, version informations, the complete mailing list archives, frequently asked questions and various tips. All threaded and fully searchable.

How to report bugs? Bugs are submitted online to our Bug Tracking System, available from PARI's home page, or directly from the URL

`http://pari.math.u-bordeaux.fr/Bugs`

Further instructions can be found on that page.

1.2 The PARI types.

The GP language is not typed in the traditional sense (it is dynamically typed); in particular, variables have no type. In library mode, the type of all PARI objects is `GEN`, a generic type. On the other hand, each object has a specific internal type, depending on the mathematical object it represents.

The crucial word is recursiveness: most of the types PARI knows about are recursive. For example, the basic internal type `t_COMPLEX` exists. However, the components (i.e. the real and imaginary part) of such a "complex number" can be of any type. The only sensible ones are integers (we are then in $\mathbf{Z}[i]$), rational numbers ($\mathbf{Q}[i]$), real numbers ($\mathbf{R}[i] = \mathbf{C}$), or even elements of $\mathbf{Z}/n\mathbf{Z}$ (in $(\mathbf{Z}/n\mathbf{Z})[t]/(t^2+1)$), or p-adic numbers when $p \equiv 3 \bmod 4$ ($\mathbf{Q}_p[i]$).

This feature must not be used too rashly in library mode: for example you are in principle allowed to create objects which are "complex numbers of complex numbers". (This is not possible under `gp`.) But do not expect PARI to make sensible use of such objects: you will mainly get nonsense.

On the other hand, one thing which *is* allowed is to have components of different, but compatible, types. For example, taking again complex numbers, the real part could be of type integer, and the imaginary part of type rational. By compatible, we mean types which can be freely mixed in operations like $+$ or $\times$. For example if the real part is of type real, the imaginary part cannot be of type intmod (integers modulo a given number n).

Let us now describe the types. As explained above, they are built recursively from basic types which are as follows. We use the letter T to designate any type; the symbolic names `t_xxx` correspond to the internal representations of the types.

type `t_INT`	$\mathbf{Z}$	Integers (with arbitrary precision)
type `t_REAL`	$\mathbf{R}$	Real numbers (with arbitrary precision)
type `t_INTMOD`	$\mathbf{Z}/n\mathbf{Z}$	Intmods (integers modulo n)
type `t_FRAC`	$\mathbf{Q}$	Rational numbers (in irreducible form)
type `t_COMPLEX`	$T[i]$	Complex numbers
type `t_PADIC`	$\mathbf{Q}_p$	p-adic numbers
type `t_QUAD`	$\mathbf{Q}[w]$	Quadratic Numbers (where $[\mathbf{Z}[w] : \mathbf{Z}] = 2$)
type `t_POLMOD`	$T[X]/P(X)T[X]$	Polmods (polynomials modulo P)
type `t_POL`	$T[X]$	Polynomials
type `t_SER`	$T((X))$	Power series (finite Laurent series)
type `t_RFRAC`	$T(X)$	Rational functions (in irreducible form)
type `t_VEC`	T^n	Row (i.e. horizontal) vectors
type `t_COL`	T^n	Column (i.e. vertical) vectors
type `t_MAT`	$\mathcal{M}_{m,n}(T)$	Matrices
type `t_LIST`	T^n	Lists
type `t_STR`		Character strings

and where the types T in recursive types can be different in each component.

The internal type `t_VECSMALL`, holds vectors of small integers, whose absolute value is bounded by 2^{31} (resp. 2^{63}) on 32-bit, resp. 64-bit, machines. They are used internally to represent permutations, polynomials or matrices over a small finite field, etc.

In addition, there exist types `t_QFR` and `t_QFI` for binary quadratic forms of respectively positive and negative discriminants, which can be used in specific operations, but which may disappear in future versions.

Every PARI object (called `GEN` in the sequel) belongs to one of these basic types. Let us have a closer look.

1.2.1 Integers and reals: they are of arbitrary and varying length (each number carrying in its internal representation its own length or precision) with the following mild restrictions (given for 32-bit machines, the restrictions for 64-bit machines being so weak as to be considered inexistent): integers must be in absolute value less than $2^{268435454}$ (i.e. roughly 80807123 digits). The precision of real numbers is also at most 80807123 significant decimal digits, and the binary exponent must be in absolute value less than 2^{29}.

Note that PARI has been optimized so that it works as fast as possible on numbers with at most a few thousand decimal digits. In particular, the native PARI kernel does not contain asymptotically fast DFT-based techniques Hence, although it is possible to use PARI to do computations with 10^7 decimal digits, better programs can be written for such huge numbers. At the very least the GMP kernel should be used at this point. (For reasons of backward compatibility, we cannot enable GMP by default, but you probably should enable it.)

Integers and real numbers are non-recursive types and are sometimes called `leaves`.

1.2.2 Intmods, rational numbers, p-adic numbers, polmods, and rational functions: these are recursive, but in a restricted way.

For intmods or polmods, there are two components: the modulus, which must be of type integer (resp. polynomial), and the representative number (resp. polynomial).

For rational numbers or rational functions, there are also only two components: the numerator and the denominator, which must both be of type integer (resp. polynomial).

Finally, p-adic numbers have three components: the prime p, the "modulus" p^k, and an approximation to the p-adic number. Here $\mathbf{Z}_p$ is considered as the projective limit $\varprojlim \mathbf{Z}/p^k\mathbf{Z}$ (via its finite quotients), and $\mathbf{Q}_p$ as its field of fractions. Like real numbers, the codewords contain an exponent, giving the p-adic valuation of the number, and also the information on the precision of the number, which is redundant with p^k, but is included for the sake of efficiency.

1.2.3 Complex numbers and quadratic numbers: quadratic numbers are numbers of the form $a + bw$, where w is such that $[\mathbf{Z}[w] : \mathbf{Z}] = 2$, and more precisely $w = \sqrt{d}/2$ when $d \equiv 0 \bmod 4$, and $w = (1 + \sqrt{d})/2$ when $d \equiv 1 \bmod 4$, where d is the discriminant of a quadratic order. Complex numbers correspond to the important special case $w = \sqrt{-1}$.

Complex numbers are partially recursive: the two components a and b can be of type `t_INT`, `t_REAL`, `t_INTMOD`, `t_FRAC`, or `t_PADIC`, and can be mixed, subject to the limitations mentioned above. For example, $a+bi$ with a and b p-adic is in $\mathbf{Q}_p[i]$, but this is equal to $\mathbf{Q}_p$ when $p \equiv 1 \bmod 4$, hence we must exclude these p when one explicitly uses a complex p-adic type. Quadratic numbers are more restricted: their components may be as above, except that `t_REAL` is not allowed.

1.2.4 Polynomials, power series, vectors, matrices and lists: they are completely recursive: their components can be of any type, and types can be mixed (however beware when doing operations). Note in particular that a polynomial in two variables is simply a polynomial with polynomial coefficients.

In the present version 2.3.5 of PARI, there are bugs in the handling of power series of power series, i.e. power series in several variables. However power series of polynomials (which are power series in several variables of a special type) are OK. This bug is difficult to correct because the mathematical problem itself contains some amount of imprecision, and it is not easy to design an intuitive generic interface for such beasts.

1.2.5 Strings: These contain objects just as they would be printed by the `gp` calculator.

1.2.6 Notes:

1.2.6.1 Exact and imprecise objects: we have already said that integers and reals are called the leaves because they are ultimately at the end of every branch of a tree representing a PARI object. Another important notion is that of an **exact object**: by definition, numbers of basic type real, p-adic or power series are imprecise, and we will say that a PARI object having one of these imprecise types anywhere in its tree is not exact. All other PARI objects will be called exact. This is an important notion since no numerical analysis is involved when dealing with exact objects.

1.2.6.2 Scalar types: the first nine basic types, from `t_INT` to `t_POLMOD`, will be called scalar types because they essentially occur as coefficients of other more complicated objects. Note that type `t_POLMOD` is used to define algebraic extensions of a base ring, and as such is a scalar type.

1.2.6.3 What is zero? This is a crucial question in all computer systems. The answer we give in PARI is the following. For exact types, all zeros are equivalent and are exact, and thus are usually represented as an integer zero. The problem becomes non-trivial for imprecise types. For p-adics the answer is as follows: every p-adic number, including 0, has an exponent e and a "mantissa" (a purist would say a significand) u which is a p-adic unit, except when the number is zero (in which case u is zero), the significand having a certain precision of k "significant words" (i.e. being defined modulo p^k). Then this p-adic zero is understood to be equal to $O(p^e)$, i.e. there are infinitely many distinct p-adic zeros. The number k is thus irrelevant.

For power series the situation is similar, with p replaced by X, i.e. a power series zero will be $O(X^e)$, the number k (here the length of the power series) being also irrelevant.

For real numbers, the precision k is also irrelevant, and a real zero will in fact be $O(2^e)$ where e is now usually a negative binary exponent. This of course will be printed as usual for a floating point number ($0.0000\cdots$ in `f` format or $0.Exx$ in `e` format) and not with a O symbol as with p-adics or power series. With respect to the natural ordering on the reals we make the following convention: whatever its exponent a real zero is smaller than any positive number, and any two real zeroes are equal.

1.3 Multiprecision kernels / Portability.

(You can skip this section if you are not interested in hardware technicalities.)

The PARI multiprecision kernel comes in three non exclusive flavours. See Appendix A for how to set up these on your system; various compilers are supported, but the GNU `gcc` compiler is the definite favourite.

A first version is written entirely in ANSI C, with a C++-compatible syntax, and should be portable without trouble to any 32 or 64-bit computer having no drastic memory constraints. We do not know any example of a computer where a port was attempted and failed.

In a second version, time-critical parts of the kernel are written in inlined assembler. At present this includes

• the whole ix86 family (Intel, AMD, Cyrix) starting at the 386, up to the Xbox gaming console, including the Opteron 64 bit processor.

• three versions for the Sparc architecture: version 7, version 8 with SuperSparc processors, and version 8 with MicroSparc I or II processors. UltraSparcs use the MicroSparc II version;

• the DEC Alpha 64-bit processor;

• the Intel Itanium 64-bit processor;

• the PowerPC equipping modern macintoshs (G3, G4, etc.);

• the HPPA processors (both 32 and 64 bit);

A third version uses the GNU MP library to implement most of its multiprecision kernel. It improves significantly on the native one for large operands, say 100 decimal digits of accuracy or more. You should enable it if GMP is present on your system. Parts of the first version are still in use within the GMP kernel, but are scheduled to disappear.

An historical version of the PARI/GP kernel, written in 1985, was specific to 680x0 based computers, and was entirely written in MC68020 assembly language. It ran on SUN-3/xx, Sony News, NeXT cubes and on 680x0 based Macs. It is no longer part of the PARI distribution; to run PARI with a 68k assembler micro-kernel, one should now use the GMP kernel.

1.4 The PARI philosophy.

The basic principle which governs PARI is that operations and functions should, firstly, give as exact a result as possible, and secondly, be permitted if they make any kind of sense.

Specifically, an exact operation between exact objects will yield an exact object. For example, dividing 1 by 3 does not give $0.33333\cdots$, but simply the rational number (1/3). To get the result as a floating point real number, evaluate 1./3 or add 0. to (1/3).

Conversely, the result of operations between imprecise objects will be as precise as possible. Consider for example the addition of two real numbers x and y. The accuracy of the result is *a priori* unpredictable; it depends on the precisions of x and y, on their sizes, and also on the size of $x + y$. From this data, PARI works out the right precision for the result. Even if it is working in calculator mode `gp` where there is a notion of default precision, which is only used to convert exact types to inexact ones.

In particular, this means that if an operation involves objects of different accuracies, some digits will be disregarded by PARI. It is a common source of errors to forget, for instance, that a real number is given as $r + 2^e\varepsilon$ where r is a rational approximation, e a binary exponent and ε is a nondescript real number less than 1 in absolute value*. Hence, any number less than 2^e may be treated as an exact zero:

```
? 0.E-28 + 1.E-100
%1 = 0.E-28
? 0.E100 + 1
%2 = 0.E100
```

As an exercise, if `a = 2^(-100)`, why do `a + 0.` and `a * 1.` differ ?

The second principle is that PARI operations are in general quite permissive. For instance taking the exponential of a vector should not make sense. However, it frequently happens that a computation comes out with a result which is a vector with many components, and one wants to get the exponential of each one. This could easily be done either under `gp` or in library mode, but in fact PARI assumes that this is exactly what you want to do when you take the exponential of a vector, so no work is necessary. Most transcendental functions work in the same way (see Chapter 3 for details).

An ambiguity would arise with square matrices. PARI always considers that you want to do componentwise function evaluation, hence to get for example the exponential of a square matrix you would need to use a function with a different name, `matexp` for instance. In the present version 2.3.5, this is not implemented.

1.5 Operations and functions.

The available operations and functions in PARI are described in detail in Chapter 3. Here is a brief summary:

1.5.1 Standard arithmetic operations.

Of course, the four standard operators `+`, `-`, `*`, `/` exist. It should once more be emphasized that division is, as far as possible, an exact operation: 4 divided by 3 gives `(4/3)`. In addition to this, operations on integers or polynomials, like `\` (Euclidean division), `%` (Euclidean remainder) exist (and for integers, `\/` computes the quotient such that the remainder has smallest possible absolute value). There is also the exponentiation operator `^`, when the exponent is of type integer; otherwise, it is considered as a transcendental function. Finally, the logical operators `!` (`not` prefix operator), `&&` (`and` operator), `||` (`or` operator) exist, giving as results `1` (true) or `0` (false).

1.5.2 Conversions and similar functions.

Many conversion functions are available to convert between different types. For example floor, ceiling, rounding, truncation, etc.... Other simple functions are included like real and imaginary part, conjugation, norm, absolute value, changing precision or creating an intmod or a polmod.

* this is actually not quite true: internally, the format is $2^b(a + \varepsilon)$, where a and b are integers

1.5.3 Transcendental functions.

They usually operate on any complex number, power series, and some also on p-adics. The list is everexpanding and of course contains all the elementary functions, plus many others. Recall that by extension, PARI usually allows a transcendental function to operate componentwise on vectors or matrices.

1.5.4 Arithmetic functions.

Apart from a few like the factorial function or the Fibonacci numbers, these are functions which explicitly use the prime factor decomposition of integers. The standard functions are included. A number of factoring methods are used by a rather sophisticated factoring engine (to name a few, Shanks's SQUFOF, Pollard's rho, Lenstra's ECM, the MPQS quadratic sieve). These routines output strong pseudoprimes, which may be certified by the APRCL test.

There is also a large package to work with algebraic number fields. All the usual operations on elements, ideals, prime ideals, etc... are available.

More sophisticated functions are also implemented, like solving Thue equations, finding integral bases and discriminants of number fields, computing class groups and fundamental units, computing in relative number field extensions, class field theory, and also many functions dealing with elliptic curves over $\mathbf{Q}$ or over local fields.

1.5.5 Other functions.

Quite a number of other functions dealing with polynomials (e.g. finding complex or p-adic roots, factoring, etc), power series (e.g. substitution, reversion), linear algebra (e.g. determinant, characteristic polynomial, linear systems), and different kinds of recursions are also included. In addition, standard numerical analysis routines like univariate integration (using the double exponential method), real root finding (when the root is bracketed), polynomial interpolation, infinite series evaluation, and plotting are included. See the last sections of Chapter 3 for details.

And now, you should really have a look at the tutorial before proceeding.

Chapter 2:
Specific Use of the gp Calculator

2.1 Introduction.

Originally, `gp` was designed as a debugging device for the PARI system library, and not much thought had been given to making it user-friendly. The situation has changed, and `gp` is very useful as a stand-alone tool. The operations and functions available in PARI and `gp` are described in the next chapter. In the present one, we describe the specific use of the `gp` programmable calculator.

EMACS: If you have GNU Emacs, you can work in a special Emacs shell, described in Section 2.14. Specific features of this Emacs shell are indicated by an EMACS sign in the left margin.

2.1.1 Startup

To start the calculator, the general command line syntax is:

`gp [-s` *stacksize*`] [-p` *primelimit*`] [`*files*`]`

where items within brackets are optional. The [*files*] argument is a list of files written in the GP scripting language, which will be loaded on startup. The ones starting with a minus sign are *flags*, setting some internal parameters of `gp`, or *defaults*. See Section 2.11 below for a list and explanation of all defaults, there are many more than just those two. These defaults can be changed by adding parameters to the input line as above, or interactively during a `gp` session or in a preferences file (also known as `gprc`).

If a preferences file (or `gprc`, to be discussed in Section 2.13) can be found, `gp` then read its and execute the commands it contains. This provides an easy way to customize `gp`. The *files* argument is processed right after the `gprc`.

A copyright message then appears which includes the version number, and a lot of useful technical information. After the copyright, the computer writes the top-level help information, some initial defaults, and then waits after printing its prompt, which is '`? `' by default . Whether extended on-line help and line editing are available or not is indicated in this `gp` banner, between the version number and the copyright message. Consider investigating the matter with the person who installed `gp` if they are not. Do this as well if there is no mention of the GMP kernel.

2.1.2 Getting help

To get help, type a `?` and hit return. A menu appears, describing the eleven main categories of available functions and how to get more detailed help. If you now type `?`n with $1 \leq n \leq 11$, you get the list of commands corresponding to category n and simultaneously to Section 3.n of this manual. If you type `?`*functionname* where *functionname* is the name of a PARI function, you will get a short explanation of this function.

If extended help (see Section 2.12.1) is available on your system, you can double or triple the `?` sign to get much more: respectively the complete description of the function (e.g. `??sqrt`), or a list of `gp` functions relevant to your query (e.g. `???"elliptic curve"` or `???"quadratic field"`).

If gp was properly installed (see Appendix A), a line editor is available to correct the command line, get automatic completions, and so on. See Section 2.15.1 or ??readline for a short summary of the line editor's commands.

If you type ?\ you will get a short description of the metacommands (keyboard shortcuts).

Finally, typing ?. will return the list of available (pre-defined) member functions. These are functions attached to specific kind of objects, used to retrieve easily some information from complicated structures (you can define your own but they won't be shown here). We will soon describe these commands in more detail.

As a general rule, under gp, commands starting with \ or with some other symbols like ? or #, are not computing commands, but are metacommands which allow you to exchange information with gp. The available metacommands can be divided into default setting commands (explained below) and simple commands (or keyboard shortcuts, to be dealt with in Section 2.12).

2.1.3 Input

Just type in an instruction, e.g. 1 + 1, or Pi. No action is undertaken until you hit the <Return> key. Then computation starts, and a result is eventually printed. To suppress printing of the result, end the expression with a ; sign. Note that many systems use ; to indicate end of input. Not so in gp: this will hide the result from you! (Which is certainly useful if it occupies several screens.)

2.1.4 Interrupt, Quit

Typing quit at the prompt ends the session and exits gp. At any point you can type Ctrl-C (that is press simultaneously the Control and C keys): the current computation is interrupted and control given back to you at the gp prompt, together with a message like

```
*** gcd: user interrupt after 840 ms.
```

telling you how much time ellapsed since the last command was typed in and in which GP function the computation was aborted. It does not mean that that much time was spent in the function, only that the evaluator was busy processing that specific function when you stopped it.

2.2 The general gp input line.

The gp calculator uses a purely interpreted language GP. The structure of this language is reminiscent of LISP with a functional notation, f(x,y) rather than (f x y): all programming constructs, such as if, while, etc... are functions*, and the main loop does not really execute, but rather evaluates (sequences of) expressions. Of course, it is by no means a true LISP.

* Not exactly, since not all their arguments need be evaluated. For instance it would be stupid to evaluate both branches of an if statement: since only one will apply, only this one is evaluated.

2.2.1 Introduction. User interaction with a `gp` session proceeds as follows. First, one types a sequence of characters at the `gp` prompt; see Section 2.15.1 for a description of the line editor. When you hit the `<Return>` key, `gp` gets your input, evaluates it, then prints the result and assigns it to an "history" array if it is not void (see next section).

More precisely, you input either a metacommand or a sequence of expressions. Metacommands, described in Section 2.12, are not part of the GP language and are simple shortcuts designed to alter gp's internal state (such as the working precision or general verbosity level), or speed up input/output.

An expression is formed by combining constants, variables, operator symbols, functions (including user-defined functions) and control statements. It always has a value, which can be any PARI object. There is a distinction between lowercase and uppercase. Also, outside of character strings, blanks are completely ignored in the input to `gp`. An expression is evaluated using the conventions about operator priorities and left to right associativity.

Several expressions are combined on a single line by separating them with semicolons (`';'`). Such an expression sequence will be called simply a *seq*. A *seq* also has a value, which is the value of the last expression in the sequence. Under `gp`, the value of the *seq*, and only this last value, becomes an history entry. The values of the other expressions in the *seq* are discarded after the execution of the *seq* is complete, except of course if they were assigned into variables. In addition, the value of the *seq* is printed if the line does not end with a semicolon `;`.

2.2.2 The gp history.

This is not to be confused with the history of your *commands*, maintained by readline. It only contains their non-void results, in sequence. Several inputs only act through side effects and produce a void result, for instance a `print` statement, a `for` loop, or a function definition.

The successive elements of the history array are called `%1`, `%2`, ... As a shortcut, the latest computed expression can also be called `%`, the previous one `%'`, the one before that `%''` and so on. The total number of history entries is `%#`.

When you suppress the printing of the result with a semicolon, its history number will not appear either, so it is often a better idea to assign it to a variable for later use than to mentally recompute what its number is. Of course, on the next line, just use `%` as usual.

This history "array" is in fact better thought of as a queue: its size is limited to 5000 entries by default, after which `gp` starts forgetting the initial entries. So `%1` becomes unavailable as `gp` prints `%5001`. You can modify the history size using `histsize`.

2.2.3 Special editing characters. A GP program can of course have more than one line. Since `gp` executes your commands as soon as you have finished typing them, there must be a way to tell it to wait for the next line or lines of input before doing anything. There are three ways of doing this.

The first one is simply to use the backslash character `\` at the end of the line that you are typing, just before hitting `<Return>`. This tells `gp` that what you will write on the next line is the physical continuation of what you have just written. In other words, it makes `gp` forget your newline character. You can type a `\` anywhere. It is interpreted as above only if (apart from ignored whitespace characters) it is immediately followed by a newline. For example, you can type

```
? 3 + \
4
```

instead of typing `3 + 4`.

The second one is a slight variation on the first, and is mostly useful when defining a user function (see Section 2.6): since an equal sign can never end a valid expression, `gp` disregards a newline immediately following an `=`.

```
? a =
123
%1 = 123
```

The third one cannot be used everywhere, but is in general much more useful. It is the use of braces `{` and `}`. An opening brace (`{`) *at the beginning of a line* (modulo spaces as usual) signals that you are typing a multi-line command, and newlines are ignored until you type a closing brace `}`. There is an important, but easily obeyed, restriction: inside an open brace-close brace pair, all your input lines are concatenated, suppressing any newlines. Thus, all newlines should occur after a semicolon (`;`), a comma (`,`) or an operator (for clarity's sake, we don't recommend splitting an identifier over two lines in this way). For instance, the following program

```
{
  a = b
  b = c
}
```

would silently produce garbage, since this is interpreted as `a=bb=c` which assigns the value of `c` to both `bb` and `a`.

2.3 The PARI types.

We see here how to input values of the different data types known to PARI. Recall that blanks are ignored in any expression which is not a string (see below).

A note on efficiency. The following types are provided for convenience, not for speed: `t_INTMOD`, `t_FRAC`, `t_PADIC`, `t_QUAD`, `t_POLMOD`, `t_RFRAC`. Indeed, they always perform a reduction of some kind after each basic operation, even though it is usually more efficient to perform a single reduction at the end of some complex computation. For instance, in a convolution product $\sum_{i+j=n} x_i y_j$ in $\mathbf{Z}/N\mathbf{Z}$ (common when multiplying polynomials!), it is wasteful to perform n reductions modulo N. In short, basic individual operations on these types are fast, but recursive objects with such components could be handled more efficiently: programming with libpari will save large constant factors here, compared to GP.

2.3.1 Integers (type `t_INT`): type the integer (with an initial `+` or `-`, if desired) with no decimal point.

2.3.2 Real numbers (type `t_REAL`): type the number with a decimal point. The internal precision of the real number is the supremum of the input precision and the default precision. For example, if the default precision is 28 digits, typing `2.` gives a number with internal precision 28, but typing a 45 significant digit real number gives a number with internal precision at least 45, although less may be printed.

You can also use scientific notation with the letter `E` or `e`, in which case the (non leading) decimal point may be omitted (like `6.02 E 23` or `1e-5`, but *not* `e10`). By definition, `0.E` N (or `0 E` N) returns a real 0 of (decimal) exponent N, whereas `0.` returns a real 0 "of default precision" (of exponent $-$`realprecision`), see Section 1.2.6.3.

2.3.3 Intmods (type `t_INTMOD`): to enter $n \bmod m$, type `Mod(n,m)`, *not* `n%m`. Internally, all operations are done on integer representatives belonging to $[0, m-1]$.

Note that this type is available for convenience, not for speed: each elementary operation involves a reduction modulo m.

2.3.4 Rational numbers (types `t_FRAC`): all fractions are automatically reduced to lowest terms, so it is impossible to work with reducible fractions. To enter n/m just type it as written. As explained in Section 3.1.4, division is *not* performed, only reduction to lowest terms.

Note that this type is available for convenience, not for speed: each elementary operation involves computing a gcd.

2.3.5 Complex numbers (type `t_COMPLEX`): to enter $x+iy$, type `x + I*y` (*not* `x+i*y`). The letter `I` stands for $\sqrt{-1}$. Recall from Chapter 1 that x and y can be of type `t_INT`, `t_REAL`, `t_INTMOD`, `t_FRAC`, or `t_PADIC`.

2.3.6 p-adic numbers (type `t_PADIC`): to enter a p-adic number, simply write a rational or integer expression and add to it `O(p^k)`, where p and k are integers. This last expression indicates three things to `gp`: first that it is dealing with a `t_PADIC` type (the fact that p is an integer, and not a polynomial, which would be used to enter a series, see Section 2.3.10), secondly the prime p, and finally the number of significant p-adic digits k.

Note that it is not checked whether p is indeed prime but results are undefined if this is not the case: you can work on 10-adics if you want, but disasters will happen as soon as you do something non-trivial like taking a square root. Note that `O(25)` is not the same as `O(5^2)`; you want the latter!

For example, you can type in the 7-adic number

```
2*7^(-1) + 3 + 4*7 + 2*7^2 + O(7^3)
```

exactly as shown, or equivalently as `905/7 + O(7^3)`.

Note that this type is available for convenience, not for speed: internally, `t_PADIC`s are stored as p-adic units modulo some p^k. Each elementary operation involves updating p^k (multiplying or dividing by powers of p) and a reduction mod p^k. In particular additions are slow.

```
? n = 1+O(2^20);   for (i=1,10^5, n++)
time = 86 ms.
? n = Mod(1,2^20); for (i=1,10^5, n++)
time = 48 ms.
? n = 1;           for (i=1,10^5, n++)
time = 38 ms.
```

2.3.7 Quadratic numbers (type `t_QUAD`): first, you must define the default quadratic order or field in which you want to work. This is done using the `quadgen` function, in the following way. Write something like

```
w = quadgen(d)
```

where `d` is the *discriminant* of the quadratic order in which you want to work (hence d is congruent to 0 or 1 modulo 4). The name `w` is of course just a suggestion, but corresponds to traditional usage. You can use any variable name that you like. However, quadratic numbers are always printed with a `w`, regardless of the discriminant. So beware, two numbers can be printed in the same way and not be equal. However `gp` will refuse to add or multiply them for example.

Now $(1, w)$ is the "canonical" integral basis of the quadratic order (i.e. $w = \sqrt{d}/2$ if $d \equiv 0 \bmod 4$, and $w = (1+\sqrt{d})/2$ if $d \equiv 1 \bmod 4$, where d is the discriminant), and to enter $x + yw$ you just type `x + y*w`.

2.3.8 Polmods (type `t_POLMOD`): exactly as for intmods, to enter $x \bmod y$ (where x and y are polynomials), type `Mod(x,y)`, not `x%y`. Note that when y is an irreducible polynomial in one variable, polmods whose modulus is y are simply algebraic numbers in the finite extension defined by the polynomial y. This allows us to work easily in number fields, finite extensions of the p-adic field $\mathbf{Q}_p$, or finite fields.

Note that this type is available for convenience, not for speed: each elementary operation involves a reduction modulo y.

Important remark. Mathematically, the variables occurring in a polmod are not free variables. But internally, a congruence class in $R[t]/(y)$ is represented by its representative of lowest degree, which is a `t_POL` in $R[t]$, and computations occur with polynomials in the variable t. PARI will not recognize that `Mod(y, y^2 + 1)` is "the same" as `Mod(x, x^2 + 1)`, since `x` and `y` are different variables.

To avoid inconsistencies, polmods must use the same variable in internal operations (i.e. between polmods) and variables of lower priority for external operations, typically between a polynomial and a polmod. See Section 2.5.4 for a definition of "priority" and a discussion of (PARI's idea of) multivariate polynomial arithmetic. For instance:

```
? Mod(x, x^2+ 1) + Mod(x, x^2 + 1)
%1 = Mod(2*x, x^2 + 1)     \\ 2i (or -2i), with i^2 = -1
? x + Mod(y, y^2 + 1)
%2 = x + Mod(y, y^2 + 1)   \\ in Q(i)[x]
? y + Mod(x, x^2 + 1)
%3 = Mod(x + y, x^2 + 1)   \\ in Q(y)[i]
```

The first two are straightforward, but the last one may not be what you want: `y` is treated here as a numerical parameter, not as a polynomial variable.

If the main variables are the same, it is allowed to mix `t_POL` and `t_POLMOD`s. The result is the expected `t_POLMOD`. For instance

```
? x + Mod(x, x^2 + 1)
%1 = Mod(2*x, x^2 + 1)
```

2.3.9 Polynomials (type `t_POL`): type the polynomial in a natural way, not forgetting to put a "$*$" between a coefficient and a formal variable (this $*$ does not appear in beautified output). Any variable name can be used except for the reserved names `I` (used exclusively for the square root of -1), `Pi` ($3.14\ldots$), `Euler` (Euler's constant), and all the function names: predefined functions, as described in Chapter 3 (use `\c` to get the complete list of them) and user-defined functions, which you ought to know about (use `\u` if you are subject to memory lapses). The total number of different variable names is limited to 16384 and 65536 on 32-bit and 64-bit machines respectively, which should be enough. If you ever need hundreds of variables, you should probably be using vectors instead. See Section 2.5.4 for a discussion of multivariate polynomial rings.

2.3.10 Power series (type `t_SER`): type a rational function or polynomial expression and add to it `O(`*expr*`^`*k*`)`, where *expr* is an expression which has non-zero valuation (it can be a polynomial, power series, or a rational function; the most common case being simply a variable name). This indicates to `gp` that it is dealing with a power series, and the desired precision is k times the valuation of *expr* with respect to the main variable of *expr*. (To check the ordering of the variables, or to modify it, use the function `reorder`; see Section 3.11.2.23.)

Caveat. Power series with inexact coefficients sometimes have a non-intuitive behaviour: if k significant terms are requested, an inexact zero is counted as significant, even if it is the coefficient of lowest degree. This means that useful higher order terms may be disregarded. If the series precision is insufficient, errors may occur (mostly division by 0), which could have been avoided by a better global understanding of the computation:

```
? A = 1/(y + 0.); B = 1. + O(y);
? B * denominator(A)
%2 = 0.E-28 + O(y)
? A/B
***   division by zero
? A*B
***   Warning: normalizing a series with 0 leading term.
***   division by zero
? A*(1/B)
***   Warning: normalizing a series with 0 leading term.
%3 = 1.0000000000000000000000000000*y^-1 + O(1)
```

If a series with a zero leading coefficient must be inverted, then as a desperation measure that coefficient is discarded, and a warning is issued:

```
? C = 0. + y + O(y^2);
? 1/C
***   Warning: normalizing a series with 0 leading term.
%2 = y^-1 + O(1)
```

The last result could be construed as a bug since it is a priori impossible to deduce such a result from the input (0. may represent any sufficiently small real number). But it was thought more useful to try and go on with an approximate computation than to raise an early exception.

In the first example above, to compute `A*(1/B)`, the denominator of `A` was converted to a power series, then inverted.

2.3.11 Rational functions (types `t_RFRAC`): as for fractions, all rational functions are automatically reduced to lowest terms. All that was said about fractions in Section 2.3.4 remains valid here.

2.3.12 Binary quadratic forms of positive or negative discriminant (type `t_QFR` and `t_QFI`): these are input using the function `Qfb` (see Chapter 3). For example `Qfb(1,2,3)` creates the binary form $x^2 + 2xy + 3y^2$. It is imaginary (of internal type `t_QFI`) since $2^2 - 4 * 3 = -8$ is negative.

Although imaginary forms could be positive or negative definite, only positive definite forms are implemented.

In the case of forms with positive discriminant (type `t_QFR`), you may add an optional fourth component (related to the regulator, more precisely to Shanks and Lenstra's "distance"), which must be a real number. See also the function `qfbprimeform` which directly creates a prime form of given discriminant (see Chapter 3).

2.3.13 Row and column vectors (types `t_VEC` and `t_COL`): to enter a row vector, type the components separated by commas "`,`", and enclosed between brackets "`[`" and "`]`", e.g. `[1,2,3]`. To enter a column vector, type the vector horizontally, and add a tilde "~" to transpose. `[ ]` yields the empty (row) vector. The function `Vec` can be used to transform any object into a vector (see Chapter 3).

2.3.14 Matrices (type `t_MAT`): to enter a matrix, type the components line by line, the components being separated by commas "`,`", the lines by semicolons "`;`", and everything enclosed in brackets "`[`" and "`]`", e.g. `[x,y; z,t; u,v]`. `[ ; ]` yields the empty (0x0) matrix. The function `Mat` can be used to transform any object into a matrix (see Chapter 3).

Note that although the internal representation is essentially the same (only the type number is different), a row vector of column vectors is *not* a matrix; for example, multiplication will not work in the same way.

Note also that it is possible to create matrices (by conversion of empty column vectors and concatenation, or using the `matrix` function) with a given positive number of columns, each of which has zero rows. It is not possible to create or represent matrices with zero columns and a nonzero number of rows.

2.3.15 Lists (type `t_LIST`): lists cannot be input directly; you have to use the function `listcreate` first, then `listput` each time you want to append a new element (but you can access the elements directly as with the vector types described above). The function `List` can be used to transform (row or column) vectors into lists (see Chapter 3).

2.3.16 Strings (type `t_STR`): to enter a string, just enclose it between double quotes `"`, like this: `"this is a string"`. The function `Str` can be used to transform any object into a string (see Chapter 3).

2.3.17 Small vectors (type `t_VECSMALL`): this is an internal type, used to code in an efficient way vectors containing only small integers, such as permutations. Most `gp` functions will refuse to operate on these objects.

2.3.18 Note on output formats. A zero real number is printed in `e` format as $0.Exx$ where xx is the (usually negative) *decimal* exponent of the number (cf. Section 1.2.6.3). This allows the user to check the accuracy of that particular zero.

When the integer part of a real number x is not known exactly because the exponent of x is greater than the internal precision, the real number is printed in `e` format.

Note also that in beautified format, a number of type integer or real is written without enclosing parentheses, while most other types have them. Hence, if you see the expression `(3.14)`, it is not of type real, but probably of type complex with zero imaginary part, or polynomial of degree 0 (to be sure, use `\x` or the function `type`).

2.4 GP operators.

Loosely speaking, an operator is a function (usually associated to basic arithmetic operations) whose name contains only non-alphanumeric characters. In practice, most of these are simple functions, which take arguments, and return a value; assignment operators also have side effects. Each of these has some fixed and unchangeable priority, which means that, in a given expression, the operations with the highest priority is performed first. Operations at the same priority level are performed in the order they were written, i.e. from left to right. Anything enclosed between parenthesis is considered a complete subexpression, and is resolved independently of the surrounding context. For instance, assuming that op_1, op_2, op_3 are standard binary operators with increasing priorities (think of `+`, `*`, `^` for instance),

$$x\ op_1\ y\ op_2\ z\ op_2\ x\ op_3\ y$$

is equivalent to

$$x\ op_1\ ((y\ op_2\ z)\ op_2\ (x\ op_3\ y)).$$

GP contains quite a lot of different operators, some of them unary (having only one argument), some binary, plus special selection operators. Unary operators are defined for either prefix (preceding their single argument: $op\ x$) or postfix (following the argument: $x\ op$) position, never both (some are syntactically correct in both positions, but with different meanings). Binary operators all use the syntax $x\ op\ y$. Most of them are well known, some are borrowed from C syntax, and a few are specific to GP. Beware that some GP operators may differ slightly from their C counterparts. For instance, GP's postfix `++` returns the *new* value, like the prefix `++` of C, and the binary shifts `<<`, `>>` have a priority which is different from (higher than) that of their C counterparts. When in doubt, just surround everything by parentheses. (Besides, your code will be more legible.)

Here is the complete list in order of decreasing priority, binary unless mentioned otherwise:

• Priority 10

`++` and `--` (unary, postfix): x`++` assigns the value $x+1$ to x, then returns the new value of x. This corresponds to the C statement `++`x (there is no prefix `++` operator in GP). x`--` does the same with $x-1$.

• Priority 9

op`=`, where *op* is any simple binary operator (i.e. a binary operator with no side effects, i.e. one of those defined below) which is not a boolean operator (comparison or logical). `x` *op*`=` y assigns (`x` *op* y) to `x`, and returns the new value of `x`. This is *not* a reference to the variable `x`, i.e. an `lvalue`; thus

```
(x += 2) = 3
```

is invalid.

• Priority 8

`=` is the assignment operator. The result of `x = y` is the value of the expression y, which is also assigned to the variable `x`. This is *not* the equality test operator; a statement like `x = 1` is always true (i.e. non-zero), and sets `x` to 1. The right hand side of the assignment operator is evaluated before the left hand side. If the left hand side cannot be modified, raise an error.

• Priority 7

`[ ]` is the selection operator. x`[`i`]` returns the i-th component of vector x; x`[i,j]`, x`[,j]` and x`[`i`,]` respectively return the entry of coordinates (i, j), the j-th column, and the i-th row of matrix x. If the assignment operator (`=`) immediately follows a sequence of selections, it assigns its right hand side to the selected component. E.g `x[1][1] = 0` is valid; but beware that `(x[1])[1] = 0` is not (because the parentheses force the complete evaluation of `x[1]`, and the result is not modifiable).

• Priority 6

`'` (unary, prefix): quote its argument (a variable name) without evaluating it.

```
? a = x + 1; x = 1;
? subst(a,x,1)
  ***   variable name expected: subst(a,x,1)
                                          ^---
? subst(a,'x,1)
%1 = 2
```

`^`: powering.

`'` (unary, postfix): derivative with respect to the main variable. If f is a (GP or user) function, $f'(x)$ is allowed. If x is a scalar, the operator performs numerical derivation, defined as $(f(x+\varepsilon) - f(x-\varepsilon))/2\varepsilon$ for a suitably small epsilon depending on current precision. It behaves as $(f(x))'$ otherwise.

`~` (unary, postfix): vector/matrix transpose.

`!` (unary, postfix): factorial. $x! = x(x-1)\cdots 1$.

`.`*member* (unary, postfix): x`.`*member* extracts *member* from structure x (see Section 2.7).

• Priority 5

`!` (unary, prefix): logical *not*. `!`x return 1 if x is equal to 0 (specifically, if `gcmp0(`x`)==1`), and 0 otherwise.

`#` (unary, prefix): cardinality; `#`x returns `length(`x`)`.

• Priority 4

`+`, `-` (unary, prefix): `-` toggles the sign of its argument, `+` has no effect whatsoever.

• Priority 3

`*`: multiplication.

`/`: exact division (`3/2`=3/2, not 1.5).

`\`, `%`: Euclidean quotient and remainder, i.e. if $x = qy + r$, with $0 \le r < y$ (if x and y are polynomials, assume instead that $\deg r < \deg y$ and that the leading terms of r and x have the same sign), then `x\y` $= q$, `x%y` $= r$.

\/: rounded Euclidean quotient for integers (rounded towards $+\infty$ when the exact quotient would be a half-integer).

<<, >>: left and right binary shift: `x<<n` $= x * 2^n$ if $n > 0$, and $x\backslash/2^{-n}$ otherwise. Right shift is defined by `x>>n` = `x<<(-n)`.

• Priority 2

+, -: addition/subtraction.

• Priority 1

<, >, <=, >=: the usual comparison operators, returning 1 for `true` and 0 for `false`. For instance, `x<=1` returns 1 if $x \leq 1$ and 0 otherwise.

<>, !=: test for (exact) inequality.

==: test for (exact) equality.

• Priority 0

&, &&: logical *and*.

|, ||: logical (inclusive) *or*. Any sequence of logical *or* and *and* operations is evaluated from left to right, and aborted as soon as the final truth value is known. Thus, for instance,

```
x && test(1/x)
type(p) == "t_INT" && isprime(p)
```

will never produce an error since the second argument need not (and will not) be processed when the first is already zero (false).

Remark: For optimal efficiency, you should use the ++, -- and *op*= operators whenever possible:

```
? a = 200000;
? i = 0; while(i<a, i=i+1)
time = 4,919 ms.
? i = 0; while(i<a, i+=1)
time = 4,478 ms.
? i = 0; while(i<a, i++)
time = 3,639 ms.
```

For the same reason, the shift operators should be preferred to multiplication:

```
? a = 1<<20000;
? i = 1; while(i<a, i=i*2);
time = 5,255 ms.
? i = 1; while(i<a, i<<=1);
time = 988 ms.
```

2.5 Variables and symbolic expressions.

2.5.1 Variable names. In GP you can use up to 16383 variable names (up to 65535 on 64-bit machines). A valid identifier name starts with a letter and contain only valid keyword characters: _ or alphanumeric characters ([_A-Za-z0-9]). You may not use built-in function names; see the list with `\c`, including the constants `Pi`, `Euler` and `I` $= \sqrt{-1}$.

Note that GP names are case sensitive. This means for instance that the symbol `i` is perfectly safe to use, and will not be mistaken for $\sqrt{-1}$, and that `o` is not synonymous to `0`.

We will see in Section 2.6 that it is possible to restrict the use of a given variable by declaring it to be `global` or `local`. This can be useful to enforce clean programming style, but is in no way mandatory.

2.5.2 Vectors and matrices. If the variable x contains a vector or list, $x[m]$ refers to its m-th entry. You can assign a result to $x[m]$ (i.e. write something like $x[k] = expr$). If x is a matrix, $x[m, n]$ referes to its (m, n) entry; you can assign a result to $x[m, n]$, but *not* to $x[m]$. If you want to assign an expression to the m-th column of a matrix x, use $x[, m] = expr$ instead. Similarly, use $x[m,] = expr$ to assign an expression to the m-th row of x. This process is recursive, so if x is a matrix of matrices of ..., an expression such as `x[1,1][,3][4] = 1` is perfectly valid (and actually identical to `x[1,1][4,3] = 1`), assuming that all matrices along the way have compatible dimensions.

2.5.3 Variables and polynomials The main thing to understand is that PARI/GP is *not* a symbolic manipulation package. One of the main consequences of this fact is that all expressions are evaluated as soon as they are written, they never stay in an abstract form*. As an important example, consider what happens when you use a variable name *before* assigning a value into it, `x` say. This is perfectly acceptable, it is considered as a monomial of degree 1 in the variable `x`.

```
? p = x^2 + 1
%1 = x^2 + 1
? x = 2;
? x^2 + 1
%3 = 5
? p
%4 = x^2 + 1
? eval(p)
%5 = 5
```

As is shown above, assigning a value to a variable, does not affect polynomials that used it; to take into account the new variable's value, one must use the function `eval` (see Section 3.7.3). It is in general preferable to use `subst`, rather than assigning values to polynomial variables.

* An obvious but important exception are character strings which are evaluated essentially to themselves (type `t_STR`). Not exactly so though, since we do some work to treat the quoted characters correctly (those preceded by a `\`).

2.5.4 Variable priorities, multivariate objects. PARI has no "sparse" representation of polynomials. So a multivariate polynomial in PARI is just a polynomial (in one variable), whose coefficients are themselves polynomials, arbitrary but for the fact that they do not involve the main variable. All computations are then just done formally on the coefficients as if the polynomial was univariate.

This is not symmetrical. So if I enter `x + y` in a clean session, what happens ? This is understood as

$$x^1 + y * x^0 \in (\mathbf{Z}[y])[x]$$

but how do we know that x is "more important" than y ? Why not $y^1 + x * y^0$, which is the same mathematical entity after all ?

The answer is that variables are ordered implicitly by the `gp` interpreter: when a new identifier (e.g x, or y as above) is input, the corresponding variable is registered as having a strictly lower priority than any variable in use at this point** . To see the ordering used by `gp` at any given time, type `reorder()`.

Given such an ordering, multivariate polynomials are stored so that the variable with the highest priority is the main variable. And so on, recursively, until all variables are exhausted. A different storage pattern (which could only be obtained via library mode) would produce an invalid object, and eventually a disaster.

In any case, if you are working with expressions involving several variables and want to have them ordered in a specific manner in the internal representation just described, the simplest is just to write down the variables one after the other under `gp` before starting any real computations. You could also define variables from your GPRC to have a consistent ordering of common variable names in all your `gp` sessions, e.g read in a file `variables.gp` containing

```
x;y;z;t;a;b;c;d;
```

If you already have started working and want to change the names of the variables in an object, use the function `changevar`. If you only want to have them ordered when the result is printed, you can also use the function `reorder`, but this won't change anything to the internal representation, and is not recommended.

Important note: PARI allows Euclidean division of multivariate polynomials, but assumes that the computation takes place in the fraction field of the coefficient ring (if it is not an integral domain, the result will a priori not make sense). This can be very tricky; for instance assume x has highest priority (which is always the case), then y:

```
? x % y
%1 = 0
? y % x
%2 = y                \\ these two take place in Q(y)[x]
? x * Mod(1,y)
%3 = Mod(1, y)*x      \\ in (Q(y)/yQ(y))[x] ~ Q[x]
? Mod(x,y)
%4 = 0
```

** This is not strictly true: if an identifier is interpreted as a user function, no variable is registered. Also, the variable x is predefined and always has the highest possible priority.

In the last example, the division by y takes place in $\mathbf{Q}(y)[x]$, hence the `Mod` object is a coset in $(\mathbf{Q}(y)[x])/(y\mathbf{Q}(y)[x])$, which is the null ring since y is invertible! So be very wary of variable ordering when your computations involve implicit divisions and many variables. This also affects functions like `numerator`/`denominator` or `content`:

```
? denominator(x / y)
%1 = 1
? denominator(y / x)
%2 = x
? content(x / y)
%3 = 1/y
? content(y / x)
%4 = y
? content(2 / x)
%5 = 2
```

Can you see why ? Hint: $x/y = (1/y) * x$ is in $\mathbf{Q}(y)[x]$ and denominator is taken with respect to $\mathbf{Q}(y)(x)$; $y/x = (y * x^0)/x$ is in $\mathbf{Q}(y)(x)$ so y is invertible in the coefficient ring. On the other hand, $2/x$ involves a single variable and the coefficient ring is simply $\mathbf{Z}$.

These problems arise because the variable ordering defines an *implicit* variable with respect to which division takes place. This is the price to pay to allow `%` and `/` operators on polynomials instead of requiring a more cumbersome `divrem(`x`, `y`, `*var*`)` (which also exists). Unfortunately, in some functions like `content` and `denominator`, there is no way to set explicitly a main variable like in `divrem` and remove the dependence on implicit orderings. This will hopefully be corrected in future versions.

2.5.5 Multivariate power series Just like multivariate polynomials, power series are fundamentally single-variable objects. It is awkward to handle many variables at once, since PARI's implementation cannot handle multivariate error terms like $O(x^i y^j)$. (It can handle the polynomial $O(y^j) \times x^i$ which is a very different thing, see below.)

The basic assumption in our model is that if variable x has higher priority than y, then y does not depend on x: setting y to a function of x after some computations with bivariate power series does not make sense a priori. This is because implicit constants in expressions like $O(x^i)$ depend on y (whereas in $O(y^j)$ they can not depend on x). For instance

```
? O(x) * y
%1 = O(x)
? O(y) * x
%2 = O(y)*x
```

Here is a more involved example:

```
? A = 1/x^2 + 1 + O(x); B = 1/x + 1 + O(x^3);
? subst(z*A, z, B)
%2 = x^-3 + x^-2 + x^-1 + 1 + O(x)
? B * A
%3 = x^-3 + x^-2 + x^-1 + O(1)
? z * A
%4 = z*x^-2 + z + O(x)
```

The discrepancy between `%2` and `%3` is surprising. Why does `%2` contain a spurious constant term, which cannot be deduced from the input ? Well, we ignored the rule that forbids to substitute an expression involving high-priority variables to a low-priority variable. The result `%4` is correct according to our rules since the implicit constant in $O(x)$ may depend on z. It is obviously wrong if z is allowed to have negative valuation in x. Of course, the correct error term should be $O(xz)$, but this is not possible in PARI.

2.6 User defined functions.

2.6.1 Definition. It is easy to define a new function in GP, which can then be used like any other function. The syntax is as follows:

`name(`*list of formal variables*`) = local(`*list of local variables*`);` *seq*

which looks better written on consecutive lines:

```
name(x0, x1, ...) =
{
  local(t0, t1, ...);
  local(...);

  ...
}
```

(the first newline is disregarded due to the preceding `=` sign, and the others because of the enclosing braces). Both lists of variables are comma-separated and allowed to be empty. The `local` statements can be omitted; as usual *seq* is any expression sequence.

`name` is the name given to the function and is subject to the same restrictions as variable names. In addition, variable names are not valid function names, you have to `kill` the variable first (the converse is true: function names can't be used as variables, see Section 3.11.2.14). Previously used function names can be recycled: you are just redefining the function. The previous definition is lost of course.

list of formal variables is the list of variables corresponding to those which you will actually use when calling your function. The number of actual parameters supplied when calling the function has to be less than the number of formal variables. Arguments are passed by value, not as variables: modifying a function's argument in the function body is allowed, but does not modify its value in the calling frame. In fact, a *copy* of the actual parameter is assigned to the formal parameter when the function is called.

Uninitialized formal variables are given a default value. An equal (`=`) sign following a variable name in the function definition, followed by any expression, gives the variable a default value. The said expression gets evaluated the moment the function is called, hence may involve the preceding function parameters (a default value for x_i may involve x_j for $j < i$). A variable for which you supply no default value is initialized to (the integer) zero. For instance

```
foo(x, y=2, z=3) = print(x ":" y ":" z)
```

defines a function which prints its arguments (at most three of them), separated by colons. This then follows the rules of default arguments generation as explained at the beginning of Section 3.0.2.

```
? foo(6,7)
```

```
6:7:3
? foo(,5)
0:5:3
? foo()
0:2:3
```

list of local variables is the list of additional temporary variables used in the function body. Note that if you omit some or all of these local variable declarations, the non-declared variables will become global, hence known outside of the function, and this may have undesirable side-effects. On the other hand, in some cases it may also be what you want. See Section 2.6.6 for details. Local variables can be given a default value as the formal variables.

Restrictions on variable use: it is not allowed to use the same variable name for different parameters of your function. Or to use a given variable both as a formal parameter and a local variable in a given function. Hence

```
? f(x,x) = 1
  ***   user function f: variable x declared twice.
```

Note: The above syntax (using the `local` keyword) was introduced in version 2.0.13. The old syntax

`name(`*list of true formal variables, list of local variables*`) =` *seq*

is still recognized but is deprecated since genuine arguments and local variables become undistinguishable.

2.6.2 Use. Once the function is defined using the above syntax, you can use it like any other function, see the example with `fun` above. In addition, you can also recall its definition exactly as you do for predefined functions, that is by writing `?`*name*. This will print the list of arguments, as well as their default values, the text of *seq*, and a short help text if one was provided using the `addhelp` function (see Section 3.11.2.1). One small difference to predefined functions is that you can never redefine the built-in functions, while you can redefine a user-defined function as many times as you want.

Typing `\u` will output the list of user-defined functions.

An amusing example of a user-defined function is the following. It is intended to illustrate both the use of user-defined functions and the power of the `sumalt` function. Although the Riemann zeta-function is included in the standard functions, let us assume that this is not the case (or that we want another implementation). One way to define it, which is probably the simplest, but certainly not the most efficient, is as follows:

```
zet(s) =
{ local(n); /* not needed, and possibly confusing (see below) */
  sumalt(n=1, (-1)^(n-1)*n^(-s)) / (1 - 2^(1-s))
}
```

This gives reasonably good accuracy and speed as long as you are not too far from the domain of convergence. Try it for s integral between -5 and 5, say, or for $s = 0.5 + i * t$ where $t = 14.134\ldots$

2.6.3 Recursive functions. Recursive functions can easily be written as long as one pays proper attention to variable scope. Here is an example, used to retrieve the coefficient array of a multivariate polynomial (a non-trivial task due to PARI's unsophisticated representation for those objects):

```
coeffs(P, nbvar) =
{
  if (type(P) != "t_POL",
    for (i=1, nbvar, P = [P]);
    return (P)
  );
  vector(poldegree(P)+1, i, coeffs(polcoeff(P, i-1), nbvar-1))
}
```

If P is a polynomial in k variables, show that after the assignment `v = coeffs(P,k)`, the coefficient of $x_1^{n_1} \dots x_k^{n_k}$ in P is given by `v[`n_1`+1][...][`n_k`+1]`.

The operating system automatically limits the recursion depth:

```
? dive(n) = if (n, dive(n-1))
? dive(5000);
  ***   deep recursion: if(n,dive(n-1))
                        ^---------------
```

There is no way to increase the recursion limit (which may be different on your machine) from within `gp`. To increase it before launching `gp`, you can use `ulimit` or `limit`, depending on your shell, and raise the process available stack space (increase `stacksize`).

2.6.4 Function which take functions as parameters ? Use the following trick (neat example due to Bill Daly):

```
calc(f, x) = eval( Str(f, "(x)") )
```

If you call this with `calc("sin", 1)`, it will return sin(1) (evaluated!).

2.6.5 Defining functions within a function ? The first idea

```
init(x) = add(y) = x+y; mul(y) = x*y;
```

does not work since in the construction `f() =` *seq*, the function body contains everything until the end of the expression. Hence executing `init` defines the wrong function `add`. The way out is to use parentheses for grouping, to that enclosed subexpressions be evaluated independently:

```
init(x) = ( add(y) = x+y ); ( mul(y) = x*y );
```

2.6.6 Variable scope.

Local variables should more appropriately be called *temporary values* since they are in fact local to the function declaring them *and* any subroutine called from within. In the following example

```
f() = local(y); ... ; g()
g() = y + 1
```

`g()` "sees" the y introduced in `f()`. True lexical scoping does not exist in GP. (See e.g. the difference between `local` and `my` in Perl.)

In an iterative constructs which use a variable name (`for`*xxx*, `prod`*xxx*, `sum`*xxx*, `vector`, `matrix`, `plot`, etc.) the given variable is also local to the construct. A value is pushed on entry and poped on exit. So, it is not necessary for a function using such an iterator to declare the variable as `local`. On the other hand, if you exit the loop prematurely, e.g. using the `break` statement, you must save the loop index in another variable since its value prior the loop will be restored upon exit: for instance

```
for(i = 1, n,
  if (ok(i), break);
);
if (i > n, return(failure));
```

is incorrect, since the value of i tested by the $(i > n)$ is quite unrelated to the loop index.

Finally, the statement `global(x, y, z, t)` (see Section 3.11.2.11) declares the corresponding variables to be global. It is then forbidden to use them as formal parameters or loop indexes as above, since the parameter would "shadow" the variable. If speed is of the essence and an object is large (e.g. a *bnf*, a huge matrix), it should be declared `global`, not passed as a parameter, since this saves an expensive copy. It is possible to declare it local and use it as a global variables from relevant subroutines, but `global` is safer.

It is strongly recommended to explicitly declare all `global` variables at the beginning of your program and all `local` variable used inside a given function, with the possible exception of loop indexes which are local to their loop. If a function accesses a variable which is not one of its formal parameters, the value used will be the one which happens to be on top of the stack at the time of the call. This could be a "global" value, or a local value belonging to any function higher in the call chain, and is in general *not* what you want to do. So, be warned.

Coming back to our previous example `zet`, since loop variables are not visible outside their loops, the variable `n` need not be declared in the function protoype.

```
zet(s) = sumalt(n=1, (-1)^(n-1)*n^(-s)) / (1 - 2^(1-s))
```

would be a better definition. One last example: what is wrong with the following definition?

```
FirstPrimeDiv(x) =
{ local(p);
  forprime(p=2, x, if (x%p == 0, break));
  p
}
? FirstPrimeDiv(10)
%1 = 0
```

Well, the index p in the `forprime` loop is local to the loop and is not visible to the outside world. Hence, it does not survive the `break` statement. More precisely, at this point the loop index is restored to its preceding value, which is 0 (local variables are initialized to 0 by default). To sum up, the routine returns the p declared local to it, not the one which was local to `forprime` and ran through consecutive prime numbers. Here is a corrected version:

```
? FirstPrimeDiv(x) = forprime(p=2, x, if (x%p == 0, return(p)))
```

Implementation note: For the curious reader, here is how values of variables are handled: a hashing function is computed from the variable name, and used as an index in `hashtable`, a table of linked list of structures (type `entree`). The linked list is searched linearly for the identifier (each list typically has less than 10 components). When the correct `entree` is found, it points to the top of the stack of values for that identifier if it is a variable, to the function itself if it is a predefined function, and to a copy of the text of the function if it is a user-defined function. When an error occurs, all of this maze (rather a tree, in fact) is searched and restored to the state preceding the last call of the main evaluator.

2.7 Member functions.

Member functions use the 'dot' notation to retrieve information from complicated structures, by default: *bid*, *ell*, *galois*, *nf*, *bnf*, *bnr* and prime ideals. The syntax `structure.member` is taken to mean: retrieve `member` from `structure`, e.g. `ell.j` returns the j-invariant of the elliptic curve `ell`, or outputs an error message if `ell` doesn't have the correct type.

To define your own member functions, use the syntax *structure.member* = *function text*, where *function text* is written as the *seq* in a standard user function (without local variables), whose only argument would be `structure`. For instance, the current implementation of the `ell` type is simply an horizontal vector, the j-invariant being the thirteenth component. It could be implemented as

```
x.j =
{
  if (type(x) != "t_VEC" || length(x) < 14,
    error("this is not a proper elliptic curve: " x)
  );
  x[13]
}
```

Typing `\um` will output the list of user-defined member functions.

You can redefine one of your own member functions simply by typing a new definition for it. On the other hand, as a safety measure, you can't redefine the built-in member functions, so typing the above text would in fact produce an error (you'd have to call it e.g. `x.myj` in order for `gp` to accept it).

Warning: contrary to user functions arguments, the structure accessed by a member function is *not* copied before being used. Any modification to the structure's components will be permanent.

Warning: it is advised not to apply a member whose name starts with `e` or `E` to an integer constant, where it would be confused with the usual floating point exponent. E.g compare

```
? x.e2 = x+1
? 1.e2
%1 = 100.000000000 \\ taken to mean 1.0E2.
? (1).e2
%2 = 2
? 1.0.e2
%3 = 2.00000000000
```

Note: Member functions were not meant to be too complicated or to depend on any data that wouldn't be global. Hence they do no have parameters (besides the implicit `structure`) or local variables. Of course, if you need some preprocessing work in there, there's nothing to prevent you from calling your own functions (using freely their local variables) from a member function. For instance, one could implement (a dreadful idea as far as efficiency goes):

```
correct_ell_if_needed(x) =
{ local(tmp);
  if (type(x) != "t_VEC", tmp = ellinit(x))
    \\  some further checks
  tmp
}
x.j = correct_ell_if_needed(x)[13];
```

2.8 Strings and Keywords.

2.8.1 Strings. GP variables can hold values of type character string (internal type `t_STR`). This section describes how they are actually used, as well as some convenient tricks (automatic concatenation and expansion, keywords) valid in string context.

As explained above, the general way to input a string is to enclose characters between quotes `"`. This is the only input construct where whitespace characters are significant: the string will contain the exact number of spaces you typed in. Besides, you can "escape" characters by putting a `\` just before them; the translation is as follows

```
\e: <Escape>
\n: <Newline>
\t: <Tab>
```

For any other character x, `\`x is expanded to x. In particular, the only way to put a `"` into a string is to escape it. Thus, for instance, `"\"a\""` would produce the string whose content is "a". This is definitely *not* the same thing as typing `"a"`, whose content is merely the one-letter string a.

You can concatenate two strings using the `concat` function. If either argument is a string, the other is automatically converted to a string if necessary (it will be evaluated first).

```
? concat("ex", 1+1)
%1 = "ex2"
? a = 2; b = "ex"; concat(b, a)
%2 = "ex2"
```

```
? concat(a, b)
%3 = "2ex"
```

Some functions expect strings for some of their arguments: `print` would be an obvious example, `Str` is a less obvious but useful one (see the end of this section for a complete list). While typing in such an argument, you will be said to be in *string context*. The rest of this section is devoted to special syntactical tricks which can be used with such arguments (and only here; you will get an error message if you try these outside of string context):

• Writing two strings alongside one another will just concatenate them, producing a longer string. Thus it is equivalent to type in `"a " "b"` or `"a b"`. A little tricky point in the first expression: the first whitespace is enclosed between quotes, and so is part of a string; while the second (before the `"b"`) is completely optional and `gp` actually suppresses it, as it would with any number of whitespace characters at this point (i.e. outside of any string).

• If you insert any expression when a string is expected, it gets "expanded": it is evaluated as a standard GP expression, and the final result (as would have been printed if you had typed it by itself) is then converted to a string, as if you had typed it directly. For instance `"a" 1+1 "b"` is equivalent to `"a2b"`: three strings get created, the middle one being the expansion of `1+1`, and these are then concatenated according to the rule described above. Another tricky point here: assume you did not assign a value to `aaa` in a GP expression before. Then typing `aaa` by itself in a string context will actually produce the correct output (i.e. the string whose content is aaa), but in a fortuitous way. This `aaa` gets expanded to the monomial of degree one in the variable `aaa`, which is of course printed as `aaa`, and thus will expand to the three letters you were expecting.

Warning: expression involving strings are not handled in a special way; even in string context, the largest possible expression is evaluated, hence `print("a"[1])` is incorrect since `"a"` is not an object whose first component can be extracted. On the other hand `print("a", [1])` is correct (two distinct argument, each converted to a string), and so is `print("a" 1)` (since `"a"1` is not a valid expression, only `"a"` gets expanded, then `1`, and the result is concatenated as explained above). In case of doubt, you can surround part of your text by parenthesis to force immediate interpretation of a subexpression: `print("a"([1]))` is another solution.

2.8.2 Keywords. Since there are cases where expansion is not desirable, we now distinguish between "Keywords" and "Strings". String is what has been described so far. Keywords are special relatives of Strings which are automatically assumed to be quoted, whether you actually type in the quotes or not. Thus expansion is never performed on them. They get concatenated, though. The analyzer supplies automatically the quotes you have "forgotten" and treats Keywords just as normal strings otherwise. For instance, if you type `"a"b+b` in Keyword context, you will get the string whose contents are ab+b. In String context, on the other hand, you would get a2*b.

All GP functions have prototypes (described in Chapter 3 below) which specify the types of arguments they expect: either generic PARI objects (GEN), or strings, or keywords, or unevaluated expression sequences. In the keyword case, only a very small set of words will actually be meaningful (the `default` function is a prominent example).

Reference: The arguments of the following functions are processed in string context:

`Str`
`addhelp` (second argument)
`default` (second argument)
`error`
`extern`
`plotstring` (second argument)
`plotterm` (first argument)
`read` and `readvec`
`system`
all the `print`*xxx* functions
all the `write`*xxx* functions

The arguments of the following functions are processed as keywords:

`alias`
`default` (first argument)
`install` (all arguments but the last)
`trap` (first argument)
`type` (second argument)
`whatnow`

2.8.3 Useful examples The function `Str` converts its arguments into strings and concatenate them. Coupled with `eval`, it is very powerful. The following example creates generic matrices:

```
? genmat(u,v,s="x") = matrix(u,v,i,j, eval( Str(s,i,j) ))
? genmat(2,3) + genmat(2,3,"m")
%1 =
[x11 + m11 x12 + m12 x13 + m13]
[x21 + m21 x22 + m22 x23 + m23]
```

Two last examples: `hist(10,20)` returns all history entries from `%10` to `%20` neatly packed into a single vector; `histlast(10)` returns the last 10 history entries:

```
hist(a,b) = vector(b-a+1, i, eval(Str("%", a-1+i)))
histlast(n) = vector(n, i, eval(Str("%", %#-i+1)))
```

2.9 Errors and error recovery.

2.9.1 Errors. There are two kind of errors: syntax errors, and errors produced by functions in the PARI library. Both kinds are fatal to your computation: `gp` will report the error, perform some cleanup (restore variables modified while evaluating the erroneous command, close open files, reclaim unused memory, etc.), and will output its usual prompt.

When reporting a syntax error, `gp` tries to give meaningful context by copying the sentence it was trying to read (whitespace and comments stripped out), indicating an error with a little caret like in

```
? factor(x^2 - 1
  ***   expected character: ',' instead of: factor(x^2-1
                                                     ^
```

possibly enlarged to a full arrow given enough trailing context

```
? if (siN(x) < eps, do_something())
  ***   expected character: '=' instead of: if(siN(x)<eps,do_something())
                                                        ^--------------------
```

Error messages will often be mysterious, because gp cannot guess what you were trying to do and the error usually occurs once gp has been sidetracked. Let's have a look at the two messages above.

The first error is a missing parenthesis, but from gp's point of view, you might as well have intended to give further arguments to `factor` (this is possible, and often useful, see the description of the function). Since gp did not see the closing parenthesis, it tried to read a second argument, first looking for the comma that would separate it form the first. The error occurred at this point. So gp tells you that it was expecting a comma and saw a blank.

The second error is even weirder. It is a simple typo, `siN` instead of `sin` and gp tells us that is was expecting an equal sign a few characters later? What happens is this: `siN` is not a recognized identifier, but from the context, it looks like a function (it is followed by an open parenthesis), then we have an argument, then a closing parenthesis. Then if `siN` were a known function we would evaluate it; but it is not, so gp assumes that you were trying to *define* it, as in

```
? if (siN(x)=sin(x), ...)
```

This is actually allowed (!) and defines the function `siN` as an alias for `sin`. As any expression a function definition has a value, which is 0, hence the test is meaningful, and false, so nothing happens. (Admittedly this doesn't look like a useful syntax but it can be interesting in other contexts to let functions define other functions. Anyway, it is allowed by the language definition.) So gp tells you in good faith that to correctly define a function, you need an equal sign between its name and its body.

Error messages from the library will usually be clearer since, by definition, they answer a correctly worded query (otherwise gp would have protested first). Also they have more mathematical content, which should be easier to grasp than a parser's logic. For instance:

```
? 1/0
  ***   division by zero
```

2.9.2 Error recovery.

It is quite annoying to wait for some program to finish and find out the hard way that there was a mistake in it (like the division by 0 above), sending you back to the prompt. First you may lose some valuable intermediate data. Also, correcting the error may not be ovious; you might have to change your program, adding a number of extra statements and tests to try and narrow down the problem.

A slightly different situation, still related to error recovery, is when you you actually foresee that some error may occur, are unable to prevent it, but quite capable of recovering from it, given the chance. Examples include lazy factorization (cf. `addprimes`), where you knowingly use a pseudo prime N as if it were prime; you may then encounter an "impossible" situation, but this would usually exhibit a factor of N, enabling you to refine the factorization and go on. Or you might run an expensive computation at low precision to guess the size of the output, hence the right precision to use. You can then encounter errors like "precision loss in truncation", e.g when trying to convert `1E1000`, known to 28 digits of accuracy, to an integer; or "division by 0", e.g inverting `0E1000` when

all accuracy has been lost, and no significant digit remains. It would be enough to restart part of the computation at a slightly higher precision.

We now describe *error trapping*, a useful mechanism which alleviates much of the pain in the first situation, and provides a satisfactory way out of the second one. Everything is handled via the `trap` function whose different modes we now describe.

2.9.3 Break loop.

A *break loop* is a special debugging mode that you enter whenever an error occurs, freezing the `gp` state, and preventing cleanup until you get out of the loop. Any error: syntax error, library error, user error (from `error`), even user interrupts like `C-c` (Control-C). When a break loop starts, a prompt is issued (`break>`). You can type in a `gp` command, which is evaluated when you hit the `<Return>` key, and the result is printed as during the main `gp` loop, except that no history of results is kept Then the break loop prompt reappears and you can type further commands as long as you do not exit the loop. If you are using readline, the history of commands is kept, and line editing is available as usual. If you type in a command that results in an error, you are sent back to the break loop prompt (errors does *not* terminate the loop).

To get out of a break loop, you can use `next`, `break`, `return`, or `C-d` (`EOF`), any of which will let `gp` perform its usual cleanup, and send you back to the `gp` prompt. If the error is not fatal, inputing an empty line, i.e hitting the `<Return>` key at the `break>` prompt, will continue the temporarily interrupted computation. An empty line has no effect in case of a fatal error, to ensure you do not get out of the loop prematurely, thus losing most debugging data during the cleanup (since user variables will be restored to their former values).

In current version 2.3.5, an error is non-fatal if and only if it was initiated by a `C-c` typed by the user.

Break loops are useful as a debugging tool to inspect the values of `gp` variables to understand why an error occurred, or to change `gp`'s state in the middle of a computation (increase debugging level, start storing results in a logfile, set variables to different values...): hit `C-c`, type in your modifications, then let the computation go on as explained above.

A break loop looks like this:

```
? for(v = -2, 2, print(1/v))
-1/2
-1
  ***   division by zero in gdiv, gdivgs or ginv
  ***   Starting break loop (type 'break' to go back to GP):
  ***   for(v=-2,2,print(1/v))
                            ^--
break>
```

So the standard error message is printed first, except now we always have context, whether the error comes from the library or the parser. The `break>` at the bottow is a prompt, and hitting `v` then `<Return>`, we see:

```
break> v
0
```

explaining the problem. We could have typed any `gp` command, not only the name of a variable, of course. There is no special set of commands becoming available during a break loop, as they would in most debuggers.

Important Note: upon startup, this mechanism is *off*. Type `trap()` (or include it in a script) to start trapping errors in this way. By default, you will be sent back to the prompt.

Technical Note: When you enter a break loop due to a PARI stack overflow, the PARI stack is reset so that you can run commands (otherwise the stack would immediately overflow again). Still, as explained above, you do not lose the value of any `gp` variable in the process.

2.9.4 Error handlers. The break loop described above is a (sophisticated) example of an *error handler*: a function that is executed whenever an error occurs, supposedly to try and recover. The break loop is quite a satisfactory error handler, but it may not be adequate for some purposes, for instance when `gp` runs in non-interactive mode, detached from a terminal.

So, you can define a different error handler, to be used in place of the break loop. This is the purpose of the *second* argument of `trap`: to specify an error handler. (We will discuss the first argument at the very end.) For instance:

```
? { trap( ,                \\ note the comma: arg1 is omitted
      print(reorder);
      writebin("crash")) }
```

After that, whenever an error occurs, the list of all user variables is printed, and they are all saved in binary format in file `crash`, ready for inspection. Of course break loops are no longer available: the new handler has replaced the default one. Besides user-defined handlers as above, there are two special handlers you can use in `trap`, which are

• `trap(, "")` (do-nothing handler): to disable the trapping mechanism and let errors propagate, which is the default situation on startup.

• `trap(, )` (omitted argument, default handler): to trap errors by a break loop.

2.9.5 Protecting code. Finally `trap` can define a temporary handler used within the scope of a code frament, protecting it from errors, by providing replacement code should the trap be activated. The expression

`trap`(, *recovery*, *statements*)

evaluates and returns the value of *statements*, unless an error occurs during the evaluation in which case the value of *recovery* is returned. As in an if/else clause, with the difference that *statements* has been partially evaluated, with possible side effects. For instance one could define a fault tolerant inversion function as follows:

```
? inv(x) = trap (, "oo", 1/x)
? for (i=-1,1, print(inv(i)))
-1
oo
1
```

Protected codes can be nested without adverse effect, the last trap seen being the first to spring.

2.9.6 Trapping specific exceptions. We have not yet seen the use of the first argument of trap, which has been omitted in all previous examples. It simply indicates that only errors of a specific type should be intercepted, to be chosen among

`accurer`: accuracy problem

`gdiver`: division by 0

`invmoder`: impossible inverse modulo

`archer`: not available on this architecture or operating system

`typeer`: wrong type

`errpile`: the PARI stack overflows

Omitting the error name means we are trapping all errors. For instance, the following can be used to check in a safe way whether `install` works correctly in your `gp`:

```
broken_install() =
{
  trap(archer, return ("OS"),
    install(addii,GG)
  );
  trap(, "USE",
    if (addii(1,1) != 2, "BROKEN")
  )
}
```

The function returns 0 if everything works (the omitted *else* clause of the `if`), `OS` if the operating system does not support `install`, `USE` if using an installed function triggers an error, and `BROKEN` if the installed function did not behave as expected.

2.10 Interfacing GP with other languages.

The PARI library was meant to be interfaced with C programs. This specific use will be dealt with extensively in Chapter 4. `gp` itself provides a convenient, if simple-minded, interpreter, which enables you to execute rather intricate scripts (see Section 3.11).

Scripts, when properly written, tend to be shorter and clearer than C programs, and are certainly easier to write, maintain or debug. You don't need to deal with memory management, garbage collection, pointers, declarations, and so on. Because of their intrinsic simplicity, they are more robust as well. They are unfortunately somewhat slower. Thus their use will remain complementary: it is suggested that you test and debug your algorithms using scripts, before actually coding them in C for the sake of speed. The GP2C compiler often eases this part.

Note also that the `install` command enables you to concentrate on critical parts of your programs only (which can of course be written with the help of other mathematical libraries than PARI!), and to efficiently import foreign functions for use under `gp` (see Section 3.11.2.13).

We are aware of four PARI-related public domain packages to embed PARI in other languages. We *neither endorse nor support* any of them; you might want to give them a try if you are familiar with the languages they are based on. The first is the `Math::Pari` Perl module (see any CPAN

mirror), written by Ilya Zakharevich. The second is `PariPython`*, by Stéfane Fermigier, which is no more maintained. Starting from Fermigier's work, Wiliam Stein has embedded PARI into his Python-based SAGE** system. Finally, Michael Stoll has integrated PARI into `CLISP` ***, which is a Common Lisp implementation by Bruno Haible, Marcus Daniels and others; this interface has been updated for pari-2 by Sam Steingold.

These provide interfaces to `gp` functions for use in `perl`, `python` or `Lisp` programs, respectively.

2.11 Defaults.

There are many internal variables in `gp`, defining how the system will behave in certain situations, unless a specific override has been given. Most of them are a matter of basic customization (colors, prompt) and will be set once and for all in your preferences file (see Section 2.13), but some of them are useful interactively (set timer on, increase precision, etc.).

The function used to manipulate these values is called `default`, which is described in Section 3.11.2.4. The basic syntax is

`default(`*def*`,` *value*`)`,

which sets the default *def* to *value*. In interactive use, most of these can be abbreviated using historic `gp` metacommands (mostly, starting with `\`), which we shall describe in the next section.

Here we will only describe the available defaults and how they are used. Just be aware that typing `default` by itself will list all of them, as well as their current values (see `\d`). Just after the default name, we give between parentheses the initial value when `gp` starts (assuming you did not tamper with it using command-line switches or a `gprc`).

Note: the suffixes `k`, `M` or `G` can be appended to a *value* which is a numeric argument, with the effect of multiplying it by 10^3, 10^6 and 10^9 respectively. Case is not taken into account there, so for instance `30k` and `30K` both stand for 30000. This is mostly useful to modify or set the defaults `primelimit` or `stacksize` which typically involve a lot of trailing zeroes.

(somewhat technical) Note: As we will see in Section 2.8, the second argument to `default` will be subject to string context expansion, which means you can use run-time values. In other words, something like

```
a = 3;
default(logfile, "\var{some filename}" a ".log")
```

logs the output in *some filename*`3.log`.

Some defaults will be expanded further when the values are used, after the above expansion has been performed:

• **time expansion**: the string is sent through the library function `strftime`. This means that `%`*char* combinations have a special meaning, usually related to the time and date. For instance, `%H` = hour (24-hour clock) and `%M` = minute [00,59] (on a Unix system, you can try `man strftime` at your shell prompt to get a complete list). This is applied to `prompt`, `psfile`, and `logfile`. For instance,

* see `http://www.fermigier.com/fermigier/PariPython/`
** see `http://modular.fas.harvard.edu/sage/`
*** see `http://clisp.cons.org`

```
default(prompt,"(%H:%M) ? ")
```

will prepend the time of day, in the form `(`*hh*`:`*mm*`)` to `gp`'s usual prompt.

• **environment expansion**: When the string contains a sequence of the form `$`*SOMEVAR*, e.g. `$HOME`, the environment is searched and if *SOMEVAR* is defined, the sequence is replaced by the corresponding value. Also the `~` symbol has the same meaning as in many shells — `~` by itself stands for your home directory, and `~user` is expanded to `user`'s home directory. This is applied to all filenames.

2.11.1 colors (default `""`): this default is only usable if `gp` is running within certain color-capable terminals. For instance `rxvt`, `color_xterm` and modern versions of `xterm` under X Windows, or standard Linux/DOS text consoles. It causes `gp` to use a small palette of colors for its output. With xterms, the colormap used corresponds to the resources `Xterm*color`n where n ranges from 0 to 15 (see the file `misc/color.dft` for an example). Accepted values for this default are strings `"`a_1`,...,`a_k`"` where $k \leq 7$ and each a_i is either

• the keyword `no` (use the default color, usually black on transparent background)

• an integer between 0 and 15 corresponding to the aforementioned colormap

• a triple $[c_0, c_1, c_2]$ where c_0 stands for foreground color, c_1 for background color, and c_2 for attributes (0 is default, 1 is bold, 4 is underline).

The output objects thus affected are respectively error messages, history numbers, prompt, input line, output, help messages, timer (that's seven of them). If $k < 7$, the remaining a_i are assumed to be *no*. For instance

```
default(colors, "9, 5, no, no, 4")
```

typesets error messages in color 9, history numbers in color 5, output in color 4, and does not affect the rest.

A set of default colors for dark (reverse video or PC console) and light backgrounds respectively is activated when `colors` is set to `darkbg`, resp. `lightbg` (or any proper prefix: `d` is recognized as an abbreviation for `darkbg`). A bold variant of `darkbg`, called `boldfg`, is provided if you find the former too pale.

EMACS: In the present version, this default is incompatible with Emacs. Changing it will just fail silently (the alternative would be to display escape sequences as is, since Emacs will refuse to interpret them). On the other hand, you can customize highlighting in your `.emacs` so as to mimic exactly this behaviour. See `emacs/pariemacs.txt`.

Technical note: If you use an old `readline` library (version number less than 2.0), you should do as in the example above and leave a_3 and a_4 (prompt and input line) strictly alone. Since old versions of `readline` did not handle escape characters correctly (or more accurately, treated them in the only sensible way since they did not care to check all your terminal capabilities: it just ignored them), changing them would result in many annoying display bugs.

The specific thing to look for is to check the `readline.h` include file, wherever your readline include files are, for the string `RL_PROMPT_START_IGNORE`. If it is there, you are safe. Another sensible way is to make some experiments, and get a more recent `readline` if yours doesn't work the way you would like it to. See the file `misc/gprc.dft` for some examples.

2.11.2 compatible (default `0`): The GP function names and syntax have changed tremendously between versions 1.xx and 2.00. To help you cope with this we provide some kind of backward compatibility, depending on the value of this default:

`compatible` = 0: no backward compatibility. In this mode, a very handy function, to be described in Section 3.11.2.29, is `whatnow`, which tells you what has become of your favourite functions, which `gp` suddenly can't seem to remember.

`compatible` = 1: warn when using obsolete functions, but otherwise accept them. The output uses the new conventions though, and there may be subtle incompatibilities between the behaviour of former and current functions, even when they share the same name (the current function is used in such cases, of course!). We thought of this one as a transitory help for `gp` old-timers. Thus, to encourage switching to `compatible`=0, it is not possible to disable the warning.

`compatible` = 2: use only the old function naming scheme (as used up to version 1.39.15), but *taking case into account*. Thus `I` ($= \sqrt{-1}$) is not the same as `i` (user variable, unbound by default), and you won't get an error message using `i` as a loop index as used to be the case.

`compatible` = 3: try to mimic exactly the former behaviour. This is not always possible when functions have changed in a fundamental way. But these differences are usually for the better (they were meant to, anyway), and will probably not be discovered by the casual user.

One adverse side effect is that any user functions and aliases that have been defined *before* changing `compatible` will get erased if this change modifies the function list, i.e. if you move between groups $\{0, 1\}$ and $\{2, 3\}$ (variables are unaffected). We of course strongly encourage you to try and get used to the setting `compatible`=0.

Note that the default `new_galois_format` is another compatibility setting, which is completely independent of `compatible`.

2.11.3 datadir (default: the location of installed precomputed data): the name of directory containing the optional data files. For now, only the `galdata` and `elldata` packages.

2.11.4 debug (default `0`): debugging level. If it is non-zero, some extra messages may be printed (some of it in French), according to what is going on (see `\g`).

2.11.5 debugfiles (default `0`): file usage debugging level. If it is non-zero, `gp` will print information on file descriptors in use, from PARI's point of view (see `\gf`).

2.11.6 debugmem (default `0`): memory debugging level. If it is non-zero, `gp` will regularly print information on memory usage. If it's greater than 2, it will indicate any important garbage collecting and the function it is taking place in (see `\gm`).

Important Note: As it noticeably slows down the performance, the first functionality (memory usage) is disabled if you're not running a version compiled for debugging (see Appendix A).

2.11.7 echo (default `0`): this is a toggle, which can be either 1 (on) or 0 (off). When `echo` mode is on, each command is reprinted before being executed. This can be useful when reading a file with the `\r` or `read` commands. For example, it is turned on at the beginning of the test files used to check whether `gp` has been built correctly (see `\e`).

2.11.8 factor_add_primes (default `0`): if this is set, the integer factorization machinery will call `addprimes` on primes factor that were difficult to find, so they are automatically tried first in other factorizations. If a routine is performing (or has performed) a factorization and is interrupted by an error or via Control-C, this let you recover the prime factors already found.

2.11.9 format (default `"g0.28"` and `"g0.38"` on 32-bit and 64-bit machines, respectively): of the form x$m.n$, where x is a letter in {`e`, `f`, `g`}, and n, m are integers. If x is `f`, real numbers will be printed in fixed floating point format with no explicit exponent (e.g. `0.000033`), unless their integer part is not defined (not enough significant digits); if the letter is `e`, they will be printed in scientific format, always with an explicit exponent (e.g. `3.3e-5`). If the letter is `g`, real numbers will be printed in `f` format, except when their absolute value is less than 2^{-32} or they are real zeroes (of arbitrary exponent), in which case they are printed in `e` format.

The number n is the number of significant digits printed for real numbers, except if $n < 0$ where all the significant digits will be printed (initial default 28, or 38 for 64-bit machines), and the number m is the number of characters to be used for printing integers, but is ignored if equal to 0 (which is the default). This is a feeble attempt at formatting.

UNIX: **2.11.10 help** (default: the location of the `gphelp` script): the name of the external help program which will be used from within `gp` when extended help is invoked, usually through a `??` or `???` request (see Section 2.12.1), or `M-H` under readline (see Section 2.15.1).

2.11.11 histsize (default `5000`): `gp` keeps a history of the last `histsize` results computed so far, which you can recover using the `%` notation (see Section 2.12.4). When this number is exceeded, the oldest values are erased. Tampering with this default is the only way to get rid of the ones you do not need anymore.

2.11.12 lines (default `0`): if set to a positive value, `gp` prints at most that many lines from each result, terminating the last line shown with `[+++]` if further material has been suppressed. The various `print` commands (see Section 3.11.2) are unaffected, so you can always type `print(%)`, `\a`, or `\b` to view the full result. If the actual screen width cannot be determined, a "line" is assumed to be 80 characters long.

2.11.13 log (default `0`): this can be either 0 (off) or 1, 2, 3 (on, see below for the various modes). When logging mode is turned on, `gp` opens a log file, whose exact name is determined by the `logfile` default. Subsequently, all the commands and results will be written to that file (see `\l`). In case a file with this precise name already existed, it will not be erased: your data will be *appended* at the end.

The specific positive values of `log` have the following meaning

1: plain logfile

2: emit color codes to the logfile (if `colors` is set).

3: write LaTeX output to the logfile (can be further customized using `TeXstyle`).

2.11.14 logfile (default `"pari.log"`): name of the log file to be used when the `log` toggle is on. Environment and time expansion are performed.

2.11.15 new_galois_format (default `0`): if this is set, the `polgalois` command will use a different, more consistent, naming scheme for Galois groups. This default is provided to ensure that scripts can control this behaviour and do not break unexpectedly. Note that the default value of 0 (unset) will change to 1 (set) in the next major version.

2.11.16 output (default `1`): there are four possible values: 0 (= *raw*), 1 (= *prettymatrix*), 2 (= *prettyprint*), or 3 (= *external prettyprint*). This means that, independently of the default `format` for reals which we explained above, you can print results in four ways: either in *raw format*, i.e. a format which is equivalent to what you input, including explicit multiplication signs, and everything typed on a line instead of two dimensional boxes. This can have several advantages, for instance it allows you to pick the result with a mouse or an editor, and to paste it somewhere else.

The second format is the *prettymatrix format*. The only difference to raw format is that matrices are printed as boxes instead of horizontally. This is prettier, but takes more space and cannot be used for input. Column vectors are still printed horizontally.

The third format is the *prettyprint format*, or beautified format. In the present version 2.3.5, this is not beautiful at all.

The fourth format is *external prettyprint*, which pipes all `gp` output in TeX format to an external prettyprinter, according to the value of `prettyprinter`. The default script (`tex2mail`) converts its input to readable two-dimensional text.

Independently of the setting of this default, an object can be printed in any of the three formats at any time using the commands `\a`, `\m` and `\b` respectively (see below).

2.11.17 parisize (default 4M, resp. 8M on a 32-bit, resp. 64-bit machine): `gp`, and in fact any program using the PARI library, needs a *stack* in which to do its computations. `parisize` is the stack size, in bytes. It is strongly recommended you increase this default (using the `-s` command-line switch, or a `gprc`) if you can afford it. Don't increase it beyond the actual amount of RAM installed on your computer or `gp` will spend most of its time paging.

In case of emergency, you can use the `allocatemem` function to increase `parisize`, once the session is started.

2.11.18 path (default `".:~:~/gp"` on UNIX systems, `".;C:\;C:\GP` on DOS, OS/2 and Windows, and `"."` otherwise): This is a list of directories, separated by colons ':' (semicolons ';' in the DOS world, since colons are pre-empted for drive names). When asked to read a file whose name does not contain `/` (i.e. no explicit path was given), `gp` will look for it in these directories, in the order they were written in `path`. Here, as usual, '.' means the current directory, and '..' its immediate parent. Environment expansion is performed.

UNIX: **2.11.19 prettyprinter** (default `"tex2mail -TeX -noindent -ragged -by_par"`) the name of an external prettyprinter to use when `output` is 3 (*alternate prettyprinter*). Note that the default `tex2mail` looks much nicer than the built-in "beautified format" (`output` = 2).

2.11.20 primelimit (default `500k`): `gp` precomputes a list of all primes less than `primelimit` at initialization time. These are used by many arithmetical functions. If you don't plan to invoke any of them, you can just set this to 1. The maximal value is a little less than 2^32 (resp 2^64) on a 32-bit (resp. 64-bit) machine.

2.11.21 prompt (default `"?  "`): a string that will be printed as prompt. Note that most usual escape sequences are available there: `\e` for Esc, `\n` for Newline, ..., `\\` for `\`. Time expansion is performed.

This string is sent through the library function `strftime` (on a Unix system, you can try `man strftime` at your shell prompt). This means that `%` constructs have a special meaning, usually related to the time and date. For instance, `%H` = hour (24-hour clock) and `%M` = minute [00,59] (use `%%` to get a real `%`).

If you use `readline`, escape sequences in your prompt will result in display bugs. If you have a relatively recent `readline` (see the comment at the end of Section 2.11.1), you can brace them with special sequences (`\[` and `\]`), and you will be safe. If these just result in extra spaces in your prompt, then you'll have to get a more recent `readline`. See the file `misc/gprc.dft` for an example.

EMACS: **Caution**: Emacs needs to know about the prompt pattern to separate your input from previous `gp` results, without ambiguity. It is not a trivial problem to adapt automatically this regular expression to an arbitrary prompt (which can be self-modifying!). Thus, in this version 2.3.5, Emacs relies on the prompt being the default one. So, do not tamper with the `prompt` variable *unless* you modify it simultaneously in your `.emacs` file (see `emacs/pariemacs.txt` and `misc/gprc.dft` for examples).

2.11.22 prompt_cont (default `""`): a string that will be printed to prompt for continuation lines (e.g. in between braces, or after a line-terminating backslash). Everything that applies to `prompt` applies to `prompt_cont` as well.

2.11.23 psfile (default `"pari.ps"`): name of the default file where `gp` is to dump its PostScript drawings (these are appended, so that no previous data are lost). Environment and time expansion are performed.

2.11.24 readline (default `1`): switches readline line-editing facilities on and off. This may be useful if you are running `gp` in a Sun `cmdtool`, which interacts badly with readline. Of course, until readline is switched on again, advanced editing features like automatic completion and editing history are not available.

2.11.25 realprecision (default `28` and `38` on 32-bit and 64-bit machines respectively): the number of significant digits and, at the same time, the number of printed digits of real numbers (see `\p`). Note that PARI internal precision works on a word basis (32 or 64 bits), hence may not coincide with the number of decimal digits you input. For instance to get 2 decimal digits you need one word of precision which, on a 32-bit machine, actually gives you 9 digits ($9 < \log_{10}(2^{32}) < 10$):

```
? default(realprecision, 2)
      realprecision = 9 significant digits (2 digits displayed)
```

2.11.26 secure (default `0`): this is a toggle which can be either 1 (on) or 0 (off). If on, the `system` and `extern` command are disabled. These two commands are potentially dangerous when you execute foreign scripts since they let `gp` execute arbitrary UNIX commands. `gp` will ask for confirmation before letting you (or a script) unset this toggle.

2.11.27 seriesprecision (default `16`): number of significant terms when converting a polynomial or rational function to a power series (see `\ps`).

2.11.28 simplify (default `1`): this is a toggle which can be either 1 (on) or 0 (off). When the PARI library computes something, the type of the result is not always the simplest possible. The only type conversions which the PARI library does automatically are rational numbers to integers (when they are of type `t_FRAC` and equal to integers), and similarly rational functions to polynomials (when they are of type `t_RFRAC` and equal to polynomials). This feature is useful in many cases, and saves time, but can be annoying at times. Hence you can disable this and, whenever you feel like it, use the function `simplify` (see Chapter 3) which allows you to simplify objects to the simplest possible types recursively (see `\y`).

2.11.29 strictmatch (default `1`): this is a toggle which can be either 1 (on) or 0 (off). If on, unused characters after a sequence has been processed will produce an error. Otherwise just a warning is printed. This can be useful when you're not sure how many parentheses you have to close after complicated nested loops.

2.11.30 TeXstyle (default `0`): the bits of this default allow `gp` to use less rigid TeX formatting commands in the logfile. This default is only taken into account when `log` = 3. The bits of `TeXstyle` have the following meaning

2: insert `\right` / `\left` pairs where appropriate.

4: insert discretionary breaks in polynomials, to enhance the probability of a good line break.

2.11.31 timer (default `0`): this is a toggle which can be either 1 (on) or 0 (off). If on, every instruction sequence (anything ended by a newline in your input) is timed, to some accuracy depending on the hardware and operating system. The time measured is the user CPU time, *not* including the time for printing the results (see `#` and `##`).

2.12 Simple metacommands.

Simple metacommands are meant as shortcuts and should not be used in GP scripts (see Section 3.11). Beware that these, as all of `gp` input, are *case sensitive*. For example, `\Q` is not identical to `\q`. In the following list, braces are used to denote optional arguments, with their default values when applicable, e.g. $\{n = 0\}$ means that if n is not there, it is assumed to be 0. Whitespace (or spaces) between the metacommand and its arguments and within arguments is optional. (This can cause problems only with `\w`, when you insist on having a filename whose first character is a digit, and with `\r` or `\w`, if the filename itself contains a space. In such cases, just use the underlying `read` or `write` function; see Section 3.11.2.30).

2.12.1 `?` {*command*}: `gp` on-line help interface. If you type `?`n where n is a number from 1 to 11, you will get the list of functions in Section 3.n of the manual (the list of sections being obtained by simply typing `?`).

These names are in general not informative enough. More details can be obtained by typing `?`*function*, which gives a short explanation of the function's calling convention and effects. Of course, to have complete information, read Chapter 3 of this manual (the source code is at your disposal as well, though a trifle less readable).

If the line before the copyright message indicates that extended help is available (this means `perl` is present on your system and the PARI distribution was correctly installed), you can add more `?` signs for extended functionalities:

`??` *keyword* yields the functions description as it stands in this manual, usually in Chapter 2 or 3. If you're not satisfied with the default chapter chosen, you can impose a given chapter by ending the keyword with `@` followed by the chapter number, e.g. `?? Hello@2` will look in Chapter 2 for section heading `Hello` (which doesn't exist, by the way).

All operators (e.g. `+`, `&&`, etc.) are accepted by this extended help, as well as a few other keywords describing key `gp` concepts, e.g. `readline` (the line editor), `integer`, `nf` ("number field" as used in most algebraic number theory computations), `ell` (elliptic curves), etc.

In case of conflicts between function and default names (e.g `log`, `simplify`), the function has higher priority. To get the default help, use

```
?? default(log)
?? default(simplify)
```

`???` *pattern* produces a list of sections in Chapter 3 of the manual related to your query. As before, if *pattern* ends by `@` followed by a chapter number, that chapter is searched instead; you also have the option to append a simple `@` (without a chapter number) to browse through the whole manual.

If your query contains dangerous characters (e.g `?` or blanks) it is advisable to enclose it within double quotes, as for GP strings (e.g `??? "elliptic curve"`).

Note that extended help is much more powerful than the short help, since it knows about operators as well: you can type `?? *` or `?? &&`, whereas a single `?` would just yield a not too helpful

```
*** unknown identifier.
```

message. Also, you can ask for extended help on section number n in Chapter 3, just by typing `??` n (where `?`n would yield merely a list of functions). Finally, a few key concepts in `gp` are documented in this way: metacommands (e.g `?? "??"`), defaults (e.g `?? psfile`) and type names (e.g `t_INT` or `integer`), as well as various miscellaneous keywords such as `edit` (short summary of line editor commands), `operator`, `member`, `"user defined"`, `nf`, `ell`, ...

Last but not least: `??` without argument will open a `dvi` previewer (`xdvi` by default, `$GPXDVI` if it is defined in your environment) containing the full user's manual. `??tutorial` and `??refcard` do the same with the tutorial and reference card respectively.

Technical note: these functionalities are provided by an external `perl` script that you are free to use outside any `gp` session (and modify to your liking, if you are perl-knowledgeable). It is called `gphelp`, lies in the `doc` subdirectory of your distribution (just make sure you run `Configure` first, see Appendix A) and is really two programs in one. The one which is used from within `gp` is `gphelp` which runs TEX on a selected part of this manual, then opens a previewer. `gphelp -detex` is a text mode equivalent, which looks often nicer especially on a colour-capable terminal (see `misc/gprc.dft` for examples). The default `help` selects which help program will be used from within `gp`. You are welcome to improve this help script, or write new ones (and we would like to know about it so that we may include them in future distributions). By the way, outside of `gp` you can give more than one keyword as argument to `gphelp`.

2.12.2 `/*...*/`: comment. Everything between the stars is ignored by `gp`. These comments can span any number of lines.

2.12.3 `\\`: one-line comment. The rest of the line is ignored by `gp`.

2.12.4 `\a` {n}: prints the object number n (%n) in raw format. If the number n is omitted, print the latest computed object (%).

2.12.5 `\b` {n}: Same as `\a`, in prettyprint (i.e. beautified) format.

2.12.6 `\c`: prints the list of all available hardcoded functions under `gp`, not including operators written as special symbols (see Section 2.4). More information can be obtained using the `?` meta-command (see above). For user-defined functions / member functions, see `\u` and `\um`.

2.12.7 `\d`: prints the defaults as described in the previous section (shortcut for `default()`, see Section 3.11.2.4).

2.12.8 `\e` {n}: switches the `echo` mode on (1) or off (0). If n is explicitly given, set echo to n.

2.12.9 `\g` {n}: sets the debugging level `debug` to the non-negative integer n.

2.12.10 `\gf` {n}: sets the file usage debugging level `debugfiles` to the non-negative integer n.

2.12.11 `\gm` {n}: sets the memory debugging level `debugmem` to the non-negative integer n.

2.12.12 `\h` {m-n}: outputs some debugging info about the hashtable. If the argument is a number n, outputs the contents of cell n. Ranges can be given in the form m-n (from cell m to cell n, $ = last cell). If a function name is given instead of a number or range, outputs info on the internal structure of the hash cell this function occupies (a `struct entree` in C). If the range is reduced to a dash ('-'), outputs statistics about hash cell usage.

2.12.13 `\l` {*logfile*}: switches `log` mode on and off. If a *logfile* argument is given, change the default logfile name to *logfile* and switch log mode on.

2.12.14 `\m`: as `\a`, but using prettymatrix format.

2.12.15 `\o` {n}: sets `output` mode to n (0: raw, 1: prettymatrix, 2: prettyprint, 3: external prettyprint).

2.12.16 `\p` {n}: sets `realprecision` to n decimal digits. Prints its current value if n is omitted.

2.12.17 `\ps` {n}: sets `seriesprecision` to n significant terms. Prints its current value if n is omitted.

2.12.18 `\q`: quits the `gp` session and returns to the system. Shortcut for the function `quit` (see Section 3.11.2.20).

2.12.19 `\r` {*filename*}: reads into `gp` all the commands contained in the named file as if they had been typed from the keyboard, one line after the other. Can be used in combination with the `\w` command (see below). Related but not equivalent to the function `read` (see Section 3.11.2.21); in particular, if the file contains more than one line of input, there will be one history entry for each of them, whereas `read` would only record the last one. If *filename* is omitted, re-read the previously used input file (fails if no file has ever been successfully read in the current session). If a `gp binary file` (see Section 3.11.2.32) is read using this command, it is silently loaded, without cluttering the history.

Assuming `gp` figures how to decompress files on your machine, this command accepts compressed files in `compressed` (`.Z`) or `gzipped` (`.gz` or `.z`) format. They will be uncompressed on the fly as `gp` reads them, without changing the files themselves.

2.12.20 `\s`: prints the state of the PARI *stack* and *heap*. This is used primarily as a debugging device for PARI.

2.12.21 `\t`: prints the internal longword format of all the PARI types. The detailed bit or byte format of the initial codeword(s) is explained in Chapter 4, but its knowledge is not necessary for a `gp` user.

2.12.22 `\u`: prints the definitions of all user-defined functions.

2.12.23 `\um`: prints the definitions of all user-defined member functions.

2.12.24 `\v`: prints the version number and implementation architecture (680x0, Sparc, Alpha, other) of the `gp` executable you are using. In library mode, you can use instead the two character strings `PARIVERSION` and `PARIINFO`, which correspond to the first two lines printed by `gp` just before the Copyright message.

2.12.25 `\w` {n} {*filename*}: writes the object number n (%n) into the named file, in raw format. If the number n is omitted, writes the latest computed object (%). If *filename* is omitted, appends to `logfile` (the GP function `write` is a trifle more powerful, as you can have arbitrary filenames).

2.12.26 `\x`: prints the complete tree with addresses and contents (in hexadecimal) of the internal representation of the latest computed object in `gp`. As for `\s`, this is used primarily as a debugging device for PARI, and the format should be self-explanatory (a $*$ before an object – typically a modulus – means the corresponding component is out of stack). However, used on a PARI integer, it can be used as a decimal→hexadecimal converter.

2.12.27 `\y` {n}: switches `simplify` on (1) or off (0). If n is explicitly given, set simplify to n.

2.12.28 `#`: switches the `timer` on or off.

2.12.29 `##`: prints the time taken by the latest computation. Useful when you forgot to turn on the `timer`.

2.13 The preferences file.

This file, called `gprc` in the sequel, is used to modify or extend `gp` default behaviour, in all `gp` sessions: e.g customize `default` values or load common user functions and aliases. `gp` opens the `gprc` file and processes the commands in there, *before* doing anything else, e.g. creating the PARI stack. If the file does not exist or cannot be read, `gp` will proceed to the initialization phase at once, eventually emitting a prompt. If any explicit command line switches are given, they override the values read from the preferences file.

2.13.1 Syntax. The syntax in the `gprc` file (and valid in this file only) is simple-minded, but should be sufficient for most purposes. The file is read line by line; as usual, white space is ignored unless surrounded by quotes and the standard multiline constructions using braces, `\`, or `=` are available (multiline comments between `/* ... */` are also recognized).

2.13.1.1 Preprocessor: Two types of lines are first dealt with by a preprocessor:

• comments are removed. This applies to all text surrounded by `/* ... */` as well as to everything following `\\` on a given line.

• lines starting with `#if` *boolean* are treated as comments if *boolean* evaluates to `false`, and read normally otherwise. The condition can be negated using either `#if not` (or `#if !`). If the rest of the current line is empty, the test applies to the next line (same behaviour as `=` under `gp`). Only three tests can be performed:

`EMACS`: `true` if `gp` is running in an Emacs or TeXmacs shell (see Section 2.14).

`READL`: `true` if `gp` is compiled with `readline` support (see Section 2.15.1).

`VERSION` *op number*: where *op* is in the set $\{>, <, <=, >=\}$, and *number* is a PARI version number of the form *Major.Minor.patch*, where the last two components can be omitted (i.e. 1 is understood as versio 1.0.0). This is `true` if `gp`'s version number satisfies the required inequality.

2.13.1.2 Commands: After the preprocessing the remaining lines are executed as sequence of expressions (as usual, separated by `;` if necessary). Only two kinds of expressions are recognized:

• *default* `=` *value*, where *default* is one of the available defaults (see Section 2.11), which will be set to *value* on actual startup. Don't forget the quotes around strings (e.g. for `prompt` or `help`).

• `read "`*some_GP_file*`"` where *some_GP_file* is a regular GP script this time, which will be read just before `gp` prompts you for commands, but after initializing the defaults. In particular, file input is delayed until the `gprc` has been fully loaded. This is the right place to input files containing `alias` commands, or your favorite macros.

For instance you could set your prompt in the following portable way:

```
\\ self modifying prompt looking like (18:03) gp >
prompt   = "(%H:%M) \e[1mgp\e[m > "

\\ readline wants non-printing characters to be braced between ^A/^B pairs
#if READL prompt = "(%H:%M) ^A\e[1m^Bgp^A\e[m^B > "

\\ escape sequences not supported under emacs
#if EMACS prompt = "(%H:%M) gp > "
```

Note that any of the last two lines could be broken in the following way

```
#if EMACS
```

```
  prompt = "(%H:%M) gp > "
```

since the preprocessor directive applies to the next line if the current one is empty.

A sample `gprc` file called `misc/gprc.dft` is provided in the standard distribution. It is a good idea to have a look at it and customize it to your needs. Since this file does not use multiline constructs, here is one (note the terminating `;` to separate the expressions):

```
#if VERSION > 2.2.3
{
  read "my_scripts";     \\ syntax errors in older versions
  new_galois_format = 1; \\ default introduced in 2.2.4
}
#if ! EMACS
{
  colors = "9, 5, no, no, 4, 1, 2";
  help   = "gphelp -detex -ch 4 -cb 0 -cu 2";
}
```

2.13.2 Where is it? When `gp` is started, it looks for a customization file, or `gprc` in the following places (in this order, only the first one found will be loaded):

• On the Macintosh (only), `gp` looks in the directory which contains the `gp` executable itself for a file called `gprc`.

• `gp` checks whether the environment variable `GPRC` is set. Under DOS, you can set it in `AUTOEXEC.BAT`. On Unix, this can be done with something like:

`GPRC=/my/dir/anyname; export GPRC` in `sh` syntax (for instance in your `.profile`),
`setenv GPRC /my/dir/anyname` in `csh` syntax (in your `.login` or `.cshrc` file).

If so, the file named by `$GPRC` is the `gprc`.

• If `GPRC` is not set, and if the environment variable `HOME` is defined, `gp` then tries

`$HOME/.gprc` on a Unix system

`$HOME\_gprc` on a DOS, OS/2, or Windows system.

• If `HOME` also leaves us clueless, we try

`~/.gprc` on a Unix system (where as usual `~` stands for your home directory), or

`\_gprc` on a DOS, OS/2, or Windows system.

• Finally, if no gprc was found among the user files mentioned above we look for `/etc/gprc` (`\etc\gprc`) for a system-wide gprc file (you will need root privileges to set up such a file yourself).

Note that on Unix systems, the `gprc`'s default name starts with a '.' and thus is hidden to regular `ls` commands; you need to type `ls -a` to list it.

2.14 Using GNU Emacs.

If GNU Emacs is installed on your machine, it is possible to use `gp` as a subprocess in Emacs. To use this, you should include in your `.emacs` file the following lines:

```
(autoload 'gp-mode "pari" nil t)
(autoload 'gp-script-mode "pari" nil t)
(autoload 'gp "pari" nil t)
(autoload 'gpman "pari" nil t)
(setq auto-mode-alist
  (cons '("\\.gp$" . gp-script-mode) auto-mode-alist))
```

which autoloads functions from `pari.el`. See also `pariemacs.txt`. These files are included in the PARI distribution and are installed at the same time as `gp`.

Once this is done, under GNU Emacs if you type `M-x gp` (where as usual `M` is the `Meta` key, i.e. Escape, or on SUN keyboards, the Left key), a special shell will be started, which in particular launches `gp` with the default stack size, prime limit and input buffer size. If you type instead `C-u M-x gp`, you will be asked for the name of the `gp` executable, the stack size and the prime limit before the execution of `gp` begins. If for any of these you simply type return, the default value will be used. On UNIX machines it will be the place you told `Configure` (usually `/usr/local/bin/gp`) for the executable, `10M` for the stack and `500k` for the prime limit.

You can then work as usual under `gp`, but with two notable advantages (which don't really matter if readline is available to you, see below). First and foremost, you have at your disposal all the facilities of a text editor like Emacs, in particular for correcting or copying blocks. Second, you can have an on-line help which is much more complete than what you obtain by typing `?name`. This is done by typing `M-?`. In the minibuffer, Emacs asks what function you want to describe, and after your reply you obtain the description which is in the users manual, including the description of functions (such as `\`, `%`) which use special symbols.

This help system can also be menu-driven, by using the command `M-\c` which opens a help menu window which enables you to choose the category of commands for which you want an explanation.

Nevertheless, if extended help is available on your system (see Section 2.12.1), you should use it instead of the above, since it's nicer (it ran through TEX) and understands many more keywords.

Finally you can use command completion in the following way. After the prompt, type the first few letters of the command, then `<TAB>` where `<TAB>` is the TAB key. If there exists a unique command starting with the letters you have typed, the command name will be completed. If not, either the list of commands starting with the letters you typed will be displayed in a separate window (which you can then kill by typing as usual `C-x 1` or by typing in more letters), or "no match found" will be displayed in the Emacs command line. If your `gp` was linked with the readline library, read the section on completion in the section below (the paragraph on online help is not relevant).

Note that if for some reason the session crashes (due to a bug in your program or in the PARI system), you will usually stay under Emacs, but the `gp` buffer will be killed. To recover it, simply type again `M-x gp` (or `C-u M-x gp`), and a new session of `gp` will be started after the old one, so you can recover what you have typed. Note that this will of course *not* work if for some reason you kill Emacs and start a new session.

You also have at your disposal a few other commands and many possible customizations (colours, prompt). Read the file `emacs/pariemacs.txt` in standard distribution for details.

2.15 Using readline.

Thanks to the initial help of Ilya Zakharevich, there is a possibility of line editing and command name completion outside of an Emacs buffer *if* you have compiled `gp` with the GNU readline library. If you do not have Emacs available, or cannot stand using it, we really advise you to make sure you get this very useful library before configuring or compiling `gp`. In fact, with `readline`, even line editing becomes *more* powerful outside an Emacs buffer!

2.15.1 A (too) short introduction to readline: The basics are as follows (read the readline user manual !), assume that `C-` stands for "the `Control` key combined with another" and the same for `M-` with the `Meta` key (generally `C-` combinations act on characters, while the `M-` ones operate on words). The `Meta` key might be called `Alt` on some keyboards, will display a black diamond on most others, and can safely be replaced by `Esc` in any case. Typing any ordinary key inserts text where the cursor stands, the arrow keys enabling you to move in the line. There are many more movement commands, which will be familiar to the Emacs user, for instance `C-a`/`C-e` will take you to the start/end of the line, `M-b`/`M-f` move the cursor backward/forward by a word, etc. Just press the `<Return>` key at any point to send your command to `gp`.

All the commands you type in are stored in a history (with multiline commands being saved as single concatenated lines). The Up and Down arrows (or `C-p`/`C-n`) will move you through it, `M-<`/`M->` sending you to the start/end of the history. `C-r`/`C-s` will start an incremental backward/forward search. You can kill text (`C-k` kills till the end of line, `M-d` to the end of current word) which you can then yank back using the `C-y` key (`M-y` will rotate the kill-ring). `C-_` will undo your last changes incrementally (`M-r` undoes all changes made to the current line). `C-t` and `M-t` will transpose the character (word) preceding the cursor and the one under the cursor.

Keeping the `M-` key down while you enter an integer (a minus sign meaning reverse behaviour) gives an argument to your next readline command (for instance `M-- C-k` will kill text back to the start of line). If you prefer Vi–style editing, `M-C-j` will toggle you to Vi mode.

Of course you can change all these default bindings. For that you need to create a file named `.inputrc` in your home directory. For instance (notice the embedding conditional in case you would want specific bindings for `gp`):

```
$if Pari-GP
  set show-all-if-ambiguous
  "\C-h": backward-delete-char
  "\e\C-h": backward-kill-word
  "\C-xd": dump-functions
  (: "\C-v()\C-b"       # can be annoying when copy-pasting !
  [: "\C-v[]\C-b"
$endif
```

`C-x C-r` will re-read this init file, incorporating any changes made to it during the current session.

Note: By default, `(` and `[` are bound to the function `pari-matched-insert` which, if "electric parentheses" are enabled (default: off) will automatically insert the matching closure (respectively `)` and `]`). This behaviour can be toggled on and off by giving the numeric argument -2 to `(` (`M--2(`), which is useful if you want, e.g to copy-paste some text into the calculator. If you do not want a toggle, you can use `M--0` / `M--1` to specifically switch it on or off).

Note: In some versions of readline (2.1 for instance), the `Alt` or `Meta` key can give funny results (output 8-bit accented characters for instance). If you do not want to fall back to the `Esc` combination, put the following two lines in your `.inputrc`:

```
set convert-meta on
set output-meta off
```

2.15.2 Command completion and online help. As in the Emacs shell, `<TAB>` will complete words for you. But, under readline, this mechanism will be context-dependent: `gp` will strive to only give you meaningful completions in a given context (it will fail sometimes, but only under rare and restricted conditions).

For instance, shortly after a `~`, we expect a user name, then a path to some file. Directly after `default(` has been typed, we would expect one of the `default` keywords. After `whatnow(` , we expect the name of an old function, which may well have disappeared from this version. After a '.', we expect a member keyword. And generally of course, we expect any GP symbol which may be found in the hashing lists: functions (both yours and GP's), and variables.

If, at any time, only one completion is meaningful, `gp` will provide it together with

• an ending comma if we are completing a default,

• a pair of parentheses if we are completing a function name. In that case hitting `<TAB>` again will provide the argument list as given by the online help*.

Otherwise, hitting `<TAB>` once more will give you the list of possible completions. Just experiment with this mechanism as often as possible, you will probably find it very convenient. For instance, you can obtain `default(seriesprecision,10)`, just by hitting `def<TAB>se<TAB>10`, which saves 18 keystrokes (out of 27).

Hitting `M-h` will give you the usual short online help concerning the word directly beneath the cursor, `M-H` will yield the extended help corresponding to the `help` default program (usually opens a dvi previewer, or runs a primitive tex-to-ASCII program). None of these disturb the line you were editing.

* recall that you can always undo the effect of the preceding keys by hitting `C-_`

Chapter 3:
Functions and Operations Available in PARI and GP

The functions and operators available in PARI and in the GP/PARI calculator are numerous and everexpanding. Here is a description of the ones available in version 2.3.5. It should be noted that many of these functions accept quite different types as arguments, but others are more restricted. The list of acceptable types will be given for each function or class of functions. Except when stated otherwise, it is understood that a function or operation which should make natural sense is legal. In this chapter, we will describe the functions according to a rough classification. The general entry looks something like:

foo$(x, \{flag = 0\})$: short description.

The library syntax is **foo**$(x, flag)$.

This means that the GP function `foo` has one mandatory argument x, and an optional one, *flag*, whose default value is 0. (The {} should not be typed, it is just a convenient notation we will use throughout to denote optional arguments.) That is, you can type `foo(x,2)`, or `foo(x)`, which is then understood to mean `foo(x,0)`. As well, a comma or closing parenthesis, where an optional argument should have been, signals to GP it should use the default. Thus, the syntax `foo(x,)` is also accepted as a synonym for our last expression. When a function has more than one optional argument, the argument list is filled with user supplied values, in order. When none are left, the defaults are used instead. Thus, assuming that `foo`'s prototype had been

$$\mathbf{foo}(\{x = 1\}, \{y = 2\}, \{z = 3\}),$$

typing in `foo(6,4)` would give you `foo(6,4,3)`. In the rare case when you want to set some far away argument, and leave the defaults in between as they stand, you can use the "empty arg" trick alluded to above: `foo(6,,1)` would yield `foo(6,2,1)`. By the way, `foo()` by itself yields `foo(1,2,3)` as was to be expected.

In this rather special case of a function having no mandatory argument, you can even omit the (): a standalone `foo` would be enough (though we do not recommend it for your scripts, for the sake of clarity). In defining GP syntax, we strove to put optional arguments at the end of the argument list (of course, since they would not make sense otherwise), and in order of decreasing usefulness so that, most of the time, you will be able to ignore them.

Finally, an optional argument (between braces) followed by a star, like $\{x\}*$, means that any number of such arguments (possibly none) can be given. This is in particular used by the various `print` routines.

Flags. A *flag* is an argument which, rather than conveying actual information to the routine, intructs it to change its default behaviour, e.g. return more or less information. All such flags are optional, and will be called *flag* in the function descriptions to follow. There are two different kind of flags

• generic: all valid values for the flag are individually described ("If *flag* is equal to 1, then...").

• binary: use customary binary notation as a compact way to represent many toggles with just one integer. Let $(p_0, \ldots, p_n)$ be a list of switches (i.e. of properties which take either the value 0 or 1), the number $2^3 + 2^5 = 40$ means that p_3 and p_5 are set (that is, set to 1), and none of the others are (that is, they are set to 0). This is announced as "The binary digits of *flag* mean 1: p_0, 2: p_1, 4: p_2", and so on, using the available consecutive powers of 2.

Mnemonics for flags. Numeric flags as mentionned above are obscure, error-prone, and quite rigid: should the authors want to adopt a new flag numbering scheme (for instance when noticing flags with the same meaning but different numeric values across a set of routines), it would break backward compatibility. The only advantage of explicit numeric values is that they are fast to type, so their use is only advised when using the calculator `gp`.

As an alternative, one can replace a numeric flag by a character string containing symbolic identifiers. For a generic flag, the mnemonic corresponding to the numeric identifier is given after it as in

```
fun(x, {flag = 0} ):

  If flag is equal to 1 = AGM, use an agm formula\dots
```

which means that one can use indifferently `fun(`x`, 1)` or `fun(`x`, AGM)`.

For a binary flag, mnemonics corresponding to the various toggles are given after each of them. They can be negated by prepending `no_` to the mnemonic, or by removing such a prefix. These toggles are grouped together using any punctuation character (such as ',' or ';'). For instance (taken from description of `ploth`$(X = a, b, expr, \{flag = 0\}, \{n = 0\})$)

Binary digits of flags mean: 1 = `Parametric`, 2 = `Recursive`, ...

so that, instead of 1, one could use the mnemonic `"Parametric; no_Recursive"`, or simply `"Parametric"` since `Recursive` is unset by default (default value of *flag* is 0, i.e. everything unset).

Pointers. If a parameter in the function prototype is prefixed with a & sign, as in

foo$(x, \&e)$

it means that, besides the normal return value, the function may assign a value to e as a side effect. When passing the argument, the & sign has to be typed in explicitly. As of version 2.3.5, this *pointer* argument is optional for all documented functions, hence the & will always appear between brackets as in `Z_issquare`$(x, \{\&e\})$.

About library programming. the *library* function `foo`, as defined at the beginning of this section, is seen to have two mandatory arguments, x and *flag*: no PARI mathematical function has been implemented so as to accept a variable number of arguments, so all arguments are mandatory when programming with the library (often, variants are provided corresponding to the various flag values). When not mentioned otherwise, the result and arguments of a function are assumed implicitly to be of type `GEN`. Most other functions return an object of type `long` integer in C (see Chapter 4). The variable or parameter names *prec* and *flag* always denote `long` integers.

The `entree` type is used by the library to implement iterators (loops, sums, integrals, etc.) when a formal variable has to successively assume a number of values in a given set. When programming with the library, it is easier and much more efficient to code loops and the like directly. Hence this type is not documented, although it does appear in a few library function prototypes below. See Section 3.9 for more details.

3.1 Standard monadic or dyadic operators.

3.1.1 `+`/`-`: The expressions `+`x and `-`x refer to monadic operators (the first does nothing, the second negates x).

The library syntax is **gneg**(x) for `-`x.

3.1.2 `+`, `-`: The expression x `+` y is the sum and x `-` y is the difference of x and y. Among the prominent impossibilities are addition/subtraction between a scalar type and a vector or a matrix, between vector/matrices of incompatible sizes and between an intmod and a real number.

The library syntax is **gadd**(x, y) x `+` y, **gsub**(x, y) for x `-` y.

3.1.3 `*`: The expression x `*` y is the product of x and y. Among the prominent impossibilities are multiplication between vector/matrices of incompatible sizes, between an intmod and a real number. Note that because of vector and matrix operations, `*` is not necessarily commutative. Note also that since multiplication between two column or two row vectors is not allowed, to obtain the scalar product of two vectors of the same length, you must multiply a line vector by a column vector, if necessary by transposing one of the vectors (using the operator `~` or the function `mattranspose`, see Section 3.8).

If x and y are binary quadratic forms, compose them. See also `qfbnucomp` and `qfbnupow`.

The library syntax is **gmul**(x, y) for x `*` y. Also available is **gsqr**(x) for x `*` x (faster of course!).

3.1.4 `/`: The expression x `/` y is the quotient of x and y. In addition to the impossibilities for multiplication, note that if the divisor is a matrix, it must be an invertible square matrix, and in that case the result is $x*y^{-1}$. Furthermore note that the result is as exact as possible: in particular, division of two integers always gives a rational number (which may be an integer if the quotient is exact) and *not* the Euclidean quotient (see x `\` y for that), and similarly the quotient of two polynomials is a rational function in general. To obtain the approximate real value of the quotient of two integers, add `0.` to the result; to obtain the approximate p-adic value of the quotient of two integers, add `O(p^k)` to the result; finally, to obtain the Taylor series expansion of the quotient of two polynomials, add `O(X^k)` to the result or use the `taylor` function (see Section 3.7.34).

The library syntax is **gdiv**(x, y) for x `/` y.

3.1.5 `\`: The expression x `\` y is the Euclidean quotient of x and y. If y is a real scalar, this is defined as `floor(`x/y`)` if $y > 0$, and `ceil(`x/y`)` if $y < 0$ and the division is not exact. Hence the remainder x `-` `(`x`\`y`)*`y is in $[0, |y|[$.

Note that when y is an integer and x a polynomial, y is first promoted to a polynomial of degree 0. When x is a vector or matrix, the operator is applied componentwise.

The library syntax is **gdivent**(x, y) for x `\` y.

3.1.6 `\/`: The expression x `\/` y evaluates to the rounded Euclidean quotient of x and y. This is the same as x `\` y except for scalar division: the quotient is such that the corresponding remainder is smallest in absolute value and in case of a tie the quotient closest to $+\infty$ is chosen (hence the remainder would belong to $]-|y|/2, |y|/2]$).

When x is a vector or matrix, the operator is applied componentwise.

The library syntax is **gdivround**(x, y) for x `\/` y.

3.1.7 %: The expression x `%` y evaluates to the modular Euclidean remainder of x and y, which we now define. If y is an integer, this is the smallest non-negative integer congruent to x modulo y. If y is a polynomial, this is the polynomial of smallest degree congruent to x modulo y. When y is a non-integral real number, x`%`y is defined as x `-` `(`x`\`y`)*`y. This coincides with the definition for y integer if and only if x is an integer, but still belongs to $[0, |y|[$. For instance:

```
? (1/2) % 3
%1 = 2
? 0.5 % 3
  ***   forbidden division t_REAL % t_INT.
? (1/2) % 3.0
%2 = 1/2
```

Note that when y is an integer and x a polynomial, y is first promoted to a polynomial of degree 0. When x is a vector or matrix, the operator is applied componentwise.

The library syntax is **gmod**(x, y) for x `%` y.

3.1.8 divrem$(x, y, \{v\})$: creates a column vector with two components, the first being the Euclidean quotient (x `\` y), the second the Euclidean remainder (x `-` `(`x`\`y`)*`y), of the division of x by y. This avoids the need to do two divisions if one needs both the quotient and the remainder. If v is present, and x, y are multivariate polynomials, divide with respect to the variable v.

Beware that `divrem(`x`,`y`)[2]` is in general not the same as x `%` y; there is no operator to obtain it in GP:

```
? divrem(1/2, 3)[2]
%1 = 1/2
? (1/2) % 3
%2 = 2
? divrem(Mod(2,9), 3)[2]
  ***   forbidden division t_INTMOD \ t_INT.
? Mod(2,9) % 6
%3 = Mod(2,3)
```

The library syntax is **divrem**(x, y, v),where v is a `long`. Also available as **gdiventres**(x, y) when v is not needed.

3.1.9 ^: The expression x`^`n is powering. If the exponent is an integer, then exact operations are performed using binary (left-shift) powering techniques. In particular, in this case x cannot be a vector or matrix unless it is a square matrix (invertible if the exponent is negative). If x is a p-adic number, its precision will increase if $v_p(n) > 0$. Powering a binary quadratic form (types `t_QFI` and `t_QFR`) returns a reduced representative of the class, provided the input is reduced. In particular, x`^1` is identical to x.

PARI is able to rewrite the multiplication $x * x$ of two *identical* objects as x^2, or `sqr`(x). Here, identical means the operands are two different labels referencing the same chunk of memory; no equality test is performed. This is no longer true when more than two arguments are involved.

If the exponent is not of type integer, this is treated as a transcendental function (see Section 3.3), and in particular has the effect of componentwise powering on vector or matrices.

As an exception, if the exponent is a rational number p/q and x an integer modulo a prime or a p-adic number, return a solution y of $y^q = x^p$ if it exists. Currently, q must not have large prime factors. Beware that

```
? Mod(7,19)^(1/2)
%1 = Mod(11, 19) /* is any square root */
? sqrt(Mod(7,19))
%2 = Mod(8, 19)  /* is the smallest square root */
? Mod(7,19)^(3/5)
%3 = Mod(1, 19)
? %3^(5/3)
%4 = Mod(1, 19)  /* Mod(7,19) is just another cubic root */
```

If the exponent is a negative integer, an inverse must be computed. For non-invertible `t_INTMOD`, this will fail and implicitly exhibit a non trivial factor of the modulus:

```
? Mod(4,6)^(-1)
  ***   impossible inverse modulo: Mod(2, 6).
```

(Here, a factor 2 is obtained directly. In general, take the gcd of the representative and the modulus.) This is most useful when performing complicated operations modulo an integer N whose factorization is unknown. Either the computation succeeds and all is well, or a factor d is discovered and the computation may be restarted modulo d or N/d.

For non-invertible `t_POLMOD`, this will fail without exhibiting a factor.

```
? Mod(x^2, x^3-x)^(-1)
  ***   non-invertible polynomial in RgXQ_inv.
? a = Mod(3,4)*y^3 + Mod(1,4); b = y^6+y^5+y^4+y^3+y^2+y+1;
? Mod(a, b)^(-1);
  ***   non-invertible polynomial in RgXQ_inv.
```

In fact the latter polynomial is invertible, but the algorithm used (subresultant) assumes the base ring is a domain. If it is not the case, as here for $\mathbf{Z}/4\mathbf{Z}$, a result will be correct but chances are an error will occur first. In this specific case, one should work with 2-adics. In general, one can try the following approach

```
? inversemod(a, b) =
{ local(m);
  m = polsylvestermatrix(polrecip(a), polrecip(b));
  m = matinverseimage(m, matid(#m)[,1]);
  Polrev( vecextract(m, Str("..", poldegree(b))), variable(b) )
}
? inversemod(a,b)
%2 = Mod(2,4)*y^5 + Mod(3,4)*y^3 + Mod(1,4)*y^2 + Mod(3,4)*y + Mod(2,4)
```

This is not guaranteed to work either since it must invert pivots. See Section 3.8.

The library syntax is **gpow**$(x, n, prec)$ for `x^n`.

3.1.10 bittest(x, n): outputs the n^{th} bit of x starting from the right (i.e. the coefficient of 2^n in the binary expansion of x). The result is 0 or 1. To extract several bits at once as a vector, pass a vector for n.

See Section 3.2.17 for the behaviour at negative arguments.

The library syntax is **bittest**(x, n), where n and the result are `long`s.

3.1.11 shift(x, n) or x `<<` n $(= x$ `>>` $(-n))$: shifts x componentwise left by n bits if $n \geq 0$ and right by $|n|$ bits if $n < 0$. A left shift by n corresponds to multiplication by 2^n. A right shift of an integer x by $|n|$ corresponds to a Euclidean division of x by $2^{|n|}$ with a remainder of the same sign as x, hence is not the same (in general) as $x\backslash 2^n$.

The library syntax is **gshift**(x, n) where n is a `long`.

3.1.12 shiftmul(x, n): multiplies x by 2^n. The difference with `shift` is that when $n < 0$, ordinary division takes place, hence for example if x is an integer the result may be a fraction, while for shifts Euclidean division takes place when $n < 0$ hence if x is an integer the result is still an integer.

The library syntax is **gmul2n**(x, n) where n is a `long`.

3.1.13 Comparison and boolean operators. The six standard comparison operators `<=`, `<`, `>=`, `>`, `==`, `!=` are available in GP, and in library mode under the names `gle`, `glt`, `gge`, `ggt`, `geq`, `gne` respectively. The library syntax is $co(x, y)$, where co is the comparison operator. The result is 1 (as a `GEN`) if the comparison is true, 0 (as a `GEN`) if it is false. For the purpose of comparison, `t_STR` objects are strictly larger than any other non-string type; two `t_STR` objects are compared using the standard lexicographic order.

The standard boolean functions `||` (inclusive or), `&&` (and) and `!` (not) are also available, and the library syntax is **gor**(x, y), **gand**(x, y) and **gnot**(x) respectively.

In library mode, it is in fact usually preferable to use the two basic functions which are **gcmp**(x, y) which gives the sign (1, 0, or -1) of $x - y$, where x and y must be in **R**, and **gequal**(x, y) which can be applied to any two PARI objects x and y and gives 1 (i.e. true) if they are equal (but not necessarily identical), 0 (i.e. false) otherwise. Comparisons to special constants are implemented and should be used instead of `gequal`: **gcmp0**(x) $(x == 0\ ?)$, **gcmp1**(x) $(x == 1\ ?)$, and **gcmp_1**(x) $(x == -1\ ?)$.

Note that `gcmp0`(x) tests whether x is equal to zero, even if x is not an exact object. To test whether x is an exact object which is equal to zero, one must use **isexactzero**(x).

Also note that the `gcmp` and `gequal` functions return a C-integer, and *not* a `GEN` like `gle` etc.

GP accepts the following synonyms for some of the above functions: since we thought it might easily lead to confusion, we don't use the customary C operators for bitwise `and` or bitwise `or` (use `bitand` or `bitor`), hence `|` and `&` are accepted as synonyms of `||` and `&&` respectively. Also, `<>` is accepted as a synonym for `!=`. On the other hand, `=` is definitely *not* a synonym for `==` since it is the assignment statement.

3.1.14 lex(x, y): gives the result of a lexicographic comparison between x and y (as -1, 0 or 1). This is to be interpreted in quite a wide sense: It is admissible to compare objects of different types (scalars, vectors, matrices), provided the scalars can be compared, as well as vectors/matrices of different lengths. The comparison is recursive.

In case all components are equal up to the smallest length of the operands, the more complex is considered to be larger. More precisely, the longest is the largest; when lengths are equal, we have matrix > vector > scalar. For example:

```
? lex([1,3], [1,2,5])
%1 = 1
? lex([1,3], [1,3,-1])
%2 = -1
? lex([1], [[1]])
%3 = -1
? lex([1], [1]~)
%4 = 0
```

The library syntax is **lexcmp**(x, y).

3.1.15 sign(x): sign (0, 1 or -1) of x, which must be of type integer, real or fraction.

The library syntax is **gsigne**(x). The result is a `long`.

3.1.16 max(x, y) and **min**(x, y): creates the maximum and minimum of x and y when they can be compared.

The library syntax is **gmax**(x, y) and **gmin**(x, y).

3.1.17 vecmax(x): if x is a vector or a matrix, returns the maximum of the elements of x, otherwise returns a copy of x. Error if x is empty.

The library syntax is **vecmax**(x).

3.1.18 vecmin(x): if x is a vector or a matrix, returns the minimum of the elements of x, otherwise returns a copy of x. Error if x is empty.

The library syntax is **vecmin**(x).

3.2 Conversions and similar elementary functions or commands.

Many of the conversion functions are rounding or truncating operations. In this case, if the argument is a rational function, the result is the Euclidean quotient of the numerator by the denominator, and if the argument is a vector or a matrix, the operation is done componentwise. This will not be restated for every function.

3.2.1 Col($x = []$): transforms the object x into a column vector. The vector will be with one component only, except when x is a vector or a quadratic form (in which case the resulting vector is simply the initial object considered as a column vector), a matrix (the column of row vectors comprising the matrix is returned), a character string (a column of individual characters is returned), but more importantly when x is a polynomial or a power series. In the case of a polynomial, the coefficients of the vector start with the leading coefficient of the polynomial, while for power series only the significant coefficients are taken into account, but this time by increasing order of degree.

The library syntax is **gtocol**(x).

3.2.2 List($x = []$): transforms a (row or column) vector x into a list. The only other way to create a `t_LIST` is to use the function `listcreate`.

This is useless in library mode.

3.2.3 Mat($x = []$): transforms the object x into a matrix. If x is already a matrix, a copy of x is created. If x is not a vector or a matrix, this creates a 1×1 matrix. If x is a row (resp. column) vector, this creates a 1-row (resp. 1-column) matrix, *unless* all elements are column (resp. row) vectors of the same length, in which case the vectors are concatenated sideways and the associated big matrix is returned.

```
? Mat(x + 1)
%1 =
[x + 1]
? Vec( matid(3) )
%2 = [[1, 0, 0]~, [0, 1, 0]~, [0, 0, 1]~]
? Mat(%)
%3 =
[1 0 0]

[0 1 0]

[0 0 1]
? Col( [1,2; 3,4] )
%4 = [[1, 2], [3, 4]]~
? Mat(%)
%5 =
[1 2]

[3 4]
```

The library syntax is **gtomat**(x).

3.2.4 Mod($x, y, \{flag = 0\}$): creates the PARI object ($x \bmod y$), i.e. an intmod or a polmod. y must be an integer or a polynomial. If y is an integer, x must be an integer, a rational number, or a p-adic number compatible with the modulus y. If y is a polynomial, x must be a scalar (which is not a polmod), a polynomial, a rational function, or a power series.

This function is not the same as x `%` y, the result of which is an integer or a polynomial.

flag is obsolete and should not be used.

The library syntax is **gmodulo**(x, y).

3.2.5 Pol$(x, \{v = x\})$: transforms the object x into a polynomial with main variable v. If x is a scalar, this gives a constant polynomial. If x is a power series, the effect is identical to `truncate` (see there), i.e. it chops off the $O(X^k)$. If x is a vector, this function creates the polynomial whose coefficients are given in x, with $x[1]$ being the leading coefficient (which can be zero).

Warning: this is *not* a substitution function. It will not transform an object containing variables of higher priority than v.

```
? Pol(x + y, y)
  *** Pol: variable must have higher priority in gtopoly.
```

The library syntax is **gtopoly**(x, v), where v is a variable number.

3.2.6 Polrev$(x, \{v = x\})$: transform the object x into a polynomial with main variable v. If x is a scalar, this gives a constant polynomial. If x is a power series, the effect is identical to `truncate` (see there), i.e. it chops off the $O(X^k)$. If x is a vector, this function creates the polynomial whose coefficients are given in x, with $x[1]$ being the constant term. Note that this is the reverse of `Pol` if x is a vector, otherwise it is identical to `Pol`.

The library syntax is **gtopolyrev**(x, v), where v is a variable number.

3.2.7 Qfb$(a, b, c, \{D = 0.\})$: creates the binary quadratic form $ax^2 + bxy + cy^2$. If $b^2 - 4ac > 0$, initialize Shanks' distance function to D. Negative definite forms are not implemented, use their positive definite counterpart instead.

The library syntax is **Qfb0**$(a, b, c, D, prec)$. Also available are **qfi**(a, b, c) (when $b^2 - 4ac < 0$), and **qfr**(a, b, c, d) (when $b^2 - 4ac > 0$).

3.2.8 Ser$(x, \{v = x\})$: transforms the object x into a power series with main variable v (x by default). If x is a scalar, this gives a constant power series with precision given by the default `serieslength` (corresponding to the C global variable `precdl`). If x is a polynomial, the precision is the greatest of `precdl` and the degree of the polynomial. If x is a vector, the precision is similarly given, and the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (i.e. the reverse of the function `Pol`).

The warning given for `Pol` also applies here: this is not a substitution function.

The library syntax is **gtoser**(x, v), where v is a variable number (i.e. a C integer).

3.2.9 Set$(\{x = [\,]\})$: converts x into a set, i.e. into a row vector of character strings, with strictly increasing entries with respect to lexicographic ordering. The components of x are put in canonical form (type `t_STR`) so as to be easily sorted. To recover an ordinary `GEN` from such an element, you can apply `eval` to it.

The library syntax is **gtoset**(x).

3.2.10 Str($\{x\}*$): converts its argument list into a single character string (type `t_STR`, the empty string if x is omitted). To recover an ordinary `GEN` from a string, apply `eval` to it. The arguments of `Str` are evaluated in string context, see Section 2.8.

```
? x2 = 0; i = 2; Str(x, i)
%1 = "x2"
? eval(%)
%2 = 0
```

This function is mostly useless in library mode. Use the pair `strtoGEN`/`GENtostr` to convert between `GEN` and `char*`. The latter returns a malloced string, which should be freed after usage.

3.2.11 Strchr(x): converts x to a string, translating each integer into a character.

```
? Strchr(97)
%1 = "a"
? Vecsmall("hello world")
%2 = Vecsmall([104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100])
? Strchr(%)
%3 = "hello world"
```

3.2.12 Strexpand($\{x\}*$): converts its argument list into a single character string (type `t_STR`, the empty string if x is omitted). Then performe environment expansion, see Section 2.11. This feature can be used to read environment variable values.

```
? Strexpand("$HOME/doc")
%1 = "/home/pari/doc"
```

The individual arguments are read in string context, see Section 2.8.

3.2.13 Strtex($\{x\}*$): translates its arguments to TeX format, and concatenates the results into a single character string (type `t_STR`, the empty string if x is omitted).

The individual arguments are read in string context, see Section 2.8.

3.2.14 Vec($x = []$): transforms the object x into a row vector. The vector will be with one component only, except when x is a vector or a quadratic form (in which case the resulting vector is simply the initial object considered as a row vector), a matrix (the vector of columns comprising the matrix is return), a character string (a vector of individual characters is returned), but more importantly when x is a polynomial or a power series. In the case of a polynomial, the coefficients of the vector start with the leading coefficient of the polynomial, while for power series only the significant coefficients are taken into account, but this time by increasing order of degree.

The library syntax is **gtovec**(x).

3.2.15 Vecsmall($x = []$): transforms the object x into a row vector of type `t_VECSMALL`. This acts as `Vec`, but only on a limited set of objects (the result must be representable as a vector of small integers). In particular, polynomials and power series are forbidden. If x is a character string, a vector of individual characters in ASCII encoding is returned (`Strchr` yields back the character string).

The library syntax is **gtovecsmall**(x).

3.2.16 binary(x): outputs the vector of the binary digits of $|x|$. Here x can be an integer, a real number (in which case the result has two components, one for the integer part, one for the fractional part) or a vector/matrix.

The library syntax is **binaire**(x).

3.2.17 bitand(x, y): bitwise `and` of two integers x and y, that is the integer

$$\sum_i (x_i \texttt{ and } y_i)2^i$$

Negative numbers behave 2-adically, i.e. the result is the 2-adic limit of `bitand`(x_n, y_n), where x_n and y_n are non-negative integers tending to x and y respectively. (The result is an ordinary integer, possibly negative.)

```
? bitand(5, 3)
%1 = 1
? bitand(-5, 3)
%2 = 3
? bitand(-5, -3)
%3 = -7
```

The library syntax is **gbitand**(x, y).

3.2.18 bitneg$(x, \{n = -1\})$: bitwise negation of an integer x, truncated to n bits, that is the integer

$$\sum_{i=0}^{n-1} \texttt{not}(x_i)2^i$$

The special case $n = -1$ means no truncation: an infinite sequence of leading 1 is then represented as a negative number.

See Section 3.2.17 for the behaviour for negative arguments.

The library syntax is **gbitneg**(x).

3.2.19 bitnegimply(x, y): bitwise negated imply of two integers x and y (or `not` $(x \Rightarrow y)$), that is the integer

$$\sum (x_i \texttt{ andnot}(y_i))2^i$$

See Section 3.2.17 for the behaviour for negative arguments.

The library syntax is **gbitnegimply**(x, y).

3.2.20 bitor(x, y): bitwise (inclusive) `or` of two integers x and y, that is the integer

$$\sum (x_i \texttt{ or } y_i)2^i$$

See Section 3.2.17 for the behaviour for negative arguments.

The library syntax is **gbitor**(x, y).

3.2.21 bittest(x, n): outputs the n^{th} bit of $|x|$ starting from the right (i.e. the coefficient of 2^n in the binary expansion of x). The result is 0 or 1. To extract several bits at once as a vector, pass a vector for n.

The library syntax is **bittest**(x, n), where n and the result are `long`s.

3.2.22 bitxor(x, y): bitwise (exclusive) `or` of two integers x and y, that is the integer

$$\sum (x_i \texttt{ xor } y_i) 2^i$$

See Section 3.2.17 for the behaviour for negative arguments.

The library syntax is **gbitxor**(x, y).

3.2.23 ceil(x): ceiling of x. When x is in $\mathbf{R}$, the result is the smallest integer greater than or equal to x. Applied to a rational function, `ceil`(x) returns the euclidian quotient of the numerator by the denominator.

The library syntax is **gceil**(x).

3.2.24 centerlift$(x, \{v\})$: lifts an element $x = a \bmod n$ of $\mathbf{Z}/n\mathbf{Z}$ to a in $\mathbf{Z}$, and similarly lifts a polmod to a polynomial. This is the same as `lift` except that in the particular case of elements of $\mathbf{Z}/n\mathbf{Z}$, the lift y is such that $-n/2 < y \leq n/2$. If x is of type fraction, complex, quadratic, polynomial, power series, rational function, vector or matrix, the lift is done for each coefficient. Reals are forbidden.

The library syntax is **centerlift0**(x, v), where v is a `long` and an omitted v is coded as -1. Also available is **centerlift**(x) = `centerlift0(`x`,-1)`.

3.2.25 changevar(x, y): creates a copy of the object x where its variables are modified according to the permutation specified by the vector y. For example, assume that the variables have been introduced in the order `x`, `a`, `b`, `c`. Then, if y is the vector `[x,c,a,b]`, the variable `a` will be replaced by `c`, `b` by `a`, and `c` by `b`, `x` being unchanged. Note that the permutation must be completely specified, e.g. `[c,a,b]` would not work, since this would replace `x` by `c`, and leave `a` and `b` unchanged (as well as `c` which is the fourth variable of the initial list). In particular, the new variable names must be distinct.

The library syntax is **changevar**(x, y).

3.2.26 components of a PARI object:

There are essentially three ways to extract the components from a PARI object.

The first and most general, is the function **component**(x, n) which extracts the n^{th}-component of x. This is to be understood as follows: every PARI type has one or two initial code words. The components are counted, starting at 1, after these code words. In particular if x is a vector, this is indeed the n^{th}-component of x, if x is a matrix, the n^{th} column, if x is a polynomial, the n^{th} coefficient (i.e. of degree $n-1$), and for power series, the n^{th} significant coefficient. The use of the function `component` implies the knowledge of the structure of the different PARI types, which can be recalled by typing `\t` under `gp`.

The library syntax is **compo**(x, n), where n is a `long`.

The two other methods are more natural but more restricted. The function **polcoeff**(x, n) gives the coefficient of degree n of the polynomial or power series x, with respect to the main variable of x (to check variable ordering, or to change it, use the function `reorder`, see Section 3.11.2.23). In particular if n is less than the valuation of x or in the case of a polynomial, greater than the degree, the result is zero (contrary to `compo` which would send an error message). If x is a power series and n is greater than the largest significant degree, then an error message is issued.

For greater flexibility, vector or matrix types are also accepted for x, and the meaning is then identical with that of `compo`.

Finally note that a scalar type is considered by `polcoeff` as a polynomial of degree zero.

The library syntax is **truecoeff**(x, n).

The third method is specific to vectors or matrices in GP. If x is a (row or column) vector, then `x[n]` represents the n^{th} component of x, i.e. `compo(x,n)`. It is more natural and shorter to write. If x is a matrix, `x[m,n]` represents the coefficient of row `m` and column `n` of the matrix, `x[m,]` represents the m^{th} *row* of x, and `x[,n]` represents the n^{th} *column* of x.

Finally note that in library mode, the macros **gcoeff** and **gmael** are available as direct accessors to a `GEN component`. See Chapter 4 for details.

3.2.27 conj(x): conjugate of x. The meaning of this is clear, except that for real quadratic numbers, it means conjugation in the real quadratic field. This function has no effect on integers, reals, intmods, fractions or p-adics. The only forbidden type is polmod (see `conjvec` for this).

The library syntax is **gconj**(x).

3.2.28 conjvec(x): conjugate vector representation of x. If x is a polmod, equal to `Mod`(a, q), this gives a vector of length degree(q) containing the complex embeddings of the polmod if q has integral or rational coefficients, and the conjugates of the polmod if q has some intmod coefficients. The order is the same as that of the `polroots` functions. If x is an integer or a rational number, the result is x. If x is a (row or column) vector, the result is a matrix whose columns are the conjugate vectors of the individual elements of x.

The library syntax is **conjvec**$(x, prec)$.

3.2.29 denominator(x): denominator of x. The meaning of this is clear when x is a rational number or function. If x is an integer or a polynomial, it is treated as a rational number of function, respectively, and the result is equal to 1. For polynomials, you probably want to use

```
denominator( content(x) )
```

instead. As for modular objects, `t_INTMOD` and `t_PADIC` have denominator 1, and the denominator of a `t_POLMOD` is the denominator of its (minimal degree) polynomial representative.

If x is a recursive structure, for instance a vector or matrix, the lcm of the denominators of its components (a common denominator) is computed. This also applies for `t_COMPLEX`s and `t_QUAD`s.

Warning: multivariate objects are created according to variable priorities, with possibly surprising side effects (x/y is a polynomial, but y/x is a rational function). See Section 2.5.4.

The library syntax is **denom**(x).

3.2.30 floor(x): floor of x. When x is in **R**, the result is the largest integer smaller than or equal to x. Applied to a rational function, `floor`(x) returns the euclidian quotient of the numerator by the denominator.

The library syntax is **gfloor**(x).

3.2.31 frac(x): fractional part of x. Identical to $x - \text{floor}(x)$. If x is real, the result is in $[0, 1[$.

The library syntax is **gfrac**(x).

3.2.32 imag(x): imaginary part of x. When x is a quadratic number, this is the coefficient of ω in the "canonical" integral basis $(1, \omega)$.

The library syntax is **gimag**(x). This returns a copy of the imaginary part. The internal routine `imag_i` is faster, since it returns the pointer and skips the copy.

3.2.33 length(x): number of non-code words in x really used (i.e. the effective length minus 2 for integers and polynomials). In particular, the degree of a polynomial is equal to its length minus 1. If x has type `t_STR`, output number of letters.

The library syntax is **glength**(x) and the result is a C long.

3.2.34 lift($x, \{v\}$): lifts an element $x = a \bmod n$ of $\mathbf{Z}/n\mathbf{Z}$ to a in **Z**, and similarly lifts a polmod to a polynomial if v is omitted. Otherwise, lifts only polmods whose modulus has main variable v (if v does not occur in x, lifts only intmods). If x is of recursive (non modular) type, the lift is done coefficientwise. For p-adics, this routine acts as `truncate`. It is not allowed to have x of type `t_REAL`.

```
? lift(Mod(5,3))
%1 = 2
? lift(3 + O(3^9))
%2 = 3
? lift(Mod(x,x^2+1))
%3 = x
? lift(x * Mod(1,3) + Mod(2,3))
%4 = x + 2
? lift(x * Mod(y,y^2+1) + Mod(2,3))
%5 = y*x + Mod(2, 3)   \\ do you understand this one ?
? lift(x * Mod(y,y^2+1) + Mod(2,3), x)
%6 = Mod(y, y^2+1) * x + Mod(2, y^2+1)
```

The library syntax is **lift0**(x, v), where v is a `long` and an omitted v is coded as -1. Also available is **lift**(x) = `lift0(`x`,-1)`.

3.2.35 norm(x): algebraic norm of x, i.e. the product of x with its conjugate (no square roots are taken), or conjugates for polmods. For vectors and matrices, the norm is taken componentwise and hence is not the L^2-norm (see `norml2`). Note that the norm of an element of $\mathbf{R}$ is its square, so as to be compatible with the complex norm.

The library syntax is **gnorm**(x).

3.2.36 norml2(x): square of the L^2-norm of x. More precisely, if x is a scalar, `norml2`(x) is defined to be x `* conj(`x`)`. If x is a (row or column) vector or a matrix, `norml2(`x`)` is defined recursively as $\sum_i$ `norml2`(x_i), where (x_i) run through the components of x. In particular, this yields the usual $\sum |x_i|^2$ (resp. $\sum |x_{i,j}|^2$) if x is a vector (resp. matrix) with complex components.

```
? norml2( [ 1, 2, 3 ] )      \\ vector
%1 = 14
? norml2( [ 1, 2; 3, 4] )   \\ matrix
%1 = 30
? norml2( I + x )
%3 = x^2 + 1
? norml2( [ [1,2], [3,4], 5, 6 ] )   \\ recursively defined
%4 = 91
```

The library syntax is **gnorml2**(x).

3.2.37 numerator(x): numerator of x. The meaning of this is clear when x is a rational number or function. If x is an integer or a polynomial, it is treated as a rational number of function, respectively, and the result is x itself. For polynomials, you probably want to use

```
numerator( content(x) )
```

instead.

In other cases, `numerator(x)` is defined to be `denominator(x)*x`. This is the case when x is a vector or a matrix, but also for `t_COMPLEX` or `t_QUAD`. In particular since a `t_PADIC` or `t_INTMOD` has denominator 1, its numerator is itself.

Warning: multivariate objects are created according to variable priorities, with possibly surprising side effects (x/y is a polynomial, but y/x is a rational function). See Section 2.5.4.

The library syntax is **numer**(x).

3.2.38 numtoperm(n, k): generates the k-th permutation (as a row vector of length n) of the numbers 1 to n. The number k is taken modulo $n!$, i.e. inverse function of `permtonum`.

The library syntax is **numtoperm**(n, k), where n is a `long`.

3.2.39 padicprec(x, p): absolute p-adic precision of the object x. This is the minimum precision of the components of x. The result is `VERYBIGINT` ($2^{31} - 1$ for 32-bit machines or $2^{63} - 1$ for 64-bit machines) if x is an exact object.

The library syntax is **padicprec**(x, p) and the result is a `long` integer.

3.2.40 permtonum(x): given a permutation x on n elements, gives the number k such that $x =$ `numtoperm(n,k)`, i.e. inverse function of `numtoperm`.

The library syntax is **permtonum**(x).

3.2.41 precision$(x, \{n\})$: gives the precision in decimal digits of the PARI object x. If x is an exact object, the largest single precision integer is returned. If n is not omitted, creates a new object equal to x with a new precision n. This is to be understood as follows:

For exact types, no change. For x a vector or a matrix, the operation is done componentwise.

For real x, n is the number of desired significant *decimal* digits. If n is smaller than the precision of x, x is truncated, otherwise x is extended with zeros.

For x a p-adic or a power series, n is the desired number of significant p-adic or X-adic digits, where X is the main variable of x.

Note that the function `precision` never changes the type of the result. In particular it is not possible to use it to obtain a polynomial from a power series. For that, see `truncate`.

The library syntax is **precision0**(x, n), where n is a `long`. Also available are **ggprecision**(x) (result is a `GEN`) and **gprec**(x, n), where n is a `long`.

3.2.42 random$(\{N = 2^{31}\})$: returns a random integer between 0 and $N-1$. N is an integer, which can be arbitrary large. This is an internal PARI function and does not depend on the system's random number generator.

The resulting integer is obtained by means of linear congruences and will not be well distributed in arithmetic progressions. The random seed may be obtained via `getrand`, and reset using `setrand`.

Note that `random(2^31)` is *not* equivalent to `random()`, although both return an integer between 0 and $2^{31}-1$. In fact, calling `random` with an argument generates a number of random words (32bit or 64bit depending on the architecture), rescaled to the desired interval. The default uses directly a 31-bit generator.

Important technical note: the implementation of this function is incorrect unless N is a power of 2 (integers less than the bound are not equally likely, some may not even occur). It is kept for backward compatibility only, and has been rewritten from scratch in the 2.4.x unstable series. Use the following script for a correct version:

```
RANDOM(N) =
{ local(n, L);

  L = 1; while (L < N, L <<= 1;);
  /* L/2 < N <= L, L power of 2 */
  until(n < N, n = random(L)); n
}
```

The library syntax is **genrand**(N). Also available are `pari_rand`() which returns a random `unsigned long` (32bit or 64bit depending on the architecture), and `pari_rand31`() which returns a 31bit `long` integer.

3.2.43 real(x): real part of x. In the case where x is a quadratic number, this is the coefficient of 1 in the "canonical" integral basis $(1, \omega)$.

The library syntax is **greal**(x). This returns a copy of the real part. The internal routine `real_i` is faster, since it returns the pointer and skips the copy.

3.2.44 round$(x, \{\&e\})$: If x is in **R**, rounds x to the nearest integer and sets e to the number of error bits, that is the binary exponent of the difference between the original and the rounded value (the "fractional part"). If the exponent of x is too large compared to its precision (i.e. $e > 0$), the result is undefined and an error occurs if e was not given.

Important remark: note that, contrary to the other truncation functions, this function operates on every coefficient at every level of a PARI object. For example

$$\text{truncate}\left(\frac{2.4 * X^2 - 1.7}{X}\right) = 2.4 * X,$$

whereas

$$\text{round}\left(\frac{2.4 * X^2 - 1.7}{X}\right) = \frac{2 * X^2 - 2}{X}.$$

An important use of `round` is to get exact results after a long approximate computation, when theory tells you that the coefficients must be integers.

The library syntax is **grndtoi**$(x, \&e)$, where e is a `long` integer. Also available is **ground**(x).

3.2.45 simplify(x): this function simplifies x as much as it can. Specifically, a complex or quadratic number whose imaginary part is an exact 0 (i.e. not an approximate one as a `0(3)` or `0.E-28`) is converted to its real part, and a polynomial of degree 0 is converted to its constant term. Simplifications occur recursively.

This function is especially useful before using arithmetic functions, which expect integer arguments:

```
? x = 1 + y - y
%1 = 1
? divisors(x)
  *** divisors: not an integer argument in an arithmetic function
? type(x)
%2 = "t_POL"
? type(simplify(x))
%3 = "t_INT"
```

Note that GP results are simplified as above before they are stored in the history. (Unless you disable automatic simplification with `\y`, that is.) In particular

```
? type(%1)
%4 = "t_INT"
```

The library syntax is **simplify**(x).

3.2.46 sizebyte(x): outputs the total number of bytes occupied by the tree representing the PARI object x.

The library syntax is **taille2**(x) which returns a `long`; **taille**(x) returns the number of *words* instead.

3.2.47 sizedigit(x): outputs a quick bound for the number of decimal digits of (the components of) x, off by at most 1. If you want the exact value, you can use `#Str(x)`, which is slower.

The library syntax is **sizedigit**(x) which returns a `long`.

3.2.48 truncate$(x, \{\&e\})$: truncates x and sets e to the number of error bits. When x is in $\mathbf{R}$, this means that the part after the decimal point is chopped away, e is the binary exponent of the difference between the original and the truncated value (the "fractional part"). If the exponent of x is too large compared to its precision (i.e. $e > 0$), the result is undefined and an error occurs if e was not given. The function applies componentwise on vector / matrices; e is then the maximal number of error bits. If x is a rational function, the result is the "integer part" (Euclidean quotient of numerator by denominator) and e is not set.

Note a very special use of `truncate`: when applied to a power series, it transforms it into a polynomial or a rational function with denominator a power of X, by chopping away the $O(X^k)$. Similarly, when applied to a p-adic number, it transforms it into an integer or a rational number by chopping away the $O(p^k)$.

The library syntax is **gcvtoi**$(x, \&e)$, where e is a `long` integer. Also available is **gtrunc**(x).

3.2.49 valuation(x, p): computes the highest exponent of p dividing x. If p is of type integer, x must be an integer, an intmod whose modulus is divisible by p, a fraction, a q-adic number with $q = p$, or a polynomial or power series in which case the valuation is the minimum of the valuation of the coefficients.

If p is of type polynomial, x must be of type polynomial or rational function, and also a power series if x is a monomial. Finally, the valuation of a vector, complex or quadratic number is the minimum of the component valuations.

If $x = 0$, the result is `VERYBIGINT` ($2^{31} - 1$ for 32-bit machines or $2^{63} - 1$ for 64-bit machines) if x is an exact object. If x is a p-adic numbers or power series, the result is the exponent of the zero. Any other type combinations gives an error.

The library syntax is **ggval**(x, p), and the result is a `long`.

3.2.50 variable(x): gives the main variable of the object x, and p if x is a p-adic number. Gives an error if x has no variable associated to it. Note that this function is useful only in GP, since in library mode the function `gvar` is more appropriate.

The library syntax is **gpolvar**(x). However, in library mode, this function should not be used. Instead, test whether x is a p-adic (type `t_PADIC`), in which case p is in $x[2]$, or call the function **gvar**(x) which returns the variable *number* of x if it exists, `BIGINT` otherwise.

3.3 Transcendental functions.

As a general rule, which of course in some cases may have exceptions, transcendental functions operate in the following way:

• If the argument is either an integer, a real, a rational, a complex or a quadratic number, it is, if necessary, first converted to a real (or complex) number using the current precision held in the default `realprecision`. Note that only exact arguments are converted, while inexact arguments such as reals are not.

In GP this is transparent to the user, but when programming in library mode, care must be taken to supply a meaningful parameter *prec* as the last argument of the function if the first argument is an exact object. This parameter is ignored if the argument is inexact.

Note that in library mode the precision argument *prec* is a word count including codewords, i.e. represents the length in words of a real number, while under `gp` the precision (which is changed by the metacommand `\p` or using `default(realprecision,...)`) is the number of significant decimal digits.

Note that some accuracies attainable on 32-bit machines cannot be attained on 64-bit machines for parity reasons. For example the default `gp` accuracy is 28 decimal digits on 32-bit machines, corresponding to *prec* having the value 5, but this cannot be attained on 64-bit machines.

After possible conversion, the function is computed. Note that even if the argument is real, the result may be complex (e.g. acos(2.0) or acosh(0.0)). Note also that the principal branch is always chosen.

• If the argument is an intmod or a p-adic, at present only a few functions like `sqrt` (square root), `sqr` (square), `log`, `exp`, powering, `teichmuller` (Teichmüller character) and `agm` (arithmetic-geometric mean) are implemented.

Note that in the case of a 2-adic number, `sqr`(x) may not be identical to $x*x$: for example if $x = 1+O(2^5)$ and $y = 1+O(2^5)$ then $x*y = 1+O(2^5)$ while `sqr`$(x) = 1+O(2^6)$. Here, $x*x$ yields the same result as `sqr`(x) since the two operands are known to be *identical*. The same statement holds true for p-adics raised to the power n, where $v_p(n) > 0$.

Remark: note that if we wanted to be strictly consistent with the PARI philosophy, we should have $x*y = (4 \bmod 8)$ and `sqr`$(x) = (4 \bmod 32)$ when both x and y are congruent to 2 modulo 4. However, since intmod is an exact object, PARI assumes that the modulus must not change, and the result is hence (0 mod 4) in both cases. On the other hand, p-adics are not exact objects, hence are treated differently.

• If the argument is a polynomial, power series or rational function, it is, if necessary, first converted to a power series using the current precision held in the variable `precdl`. Under `gp` this again is transparent to the user. When programming in library mode, however, the global variable `precdl` must be set before calling the function if the argument has an exact type (i.e. not a power series). Here `precdl` is not an argument of the function, but a global variable.

Then the Taylor series expansion of the function around $X = 0$ (where X is the main variable) is computed to a number of terms depending on the number of terms of the argument and the function being computed.

• If the argument is a vector or a matrix, the result is the componentwise evaluation of the function. In particular, transcendental functions on square matrices, which are not implemented in the present version 2.3.5, will have a different name if they are implemented some day.

3.3.1 `^`: If y is not of type integer, `x^y` has the same effect as `exp(y*log(x))`. It can be applied to p-adic numbers as well as to the more usual types.

The library syntax is **gpow**$(x, y, prec)$.

3.3.2 Euler: Euler's constant $\gamma = 0.57721\cdots$. Note that `Euler` is one of the few special reserved names which cannot be used for variables (the others are `I` and `Pi`, as well as all function names).

The library syntax is **mpeuler**(*prec*) where *prec must* be given. Note that this creates γ on the PARI stack, but a copy is also created on the heap for quicker computations next time the function is called.

3.3.3 I: the complex number $\sqrt{-1}$.

The library syntax is the global variable `gi` (of type `GEN`).

3.3.4 Pi: the constant π ($3.14159\cdots$).

The library syntax is **mppi**(*prec*) where *prec must* be given. Note that this creates π on the PARI stack, but a copy is also created on the heap for quicker computations next time the function is called.

3.3.5 abs(x): absolute value of x (modulus if x is complex). Rational functions are not allowed. Contrary to most transcendental functions, an exact argument is *not* converted to a real number before applying `abs` and an exact result is returned if possible.

```
? abs(-1)
%1 = 1
? abs(3/7 + 4/7*I)
%2 = 5/7
? abs(1 + I)
%3 = 1.4142135623730950488016887242
```

If x is a polynomial, returns $-x$ if the leading coefficient is real and negative else returns x. For a power series, the constant coefficient is considered instead.

The library syntax is **gabs**(x, *prec*).

3.3.6 acos(x): principal branch of $\cos^{-1}(x)$, i.e. such that $\mathrm{Re}(\mathrm{acos}(x)) \in [0, \pi]$. If $x \in \mathbf{R}$ and $|x| > 1$, then $\mathrm{acos}(x)$ is complex.

The library syntax is **gacos**(x, *prec*).

3.3.7 acosh(x): principal branch of $\cosh^{-1}(x)$, i.e. such that $\mathrm{Im}(\mathrm{acosh}(x)) \in [0, \pi]$. If $x \in \mathbf{R}$ and $x < 1$, then $\mathrm{acosh}(x)$ is complex.

The library syntax is **gach**(x, *prec*).

3.3.8 agm(x, y): arithmetic-geometric mean of x and y. In the case of complex or negative numbers, the principal square root is always chosen. p-adic or power series arguments are also allowed. Note that a p-adic agm exists only if x/y is congruent to 1 modulo p (modulo 16 for $p = 2$). x and y cannot both be vectors or matrices.

The library syntax is **agm**(x, y, *prec*).

3.3.9 arg(x): argument of the complex number x, such that $-\pi < \arg(x) \leq \pi$.

The library syntax is **garg**(x, *prec*).

3.3.10 asin(x): principal branch of $\sin^{-1}(x)$, i.e. such that $\mathrm{Re}(\mathrm{asin}(x)) \in [-\pi/2, \pi/2]$. If $x \in \mathbf{R}$ and $|x| > 1$ then $\mathrm{asin}(x)$ is complex.

The library syntax is **gasin**(x, *prec*).

3.3.11 asinh(x): principal branch of $\sinh^{-1}(x)$, i.e. such that $\mathrm{Im}(\mathrm{asinh}(x)) \in [-\pi/2, \pi/2]$.

The library syntax is **gash**(x, *prec*).

3.3.12 atan(x): principal branch of $\tan^{-1}(x)$, i.e. such that $\mathrm{Re}(\mathrm{atan}(x)) \in]-\pi/2, \pi/2[$.

The library syntax is **gatan**$(x, prec)$.

3.3.13 atanh(x): principal branch of $\tanh^{-1}(x)$, i.e. such that $\mathrm{Im}(\mathrm{atanh}(x)) \in]-\pi/2, \pi/2]$. If $x \in \mathbf{R}$ and $|x| > 1$ then atanh(x) is complex.

The library syntax is **gath**$(x, prec)$.

3.3.14 bernfrac(x): Bernoulli number B_x, where $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6$,..., expressed as a rational number. The argument x should be of type integer.

The library syntax is **bernfrac**(x).

3.3.15 bernreal(x): Bernoulli number B_x, as `bernfrac`, but B_x is returned as a real number (with the current precision).

The library syntax is **bernreal**$(x, prec)$.

3.3.16 bernvec(x): creates a vector containing, as rational numbers, the Bernoulli numbers B_0, B_2,..., B_{2x}. This routine is obsolete. Use `bernfrac` instead each time you need a Bernoulli number in exact form.

Note: this routine is implemented using repeated independent calls to `bernfrac`, which is faster than the standard recursion in exact arithmetic. It is only kept for backward compatibility: it is not faster than individual calls to `bernfrac`, its output uses a lot of memory space, and coping with the index shift is awkward.

The library syntax is **bernvec**(x).

3.3.17 besselh1(nu, x): H^1-Bessel function of index nu and argument x.

The library syntax is **hbessel1**$(nu, x, prec)$.

3.3.18 besselh2(nu, x): H^2-Bessel function of index nu and argument x.

The library syntax is **hbessel2**$(nu, x, prec)$.

3.3.19 besseli(nu, x): I-Bessel function of index nu and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

The library syntax is **ibessel**$(nu, x, prec)$.

3.3.20 besselj(nu, x): J-Bessel function of index nu and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

The library syntax is **jbessel**$(nu, x, prec)$.

3.3.21 besseljh(n, x): J-Bessel function of half integral index. More precisely, `besseljh`(n, x) computes $J_{n+1/2}(x)$ where n must be of type integer, and x is any element of $\mathbf{C}$. In the present version 2.3.5, this function is not very accurate when x is small.

The library syntax is **jbesselh**$(n, x, prec)$.

3.3.22 besselk$(nu, x, \{flag = 0\})$: K-Bessel function of index nu (which can be complex) and argument x. Only real and positive arguments x are allowed in the present version 2.3.5. If *flag* is equal to 1, uses another implementation of this function which is faster when $x \gg 1$.

The library syntax is **kbessel**$(nu, x, prec)$ and **kbessel2**$(nu, x, prec)$ respectively.

3.3.23 besseln(nu, x): N-Bessel function of index nu and argument x.

The library syntax is **nbessel**$(nu, x, prec)$.

3.3.24 cos(x): cosine of x.

The library syntax is **gcos**$(x, prec)$.

3.3.25 cosh(x): hyperbolic cosine of x.

The library syntax is **gch**$(x, prec)$.

3.3.26 cotan(x): cotangent of x.

The library syntax is **gcotan**$(x, prec)$.

3.3.27 dilog(x): principal branch of the dilogarithm of x, i.e. analytic continuation of the power series $\log_2(x) = \sum_{n \geq 1} x^n/n^2$.

The library syntax is **dilog**$(x, prec)$.

3.3.28 eint1$(x, \{n\})$: exponential integral $\int_x^\infty \frac{e^{-t}}{t}\, dt$ $(x \in \mathbf{R})$

If n is present, outputs the n-dimensional vector $[\texttt{eint1}(x), \ldots, \texttt{eint1}(nx)]$ $(x \geq 0)$. This is faster than repeatedly calling `eint1(`i `* x)`.

The library syntax is **veceint1**$(x, n, prec)$. Also available is **eint1**$(x, prec)$.

3.3.29 erfc(x): complementary error function $(2/\sqrt{\pi}) \int_x^\infty e^{-t^2}\, dt$ $(x \in \mathbf{R})$.

The library syntax is **erfc**$(x, prec)$.

3.3.30 eta$(x, \{flag = 0\})$: Dedekind's η function, without the $q^{1/24}$. This means the following: if x is a complex number with positive imaginary part, the result is $\prod_{n=1}^\infty (1 - q^n)$, where $q = e^{2i\pi x}$. If x is a power series (or can be converted to a power series) with positive valuation, the result is $\prod_{n=1}^\infty (1 - x^n)$.

If $flag = 1$ and x can be converted to a complex number (i.e. is not a power series), computes the true η function, including the leading $q^{1/24}$.

The library syntax is **eta**$(x, prec)$.

3.3.31 exp(x): exponential of x. p-adic arguments with positive valuation are accepted.

The library syntax is **gexp**$(x, prec)$.

3.3.32 gammah(x): gamma function evaluated at the argument $x + 1/2$.

The library syntax is **ggamd**$(x, prec)$.

3.3.33 gamma(x): gamma function of x.

The library syntax is **ggamma**$(x, prec)$.

3.3.34 hyperu(a, b, x): U-confluent hypergeometric function with parameters a and b. The parameters a and b can be complex but the present implementation requires x to be positive.

The library syntax is **hyperu**$(a, b, x, prec)$.

3.3.35 incgam(s, x, y): incomplete gamma function. The argument x and s are complex numbers (x must be a positive real number if $s = 0$). The result returned is $\int_x^\infty e^{-t} t^{s-1}\, dt$. When y is given, assume (of course without checking!) that $y = \Gamma(s)$. For small x, this will speed up the computation.

The library syntax is **incgam**$(s, x, prec)$ and **incgam0**$(s, x, y, prec)$, respectively (an omitted y is coded as `NULL`).

3.3.36 incgamc(s, x): complementary incomplete gamma function. The arguments x and s are complex numbers such that s is not a pole of Γ and $|x|/(|s|+1)$ is not much larger than 1 (otherwise the convergence is very slow). The result returned is $\int_0^x e^{-t} t^{s-1}\, dt$.

The library syntax is **incgamc**$(s, x, prec)$.

3.3.37 log(x): principal branch of the natural logarithm of x, i.e. such that $\mathrm{Im}(\log(x)) \in]-\pi, \pi]$. The result is complex (with imaginary part equal to π) if $x \in \mathbf{R}$ and $x < 0$. In general, the algorithm uses the formula

$$\log(x) \approx \frac{\pi}{2\mathrm{agm}(1, 4/s)} - m \log 2,$$

if $s = x2^m$ is large enough. (The result is exact to B bits provided $s > 2^{B/2}$.) At low accuracies, the series expansion near 1 is used.

p-adic arguments are also accepted for x, with the convention that $\log(p) = 0$. Hence in particular $\exp(\log(x))/x$ is not in general equal to 1 but to a $(p-1)$-th root of unity (or ± 1 if $p = 2$) times a power of p.

The library syntax is **glog**$(x, prec)$.

3.3.38 lngamma(x): principal branch of the logarithm of the gamma function of x. This function is analytic on the complex plane with non-positive integers removed. Can have much larger arguments than `gamma` itself. The p-adic `lngamma` function is not implemented.

The library syntax is **glngamma**$(x, prec)$.

3.3.39 polylog$(m, x, flag = 0)$: one of the different polylogarithms, depending on *flag*:

If *flag* $= 0$ or is omitted: m^{th} polylogarithm of x, i.e. analytic continuation of the power series $\mathrm{Li}_m(x) = \sum_{n\geq 1} x^n/n^m$ $(x < 1)$. Uses the functional equation linking the values at x and $1/x$ to restrict to the case $|x| \leq 1$, then the power series when $|x|^2 \leq 1/2$, and the power series expansion in $\log(x)$ otherwise.

Using *flag*, computes a modified m^{th} polylogarithm of x. We use Zagier's notations; let $\Re_m$ denotes $\Re$ or $\Im$ depending whether m is odd or even:

If *flag* $= 1$: compute $\tilde{D}_m(x)$, defined for $|x| \leq 1$ by

$$\Re_m\left(\sum_{k=0}^{m-1} \frac{(-\log|x|)^k}{k!}\mathrm{Li}_{m-k}(x) + \frac{(-\log|x|)^{m-1}}{m!}\log|1-x|\right).$$

If *flag* $= 2$: compute $D_m(x)$, defined for $|x| \leq 1$ by

$$\Re_m\left(\sum_{k=0}^{m-1} \frac{(-\log|x|)^k}{k!}\mathrm{Li}_{m-k}(x) - \frac{1}{2}\frac{(-\log|x|)^m}{m!}\right).$$

If *flag* $= 3$: compute $P_m(x)$, defined for $|x| \leq 1$ by

$$\Re_m\left(\sum_{k=0}^{m-1} \frac{2^k B_k}{k!}(\log|x|)^k\mathrm{Li}_{m-k}(x) - \frac{2^{m-1}B_m}{m!}(\log|x|)^m\right).$$

These three functions satisfy the functional equation $f_m(1/x) = (-1)^{m-1}f_m(x)$.

The library syntax is **polylog0**$(m, x, flag, prec)$.

3.3.40 psi(x): the ψ-function of x, i.e. the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

The library syntax is **gpsi**$(x, prec)$.

3.3.41 sin(x): sine of x.

The library syntax is **gsin**$(x, prec)$.

3.3.42 sinh(x): hyperbolic sine of x.

The library syntax is **gsh**$(x, prec)$.

3.3.43 sqr(x): square of x. This operation is not completely straightforward, i.e. identical to $x*x$, since it can usually be computed more efficiently (roughly one-half of the elementary multiplications can be saved). Also, squaring a 2-adic number increases its precision. For example,

```
? (1 + O(2^4))^2
%1 = 1 + O(2^5)
? (1 + O(2^4)) * (1 + O(2^4))
%2 = 1 + O(2^4)
```

Note that this function is also called whenever one multiplies two objects which are known to be *identical*, e.g. they are the value of the same variable, or we are computing a power.

```
? x = (1 + O(2^4)); x * x
%3 = 1 + O(2^5)
? (1 + O(2^4))^4
%4 = 1 + O(2^6)
```

(note the difference between `%2` and `%3` above).

The library syntax is **gsqr**(x).

3.3.44 sqrt(x): principal branch of the square root of x, i.e. such that $\mathrm{Arg}(\mathrm{sqrt}(x)) \in]-\pi/2, \pi/2]$, or in other words such that $\Re(\mathrm{sqrt}(x)) > 0$ or $\Re(\mathrm{sqrt}(x)) = 0$ and $\Im(\mathrm{sqrt}(x)) \geq 0$. If $x \in \mathbf{R}$ and $x < 0$, then the result is complex with positive imaginary part.

Intmod a prime and p-adics are allowed as arguments. In that case, the square root (if it exists) which is returned is the one whose first p-adic digit (or its unique p-adic digit in the case of intmods) is in the interval $[0, p/2]$. When the argument is an intmod a non-prime (or a non-prime-adic), the result is undefined.

The library syntax is **gsqrt**($x, prec$).

3.3.45 sqrtn($x, n, \{\&z\}$): principal branch of the nth root of x, i.e. such that $\mathrm{Arg}(\mathrm{sqrt}(x)) \in]-\pi/n, \pi/n]$. Intmod a prime and p-adics are allowed as arguments.

If z is present, it is set to a suitable root of unity allowing to recover all the other roots. If it was not possible, z is set to zero. In the case this argument is present and no square root exist, 0 is returned instead or raising an error.

```
? sqrtn(Mod(2,7), 2)
%1 = Mod(4, 7)
? sqrtn(Mod(2,7), 2, &z); z
%2 = Mod(6, 7)
? sqrtn(Mod(2,7), 3)
  *** sqrtn: nth-root does not exist in gsqrtn.
? sqrtn(Mod(2,7), 3,  &z)
%2 = 0
? z
%3 = 0
```

The following script computes all roots in all possible cases:

```
sqrtnall(x,n)=
{
```

```
  local(V,r,z,r2);
  r = sqrtn(x,n, &z);
  if (!z, error("Impossible case in sqrtn"));
  if (type(x) == "t_INTMOD" || type(x)=="t_PADIC" ,
    r2 = r*z; n = 1;
    while (r2!=r, r2*=z;n++));
  V = vector(n); V[1] = r;
  for(i=2, n, V[i] = V[i-1]*z);
  V
}
addhelp(sqrtnall,"sqrtnall(x,n):compute the vector of nth-roots of x");
```

The library syntax is **gsqrtn**$(x, n, \&z, prec)$.

3.3.46 tan(x): tangent of x.

The library syntax is **gtan**$(x, prec)$.

3.3.47 tanh(x): hyperbolic tangent of x.

The library syntax is **gth**$(x, prec)$.

3.3.48 teichmuller(x): Teichmüller character of the p-adic number x, i.e. the unique $(p-1)$-th root of unity congruent to $x/p^{v_p(x)}$ modulo p.

The library syntax is **teich**(x).

3.3.49 theta(q, z): Jacobi sine theta-function.

The library syntax is **theta**$(q, z, prec)$.

3.3.50 thetanullk(q, k): k-th derivative at $z = 0$ of `theta`(q, z).

The library syntax is **thetanullk**$(q, k, prec)$, where k is a `long`.

3.3.51 weber$(x, \{flag = 0\})$: one of Weber's three f functions. If $flag = 0$, returns

$$f(x) = \exp(-i\pi/24) \cdot \eta((x+1)/2) \,/\, \eta(x) \quad \text{such that} \quad j = (f^{24} - 16)^3/f^{24}\,,$$

where j is the elliptic j-invariant (see the function `ellj`). If $flag = 1$, returns

$$f_1(x) = \eta(x/2) \,/\, \eta(x) \quad \text{such that} \quad j = (f_1^{24} + 16)^3/f_1^{24}\,.$$

Finally, if $flag = 2$, returns

$$f_2(x) = \sqrt{2}\eta(2x) \,/\, \eta(x) \quad \text{such that} \quad j = (f_2^{24} + 16)^3/f_2^{24}.$$

Note the identities $f^8 = f_1^8 + f_2^8$ and $f f_1 f_2 = \sqrt{2}$.

The library syntax is **weber0**$(x, flag, prec)$. Associated to the various values of *flag*, the following functions are also available: **werberf**$(x, prec)$, **werberf1**$(x, prec)$ or **werberf2**$(x, prec)$.

3.3.52 zeta(s): For s a complex number, Riemann's zeta function $\zeta(s) = \sum_{n\geq 1} n^{-s}$, computed using the Euler-Maclaurin summation formula, except when s is of type integer, in which case it is computed using Bernoulli numbers for $s \leq 0$ or $s > 0$ and even, and using modular forms for $s > 0$ and odd.

For s a p-adic number, Kubota-Leopoldt zeta function at s, that is the unique continuous p-adic function on the p-adic integers that interpolates the values of $(1-p^{-k})\zeta(k)$ at negative integers k such that $k \equiv 1 \pmod{p-1}$ (resp. k is odd) if p is odd (resp. $p = 2$).

The library syntax is **gzeta**(s, *prec*).

3.4 Arithmetic functions.

These functions are by definition functions whose natural domain of definition is either $\mathbf{Z}$ (or $\mathbf{Z}_{>0}$), or sometimes polynomials over a base ring. Functions which concern polynomials exclusively will be explained in the next section. The way these functions are used is completely different from transcendental functions: in general only the types integer and polynomial are accepted as arguments. If a vector or matrix type is given, the function will be applied on each coefficient independently.

In the present version 2.3.5, all arithmetic functions in the narrow sense of the word — Euler's totient function, the Moebius function, the sums over divisors or powers of divisors etc.— call, after trial division by small primes, the same versatile factoring machinery described under `factorint`. It includes Shanks SQUFOF, Pollard Rho, ECM and MPQS stages, and has an early exit option for the functions **moebius** and (the integer function underlying) **issquarefree**. Note that it relies on a (fairly strong) probabilistic primality test, see `ispseudoprime`.

3.4.1 addprimes({$x = []$}): adds the integers contained in the vector x (or the single integer x) to a special table of "user-defined primes", and returns that table. Whenever `factor` is subsequently called, it will trial divise by the elements in this table. If x is empty or omitted, just returns the current list of extra primes.

The entries in x are not checked for primality, and in fact they need only be positive integers. The algorithm makes sure that all elements in the table are pairwise coprime, so it may end up containing divisors of the input integers.

It is a useful trick to add known composite numbers, which the function `factor`($x, 0$) was not able to factor. In case the message "impossible inverse modulo ⟨*some INTMOD*⟩" shows up afterwards, you have just stumbled over a non-trivial factor. Note that the arithmetic functions in the narrow sense, like **eulerphi**, do *not* use this extra table.

To remove primes from the list use `removeprimes`.

The library syntax is **addprimes**(x).

3.4.2 bestappr$(x, A, \{B\})$: if B is omitted, finds the best rational approximation to $x \in \mathbf{R}$ (or $\mathbf{R}[X]$, or $\mathbf{R}^n$, ...) with denominator at most equal to A using continued fractions.

If B is present, x is assumed to be of type `t_INTMOD` modulo M (or a recursive combination of those), and the routine returns the unique fraction a/b in coprime integers $a \leq A$ and $b \leq B$ which is congruent to x modulo M. If $M \leq 2AB$, uniqueness is not guaranteed and the function fails with an error message. If rational reconstruction is not possible (no such a/b exists for at least one component of x), returns -1.

The library syntax is **bestappr0**(x, A, B). Also available is **bestappr**(x, A) corresponding to an omitted B.

3.4.3 bezout(x, y): finds u and v minimal in a natural sense such that $x * u + y * v = \gcd(x, y)$. The arguments must be both integers or both polynomials, and the result is a row vector with three components u, v, and $\gcd(x, y)$.

The library syntax is **vecbezout**(x, y) to get the vector, or **gbezout**$(x, y, \&u, \&v)$ which gives as result the address of the created gcd, and puts the addresses of the corresponding created objects into u and v.

3.4.4 bezoutres(x, y): as `bezout`, with the resultant of x and y replacing the gcd. The algorithm uses (subresultant) assumes the base ring is a domain.

The library syntax is **vecbezoutres**(x, y) to get the vector, or **subresext**$(x, y, \&u, \&v)$ which gives as result the address of the created gcd, and puts the addresses of the corresponding created objects into u and v.

3.4.5 bigomega(x): number of prime divisors of $|x|$ counted with multiplicity. x must be an integer.

The library syntax is **bigomega**(x), the result is a `long`.

3.4.6 binomial(x, y): binomial coefficient $\binom{x}{y}$. Here y must be an integer, but x can be any PARI object.

The library syntax is **binomial**(x, y), where y must be a `long`.

3.4.7 chinese$(x, \{y\})$: if x and y are both intmods or both polmods, creates (with the same type) a z in the same residue class as x and in the same residue class as y, if it is possible.

This function also allows vector and matrix arguments, in which case the operation is recursively applied to each component of the vector or matrix. For polynomial arguments, it is applied to each coefficient.

If y is omitted, and x is a vector, `chinese` is applied recursively to the components of x, yielding a residue belonging to the same class as all components of x.

Finally `chinese`$(x, x) = x$ regardless of the type of x; this allows vector arguments to contain other data, so long as they are identical in both vectors.

The library syntax is **chinese**(x, y). Also available is `chinese1`(x), corresponding to an ommitted `y`.

3.4.8 content(x): computes the gcd of all the coefficients of x, when this gcd makes sense. This is the natural definition if x is a polynomial (and by extension a power series) or a vector/matrix. This is in general a weaker notion than the *ideal* generated by the coefficients:

```
? content(2*x+y)
%1 = 1              \\ = gcd(2,y) over Q[y]
```

If x is a scalar, this simply returns the absolute value of x if x is rational (`t_INT` or `t_FRAC`), and either 1 (inexact input) or x (exact input) otherwise; the result should be identical to `gcd(x, 0)`.

The content of a rational function is the ratio of the contents of the numerator and the denominator. In recursive structures, if a matrix or vector *coefficient* x appears, the gcd is taken not with x, but with its content:

```
? content([ [2], 4*matid(3) ])
%1 = 2
```

The library syntax is **content**(x).

3.4.9 contfrac$(x, \{b\}, \{nmax\})$: creates the row vector whose components are the partial quotients of the continued fraction expansion of x. That is a result $[a_0, \ldots, a_n]$ means that $x \approx a_0 + 1/(a_1 + \ldots + 1/a_n)\ldots)$. The output is normalized so that $a_n \neq 1$ (unless we also have $n = 0$).

The number of partial quotients n is limited to $nmax$. If x is a real number, the expansion stops at the last significant partial quotient if $nmax$ is omitted. x can also be a rational function or a power series.

If a vector b is supplied, the numerators will be equal to the coefficients of b (instead of all equal to 1 as above). The length of the result is then equal to the length of b, unless a partial remainder is encountered which is equal to zero, in which case the expansion stops. In the case of real numbers, the stopping criterion is thus different from the one mentioned above since, if b is too long, some partial quotients may not be significant.

If b is an integer, the command is understood as `contfrac(`$x, nmax$`)`.

The library syntax is **contfrac0**$(x, b, nmax)$. Also available are **gboundcf**$(x, nmax)$, **gcf**(x), or **gcf2**(b, x), where $nmax$ is a C integer.

3.4.10 contfracpnqn(x): when x is a vector or a one-row matrix, x is considered as the list of partial quotients $[a_0, a_1, \ldots, a_n]$ of a rational number, and the result is the 2 by 2 matrix $[p_n, p_{n-1}; q_n, q_{n-1}]$ in the standard notation of continued fractions, so $p_n/q_n = a_0 + 1/(a_1 + \ldots + 1/a_n)\ldots)$. If x is a matrix with two rows $[b_0, b_1, \ldots, b_n]$ and $[a_0, a_1, \ldots, a_n]$, this is then considered as a generalized continued fraction and we have similarly $p_n/q_n = 1/b_0(a_0 + b_1/(a_1 + \ldots + b_n/a_n)\ldots)$. Note that in this case one usually has $b_0 = 1$.

The library syntax is **pnqn**(x).

3.4.11 core$(n, \{flag = 0\})$: if n is a non-zero integer written as $n = df^2$ with d squarefree, returns d. If *flag* is non-zero, returns the two-element row vector $[d, f]$.

The library syntax is **core0**$(n, flag)$. Also available are **core**(n) (= **core0**$(n, 0)$) and **core2**(n) (= **core0**$(n, 1)$).

3.4.12 coredisc$(n, \{flag\})$: if n is a non-zero integer written as $n = df^2$ with d fundamental discriminant (including 1), returns d. If *flag* is non-zero, returns the two-element row vector $[d, f]$. Note that if n is not congruent to 0 or 1 modulo 4, f will be a half integer and not an integer.

The library syntax is **coredisc0**$(n, flag)$. Also available are **coredisc**(n) (= **coredisc**$(n, 0)$) and **coredisc2**(n) (= **coredisc**$(n, 1)$).

3.4.13 dirdiv(x, y): x and y being vectors of perhaps different lengths but with $y[1] \neq 0$ considered as Dirichlet series, computes the quotient of x by y, again as a vector.

The library syntax is **dirdiv**(x, y).

3.4.14 direuler$(p = a, b, expr, \{c\})$: computes the Dirichlet series associated to the Euler product of expression *expr* as p ranges through the primes from a to b. *expr* must be a polynomial or rational function in another variable than p (say X) and $expr(X)$ is understood as the local factor $expr(p^{-s})$.

The series is output as a vector of coefficients. If c is present, output only the first c coefficients in the series. The following command computes the **sigma** function, associated to $\zeta(s)\zeta(s-1)$:

```
? direuler(p=2, 10, 1/((1-X)*(1-p*X)))
%1 = [1, 3, 4, 7, 6, 12, 8, 15, 13, 18]
```

The library syntax is **direuler**(`void *E, GEN (*eval)(GEN,void*), GEN a, GEN b`)

3.4.15 dirmul(x, y): x and y being vectors of perhaps different lengths considered as Dirichlet series, computes the product of x by y, again as a vector.

The library syntax is **dirmul**(x, y).

3.4.16 divisors(x): creates a row vector whose components are the divisors of x. The factorization of x (as output by `factor`) can be used instead.

By definition, these divisors are the products of the irreducible factors of n, as produced by `factor(n)`, raised to appropriate powers (no negative exponent may occur in the factorization). If n is an integer, they are the positive divisors, in increasing order.

The library syntax is **divisors**(x).

3.4.17 eulerphi(x): Euler's ϕ (totient) function of $|x|$, in other words $|(\mathbf{Z}/x\mathbf{Z})^*|$. x must be of type integer.

The library syntax is **phi**(x).

3.4.18 factor$(x, \{lim = -1\})$: general factorization function. If x is of type integer, rational, polynomial or rational function, the result is a two-column matrix, the first column being the irreducibles dividing x (prime numbers or polynomials), and the second the exponents. If x is a vector or a matrix, the factoring is done componentwise (hence the result is a vector or matrix of two-column matrices). By definition, 0 is factored as 0^1.

If x is of type integer or rational, the factors are *pseudoprimes* (see `ispseudoprime`), and in general not rigorously proven primes. In fact, any factor which is $\leq 10^{13}$ is a genuine prime number. Use `isprime` to prove primality of other factors, as in

```
fa = factor(2^2^7 +1)
isprime( fa[,1] )
```

An argument *lim* can be added, meaning that we look only for prime factors $p < lim$, or up to `primelimit`, whichever is lowest (except when $lim = 0$ where the effect is identical to setting $lim =$ `primelimit`). In this case, the remaining part may actually be a proven composite! See `factorint` for more information about the algorithms used.

The polynomials or rational functions to be factored must have scalar coefficients. In particular PARI does *not* know how to factor multivariate polynomials. See `factormod` and `factorff` for the algorithms used over finite fields, `factornf` for the algorithms over number fields. Over **Q**, van Hoeij's method is used, which is able to cope with hundreds of modular factors.

Note that PARI tries to guess in a sensible way over which ring you want to factor. Note also that factorization of polynomials is done up to multiplication by a constant. In particular, the factors of rational polynomials will have integer coefficients, and the content of a polynomial or rational function is discarded and not included in the factorization. If needed, you can always ask for the content explicitly:

```
? factor(t^2 + 5/2*t + 1)
%1 =
[2*t + 1 1]

[t + 2 1]

? content(t^2 + 5/2*t + 1)
%2 = 1/2
```

See also `factornf` and `nffactor`.

The library syntax is **factor0**(x, lim), where *lim* is a C integer. Also available are **factor**(x) $(=$ **factor0**$(x, -1))$, **smallfact**(x) $(=$ **factor0**$(x, 0))$.

3.4.19 factorback$(f, \{e\}, \{nf\})$: gives back the factored object corresponding to a factorization. The integer 1 corresponds to the empty factorization. If the last argument is of number field type (e.g. created by `nfinit`), assume we are dealing with an ideal factorization in the number field. The resulting ideal product is given in HNF form.

If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1 (the product of the $f[i]$ is returned). Finally, f can be a regular factorization, as produced with any `factor` command. A few examples:

```
? factorback([2,2; 3,1])
```

```
%1 = 12
? factorback([2,2], [3,1])
%2 = 12
? factorback([5,2,3])
%3 = 30
? factorback([2,2], [3,1], nfinit(x^3+2))
%4 =
[16 0 0]

[0 16 0]

[0 0 16]
? nf = nfinit(x^2+1); fa = idealfactor(nf, 10)
%5 =
[[2, [1, 1]~, 2, 1, [1, 1]~] 2]

[[5, [-2, 1]~, 1, 1, [2, 1]~] 1]

[[5, [2, 1]~, 1, 1, [-2, 1]~] 1]
? factorback(fa)
  ***   forbidden multiplication t_VEC * t_VEC.
? factorback(fa, nf)
%6 =
[10 0]

[0 10]
```

In the fourth example, 2 and 3 are interpreted as principal ideals in a cubic field. In the fifth one, `factorback(fa)` is meaningless since we forgot to indicate the number field, and the entries in the first column of `fa` can't be multiplied.

The library syntax is **factorback0**(f, e, nf), where an omitted nf or e is entered as `NULL`. Also available is `factorback`(f, nf) (case $e =$ `NULL`) where an omitted nf is entered as `NULL`.

3.4.20 factorcantor(x, p): factors the polynomial x modulo the prime p, using distinct degree plus Cantor-Zassenhaus. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two-column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. If you want only the *degrees* of the irreducible polynomials (for example for computing an L-function), use `factormod`$(x, p, 1)$. Note that the `factormod` algorithm is usually faster than `factorcantor`.

The library syntax is **factcantor**(x, p).

3.4.21 factorff(x, p, a): factors the polynomial x in the field $\mathbf{F}_q$ defined by the irreducible polynomial a over $\mathbf{F}_p$. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two-column matrix: the first column contains the irreducible factors of x, and the second their exponents. If all the coefficients of x are in $\mathbf{F}_p$, a much faster algorithm is applied, using the computation of isomorphisms between finite fields.

The library syntax is **factorff**(x, p, a).

3.4.22 factorial(x) or x!: factorial of x. The expression x! gives a result which is an integer, while `factorial`(x) gives a real number.

The library syntax is **mpfact**(x) for x! and **mpfactr**($x, prec$) for `factorial`(x). x must be a `long` integer and not a PARI integer.

3.4.23 factorint($n, \{flag = 0\}$): factors the integer n into a product of pseudoprimes (see `ispseudoprime`), using a combination of the Shanks SQUFOF and Pollard Rho method (with modifications due to Brent), Lenstra's ECM (with modifications by Montgomery), and MPQS (the latter adapted from the LiDIA code with the kind permission of the LiDIA maintainers), as well as a search for pure powers with exponents≤ 10. The output is a two-column matrix as for `factor`. Use `isprime` on the result if you want to guarantee primality.

This gives direct access to the integer factoring engine called by most arithmetical functions. *flag* is optional; its binary digits mean 1: avoid MPQS, 2: skip first stage ECM (we may still fall back to it later), 4: avoid Rho and SQUFOF, 8: don't run final ECM (as a result, a huge composite may be declared to be prime). Note that a (strong) probabilistic primality test is used; thus composites might (very rarely) not be detected.

You are invited to play with the flag settings and watch the internals at work by using `gp`'s `debuglevel` default parameter (level 3 shows just the outline, 4 turns on time keeping, 5 and above show an increasing amount of internal details). If you see anything funny happening, please let us know.

The library syntax is **factorint**($n, flag$).

3.4.24 factormod($x, p, \{flag = 0\}$): factors the polynomial x modulo the prime integer p, using Berlekamp. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two-column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. If *flag* is non-zero, outputs only the *degrees* of the irreducible polynomials (for example, for computing an L-function). A different algorithm for computing the mod p factorization is `factorcantor` which is sometimes faster.

The library syntax is **factormod**($x, p, flag$). Also available are **factmod**(x, p) (which is equivalent to **factormod**($x, p, 0$)) and **simplefactmod**(x, p) (= **factormod**($x, p, 1$)).

3.4.25 fibonacci(x): x^{th} Fibonacci number.

The library syntax is **fibo**(x). x must be a `long`.

3.4.26 ffinit($p, n, \{v = x\}$): computes a monic polynomial of degree n which is irreducible over $\mathbf{F}_p$. For instance if `P = ffinit(3,2,y)`, you can represent elements in $\mathbf{F}_{3^2}$ as polmods modulo `P`. This function uses a fast variant of Adleman-Lenstra's algorithm.

The library syntax is **ffinit**(p, n, v), where v is a variable number.

3.4.27 gcd$(x, \{y\})$: creates the greatest common divisor of x and y. x and y can be of quite general types, for instance both rational numbers. If y is omitted and x is a vector, returns the gcd of all components of x, i.e. this is equivalent to `content(x)`.

When x and y are both given and one of them is a vector/matrix type, the GCD is again taken recursively on each component, but in a different way. If y is a vector, resp. matrix, then the result has the same type as y, and components equal to `gcd(x, y[i])`, resp. `gcd(x, y[,i])`. Else if x is a vector/matrix the result has the same type as x and an analogous definition. Note that for these types, `gcd` is not commutative.

The algorithm used is a naive Euclid except for the following inputs:

• integers: use modified right-shift binary ("plus-minus" variant).

• univariate polynomials with coeffients in the same number field (in particular rational): use modular gcd algorithm.

• general polynomials: use the subresultant algorithm if coefficient explosion is likely (exact, non modular, coefficients).

The library syntax is **ggcd**(x, y). For general polynomial inputs, **srgcd**(x, y) is also available. For univariate *rational* polynomials, one also has **modulargcd**(x, y).

3.4.28 hilbert$(x, y, \{p\})$: Hilbert symbol of x and y modulo p. If x and y are of type integer or fraction, an explicit third parameter p must be supplied, $p = 0$ meaning the place at infinity. Otherwise, p needs not be given, and x and y can be of compatible types integer, fraction, real, intmod a prime (result is undefined if the modulus is not prime), or p-adic.

The library syntax is **hil**(x, y, p).

3.4.29 isfundamental(x): true (1) if x is equal to 1 or to the discriminant of a quadratic field, false (0) otherwise.

The library syntax is **gisfundamental**(x), but the simpler function **isfundamental**(x) which returns a `long` should be used if x is known to be of type integer.

3.4.30 ispower$(x, \{k\}, \{\&n\})$: if k is given, returns true (1) if x is a k-th power, false (0) if not. In this case, x may be an integer or polynomial, a rational number or function, or an intmod a prime or p-adic.

If k is omitted, only integers and fractions are allowed and the function returns the maximal $k \geq 2$ such that $x = n^k$ is a perfect power, or 0 if no such k exist; in particular `ispower(-1)`, `ispower(0)`, and `ispower(1)` all return 0.

If a third argument $\&n$ is given and a k-th root was computed in the process, then n is set to that root.

The library syntax is **ispower**$(x, k, \&n)$, the result is a `long`. Omitted k or n are coded as `NULL`.

3.4.31 isprime$(x, \{flag = 0\})$: true (1) if x is a (proven) prime number, false (0) otherwise. This can be very slow when x is indeed prime and has more than 1000 digits, say. Use `ispseudoprime` to quickly check for pseudo primality. See also `factor`.

If *flag* $= 0$, use a combination of Baillie-PSW pseudo primality test (see `ispseudoprime`), Selfridge "$p-1$" test if $x-1$ is smooth enough, and Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL) for general x.

If *flag* $= 1$, use Selfridge-Pocklington-Lehmer "$p-1$" test and output a primality certificate as follows: return 0 if x is composite, 1 if x is small enough that passing Baillie-PSW test guarantees its primality (currently $x < 10^{13}$), 2 if x is a large prime whose primality could only sensibly be proven (given the algorithms implemented in PARI) using the APRCL test. Otherwise (x is large and $x-1$ is smooth) output a three column matrix as a primality certificate. The first column contains the prime factors p of $x-1$, the second the corresponding elements a_p as in Proposition 8.3.1 in GTM 138, and the third the output of isprime(p,1). The algorithm fails if one of the pseudo-prime factors is not prime, which is exceedingly unlikely (and well worth a bug report).

If *flag* $= 2$, use APRCL.

The library syntax is **gisprime**$(x, flag)$, but the simpler function **isprime**(x) which returns a `long` should be used if x is known to be of type integer.

3.4.32 ispseudoprime$(x, \{flag\})$: true (1) if x is a strong pseudo prime (see below), false (0) otherwise. If this function returns false, x is not prime; if, on the other hand it returns true, it is only highly likely that x is a prime number. Use `isprime` (which is of course much slower) to prove that x is indeed prime.

If *flag* $= 0$, checks whether x is a Baillie-Pomerance-Selfridge-Wagstaff pseudo prime (strong Rabin-Miller pseudo prime for base 2, followed by strong Lucas test for the sequence $(P, -1)$, P smallest positive integer such that $P^2 - 4$ is not a square mod x).

There are no known composite numbers passing this test (in particular, all composites $\leq 10^{13}$ are correctly detected), although it is expected that infinitely many such numbers exist.

If *flag* > 0, checks whether x is a strong Miller-Rabin pseudo prime for *flag* randomly chosen bases (with end-matching to catch square roots of -1).

The library syntax is **gispseudoprime**$(x, flag)$, but the simpler function **ispseudoprime**(x) which returns a `long` should be used if x is known to be of type integer.

3.4.33 issquare$(x, \{\&n\})$: true (1) if x is a square, false (0) if not. What "being a square" means depends on the type of x: all `t_COMPLEX` are squares, as well as all non-negative `t_REAL`; for exact types such as `t_INT`, `t_FRAC` and `t_INTMOD`, squares are numbers of the form s^2 with s in $\mathbf{Z}$, $\mathbf{Q}$ and $\mathbf{Z}/N\mathbf{Z}$ respectively.

```
? issquare(3)          \\ as an integer
%1 = 0
? issquare(3.)         \\ as a real number
%2 = 1
? issquare(Mod(7, 8))  \\ in Z/8Z
%3 = 0
? issquare( 5 + O(13^4) )  \\ in Q_13
%4 = 0
```

If n is given and an exact square root had to be computed in the checking process, puts that square root in n. This is the case when x is a `t_INT`, `t_FRAC`, `t_POL` or `t_RFRAC` (or a vector of such objects):

```
? issquare(4, &n)
%1 = 1
? n
%2 = 2
? issquare([4, x^2], &n)
%3 = [1, 1]  \\ both are squares
? n
%4 = [2, x]  \\ the square roots
```

This will *not* work for `t_INTMOD` (use quadratic reciprocity) or `t_SER` (only check the leading coefficient).

The library syntax is **gissquarerem**$(x, \&n)$. Also available is **gissquare**(x).

3.4.34 issquarefree(x): true (1) if x is squarefree, false (0) if not. Here x can be an integer or a polynomial.

The library syntax is **gissquarefree**(x), but the simpler function **issquarefree**(x) which returns a `long` should be used if x is known to be of type integer. This **issquarefree** is just the square of the Moebius function, and is computed as a multiplicative arithmetic function much like the latter.

3.4.35 kronecker(x, y): Kronecker symbol $(x|y)$, where x and y must be of type integer. By definition, this is the extension of Legendre symbol to $\mathbf{Z} \times \mathbf{Z}$ by total multiplicativity in both arguments with the following special rules for $y = 0, -1$ or 2:

- $(x|0) = 1$ if $|x| = 1$ and 0 otherwise.
- $(x|-1) = 1$ if $x \geq 0$ and -1 otherwise.
- $(x|2) = 0$ if x is even and 1 if $x = 1, -1 \bmod 8$ and -1 if $x = 3, -3 \bmod 8$.

The library syntax is **kronecker**(x, y), the result (0 or ± 1) is a `long`.

3.4.36 lcm$(x, \{y\})$: least common multiple of x and y, i.e. such that $\mathrm{lcm}(x, y) * \gcd(x, y) = \mathrm{abs}(x * y)$. If y is omitted and x is a vector, returns the lcm of all components of x.

When x and y are both given and one of them is a vector/matrix type, the LCM is again taken recursively on each component, but in a different way. If y is a vector, resp. matrix, then the result has the same type as y, and components equal to `lcm(x, y[i])`, resp. `lcm(x, y[,i])`. Else if x is a vector/matrix the result has the same type as x and an analogous definition. Note that for these types, `lcm` is not commutative.

Note that `lcm(v)` is quite different from

```
l = v[1]; for (i = 1, #v, l = lcm(l, v[i]))
```

Indeed, `lcm(v)` is a scalar, but `l` may not be (if one of the `v[i]` is a vector/matrix). The computation uses a divide-conquer tree and should be much more efficient, especially when using the GMP multiprecision kernel (and more subquadratic algorithms become available):

```
? v = vector(10^4, i, random);
```

```
? lcm(v);
time = 323 ms.
? l = v[1]; for (i = 1, #v, l = lcm(l, v[i]))
time = 833 ms.
```

The library syntax is **glcm**(x, y).

3.4.37 moebius(x): Moebius μ-function of $|x|$. x must be of type integer.

The library syntax is **mu**(x), the result (0 or ± 1) is a `long`.

3.4.38 nextprime(x): finds the smallest pseudoprime (see `ispseudoprime`) greater than or equal to x. x can be of any real type. Note that if x is a pseudoprime, this function returns x and not the smallest pseudoprime strictly larger than x. To rigorously prove that the result is prime, use `isprime`.

The library syntax is **nextprime**(x).

3.4.39 numdiv(x): number of divisors of $|x|$. x must be of type integer.

The library syntax is **numbdiv**(x).

3.4.40 numbpart(n): gives the number of unrestricted partitions of n, usually called $p(n)$ in the litterature; in other words the number of nonnegative integer solutions to $a + 2b + 3c + \cdots = n$. n must be of type integer and $1 \leq n < 10^{15}$. The algorithm uses the Hardy-Ramanujan-Rademacher formula.

The library syntax is **numbpart**(n).

3.4.41 omega(x): number of distinct prime divisors of $|x|$. x must be of type integer.

The library syntax is **omega**(x), the result is a `long`.

3.4.42 precprime(x): finds the largest pseudoprime (see `ispseudoprime`) less than or equal to x. x can be of any real type. Returns 0 if $x \leq 1$. Note that if x is a prime, this function returns x and not the largest prime strictly smaller than x. To rigorously prove that the result is prime, use `isprime`.

The library syntax is **precprime**(x).

3.4.43 prime(x): the x^{th} prime number, which must be among the precalculated primes.

The library syntax is **prime**(x). x must be a `long`.

3.4.44 primepi(x): the prime counting function. Returns the number of primes p, $p \leq x$. Uses a naive algorithm so that x must be less than `primelimit`.

The library syntax is **primepi**(x).

3.4.45 primes(x): creates a row vector whose components are the first x prime numbers, which must be among the precalculated primes.

The library syntax is **primes**(x). x must be a `long`.

3.4.46 qfbclassno$(D, \{flag = 0\})$: ordinary class number of the quadratic order of discriminant D. In the present version 2.3.5, a $O(D^{1/2})$ algorithm is used for $D > 0$ (using Euler product and the functional equation) so D should not be too large, say $D < 10^8$, for the time to be reasonable. On the other hand, for $D < 0$ one can reasonably compute `qfbclassno(D)` for $|D| < 10^{25}$, since the routine uses Shanks's method which is in $O(|D|^{1/4})$. For larger values of $|D|$, see `quadclassunit`.

If *flag* $= 1$, compute the class number using Euler products and the functional equation. However, it is in $O(|D|^{1/2})$.

Important warning. For $D < 0$, this function may give incorrect results when the class group has a low exponent (has many cyclic factors), because implementing Shanks's method in full generality slows it down immensely. It is therefore strongly recommended to double-check results using either the version with *flag* $= 1$ or the function `quadclassunit`.

Warning. contrary to what its name implies, this routine does not compute the number of classes of binary primitive forms of discriminant D, which is equal to the *narrow* class number. The two notions are the same when $D < 0$ or the fundamental unit ε has negative norm; when $D > 0$ and $N\varepsilon > 0$, the number of classes of forms is twice the ordinary class number. This is a problem which we cannot fix for backward compatibility reasons. Use the following routine if you are only interested in the number of classes of forms:

```
QFBclassno(D) =
  qfbclassno(D) * if (D < 0 || norm(quadunit(D)) < 0, 1, 2)
```

Here are a few examples:

```
? qfbclassno(400000028)
time = 3,140 ms.
%1 = 1
? quadclassunit(400000028).no
time = 20 ms. \\ much faster
%2 = 1
? qfbclassno(-400000028)
time = 0 ms.
%3 = 7253 \\ correct, and fast enough
? quadclassunit(-400000028).no
time = 0 ms.
%4 = 7253
```

The library syntax is **qfbclassno0**$(D, flag)$. Also available: **classno**(D) $(=$ **qfbclassno**$(D))$, **classno2**(D) $(=$ **qfbclassno**$(D, 1))$, and finally we have the function **hclassno**(D) which computes the class number of an imaginary quadratic field by counting reduced forms, an $O(|D|)$ algorithm. See also `qfbhclassno`.

3.4.47 qfbcompraw(x, y) composition of the binary quadratic forms x and y, without reduction of the result. This is useful e.g. to compute a generating element of an ideal.

The library syntax is **compraw**(x, y).

3.4.48 qfbhclassno(x): Hurwitz class number of x, where x is non-negative and congruent to 0 or 3 modulo 4. For $x > 5 \cdot 10^5$, we assume the GRH, and use `quadclassunit` with default parameters.

The library syntax is **hclassno**(x).

3.4.49 qfbnucomp(x, y, l): composition of the primitive positive definite binary quadratic forms x and y (type `t_QFI`) using the NUCOMP and NUDUPL algorithms of Shanks, à la Atkin. l is any positive constant, but for optimal speed, one should take $l = |D|^{1/4}$, where D is the common discriminant of x and y. When x and y do not have the same discriminant, the result is undefined.

The current implementation is straightforward and in general *slower* than the generic routine (since the latter take advantadge of asymptotically fast operations and careful optimizations).

The library syntax is **nucomp**(x, y, l). The auxiliary function **nudupl**(x, l) can be used when $x = y$.

3.4.50 qfbnupow(x, n): n-th power of the primitive positive definite binary quadratic form x using Shanks's NUCOMP and NUDUPL algorithms (see `qfbnucomp`, in particular the final warning).

The library syntax is **nupow**(x, n).

3.4.51 qfbpowraw(x, n): n-th power of the binary quadratic form x, computed without doing any reduction (i.e. using `qfbcompraw`). Here n must be non-negative and $n < 2^{31}$.

The library syntax is **powraw**(x, n) where n must be a `long` integer.

3.4.52 qfbprimeform(x, p): prime binary quadratic form of discriminant x whose first coefficient is the prime number p. By abuse of notation, $p = \pm 1$ is a valid special case which returns the unit form. Returns an error if x is not a quadratic residue mod p. In the case where $x > 0$, $p < 0$ is allowed, and the "distance" component of the form is set equal to zero according to the current precision. (Note that negative definite `t_QFI` are not implemented.)

The library syntax is **primeform**$(x, p, prec)$, where the third variable *prec* is a `long`, but is only taken into account when $x > 0$.

3.4.53 qfbred$(x, \{flag = 0\}, \{D\}, \{isqrtD\}, \{sqrtD\})$: reduces the binary quadratic form x (updating Shanks's distance function if x is indefinite). The binary digits of *flag* are toggles meaning

1: perform a single reduction step

2: don't update Shanks's distance

D, *isqrtD*, *sqrtD*, if present, supply the values of the discriminant, $\left\lfloor \sqrt{D} \right\rfloor$, and $\sqrt{D}$ respectively (no checking is done of these facts). If $D < 0$ these values are useless, and all references to Shanks's distance are irrelevant.

The library syntax is **qfbred0**$(x, flag, D, isqrtD, sqrtD)$. Use `NULL` to omit any of D, *isqrtD*, *sqrtD*.

Also available are

redimag(x) (= **qfbred**(x) where x is definite),

and for indefinite forms:

redreal(x) (= **qfbred**(x)),

rhoreal(x) (= **qfbred**$(x, 1)$),

redrealnod(x, sq) (= **qfbred**$(x, 2, , isqrtD)$),

rhorealnod(x, sq) (= **qfbred**$(x, 3, , isqrtD)$).

3.4.54 qfbsolve(Q, p): Solve the equation $Q(x, y) = p$ over the integers, where Q is a binary quadratic form and p a prime number.

Return $[x, y]$ as a two-components vector, or zero if there is no solution. Note that this function returns only one solution and not all the solutions.

Let $D = \mathrm{disc}Q$. The algorithm used runs in probabilistic polynomial time in p (through the computation of a square root of D modulo p); it is polynomial time in D if Q is imaginary, but exponential time if Q is real (through the computation of a full cycle of reduced forms). In the latter case, note that `bnfisprincipal` provides a solution in heuristic subexponential time in D assuming the GRH.

The library syntax is **qfbsolve**(Q, n).

3.4.55 quadclassunit$(D, \{flag = 0\}, \{tech = []\})$: Buchmann-McCurley's sub-exponential algorithm for computing the class group of a quadratic order of discriminant D.

This function should be used instead of `qfbclassno` or `quadregula` when $D < -10^{25}$, $D > 10^{10}$, or when the *structure* is wanted. It is a special case of `bnfinit`, which is slower, but more robust.

If *flag* is non-zero *and* $D > 0$, computes the narrow class group and regulator, instead of the ordinary (or wide) ones. In the current version 2.3.5, this does not work at all: use the general function `bnfnarrow`.

Optional parameter *tech* is a row vector of the form $[c_1, c_2]$, where $c_1 \leq c_2$ are positive real numbers which control the execution time and the stack size. For a given c_1, set $c_2 = c_1$ to get maximum speed. To get a rigorous result under GRH, you must take $c_2 \geq 6$. Reasonable values for c_1 are between 0.1 and 2. More precisely, the algorithm will *assume* that prime ideals of norm less than $c_2(\log|D|)^2$ generate the class group, but the bulk of the work is done with prime ideals of norm less than $c_1(\log|D|)^2$. A larger c_1 means that relations are easier to find, but more relations are needed and the linear algebra will be harder. The default is $c_1 = c_2 = 0.2$, so the result is *not* rigorously proven.

The result is a vector v with 3 components if $D < 0$, and 4 otherwise. The correspond respectively to

- $v[1]$: the class number
- $v[2]$: a vector giving the structure of the class group as a product of cyclic groups;
- $v[3]$: a vector giving generators of those cyclic groups (as binary quadratic forms).
- $v[4]$: (omitted if $D < 0$) the regulator, computed to an accuracy which is the maximum of an internal accuracy determined by the program and the current default (note that once the regulator is known to a small accuracy it is trivial to compute it to very high accuracy, see the tutorial).

The library syntax is **quadclassunit0**$(D, flag, tech)$. Also available are **buchimag**(D, c_1, c_2) and **buchreal**$(D, flag, c_1, c_2)$.

3.4.56 quaddisc(x): discriminant of the quadratic field $\mathbf{Q}(\sqrt{x})$, where $x \in \mathbf{Q}$.

The library syntax is **quaddisc**(x).

3.4.57 quadhilbert$(D, \{pq\})$: relative equation defining the Hilbert class field of the quadratic field of discriminant D.

If $D < 0$, uses complex multiplication (Schertz's variant). The technical component pq, if supplied, is a vector $[p, q]$ where p, q are the prime numbers needed for the Schertz's method. More precisely, prime ideals above p and q should be non-principal and coprime to all reduced representatives of the class group. In addition, if one of these ideals has order 2 in the class group, they should have the same class. Finally, for efficiency, $\gcd(24, (p-1)(q-1))$ should be as large as possible. The routine returns 0 if $[p, q]$ is not suitable.

If $D > 0$ Stark units are used and (in rare cases) a vector of extensions may be returned whose compositum is the requested class field. See `bnrstark` for details.

The library syntax is **quadhilbert**$(D, pq, prec)$.

3.4.58 quadgen(D): creates the quadratic number $\omega = (a + \sqrt{D})/2$ where $a = 0$ if $x \equiv 0 \bmod 4$, $a = 1$ if $D \equiv 1 \bmod 4$, so that $(1, \omega)$ is an integral basis for the quadratic order of discriminant D. D must be an integer congruent to 0 or 1 modulo 4, which is not a square.

The library syntax is **quadgen**(x).

3.4.59 quadpoly$(D, \{v = x\})$: creates the "canonical" quadratic polynomial (in the variable v) corresponding to the discriminant D, i.e. the minimal polynomial of `quadgen`(D). D must be an integer congruent to 0 or 1 modulo 4, which is not a square.

The library syntax is **quadpoly0**(x, v).

3.4.60 quadray$(D, f, \{lambda\})$: relative equation for the ray class field of conductor f for the quadratic field of discriminant D using analytic methods. A `bnf` for $x^2 - D$ is also accepted in place of D.

For $D < 0$, uses the σ function. If supplied, *lambda* is is the technical element λ of `bnf` necessary for Schertz's method. In that case, returns 0 if λ is not suitable.

For $D > 0$, uses Stark's conjecture, and a vector of relative equations may be returned. See `bnrstark` for more details.

The library syntax is **quadray**$(D, f, lambda, prec)$, where an omitted `lambda` is coded as `NULL`.

3.4.61 quadregulator(x): regulator of the quadratic field of positive discriminant x. Returns an error if x is not a discriminant (fundamental or not) or if x is a square. See also `quadclassunit` if x is large.

The library syntax is **regula**$(x, prec)$.

3.4.62 quadunit(D): fundamental unit of the real quadratic field $\mathbf{Q}(\sqrt{D})$ where D is the positive discriminant of the field. If D is not a fundamental discriminant, this probably gives the fundamental unit of the corresponding order. D must be an integer congruent to 0 or 1 modulo 4, which is not a square; the result is a quadratic number (see Section 3.4.58).

The library syntax is **fundunit**(x).

3.4.63 removeprimes($\{x = []\}$): removes the primes listed in x from the prime number table. In particular `removeprimes(addprimes)` empties the extra prime table. x can also be a single integer. List the current extra primes if x is omitted.

The library syntax is **removeprimes**(x).

3.4.64 sigma($x, \{k = 1\}$): sum of the k^{th} powers of the positive divisors of $|x|$. x and k must be of type integer.

The library syntax is **sumdiv**(x) (= **sigma**(x)) or **gsumdivk**(x, k) (= **sigma**(x, k)), where k is a C long integer.

3.4.65 sqrtint(x): integer square root of x, which must be a non-negative integer. The result is non-negative and rounded towards zero.

The library syntax is **sqrti**(x). Also available is `sqrtremi`($x, \&r$) which returns s such that $s^2 = x + r$, with $0 \le r \le 2s$.

3.4.66 zncoppersmith($P, N, X, \{B = N\}$): finds all integers x_0 with $|x_0| \le X$ such that

$$\gcd(N, P(x_0)) \ge B.$$

If N is prime or a prime power, `polrootsmod` or `polrootspadic` will be much faster. X must be smaller than $\exp(\log^2 B/(\deg(P) \log N))$.

The library syntax is **zncoppersmith**(P, N, X, B), where an omitted B is coded as `NULL`.

3.4.67 znlog(x, g): g must be a primitive root mod a prime p, and the result is the discrete log of x in the multiplicative group $(\mathbf{Z}/p\mathbf{Z})^*$. This function uses a simple-minded combination of Pohlig-Hellman algorithm and Shanks baby-step/giant-step which requires $O(\sqrt{q})$ storage, where q is the largest prime factor of $p - 1$. Hence it cannot be used when the largest prime divisor of $p - 1$ is greater than about 10^{13}.

The library syntax is **znlog**(x, g).

3.4.68 znorder($x, \{o\}$): x must be an integer mod n, and the result is the order of x in the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$. Returns an error if x is not invertible. If optional parameter o is given it is assumed to be a multiple of the order (used to limit the search space).

The library syntax is **znorder**(x, o), where an omitted o is coded as `NULL`. Also available is **order**(x).

3.4.69 znprimroot(n): returns a primitive root (generator) of $(\mathbf{Z}/n\mathbf{Z})^*$, whenever this latter group is cyclic ($n = 4$ or $n = 2p^k$ or $n = p^k$, where p is an odd prime and $k \ge 0$).

The library syntax is **gener**(x).

3.4.70 znstar(n): gives the structure of the multiplicative group $(\mathbf{Z}/n\mathbf{Z})^*$ as a 3-component row vector v, where $v[1] = \phi(n)$ is the order of that group, $v[2]$ is a k-component row-vector d of integers $d[i]$ such that $d[i] > 1$ and $d[i] \mid d[i-1]$ for $i \ge 2$ and $(\mathbf{Z}/n\mathbf{Z})^* \simeq \prod_{i=1}^{k} (\mathbf{Z}/d[i]\mathbf{Z})$, and $v[3]$ is a k-component row vector giving generators of the image of the cyclic groups $\mathbf{Z}/d[i]\mathbf{Z}$.

The library syntax is **znstar**(n).

3.5 Functions related to elliptic curves.

We have implemented a number of functions which are useful for number theorists working on elliptic curves. We always use Tate's notations. The functions assume that the curve is given by a general Weierstrass model

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where a priori the a_i can be of any scalar type. This curve can be considered as a five-component vector `E=[a1,a2,a3,a4,a6]`. Points on `E` are represented as two-component vectors `[x,y]`, except for the point at infinity, i.e. the identity element of the group law, represented by the one-component vector `[0]`.

It is useful to have at one's disposal more information. This is given by the function `ellinit` (see there), which initalizes and returns an *ell* structure by default. If a specific flag is added, a shortened *sell*, for small *ell*, is returned, which is much faster to compute but contains less information. The following member functions are available to deal with the output of `ellinit`, both *ell* and *sell*:

`a1`–`a6`, `b2`–`b8`, `c4`–`c6` : coefficients of the elliptic curve.
`area` : volume of the complex lattice defining E.
`disc` : discriminant of the curve.
`j` : j-invariant of the curve.
`omega` : $[\omega_1, \omega_2]$, periods forming a basis of the complex lattice defining E (ω_1 is the real period, and ω_2/ω_1 belongs to Poincaré's half-plane).
`eta` : quasi-periods $[\eta_1, \eta_2]$, such that $\eta_1\omega_2 - \eta_2\omega_1 = i\pi$.
`roots` : roots of the associated Weierstrass equation.
`tate` : $[u^2, u, v]$ in the notation of Tate.
`w` : Mestre's w (this is technical).

The member functions `area`, `eta` and `omega` are only available for curves over $\mathbf{Q}$. Conversely, `tate` and `w` are only available for curves defined over $\mathbf{Q}_p$. The use of member functions is best described by an example:

```
? E = ellinit([0,0,0,0,1]); \\ The curve y^2 = x^3 + 1
? E.a6
%2 = 1
? E.c6
%3 = -864
? E.disc
%4 = -432
```

Some functions, in particular those relative to height computations (see `ellheight`) require also that the curve be in minimal Weierstrass form, which is duly stressed in their description below. This is achieved by the function `ellminimalmodel`. *Using a non-minimal model in such a routine will yield a wrong result!*

All functions related to elliptic curves share the prefix `ell`, and the precise curve we are interested in is always the first argument, in either one of the three formats discussed above, unless otherwise specified. The requirements are given as the *minimal* ones: any richer structure may replace the ones requested. For instance, in functions which have no use for the extra information given by an *ell* structure, the curve can be given either as a five-component vector, as an *sell*, or as an *ell*; if an *sell* is requested, an *ell* may equally be given.

3.5.1 elladd$(E, z1, z2)$: sum of the points $z1$ and $z2$ on the elliptic curve corresponding to E.

The library syntax is **addell**$(E, z1, z2)$.

3.5.2 ellak(E, n): computes the coefficient a_n of the L-function of the elliptic curve E, i.e. in principle coefficients of a newform of weight 2 assuming Taniyama-Weil conjecture (which is now known to hold in full generality thanks to the work of Breuil, Conrad, Diamond, Taylor and Wiles). E must be an *sell* as output by `ellinit`. For this function to work for every n and not just those prime to the conductor, E must be a minimal Weierstrass equation. If this is not the case, use the function `ellminimalmodel` before using `ellak`.

The library syntax is **akell**(E, n).

3.5.3 ellan(E, n): computes the vector of the first n a_k corresponding to the elliptic curve E. All comments in `ellak` description remain valid.

The library syntax is **anell**(E, n), where n is a C integer.

3.5.4 ellap$(E, p, \{flag = 0\})$: computes the a_p corresponding to the elliptic curve E and the prime number p. These are defined by the equation $\#E(\mathbf{F}_p) = p + 1 - a_p$, where $\#E(\mathbf{F}_p)$ stands for the number of points of the curve E over the finite field $\mathbf{F}_p$. When *flag* is 0, this uses the baby-step giant-step method and a trick due to Mestre. This runs in time $O(p^{1/4})$ and requires $O(p^{1/4})$ storage, hence becomes unreasonable when p has about 30 digits.

If *flag* is 1, computes the a_p as a sum of Legendre symbols. This is slower than the previous method as soon as p is greater than 100, say.

No checking is done that p is indeed prime. E must be an *sell* as output by `ellinit`, defined over $\mathbf{Q}$, $\mathbf{F}_p$ or $\mathbf{Q}_p$. E must be given by a Weierstrass equation minimal at p.

The library syntax is **ellap0**$(E, p, flag)$. Also available are **apell**(E, p), corresponding to $flag = 0$, and **apell2**(E, p) ($flag = 1$).

3.5.5 ellbil$(E, z1, z2)$: if $z1$ and $z2$ are points on the elliptic curve E, assumed to be integral given by a minimal model, this function computes the value of the canonical bilinear form on $z1$, $z2$:

$$(h(E, z1+z2) - h(E, z1) - h(E, z2))/2$$

where `+` denotes of course addition on E. In addition, $z1$ or $z2$ (but not both) can be vectors or matrices.

The library syntax is **bilhell**$(E, z1, z2, prec)$.

3.5.6 ellchangecurve(E, v): changes the data for the elliptic curve E by changing the coordinates using the vector `v=[u,r,s,t]`, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$. E must be an *sell* as output by `ellinit`.

The library syntax is **coordch**(E, v).

3.5.7 ellchangepoint(x, v): changes the coordinates of the point or vector of points x using the vector `v=[u,r,s,t]`, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$ (see also `ellchangecurve`).

The library syntax is **pointch**(x, v).

3.5.8 ellconvertname(*name*): converts an elliptic curve name, as found in the `elldata` database, from a string to a triplet [*conductor*, *isogeny class*, *index*]. It will also convert a triplet back to a curve name. Examples:

```
? ellconvertname("123b1")
%1 = [123, 1, 1]
? ellconvertname(%)
%2 = "123b1"
```

The library syntax is **ellconvertname**(*name*).

3.5.9 elleisnum($E, k, \{flag = 0\}$): E being an elliptic curve as output by `ellinit` (or, alternatively, given by a 2-component vector $[\omega_1, \omega_2]$ representing its periods), and k being an even positive integer, computes the numerical value of the Eisenstein series of weight k at E, namely

$$(2i\pi/\omega_2)^k\Big(1 + 2/\zeta(1-k)\sum_{n\geq 0} n^{k-1}q^n/(1-q^n)\Big),$$

where $q = e(\omega_1/\omega_2)$.

When *flag* is non-zero and $k = 4$ or 6, returns the elliptic invariants g_2 or g_3, such that

$$y^2 = 4x^3 - g_2x - g_3$$

is a Weierstrass equation for E.

The library syntax is **elleisnum**($E, k, flag$).

3.5.10 elleta(*om*): returns the two-component row vector $[\eta_1, \eta_2]$ of quasi-periods associated to $\mathtt{om} = [\omega_1, \omega_2]$

The library syntax is **elleta**(*om*, *prec*)

3.5.11 ellgenerators(E): returns a $\mathbf{Z}$-basis of the free part of the Mordell-Weil group associated to E. This function depends on the `elldata` database being installed and referencing the curve, and so is only available for curves over $\mathbf{Z}$ of small conductors.

The library syntax is **ellgenerators**(E).

3.5.12 ellglobalred(E): calculates the arithmetic conductor, the global minimal model of E and the global Tamagawa number c. E must be an *sell* as output by `ellinit`, *and is supposed to have all its coefficients* a_i *in* $\mathbf{Q}$. The result is a 3 component vector $[N, v, c]$. N is the arithmetic conductor of the curve. v gives the coordinate change for E over $\mathbf{Q}$ to the minimal integral model (see `ellminimalmodel`). Finally c is the product of the local Tamagawa numbers c_p, a quantity which enters in the Birch and Swinnerton-Dyer conjecture.

The library syntax is **ellglobalred**(E).

3.5.13 ellheight$(E, z, \{flag = 2\})$: global N"'eron-Tate height of the point z on the elliptic curve E (defined over $\mathbf{Q}$), given by a standard minimal integral model. E must be an `ell` as output by `ellinit`. *flag*selects the algorithm used to compute the archimedean local height. If $flag = 0$, this computation is done using sigma and theta-functions and a trick due to J. Silverman. If $flag = 1$, use Tate's 4^n algorithm. If $flag = 2$, use Mestre's AGM algorithm. The latter is much faster than the other two, both in theory (converges quadratically) and in practice.

The library syntax is **ellheight0**$(E, z, flag, prec)$. Also available are **ghell**$(E, z, prec)$ $(flag = 0)$ and **ghell2**$(E, z, prec)$ $(flag = 1)$.

3.5.14 ellheightmatrix(E, x): x being a vector of points, this function outputs the Gram matrix of x with respect to the Néron-Tate height, in other words, the (i, j) component of the matrix is equal to `ellbil(`E`,x[`i`],x[`j`])`. The rank of this matrix, at least in some approximate sense, gives the rank of the set of points, and if x is a basis of the Mordell-Weil group of E, its determinant is equal to the regulator of E. Note that this matrix should be divided by 2 to be in accordance with certain normalizations. E is assumed to be integral, given by a minimal model.

The library syntax is **mathell**$(E, x, prec)$.

3.5.15 ellidentify(E): look up the elliptic curve E (over $\mathbf{Z}$) in the `elldata` database and return `[[N, M, G], C]` where N is the name of the curve in J. E. Cremona database, M the minimal model, G a $\mathbf{Z}$-basis of the free part of the Mordell-Weil group of E and C the coordinates change (see `ellchangecurve`).

The library syntax is **ellidentify**(E).

3.5.16 ellinit$(E, \{flag = 0\})$: initialize an `ell` structure, associated to the elliptic curve E. E is a 5-component vector $[a_1, a_2, a_3, a_4, a_6]$ defining the elliptic curve with Weierstrass equation

$$Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6$$

or a string, in this case the coefficients of the curve with matching name are looked in the `elldata` database if available. For the time being, only curves over a prime field $\mathbf{F}_p$ and over the p-adic or real numbers (including rational numbers) are fully supported. Other domains are only supported for very basic operations such as point addition.

The result of `ellinit` is a an *ell* structure by default, and a shorted *sell* if $flag = 1$. Both contain the following information in their components:

$$a_1, a_2, a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta, j.$$

All are accessible via member functions. In particular, the discriminant is E`.disc`, and the j-invariant is E`.j`.

The other six components are only present if *flag* is 0 or omitted. Their content depends on whether the curve is defined over $\mathbf{R}$ or not:

• When E is defined over $\mathbf{R}$, E`.roots` is a vector whose three components contain the roots of the right hand side of the associated Weierstrass equation.

$$(y + a_1x/2 + a_3/2)^2 = g(x)$$

If the roots are all real, then they are ordered by decreasing value. If only one is real, it is the first component.

Then $\omega_1 =$`E.omega[1]` is the real period of E (integral of $dx/(2y+a_1x+a_3)$ over the connected component of the identity element of the real points of the curve), and $\omega_2 =$`E.omega[2]` is a complex period. In other words, `E.omega` forms a basis of the complex lattice defining E, with $\tau = \frac{\omega_2}{\omega_1}$ having positive imaginary part.

`E.eta` is a row vector containing the corresponding values η_1 and η_2 such that $\eta_1\omega_2 - \eta_2\omega_1 = i\pi$.

Finally, `E.area` is the volume of the complex lattice defining E.

• When E is defined over $\mathbf{Q}_p$, the p-adic valuation of j must be negative. Then `E.roots` is the vector with a single component equal to the p-adic root of the associated Weierstrass equation corresponding to -1 under the Tate parametrization.

`E.tate` yields the three-component vector $[u^2, u, q]$, in the notations of Tate. If the u-component does not belong to $\mathbf{Q}_p$, it is set to zero.

`E.w` is Mestre's w (this is technical).

For all other base fields or rings, the last six components are arbitrarily set equal to zero. See also the description of member functions related to elliptic curves at the beginning of this section.

The library syntax is **ellinit0**$(E, \mathit{flag}, \mathit{prec})$. Also available are **initell**(E, prec) ($\mathit{flag} = 0$) and **smallinitell**(E, prec) ($\mathit{flag} = 1$).

3.5.17 ellisoncurve(E, z): gives 1 (i.e. true) if the point z is on the elliptic curve E, 0 otherwise. If E or z have imprecise coefficients, an attempt is made to take this into account, i.e. an imprecise equality is checked, not a precise one. It is allowed for z to be a vector of points in which case a vector (of the same type) is returned.

The library syntax is **ellisoncurve**(E, z). Also available is **oncurve**(E, z) which returns a `long` but does not accept vector of points.

3.5.18 ellj(x): elliptic j-invariant. x must be a complex number with positive imaginary part, or convertible into a power series or a p-adic number with positive valuation.

The library syntax is **jell**(x, prec).

3.5.19 elllocalred(E, p): calculates the Kodaira type of the local fiber of the elliptic curve E at the prime p. E must be an *sell* as output by `ellinit`, and is assumed to have all its coefficients a_i in $\mathbf{Z}$. The result is a 4-component vector $[f, \mathit{kod}, v, c]$. Here f is the exponent of p in the arithmetic conductor of E, and *kod* is the Kodaira type which is coded as follows:

1 means good reduction (type I_0), 2, 3 and 4 mean types II, III and IV respectively, $4 + \nu$ with $\nu > 0$ means type I_ν; finally the opposite values -1, -2, etc. refer to the starred types I_0^*, II^*, etc. The third component v is itself a vector $[u, r, s, t]$ giving the coordinate changes done during the local reduction. Normally, this has no use if u is 1, that is, if the given equation was already minimal. Finally, the last component c is the local Tamagawa number c_p.

The library syntax is **elllocalred**(E, p).

3.5.20 elllseries$(E, s, \{A = 1\})$: E being an *sell* as output by `ellinit`, this computes the value of the L-series of E at s. It is assumed that E is defined over $\mathbf{Q}$, not necessarily minimal. The optional parameter A is a cutoff point for the integral, which must be chosen close to 1 for best speed. The result must be independent of A, so this allows some internal checking of the function.

Note that if the conductor of the curve is large, say greater than 10^{12}, this function will take an unreasonable amount of time since it uses an $O(N^{1/2})$ algorithm.

The library syntax is **elllseries**$(E, s, A, prec)$ where *prec* is a `long` and an omitted A is coded as `NULL`.

3.5.21 ellminimalmodel$(E, \{\&v\})$: return the standard minimal integral model of the rational elliptic curve E. If present, sets v to the corresponding change of variables, which is a vector $[u, r, s, t]$ with rational components. The return value is identical to that of `ellchangecurve(E, v)`.

The resulting model has integral coefficients, is everywhere minimal, a_1 is 0 or 1, a_2 is 0, 1 or -1 and a_3 is 0 or 1. Such a model is unique, and the vector v is unique if we specify that u is positive, which we do.

The library syntax is **ellminimalmodel**$(E, \&v)$, where an omitted v is coded as `NULL`.

3.5.22 ellorder(E, z): gives the order of the point z on the elliptic curve E if it is a torsion point, zero otherwise. In the present version 2.3.5, this is implemented only for elliptic curves defined over $\mathbf{Q}$.

The library syntax is **orderell**(E, z).

3.5.23 ellordinate(E, x): gives a 0, 1 or 2-component vector containing the y-coordinates of the points of the curve E having x as x-coordinate.

The library syntax is **ordell**(E, x).

3.5.24 ellpointtoz(E, z): if E is an elliptic curve with coefficients in $\mathbf{R}$, this computes a complex number t (modulo the lattice defining E) corresponding to the point z, i.e. such that, in the standard Weierstrass model, $\wp(t) = z[1], \wp'(t) = z[2]$. In other words, this is the inverse function of `ellztopoint`. More precisely, if $(w1, w2)$ are the real and complex periods of E, t is such that $0 \leq \Re(t) < w1$ and $0 \leq \Im(t) < \Im(w2)$.

If E has coefficients in $\mathbf{Q}_p$, then either Tate's u is in $\mathbf{Q}_p$, in which case the output is a p-adic number t corresponding to the point z under the Tate parametrization, or only its square is, in which case the output is $t + 1/t$. E must be an *ell* as output by `ellinit`.

The library syntax is **zell**$(E, z, prec)$.

3.5.25 ellpow(E, z, n): computes n times the point z for the group law on the elliptic curve E. Here, n can be in $\mathbf{Z}$, or n can be a complex quadratic integer if the curve E has complex multiplication by n (if not, an error message is issued).

The library syntax is **powell**(E, z, n).

3.5.26 ellrootno$(E, \{p = 1\})$: E being an *sell* as output by `ellinit`, this computes the local (if $p \neq 1$) or global (if $p = 1$) root number of the L-series of the elliptic curve E. Note that the global root number is the sign of the functional equation and conjecturally is the parity of the rank of the Mordell-Weil group. The equation for E must have coefficients in $\mathbf{Q}$ but need *not* be minimal.

The library syntax is **ellrootno**(E, p) and the result (equal to ± 1) is a `long`.

3.5.27 ellsigma$(E, z, \{flag = 0\})$: value of the Weierstrass σ function of the lattice associated to E as given by `ellinit` (alternatively, E can be given as a lattice $[\omega_1, \omega_2]$).

If $flag = 1$, computes an (arbitrary) determination of $\log(\sigma(z))$.

If $flag = 2, 3$, same using the product expansion instead of theta series. The library syntax is **ellsigma**$(E, z, flag)$

3.5.28 ellsearch(N): if N is an integer, it is taken as a conductor else if N is a string, it can be a curve name ("11a1"), a isogeny class ("11a") or a conductor "11". This function finds all curves in the `elldata` database with the given property.

If N is a full curve name, the output format is $[N, [a_1, a_2, a_3, a_4, a_6], G]$ where $[a_1, a_2, a_3, a_4, a_6]$ are the coefficients of the Weierstrass equation of the curve and G is a $\mathbf{Z}$-basis of the free part of the Mordell-Weil group associated to the curve.

If N is not a full-curve name, the output is the list (as a vector) of all matching curves in the above format.

The library syntax is **ellsearch**(N). Also available is **ellsearchcurve**(N) that only accept complete curve names.

3.5.29 ellsub$(E, z1, z2)$: difference of the points $z1$ and $z2$ on the elliptic curve corresponding to E.

The library syntax is **subell**$(E, z1, z2)$.

3.5.30 elltaniyama(E): computes the modular parametrization of the elliptic curve E, where E is an *sell* as output by `ellinit`, in the form of a two-component vector $[u, v]$ of power series, given to the current default series precision. This vector is characterized by the following two properties. First the point $(x, y) = (u, v)$ satisfies the equation of the elliptic curve. Second, the differential $du/(2v + a_1u + a_3)$ is equal to $f(z)dz$, a differential form on $H/\Gamma_0(N)$ where N is the conductor of the curve. The variable used in the power series for u and v is x, which is implicitly understood to be equal to $\exp(2i\pi z)$. It is assumed that the curve is a *strong* Weil curve, and that the Manin constant is equal to 1. The equation of the curve E must be minimal (use `ellminimalmodel` to get a minimal equation).

The library syntax is **elltaniyama**$(E, prec)$, and the precision of the result is determined by `prec`.

3.5.31 elltors$(E, \{flag = 0\})$: if E is an elliptic curve *defined over* $\mathbf{Q}$, outputs the torsion subgroup of E as a 3-component vector `[t,v1,v2]`, where `t` is the order of the torsion group, `v1` gives the structure of the torsion group as a product of cyclic groups (sorted by decreasing order), and `v2` gives generators for these cyclic groups. E must be an *ell* as output by `ellinit`.

```
? E = ellinit([0,0,0,-1,0]);
? elltors(E)
%1 = [4, [2, 2], [[0, 0], [1, 0]]]
```

Here, the torsion subgroup is isomorphic to $\mathbf{Z}/2\mathbf{Z} \times \mathbf{Z}/2\mathbf{Z}$, with generators $[0,0]$ and $[1,0]$.

If *flag* $= 0$, use Doud's algorithm: bound torsion by computing $\#E(\mathbf{F}_p)$ for small primes of good reduction, then look for torsion points using Weierstrass parametrization (and Mazur's classification).

If *flag* $= 1$, use Lutz-Nagell (*much* slower), E is allowed to be an *sell*.

The library syntax is **elltors0**$(E, flag)$.

3.5.32 ellwp$(E, \{z = x\}, \{flag = 0\})$:

Computes the value at z of the Weierstrass $\wp$ function attached to the elliptic curve E as given by `ellinit` (alternatively, E can be given as a lattice $[\omega_1, \omega_2]$).

If z is omitted or is a simple variable, computes the *power series* expansion in z (starting $z^{-2} + O(z^2)$). The number of terms to an *even* power in the expansion is the default serieslength in gp, and the second argument (C long integer) in library mode.

Optional *flag* is (for now) only taken into account when z is numeric, and means 0: compute only $\wp(z)$, 1: compute $[\wp(z), \wp'(z)]$.

The library syntax is **ellwp0**$(E, z, flag, prec, precdl)$. Also available is **weipell**$(E, precdl)$ for the power series.

3.5.33 ellzeta(E, z): value of the Weierstrass ζ function of the lattice associated to E as given by `ellinit` (alternatively, E can be given as a lattice $[\omega_1, \omega_2]$).

The library syntax is **ellzeta**(E, z).

3.5.34 ellztopoint(E, z): E being an *ell* as output by `ellinit`, computes the coordinates $[x, y]$ on the curve E corresponding to the complex number z. Hence this is the inverse function of `ellpointtoz`. In other words, if the curve is put in Weierstrass form, $[x, y]$ represents the Weierstrass \$"wp\$-function and its derivative. If z is in the lattice defining E over $\mathbf{C}$, the result is the point at infinity [0].

The library syntax is **pointell**$(E, z, prec)$.

3.6 Functions related to general number fields.

In this section can be found functions which are used almost exclusively for working in general number fields. Other less specific functions can be found in the next section on polynomials. Functions related to quadratic number fields are found in section Section 3.4 (Arithmetic functions).

3.6.1 Number field structures

Let $K = \mathbf{Q}[X]/(T)$ a number field, $\mathbf{Z}_K$ its ring of integers, $T \in \mathbf{Z}[X]$ is monic. Three basic number field structures can be associated to K in GP:

• *nf* denotes a number field, i.e. a data structure output by `nfinit`. This contains the basic arithmetic data associated to the number field: signature, maximal order (given by a basis `nf.zk`), discriminant, defining polynomial T, etc.

• *bnf* denotes a "Buchmann's number field", i.e. a data structure output by `bnfinit`. This contains *nf* and the deeper invariants of the field: units $U(K)$, class group $\mathrm{Cl}(K)$, as well as technical data required to solve the two associated discrete logarithm problems.

• *bnr* denotes a "ray number field", i.e. a data structure output by `bnrinit`, corresponding to the ray class group structure of the field, for some modulus f. It contains a *bnf*, the modulus f, the ray class group $\mathrm{Cl}_f(K)$ and data associated to the discrete logarithm problem therein.

3.6.2 Algebraic numbers and ideals

An *algebraic number* belonging to $K = \mathbf{Q}[X]/(T)$ is given as

• a `t_INT`, `t_FRAC` or `t_POL` (implicitly modulo T), or

• a `t_POLMOD` (modulo T), or

• a `t_COL v` of dimension $N = [K : \mathbf{Q}]$, representing the element in terms of the computed integral basis, as `sum(i = 1, N, v[i] * nf.zk[i])`. Note that a `t_VEC` will not be recognized.

An *ideal* is given in any of the following ways:

• an algebraic number in one of the above forms, defining a principal ideal.

• a prime ideal, i.e. a 5-component vector in the format output by `idealprimedec`.

• a `t_MAT`, square and in Hermite Normal Form (or at least upper triangular with non-negative coefficients), whose columns represent a basis of the ideal.

One may use `idealhnf` to convert an ideal to the last (preferred) format.

Note. Some routines accept non-square matrices, but using this format is strongly discouraged. Nevertheless, their behaviour is as follows: If strictly less than $N = [K : \mathbf{Q}]$ generators are given, it is assumed they form a $\mathbf{Z}_K$-basis. If N or more are given, a $\mathbf{Z}$-basis is assumed. If exactly N are given, it is further assumed the matrix is in HNF. If any of these assumptions is not correct the behaviour of the routine is undefined.

• an *idele* is a 2-component vector, the first being an ideal as above, the second being a $R_1 + R_2$-component row vector giving Archimedean information, as complex numbers.

3.6.3 Finite abelian groups

A finite abelian group G in user-readable format is given by its Smith Normal Form as a pair $[h, d]$ or triple $[h, d, g]$. Here h is the cardinality of G, (d_i) is the vector of elementary divisors, and (g_i) is a vector of generators. In short, $G = \oplus_{i \leq n}(\mathbf{Z}/d_i\mathbf{Z})g_i$, with $d_n \mid \ldots \mid d_2 \mid d_1$ and $\prod d_i = h$. This information can also be retrieved as G.`no`, G.`cyc` and G.`gen`.

• a *character* on the abelian group $\oplus(\mathbf{Z}/d_i\mathbf{Z})g_i$ is given by a row vector $\chi = [a_1, \ldots, a_n]$ such that $\chi(\prod g_i^{n_i}) = \exp(2i\pi \sum a_i n_i/d_i)$.

• given such a structure, a *subgroup* H is input as a square matrix, whose column express generators of H on the given generators g_i. Note that the absolute value of the determinant of that matrix is equal to the index $(G : H)$.

3.6.4 Relative extensions

When defining a relative extension, the base field *nf* must be defined by a variable having a lower priority (see Section 2.5.4) than the variable defining the extension. For example, you may use the variable name y to define the base field, and x to define the relative extension.

• *rnf* denotes a relative number field, i.e. a data structure output by `rnfinit`.

• A *relative matrix* is a matrix whose entries are elements of a (fixed) number field *nf*, always expressed as column vectors on the integral basis *nf*`.zk`. Hence it is a matrix of vectors.

• An *ideal list* is a row vector of (fractional) ideals of the number field *nf*.

• A *pseudo-matrix* is a pair (A, I) where A is a relative matrix and I an ideal list whose length is the same as the number of columns of A. This pair is represented by a 2-component row vector.

• The *projective module* generated by a pseudo-matrix (A, I) is the sum $\sum_i \mathbf{a}_j A_j$ where the $\mathbf{a}_j$ are the ideals of I and A_j is the j-th column of A.

• A pseudo-matrix (A, I) is a *pseudo-basis* of the module it generates if A is a square matrix with non-zero determinant and all the ideals of I are non-zero. We say that it is in Hermite Normal Form (HNF) if it is upper triangular and all the elements of the diagonal are equal to 1.

• The *determinant* of a pseudo-basis (A, I) is the ideal equal to the product of the determinant of A by all the ideals of I. The determinant of a pseudo-matrix is the determinant of any pseudo-basis of the module it generates.

3.6.5 Class field theory

A *modulus*, in the sense of class field theory, is a divisor supported on the non-complex places of K. In PARI terms, this means either an ordinary ideal I as above (no archimedean component), or a pair $[I, a]$, where a is a vector with r_1 $\{0, 1\}$-components, corresponding to the infinite part of the divisor. More precisely, the i-th component of a corresponds to the real embedding associated to the i-th real root of `K.roots`. (That ordering is not canonical, but well defined once a defining polynomial for K is chosen.) For instance, `[1, [1,1]]` is a modulus for a real quadratic field, allowing ramification at any of the two places at infinity.

A *bid* or "big ideal" is a structure output by `idealstar` needed to compute in $(\mathbf{Z}_K/I)^*$, where I is a modulus in the above sense. If is a finite abelian group as described above, supplemented by technical data needed to solve discrete log problems.

Finally we explain how to input ray number fields (or *bnr*), using class field theory. These are defined by a triple $a1$, $a2$, $a3$, where the defining set $[a1, a2, a3]$ can have any of the following forms: $[bnr]$, $[bnr, subgroup]$, $[bnf, module]$, $[bnf, module, subgroup]$.

• *bnf* is as output by `bnfinit`, where units are mandatory unless the modulus is trivial; *bnr* is as output by `bnrinit`. This is the ground field K.

• *module* is a modulus $\mathfrak{f}$, as described above.

• *subgroup* a subgroup of the ray class group modulo $\mathfrak{f}$ of K. As described above, this is input as a square matrix expressing generators of a subgroup of the ray class group *bnr*`.clgp` on the given generators.

The corresponding *bnr* is the subfield of the ray class field of K modulo $\mathfrak{f}$, fixed by the given subgroup.

3.6.6 General use

All the functions which are specific to relative extensions, number fields, Buchmann's number fields, Buchmann's number rays, share the prefix `rnf`, `nf`, `bnf`, `bnr` respectively. They take as first argument a number field of that precise type, respectively output by `rnfinit`, `nfinit`, `bnfinit`, and `bnrinit`.

However, and even though it may not be specified in the descriptions of the functions below, it is permissible, if the function expects a *nf*, to use a *bnf* instead, which contains much more information. On the other hand, if the function requires a `bnf`, it will *not* launch `bnfinit` for you, which is a costly operation. Instead, it will give you a specific error message. In short, the types

$$\texttt{nf} \leq \texttt{bnf} \leq \texttt{bnr}$$

are ordered, each function requires a minimal type to work properly, but you may always substitute a larger type.

The data types corresponding to the structures described above are rather complicated. Thus, as we already have seen it with elliptic curves, GP provides "member functions" to retrieve data from these structures (once they have been initialized of course). The relevant types of number fields are indicated between parentheses:

`bid` (*bnr*,) : bid ideal structure.
`bnf` (*bnr*, *bnf*) : Buchmann's number field.
`clgp` (*bnr*, *bnf*) : classgroup. This one admits the following three subclasses:
  `cyc` : cyclic decomposition (SNF).
  `gen` : generators.
  `no` : number of elements.
`diff` (*bnr*, *bnf*, *nf*) : the different ideal.
`codiff` (*bnr*, *bnf*, *nf*) : the codifferent (inverse of the different in the ideal group).
`disc` (*bnr*, *bnf*, *nf*) : discriminant.
`fu` (*bnr*, *bnf*, *nf*) : fundamental units.
`index` (*bnr*, *bnf*, *nf*) : index of the power order in the ring of integers.
`nf` (*bnr*, *bnf*, *nf*) : number field.
`r1` (*bnr*, *bnf*, *nf*) : the number of real embeddings.
`r2` (*bnr*, *bnf*, *nf*) : the number of pairs of complex embeddings.
`reg` (*bnr*, *bnf*,) : regulator.

`roots` (*bnr*, *bnf*, *nf*) : roots of the polynomial generating the field.
`t2` (*bnr*, *bnf*, *nf*) : the T2 matrix (see `nfinit`).
`tu` (*bnr*, *bnf*,) : a generator for the torsion units.
`tufu` (*bnr*, *bnf*,) : $[w, u_1, ..., u_r]$, (u_i) is a vector of fundamental units, w generates the torsion units.
`zk` (*bnr*, *bnf*, *nf*) : integral basis, i.e. a $\mathbf{Z}$-basis of the maximal order.

For instance, assume that *bnf* = `bnfinit`(*pol*), for some polynomial. Then *bnf*`.clgp` retrieves the class group, and *bnf*`.clgp.no` the class number. If we had set *bnf* = `nfinit`(*pol*), both would have output an error message. All these functions are completely recursive, thus for instance *bnr*`.bnf.nf.zk` will yield the maximal order of *bnr*, which you could get directly with a simple *bnr*`.zk`.

3.6.7 Class group, units, and the GRH

Some of the functions starting with `bnf` are implementations of the sub-exponential algorithms for finding class and unit groups under GRH, due to Hafner-McCurley, Buchmann and Cohen-Diaz-Olivier. The general call to the functions concerning class groups of general number fields (i.e. excluding `quadclassunit`) involves a polynomial P and a technical vector

$$tech = [c, c2, nrpid],$$

where the parameters are to be understood as follows:

P is the defining polynomial for the number field, which must be in $\mathbf{Z}[X]$, irreducible and monic. In fact, if you supply a non-monic polynomial at this point, `gp` issues a warning, then *transforms your polynomial* so that it becomes monic. The `nfinit` routine will return a different result in this case: instead of `res`, you get a vector `[res,Mod(a,Q)]`, where `Mod(a,Q) = Mod(X,P)` gives the change of variables. In all other routines, the variable change is simply lost.

The numbers $c \leq c_2$ are positive real numbers which control the execution time and the stack size. For a given c, set $c_2 = c$ to get maximum speed. To get a rigorous result under GRH you must take $c2 \geq 12$ (or $c2 \geq 6$ in P is quadratic). Reasonable values for c are between 0.1 and 2. The default is $c = c_2 = 0.3$.

nrpid is the maximal number of small norm relations associated to each ideal in the factor base. Set it to 0 to disable the search for small norm relations. Otherwise, reasonable values are between 4 and 20. The default is 4.

Warning. Make sure you understand the above! By default, most of the `bnf` routines depend on the correctness of a heuristic assumption which is stronger than the GRH. In particular, any of the class number, class group structure, class group generators, regulator and fundamental units may be wrong, independently of each other. Any result computed from such a `bnf` may be wrong. The only guarantee is that the units given generate a subgroup of finite index in the full unit group. In practice, very few counter-examples are known, requiring unlucky random seeds. No counter-example has been reported for $c_2 = 0.5$ (which should be almost as fast as $c_2 = 0.3$, and shall very probably become the default). If you use $c_2 = 12$, then everything is correct assuming the GRH holds. You can use `bnfcertify` to certify the computations unconditionally.

Remarks.

Apart from the polynomial P, you do not need to supply the technical parameters (under the library you still need to send at least an empty vector, coded as `NULL`). However, should you choose to set some of them, they *must* be given in the requested order. For example, if you want to specify a given value of *nrpid*, you must give some values as well for c and c_2, and provide a vector $[c, c_2, nrpid]$.

Note also that you can use an *nf* instead of P, which avoids recomputing the integral basis and analogous quantities.

3.6.8 bnfcertify(*bnf*): *bnf* being as output by `bnfinit`, checks whether the result is correct, i.e. whether it is possible to remove the assumption of the Generalized Riemann Hypothesis. It is correct if and only if the answer is 1. If it is incorrect, the program may output some error message, or loop indefinitely. You can check its progress by increasing the debug level.

The library syntax is **certifybuchall**(*bnf*), and the result is a C long.

3.6.9 bnfclassunit($P, \{flag = 0\}, \{tech = [\,]\}$): *this function is DEPRECATED, use* `bnfinit`.

Buchmann's sub-exponential algorithm for computing the class group, the regulator and a system of fundamental units of the general algebraic number field K defined by the irreducible polynomial P with integer coefficients.

The result of this function is a vector v with many components, which for ease of presentation is in fact output as a one column matrix. It is *not* a *bnf*, you need `bnfinit` for that. First we describe the default behaviour (*flag* $= 0$):

$v[1]$ is equal to the polynomial P.

$v[2]$ is the 2-component vector $[r1, r2]$, where $r1$ and $r2$ are as usual the number of real and half the number of complex embeddings of the number field K.

$v[3]$ is the 2-component vector containing the field discriminant and the index.

$v[4]$ is an integral basis in Hermite normal form.

$v[5]$ (v`.clgp`) is a 3-component vector containing the class number (v`.clgp.no`), the structure of the class group as a product of cyclic groups of order n_i (v`.clgp.cyc`), and the corresponding generators of the class group of respective orders n_i (v`.clgp.gen`).

$v[6]$ (v`.reg`) is the regulator computed to an accuracy which is the maximum of an internally determined accuracy and of the default.

$v[7]$ is deprecated, maintained for backward compatibility and always equal to 1.

$v[8]$ (v`.tu`) a vector with 2 components, the first being the number w of roots of unity in K and the second a primitive w-th root of unity expressed as a polynomial.

$v[9]$ (v`.fu`) is a system of fundamental units also expressed as polynomials.

If *flag* $= 1$, and the precision happens to be insufficient for obtaining the fundamental units, the internal precision is doubled and the computation redone, until the exact results are obtained. Be warned that this can take a very long time when the coefficients of the fundamental units on the integral basis are very large, for example in large real quadratic fields. For this case, there are alternate compact representations for algebraic numbers, implemented in PARI but currently not available in GP.

If *flag* = 2, the fundamental units and roots of unity are not computed. Hence the result has only 7 components, the first seven ones.

The library syntax is **bnfclassunit0**(P, *flag*, *tech*, *prec*).

3.6.10 bnfclgp(P, {*tech* = []}): as `bnfinit`, but only outputs `bnf.clgp`, i.e. the class group.

The library syntax is **classgrouponly**(P, *tech*, *prec*), where *tech* is as described under `bnfinit`.

3.6.11 bnfdecodemodule(nf, m): if m is a module as output in the first component of an extension given by `bnrdisclist`, outputs the true module.

The library syntax is **decodemodule**(nf, m).

3.6.12 bnfinit(P, {*flag* = 0}, {*tech* = []}): initializes a *bnf* structure. Used in programs such as `bnfisprincipal`, `bnfisunit` or `bnfnarrow`. By default, the results are conditional on a heuristic strengthening of the GRH, see 3.6.7. The result is a 10-component vector *bnf*.

This implements Buchmann's sub-exponential algorithm for computing the class group, the regulator and a system of fundamental units of the general algebraic number field K defined by the irreducible polynomial P with integer coefficients.

If the precision becomes insufficient, `gp` outputs a warning (`fundamental units too large, not given`) and does not strive to compute the units by default (*flag* = 0).

When *flag* = 1, we insist on finding the fundamental units exactly. Be warned that this can take a very long time when the coefficients of the fundamental units on the integral basis are very large. If the fundamental units are simply too large to be represented in this form, an error message is issued. They could be obtained using the so-called compact representation of algebraic numbers as a formal product of algebraic integers. The latter is implemented internally but not publicly accessible yet.

When *flag* = 2, on the contrary, it is initially agreed that units are not computed. Note that the resulting *bnf* will not be suitable for `bnrinit`, and that this flag provides negligible time savings compared to the default. In short, it is deprecated.

When *flag* = 3, computes a very small version of `bnfinit`, a "small Buchmann's number field" (or *sbnf* for short) which contains enough information to recover the full *bnf* vector very rapidly, but which is much smaller and hence easy to store and print. It is supposed to be used in conjunction with `bnfmake`.

tech is a technical vector (empty by default, see 3.6.7). Careful use of this parameter may speed up your computations considerably.

The components of a *bnf* or *sbnf* are technical and never used by the casual user. In fact: *never access a component directly, always use a proper member function.* However, for the sake of completeness and internal documentation, their description is as follows. We use the notations explained in the book by H. Cohen, *A Course in Computational Algebraic Number Theory*, Graduate Texts in Maths **138**, Springer-Verlag, 1993, Section 6.5, and subsection 6.5.5 in particular.

bnf[1] contains the matrix W, i.e. the matrix in Hermite normal form giving relations for the class group on prime ideal generators $(\wp_i)_{1\le i\le r}$.

bnf[2] contains the matrix B, i.e. the matrix containing the expressions of the prime ideal factorbase in terms of the $\wp_i$. It is an $r \times c$ matrix.

bnf[3] contains the complex logarithmic embeddings of the system of fundamental units which has been found. It is an $(r_1 + r_2) \times (r_1 + r_2 - 1)$ matrix.

bnf[4] contains the matrix M''_C of Archimedean components of the relations of the matrix $(W|B)$.

bnf[5] contains the prime factor base, i.e. the list of prime ideals used in finding the relations.

bnf[6] used to contain a permutation of the prime factor base, but has been obsoleted. It contains a dummy 0.

bnf[7] or *bnf*`.nf` is equal to the number field data *nf* as would be given by `nfinit`.

bnf[8] is a vector containing the classgroup *bnf*`.clgp` as a finite abelian group, the regulator *bnf*`.reg`, a 1 (used to contain an obsolete "check number"), the number of roots of unity and a generator *bnf*`.tu`, the fundamental units *bnf*`.fu`.

bnf[9] is a 3-element row vector used in `bnfisprincipal` only and obtained as follows. Let $D = UWV$ obtained by applying the Smith normal form algorithm to the matrix W (= *bnf*[1]) and let U_r be the reduction of U modulo D. The first elements of the factorbase are given (in terms of `bnf.gen`) by the columns of U_r, with Archimedean component g_a; let also GD_a be the Archimedean components of the generators of the (principal) ideals defined by the `bnf.gen[i]^ bnf.cyc[i]`. Then *bnf*[9] $= [U_r, g_a, GD_a]$.

bnf[10] is by default unused and set equal to 0. This field is used to store further information about the field as it becomes available, which is rarely needed, hence would be too expensive to compute during the initial `bnfinit` call. For instance, the generators of the principal ideals `bnf.gen[i]^bnf.cyc[i]` (during a call to `bnrisprincipal`), or those corresponding to the relations in W and B (when the `bnf` internal precision needs to be increased).

An *sbnf* is a 12 component vector v, as follows. Let *bnf* be the result of a full `bnfinit`, complete with units. Then v[1] is the polynomial P, v[2] is the number of real embeddings r_1, v[3] is the field discriminant, v[4] is the integral basis, v[5] is the list of roots as in the sixth component of `nfinit`, v[6] is the matrix MD of `nfinit` giving a $\mathbf{Z}$-basis of the different, v[7] is the matrix `W` = *bnf*[1], v[8] is the matrix `matalpha` = *bnf*[2], v[9] is the prime ideal factor base *bnf*[5] coded in a compact way, and ordered according to the permutation *bnf*[6], v[10] is the 2-component vector giving the number of roots of unity and a generator, expressed on the integral basis, v[11] is the list of fundamental units, expressed on the integral basis, v[12] is a vector containing the algebraic numbers alpha corresponding to the columns of the matrix `matalpha`, expressed on the integral basis.

Note that all the components are exact (integral or rational), except for the roots in v[5]. Note also that member functions will *not* work on *sbnf*, you have to use `bnfmake` explicitly first.

The library syntax is **bnfinit0**(P, *flag*, *tech*, *prec*).

3.6.13 bnfisintnorm(bnf, x): computes a complete system of solutions (modulo units of positive norm) of the absolute norm equation $\mathrm{Norm}(a) = x$, where a is an integer in *bnf*. If *bnf* has not been certified, the correctness of the result depends on the validity of GRH.

See also `bnfisnorm`.

The library syntax is **bnfisintnorm**(bnf, x).

3.6.14 bnfisnorm$(bnf, x, \{flag = 1\})$: tries to tell whether the rational number x is the norm of some element y in *bnf*. Returns a vector $[a, b]$ where $x = Norm(a) * b$. Looks for a solution which is an S-unit, with S a certain set of prime ideals containing (among others) all primes dividing x. If *bnf* is known to be Galois, set $flag = 0$ (in this case, x is a norm iff $b = 1$). If *flag* is non zero the program adds to S the following prime ideals, depending on the sign of *flag*. If $flag > 0$, the ideals of norm less than *flag*. And if $flag < 0$ the ideals dividing *flag*.

Assuming GRH, the answer is guaranteed (i.e. x is a norm iff $b = 1$), if S contains all primes less than $12\log(\mathrm{disc}(Bnf))^2$, where *Bnf* is the Galois closure of *bnf*.

See also `bnfisintnorm`.

The library syntax is **bnfisnorm**$(bnf, x, flag, prec)$, where *flag* and *prec* are `long`s.

3.6.15 bnfissunit(bnf, sfu, x): *bnf* being output by `bnfinit`, *sfu* by `bnfsunit`, gives the column vector of exponents of x on the fundamental S-units and the roots of unity. If x is not a unit, outputs an empty vector.

The library syntax is **bnfissunit**(bnf, sfu, x).

3.6.16 bnfisprincipal$(bnf, x, \{flag = 1\})$: *bnf* being the number field data output by `bnfinit`, and x being either a $\mathbf{Z}$-basis of an ideal in the number field (not necessarily in HNF) or a prime ideal in the format output by the function `idealprimedec`, this function tests whether the ideal is principal or not. The result is more complete than a simple true/false answer: it gives a row vector $[v_1, v_2]$, where

v_1 is the vector of components c_i of the class of the ideal x in the class group, expressed on the generators g_i given by `bnfinit` (specifically *bnf*`.gen`). The c_i are chosen so that $0 \le c_i < n_i$ where n_i is the order of g_i (the vector of n_i being *bnf*`.cyc`).

v_2 gives on the integral basis the components of α such that $x = \alpha \prod_i g_i^{c_i}$. In particular, x is principal if and only if v_1 is equal to the zero vector. In the latter case, $x = \alpha\mathbf{Z}_K$ where α is given by v_2. Note that if α is too large to be given, a warning message will be printed and v_2 will be set equal to the empty vector.

If $flag = 0$, outputs only v_1, which is much easier to compute.

If $flag = 2$, does as if *flag* were 0, but doubles the precision until a result is obtained.

If $flag = 3$, as in the default behaviour ($flag = 1$), but doubles the precision until a result is obtained.

The user is warned that these two last setting may induce *very* lengthy computations.

The library syntax is **isprincipalall**$(bnf, x, flag)$.

3.6.17 bnfisunit(bnf, x): *bnf* being the number field data output by `bnfinit` and x being an algebraic number (type integer, rational or polmod), this outputs the decomposition of x on the fundamental units and the roots of unity if x is a unit, the empty vector otherwise. More precisely, if $u_1, \ldots, u_r$ are the fundamental units, and ζ is the generator of the group of roots of unity (`bnf.tu`), the output is a vector $[x_1, \ldots, x_r, x_{r+1}]$ such that $x = u_1^{x_1} \cdots u_r^{x_r} \cdot \zeta^{x_{r+1}}$. The x_i are integers for $i \le r$ and is an integer modulo the order of ζ for $i = r + 1$.

The library syntax is **isunit**(bnf, x).

3.6.18 bnfmake(*sbnf*): *sbnf* being a "small *bnf*" as output by `bnfinit`$(x, 3)$, computes the complete `bnfinit` information. The result is *not* identical to what `bnfinit` would yield, but is functionally identical. The execution time is very small compared to a complete `bnfinit`. Note that if the default precision in `gp` (or *prec* in library mode) is greater than the precision of the roots *sbnf*[5], these are recomputed so as to get a result with greater accuracy.

Note that the member functions are *not* available for *sbnf*, you have to use `bnfmake` explicitly first.

The library syntax is **makebigbnf**(*sbnf*, *prec*), where *prec* is a C long integer.

3.6.19 bnfnarrow(*bnf*): *bnf* being as output by `bnfinit`, computes the narrow class group of *bnf*. The output is a 3-component row vector v analogous to the corresponding class group component *bnf*`.clgp` (*bnf*`[8][1]`): the first component is the narrow class number v`.no`, the second component is a vector containing the SNF cyclic components v`.cyc` of the narrow class group, and the third is a vector giving the generators of the corresponding v`.gen` cyclic groups. Note that this function is a special case of `bnrinit`.

The library syntax is **buchnarrow**(*bnf*).

3.6.20 bnfsignunit(*bnf*): *bnf* being as output by `bnfinit`, this computes an $r_1 \times (r_1 + r_2 - 1)$ matrix having ± 1 components, giving the signs of the real embeddings of the fundamental units. The following functions compute generators for the totally positive units:

```
/* exponents of totally positive units generators on bnf.tufu */
tpuexpo(bnf)=
{ local(S,d,K);

  S = bnfsignunit(bnf); d = matsize(S);
  S = matrix(d[1],d[2], i,j, if (S[i,j] < 0, 1,0));
  S = concat(vectorv(d[1],i,1), S);   \\ add sign(-1)
  K = lift(matker(S * Mod(1,2)));
  if (K, mathnfmodid(K, 2), 2*matid(d[1]))
}

/* totally positive units */
tpu(bnf)=
{ local(vu = bnf.tufu, ex = tpuexpo(bnf));

  vector(#ex-1, i, factorback(vu, ex[,i+1]))  \\ ex[,1] is 1
}
```

The library syntax is **signunits**(*bnf*).

3.6.21 bnfreg(*bnf*): *bnf* being as output by `bnfinit`, computes its regulator.

The library syntax is **regulator**(*bnf*, *tech*, *prec*), where *tech* is as in `bnfinit`.

3.6.22 bnfsunit(bnf, S): computes the fundamental S-units of the number field *bnf* (output by `bnfinit`), where S is a list of prime ideals (output by `idealprimedec`). The output is a vector v with 6 components.

$v[1]$ gives a minimal system of (integral) generators of the S-unit group modulo the unit group.

$v[2]$ contains technical data needed by `bnfissunit`.

$v[3]$ is an empty vector (used to give the logarithmic embeddings of the generators in $v[1]$ in version 2.0.16).

$v[4]$ is the S-regulator (this is the product of the regulator, the determinant of $v[2]$ and the natural logarithms of the norms of the ideals in S).

$v[5]$ gives the S-class group structure, in the usual format (a row vector whose three components give in order the S-class number, the cyclic components and the generators).

$v[6]$ is a copy of S.

The library syntax is **bnfsunit**$(bnf, S, prec)$.

3.6.23 bnfunit(bnf): *bnf* being as output by `bnfinit`, outputs the vector of fundamental units of the number field.

This function is mostly useless, since it will only succeed if *bnf* contains the units, in which case `bnf.fu` is recommanded instead, or *bnf* was produced with `bnfinit(,,2)`, which is itself deprecated.

The library syntax is **buchfu**(bnf).

3.6.24 bnrL1$(bnr, \{subgroup\}, \{flag = 0\})$: *bnr* being the number field data which is output by `bnrinit(,,1)` and *subgroup* being a square matrix defining a congruence subgroup of the ray class group corresponding to *bnr* (the trivial congruence subgroup if omitted), returns for each character χ of the ray class group which is trivial on this subgroup, the value at $s = 1$ (or $s = 0$) of the abelian L-function associated to χ. For the value at $s = 0$, the function returns in fact for each character χ a vector $[r_\chi, c_\chi]$ where r_χ is the order of $L(s, \chi)$ at $s = 0$ and c_χ the first non-zero term in the expansion of $L(s, \chi)$ at $s = 0$; in other words

$$L(s, \chi) = c_\chi \cdot s^{r_\chi} + O(s^{r_\chi+1})$$

near 0. *flag* is optional, default value is 0; its binary digits mean 1: compute at $s = 1$ if set to 1 or $s = 0$ if set to 0, 2: compute the primitive L-functions associated to χ if set to 0 or the L-function with Euler factors at prime ideals dividing the modulus of *bnr* removed if set to 1 (this is the so-called $L_S(s, \chi)$ function where S is the set of infinite places of the number field together with the finite prime ideals dividing the modulus of *bnr*, see the example below), 3: returns also the character. Example:

```
bnf = bnfinit(x^2 - 229);
bnr = bnrinit(bnf,1,1);
bnrL1(bnr)
```

returns the order and the first non-zero term of the abelian L-functions $L(s, \chi)$ at $s = 0$ where χ runs through the characters of the class group of $\mathbf{Q}(\sqrt{229})$. Then

```
bnr2 = bnrinit(bnf,2,1);
```

```
bnrL1(bnr2,,2)
```

returns the order and the first non-zero terms of the abelian L-functions $L_S(s,\chi)$ at $s=0$ where χ runs through the characters of the class group of $\mathbf{Q}(\sqrt{229})$ and S is the set of infinite places of $\mathbf{Q}(\sqrt{229})$ together with the finite prime 2. Note that the ray class group modulo 2 is in fact the class group, so `bnrL1(bnr2,0)` returns exactly the same answer as `bnrL1(bnr,0)`.

The library syntax is **bnrL1**(*bnr*, *subgroup*, *flag*, *prec*), where an omitted *subgroup* is coded as `NULL`.

3.6.25 bnrclass(*bnf*, *ideal*, {*flag* = 0}): *this function is DEPRECATED, use* `bnrinit`.

bnf being as output by `bnfinit` (the units are mandatory unless the ideal is trivial), and *ideal* being a modulus, computes the ray class group of the number field for the modulus *ideal*, as a finite abelian group.

The library syntax is **bnrclass0**(*bnf*, *ideal*, *flag*).

3.6.26 bnrclassno(*bnf*, *I*): *bnf* being as output by `bnfinit` (units are mandatory unless the ideal is trivial), and I being a modulus, computes the ray class number of the number field for the modulus I. This is faster than `bnrinit` and should be used if only the ray class number is desired. See `bnrclassnolist` if you need ray class numbers for all moduli less than some bound.

The library syntax is **bnrclassno**(*bnf*, *I*).

3.6.27 bnrclassnolist(*bnf*, *list*): *bnf* being as output by `bnfinit`, and *list* being a list of moduli (with units) as output by `ideallist` or `ideallistarch`, outputs the list of the class numbers of the corresponding ray class groups. To compute a single class number, `bnrclassno` is more efficient.

```
? bnf = bnfinit(x^2 - 2);
? L = ideallist(bnf, 100, 2);
? H = bnrclassnolist(bnf, L);
? H[98]
%4 = [1, 3, 1]
? l = L[1][98]; ids = vector(#l, i, l[i].mod[1])
%5 = [[98, 88; 0, 1], [14, 0; 0, 7], [98, 10; 0, 1]]
```

The weird `l[i].mod[1]`, is the first component of `l[i].mod`, i.e. the finite part of the conductor. (This is cosmetic: since by construction the archimedean part is trivial, I do not want to see it). This tells us that the ray class groups modulo the ideals of norm 98 (printed as `%5`) have respectively order 1, 3 and 1. Indeed, we may check directly :

```
? bnrclassno(bnf, ids[2])
%6 = 3
```

The library syntax is **bnrclassnolist**(*bnf*, *list*).

3.6.28 bnrconductor$(a_1, \{a_2\}, \{a_3\}, \{flag = 0\})$: conductor f of the subfield of a ray class field as defined by $[a_1, a_2, a_3]$ (see `bnr` at the beginning of this section).

If *flag* $= 0$, returns f.

If *flag* $= 1$, returns $[f, Cl_f, H]$, where Cl_f is the ray class group modulo f, as a finite abelian group; finally H is the subgroup of Cl_f defining the extension.

If *flag* $= 2$, returns $[f, bnr(f), H]$, as above except Cl_f is replaced by a `bnr` structure, as output by `bnrinit`$(, f, 1)$.

The library syntax is **conductor**$(bnr, subgroup, flag)$, where an omitted subgroup (trivial subgroup, i.e. ray class field) is input as `NULL`, and *flag* is a C long.

3.6.29 bnrconductorofchar(bnr, chi): *bnr* being a big ray number field as output by `bnrinit`, and *chi* being a row vector representing a character as expressed on the generators of the ray class group, gives the conductor of this character as a modulus.

The library syntax is **bnrconductorofchar**(bnr, chi).

3.6.30 bnrdisc$(a1, \{a2\}, \{a3\}, \{flag = 0\})$: $a1$, $a2$, $a3$ defining a big ray number field L over a ground field K (see `bnr` at the beginning of this section for the meaning of $a1$, $a2$, $a3$), outputs a 3-component row vector $[N, R_1, D]$, where N is the (absolute) degree of L, R_1 the number of real places of L, and D the discriminant of $L/\mathbf{Q}$, including sign (if *flag* $= 0$).

If *flag* $= 1$, as above but outputs relative data. N is now the degree of L/K, R_1 is the number of real places of K unramified in L (so that the number of real places of L is equal to R_1 times the relative degree N), and D is the relative discriminant ideal of L/K.

If *flag* $= 2$, as the default case, except that if the modulus is not the exact conductor corresponding to the L, no data is computed and the result is 0.

If *flag* $= 3$, as case 2, but output relative data.

The library syntax is **bnrdisc0**$(a1, a2, a3, flag)$.

3.6.31 bnrdisclist$(bnf, bound, \{arch\})$: *bnf* being as output by `bnfinit` (with units), computes a list of discriminants of Abelian extensions of the number field by increasing modulus norm up to bound *bound*. The ramified Archimedean places are given by *arch*; all possible values are taken if *arch* is omitted.

The alternative syntax `bnrdisclist`$(bnf, list)$ is supported, where *list* is as output by `ideallist` or `ideallistarch` (with units), in which case *arch* is disregarded.

The output v is a vector of vectors, where $v[i][j]$ is understood to be in fact $V[2^{15}(i-1)+j]$ of a unique big vector V. (This akward scheme allows for larger vectors than could be otherwise represented.)

$V[k]$ is itself a vector W, whose length is the number of ideals of norm k. We consider first the case where *arch* was specified. Each component of W corresponds to an ideal m of norm k, and gives invariants associated to the ray class field L of *bnf* of conductor $[m, arch]$. Namely, each contains a vector $[m, d, r, D]$ with the following meaning: m is the prime ideal factorization of the modulus, $d = [L : \mathbf{Q}]$ is the absolute degree of L, r is the number of real places of L, and D is the factorization of its absolute discriminant. We set $d = r = D = 0$ if m is not the finite part of a conductor.

If *arch* was omitted, all $t = 2^{r_1}$ possible values are taken and a component of W has the form $[m, [[d_1, r_1, D_1], \ldots, [d_t, r_t, D_t]]]$, where m is the finite part of the conductor as above, and $[d_i, r_i, D_i]$ are the invariants of the ray class field of conductor $[m, v_i]$, where v_i is the i-th archimedean component, ordered by inverse lexicographic order; so $v_1 = [0, \ldots, 0]$, $v_2 = [1, 0 \ldots, 0]$, etc. Again, we set $d_i = r_i = D_i = 0$ if $[m, v_i]$ is not a conductor.

Finally, each prime ideal $pr = [p, \alpha, e, f, \beta]$ in the prime factorization m is coded as the integer $p \cdot n^2 + (f-1) \cdot n + (j-1)$, where n is the degree of the base field and j is such that

`pr = idealprimedec(`*nf*`,p)[j]`.

m can be decoded using `bnfdecodemodule`.

Note that to compute such data for a single field, either `bnrclassno` or `bnrdisc` is more efficient.

The library syntax is **bnrdisclist0**(*bnf*, *bound*, *arch*).

3.6.32 bnrinit(*bnf*, *f*, {*flag* = 0}): *bnf* is as output by `bnfinit`, *f* is a modulus, initializes data linked to the ray class group structure corresponding to this module, a so-called *bnr* structure. The following member functions are available on the result: `.bnf` is the underlying *bnf*, `.mod` the modulus, `.bid` the *bid* structure associated to the modulus; finally, `.clgp`, `.no`, `.cyc`, `clgp` refer to the ray class group (as a finite abelian group), its cardinality, its elementary divisors, its generators.

The last group of functions are different from the members of the underlying *bnf*, which refer to the class group; use *bnr*`.bnf.`*xxx* to access these, e.g. *bnr*`.bnf.cyc` to get the cyclic decomposition of the class group.

They are also different from the members of the underlying *bid*, which refer to $(\vee_K/f)^*$; use *bnr*`.bid.`*xxx* to access these, e.g. *bnr*`.bid.no` to get $\phi(f)$.

If *flag* = 0 (default), the generators of the ray class group are not computed, which saves time. Hence *bnr*`.gen` would produce an error.

If *flag* = 1, as the default, except that generators are computed.

The library syntax is **bnrinit0**(*bnf*, *f*, *flag*).

3.6.33 bnrisconductor($a1$, {$a2$}, {$a3$}): $a1$, $a2$, $a3$ represent an extension of the base field, given by class field theory for some modulus encoded in the parameters. Outputs 1 if this modulus is the conductor, and 0 otherwise. This is slightly faster than `bnrconductor`.

The library syntax is **bnrisconductor**($a1, a2, a3$) and the result is a `long`.

3.6.34 bnrisprincipal(*bnr*, *x*, {*flag* = 1}): *bnr* being the number field data which is output by `bnrinit`$(,,1)$ and *x* being an ideal in any form, outputs the components of *x* on the ray class group generators in a way similar to `bnfisprincipal`. That is a 2-component vector v where $v[1]$ is the vector of components of *x* on the ray class group generators, $v[2]$ gives on the integral basis an element α such that $x = \alpha \prod_i g_i^{x_i}$.

If *flag* = 0, outputs only v_1. In that case, *bnr* need not contain the ray class group generators, i.e. it may be created with `bnrinit`$(,,0)$

The library syntax is **bnrisprincipal**(*bnr*, *x*, *flag*).

3.6.35 bnrrootnumber$(bnr, chi, \{flag = 0\})$: if $\chi = chi$ is a (not necessarily primitive) character over bnr, let $L(s,\chi) = \sum_{id} \chi(id)N(id)^{-s}$ be the associated Artin L-function. Returns the so-called Artin root number, i.e. the complex number $W(\chi)$ of modulus 1 such that

$$\Lambda(1-s,\chi) = W(\chi)\Lambda(s,\overline{\chi})$$

where $\Lambda(s,\chi) = A(\chi)^{s/2}\gamma_\chi(s)L(s,\chi)$ is the enlarged L-function associated to L.

The generators of the ray class group are needed, and you can set $flag = 1$ if the character is known to be primitive. Example:

```
bnf = bnfinit(x^2 - 145);
bnr = bnrinit(bnf,7,1);
bnrrootnumber(bnr, [5])
```

returns the root number of the character χ of $\mathrm{Cl}_7(\mathbf{Q}(\sqrt{145}))$ such that $\chi(g) = \zeta^5$, where g is the generator of the ray-class field and $\zeta = e^{2i\pi/N}$ where N is the order of g ($N = 12$ as `bnr.cyc` readily tells us).

The library syntax is **bnrrootnumber**$(bnf, chi, flag)$

3.6.36 bnrstark$(bnr, \{subgroup\})$: *bnr* being as output by `bnrinit(,,1)`, finds a relative equation for the class field corresponding to the modulus in *bnr* and the given congruence subgroup (as usual, omit *subgroup* if you want the whole ray class group).

The routine uses Stark units and needs to find a suitable auxilliary conductor, which may not exist when the class field is not cyclic over the base. In this case `bnrstark` is allowed to return a vector of polynomials defining *independent* relative extensions, whose compositum is the requested class field. It was decided that it was more useful to keep the extra information thus made available, hence the user has to take the compositum herself.

The main variable of *bnr* must not be x, and the ground field and the class field must be totally real. When the base field is $\mathbf{Q}$, the vastly simpler `galoissubcyclo` is used instead. Here is an example:

```
bnf = bnfinit(y^2 - 3);
bnr = bnrinit(bnf, 5, 1);
pol = bnrstark(bnr)
```

returns the ray class field of $\mathbf{Q}(\sqrt{3})$ modulo 5. Usually, one wants to apply to the result one of

```
rnfpolredabs(bnf, pol, 16)      \\ compute a reduced relative polynomial
rnfpolredabs(bnf, pol, 16 + 2) \\ compute a reduced absolute polynomial
```

The library syntax is **bnrstark**$(bnr, subgroup)$, where an omitted *subgroup* is coded by `NULL`.

3.6.37 dirzetak(nf, b): gives as a vector the first b coefficients of the Dedekind zeta function of the number field *nf* considered as a Dirichlet series.

The library syntax is **dirzetak**(nf, b).

3.6.38 factornf(x, t): factorization of the univariate polynomial x over the number field defined by the (univariate) polynomial t. x may have coefficients in $\mathbf{Q}$ or in the number field. The algorithm reduces to factorization over $\mathbf{Q}$ (Trager's trick). The direct approach of `nffactor`, which uses van Hoeij's method in a relative setting, is in general faster.

The main variable of t must be of *lower* priority than that of x (see Section 2.5.4). However if non-rational number field elements occur (as polmods or polynomials) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t. For example

```
? factornf(x^2 + Mod(y, y^2+1), y^2+1);
? factornf(x^2 + y, y^2+1); \\ these two are OK
? factornf(x^2 + Mod(z,z^2+1), y^2+1)
  *** factornf: inconsistent data in rnf function.
? factornf(x^2 + z, y^2+1)
  *** factornf: incorrect variable in rnf function.
```

The library syntax is **polfnf**(x, t).

3.6.39 galoisexport$(gal, \{flag = 0\})$: *gal* being be a Galois field as output by `galoisinit`, export the underlying permutation group as a string suitable for (no flags or $flag = 0$) GAP or ($flag = 1$) Magma. The following example compute the index of the underlying abstract group in the GAP library:

```
? G = galoisinit(x^6+108);
? s = galoisexport(G)
%2 = "Group((1, 2, 3)(4, 5, 6), (1, 4)(2, 6)(3, 5))"
? extern("echo \"IdGroup("s");\" | gap -q")
%3 = [6, 1]
? galoisidentify(G)
%4 = [6, 1]
```

This command also accepts subgroups returned by `galoissubgroups`.

The library syntax is **galoisexport**$(gal, flag)$.

3.6.40 galoisfixedfield$(gal, perm, \{flag = 0\}, \{v = y\})$: *gal* being be a Galois field as output by `galoisinit` and *perm* an element of *gal.group* or a vector of such elements, computes the fixed field of *gal* by the automorphism defined by the permutations *perm* of the roots *gal.roots*. P is guaranteed to be squarefree modulo *gal.p*.

If no flags or $flag = 0$, output format is the same as for `nfsubfield`, returning $[P, x]$ such that P is a polynomial defining the fixed field, and x is a root of P expressed as a polmod in *gal.pol*.

If $flag = 1$ return only the polynomial P.

If $flag = 2$ return $[P, x, F]$ where P and x are as above and F is the factorization of *gal.pol* over the field defined by P, where variable v (y by default) stands for a root of P. The priority of v must be less than the priority of the variable of *gal.pol* (see Section 2.5.4). Example:

```
? G = galoisinit(x^4+1);
? galoisfixedfield(G,G.group[2],2)
%2 = [x^2 + 2, Mod(x^3 + x, x^4 + 1), [x^2 - y*x - 1, x^2 + y*x - 1]]
```

computes the factorization $x^4 + 1 = (x^2 - \sqrt{-2}x - 1)(x^2 + \sqrt{-2}x - 1)$

The library syntax is **galoisfixedfield**$(gal, perm, flag,$v$)$, where v is a variable number, an omitted v being coded by -1.

3.6.41 galoisidentify(gal): *gal* being be a Galois field as output by `galoisinit`, output the isomorphism class of the underlying abstract group as a two-components vector $[o, i]$, where o is the group order, and i is the group index in the GAP4 Small Group library, by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien.

This command also accepts subgroups returned by `galoissubgroups`.

The current implementation is limited to degree less or equal to 127. Some larger "easy" orders are also supported.

The output is similar to the output of the function `IdGroup` in GAP4. Note that GAP4 `IdGroup` handles all groups of order less than 2000 except 1024, so you can use `galoisexport` and GAP4 to identify large Galois groups.

The library syntax is **galoisidentify**(gal).

3.6.42 galoisinit$(pol, \{den\})$: computes the Galois group and all necessary information for computing the fixed fields of the Galois extension $K/\mathbf{Q}$ where K is the number field defined by *pol* (monic irreducible polynomial in $\mathbf{Z}[X]$ or a number field as output by `nfinit`). The extension $K/\mathbf{Q}$ must be Galois with Galois group "weakly" super-solvable (see `nfgaloisconj`)

This is a prerequisite for most of the `galois`xxx routines. For instance:

```
P = x^6 + 108;
G = galoisinit(P);
L = galoissubgroups(G);
vector(#L, i, galoisisabelian(L[i],1))
vector(#L, i, galoisidentify(L[i]))
```

The output is an 8-component vector *gal*.

gal[1] contains the polynomial *pol* (*gal*`.pol`).

gal[2] is a three-components vector $[p, e, q]$ where p is a prime number (*gal*`.p`) such that *pol* totally split modulo p , e is an integer and $q = p^e$ (*gal*`.mod`) is the modulus of the roots in *gal*`.roots`.

gal[3] is a vector L containing the p-adic roots of *pol* as integers implicitly modulo *gal*`.mod`. (*gal*`.roots`).

gal[4] is the inverse of the Van der Monde matrix of the p-adic roots of *pol*, multiplied by *gal*[5].

gal[5] is a multiple of the least common denominator of the automorphisms expressed as polynomial in a root of *pol*.

gal[6] is the Galois group G expressed as a vector of permutations of L (*gal*`.group`).

gal[7] is a generating subset $S = [s_1, \ldots, s_g]$ of G expressed as a vector of permutations of L (*gal*`.gen`).

gal[8] contains the relative orders $[o_1, \ldots, o_g]$ of the generators of S (*gal*`.orders`).

Let H be the maximal normal supersolvable subgroup of G, we have the following properties:

- if $G/H \simeq A_4$ then $[o_1, \ldots, o_g]$ ends by $[2, 2, 3]$.
- if $G/H \simeq S_4$ then $[o_1, \ldots, o_g]$ ends by $[2, 2, 3, 2]$.
- else G is super-solvable.

• for $1 \le i \le g$ the subgroup of G generated by $[s_1, \dots, s_g]$ is normal, with the exception of $i = g - 2$ in the second case and of $i = g - 3$ in the third.

• the relative order o_i of s_i is its order in the quotient group $G/\langle s_1, \dots, s_{i-1}\rangle$, with the same exceptions.

• for any $x \in G$ there exists a unique family $[e_1, \dots, e_g]$ such that (no exceptions):

– for $1 \le i \le g$ we have $0 \le e_i < o_i$

– $x = g_1^{e_1} g_2^{e_2} \dots g_n^{e_n}$

If present *den* must be a suitable value for *gal*[5].

The library syntax is **galoisinit**(*gal*, *den*).

3.6.43 galoisisabelian(*gal*, *fl* = 0): *gal* being as output by `galoisinit`, return 0 if *gal* is not an abelian group, and the HNF matrix of *gal* over `gal.gen` if *fl* = 0, 1 if *fl* = 1.

This command also accepts subgroups returned by `galoissubgroups`.

The library syntax is **galoisisabelian**(*gal*, *fl*) where *fl* is a C long integer.

3.6.44 galoispermtopol(*gal*, *perm*): *gal* being a Galois field as output by `galoisinit` and *perm* a element of *gal.group*, return the polynomial defining the Galois automorphism, as output by `nfgaloisconj`, associated with the permutation *perm* of the roots *gal.roots*. *perm* can also be a vector or matrix, in this case, `galoispermtopol` is applied to all components recursively.

Note that

```
G = galoisinit(pol);
galoispermtopol(G, G[6])~
```

is equivalent to `nfgaloisconj(pol)`, if degree of *pol* is greater or equal to 2.

The library syntax is **galoispermtopol**(*gal*, *perm*).

3.6.45 galoissubcyclo(*N*, *H*, {*fl* = 0}, {*v*}): computes the subextension of $\mathbf{Q}(\zeta_n)$ fixed by the subgroup $H \subset (\mathbf{Z}/n\mathbf{Z})^*$. By the Kronecker-Weber theorem, all abelian number fields can be generated in this way (uniquely if n is taken to be minimal).

The pair (n, H) is deduced from the parameters (N, H) as follows

• N an integer: then $n = N$; H is a generator, i.e. an integer or an integer modulo n; or a vector of generators.

• N the output of `znstar(`n`)`. H as in the first case above, or a matrix, taken to be a HNF left divisor of the SNF for $(\mathbf{Z}/n\mathbf{Z})^*$ (of type N`.cyc`), giving the generators of H in terms of N`.gen`.

• N the output of `bnrinit(bnfinit(y),` m`, 1)` where m is a module. H as in the first case, or a matrix taken to be a HNF left divisor of the SNF for the ray class group modulo m (of type N`.cyc`), giving the generators of H in terms of N`.gen`.

In this last case, beware that H is understood relatively to N; in particular, if the infinite place does not divide the module, e.g if m is an integer, then it is not a subgroup of $(\mathbf{Z}/n\mathbf{Z})^*$, but of its quotient by $\{\pm 1\}$.

If $fl = 0$, compute a polynomial (in the variable v) defining the the subfield of $\mathbf{Q}(\zeta_n)$ fixed by the subgroup H of $(\mathbf{Z}/n\mathbf{Z})^*$.

If $fl = 1$, compute only the conductor of the abelian extension, as a module.

If $fl = 2$, output $[pol, N]$, where *pol* is the polynomial as output when $fl = 0$ and N the conductor as output when $fl = 1$.

The following function can be used to compute all subfields of $\mathbf{Q}(\zeta_n)$ (of exact degree `d`, if `d` is set):

```
subcyclo(n, d = -1)=
{
  local(bnr,L,IndexBound);
  IndexBound = if (d < 0, n, [d]);
  bnr = bnrinit(bnfinit(y), [n,[1]], 1);
  L = subgrouplist(bnr, IndexBound, 1);
  vector(#L,i, galoissubcyclo(bnr,L[i]));
}
```

Setting `L = subgrouplist(bnr, IndexBound)` would produce subfields of exact conductor $n\infty$.

The library syntax is **galoissubcyclo**(N, H, fl, v) where *fl* is a C long integer, and v a variable number.

3.6.46 galoissubfields$(G, \{fl = 0\}, \{v\})$: Output all the subfields of the Galois group G, as a vector. This works by applying `galoisfixedfield` to all subgroups. The meaning of the flag *fl* is the same as for `galoisfixedfield`.

The library syntax is **galoissubfields**(G, fl, v), where *fl* is a long and v a variable number.

3.6.47 galoissubgroups(gal): Output all the subgroups of the Galois group `gal`. A subgroup is a vector [*gen*, *orders*], with the same meaning as for *gal.gen* and *gal.orders*. Hence *gen* is a vector of permutations generating the subgroup, and *orders* is the relatives orders of the generators. The cardinal of a subgroup is the product of the relative orders. Such subgroup can be used instead of a Galois group in the following command: `galoisisabelian`, `galoissubgroups`, `galoisexport` and `galoisidentify`.

To get the subfield fixed by a subgroup *sub* of *gal*, use

```
galoisfixedfield(gal,sub[1])
```

The library syntax is **galoissubgroups**(gal).

3.6.48 idealadd(nf, x, y): sum of the two ideals x and y in the number field nf. When x and y are given by $\mathbf{Z}$-bases, this does not depend on nf and can be used to compute the sum of any two $\mathbf{Z}$-modules. The result is given in HNF.

The library syntax is **idealadd**(nf, x, y).

3.6.49 idealaddtoone$(nf, x, \{y\})$: x and y being two co-prime integral ideals (given in any form), this gives a two-component row vector $[a, b]$ such that $a \in x$, $b \in y$ and $a + b = 1$.

The alternative syntax `idealaddtoone`(nf, v), is supported, where v is a k-component vector of ideals (given in any form) which sum to $\mathbf{Z}_K$. This outputs a k-component vector e such that $e[i] \in x[i]$ for $1 \le i \le k$ and $\sum_{1 \le i \le k} e[i] = 1$.

The library syntax is **idealaddtoone0**(nf, x, y), where an omitted y is coded as `NULL`.

3.6.50 idealappr$(nf, x, \{flag = 0\})$: if x is a fractional ideal (given in any form), gives an element α in nf such that for all prime ideals $\wp$ such that the valuation of x at $\wp$ is non-zero, we have $v_\wp(\alpha) = v_\wp(x)$, and. $v_\wp(\alpha) \geq 0$ for all other $\wp$.

If *flag* is non-zero, x must be given as a prime ideal factorization, as output by `idealfactor`, but possibly with zero or negative exponents. This yields an element α such that for all prime ideals $\wp$ occurring in x, $v_\wp(\alpha)$ is equal to the exponent of $\wp$ in x, and for all other prime ideals, $v_\wp(\alpha) \geq 0$. This generalizes `idealappr`$(nf, x, 0)$ since zero exponents are allowed. Note that the algorithm used is slightly different, so that `idealappr(`nf`,idealfactor(`nf`,x))` may not be the same as `idealappr(`nf`,x,1)`.

The library syntax is **idealappr0**$(nf, x, flag)$.

3.6.51 idealchinese(nf, x, y): x being a prime ideal factorization (i.e. a 2 by 2 matrix whose first column contain prime ideals, and the second column integral exponents), y a vector of elements in nf indexed by the ideals in x, computes an element b such that

$v_\wp(b - y_\wp) \geq v_\wp(x)$ for all prime ideals in x and $v_\wp(b) \geq 0$ for all other $\wp$.

The library syntax is **idealchinese**(nf, x, y).

3.6.52 idealcoprime(nf, x, y): given two integral ideals x and y in the number field nf, finds a β in the field, expressed on the integral basis $nf[7]$, such that $\beta \cdot x$ is an integral ideal coprime to y.

The library syntax is **idealcoprime**(nf, x, y).

3.6.53 idealdiv$(nf, x, y, \{flag = 0\})$: quotient $x \cdot y^{-1}$ of the two ideals x and y in the number field nf. The result is given in HNF.

If *flag* is non-zero, the quotient $x \cdot y^{-1}$ is assumed to be an integral ideal. This can be much faster when the norm of the quotient is small even though the norms of x and y are large.

The library syntax is **idealdiv0**$(nf, x, y, flag)$. Also available are **idealdiv**(nf, x, y) $(flag = 0)$ and **idealdivexact**(nf, x, y) $(flag = 1)$.

3.6.54 idealfactor(nf, x): factors into prime ideal powers the ideal x in the number field nf. The output format is similar to the `factor` function, and the prime ideals are represented in the form output by the `idealprimedec` function, i.e. as 5-element vectors.

The library syntax is **idealfactor**(nf, x).

3.6.55 idealhnf$(nf, a, \{b\})$: gives the Hermite normal form matrix of the ideal a. The ideal can be given in any form whatsoever (typically by an algebraic number if it is principal, by a $\mathbf{Z}_K$-system of generators, as a prime ideal as given by `idealprimedec`, or by a $\mathbf{Z}$-basis).

If b is not omitted, assume the ideal given was $a\mathbf{Z}_K + b\mathbf{Z}_K$, where a and b are elements of K given either as vectors on the integral basis $nf[7]$ or as algebraic numbers.

The library syntax is **idealhnf0**(nf, a, b) where an omitted b is coded as `NULL`. Also available is **idealhermite**(nf, a) (b omitted).

3.6.56 idealintersect(nf, A, B): intersection of the two ideals A and B in the number field nf. The result is given in HNF.

```
? nf = nfinit(x^2+1);
? idealintersect(nf, 2, x+1)
%2 =
[2 0]

[0 2]
```

This function does not apply to general **Z**-modules, e.g. orders, since its arguments are replaced by the ideals they generate. The following script intersects **Z**-modules A and B given by matrices of compatible dimensions with integer coefficients:

```
ZM_intersect(A,B) =
{ local( Ker = matkerint(concat(A,B)) );
  mathnf(A * vecextract(Ker, Str("..", #A), ".."))
}
```

The library syntax is **idealintersect**(nf, A, B).

3.6.57 idealinv(nf, x): inverse of the ideal x in the number field nf. The result is the Hermite normal form of the inverse of the ideal, together with the opposite of the Archimedean information if it is given.

The library syntax is **idealinv**(nf, x).

3.6.58 ideallist$(nf, bound, \{flag = 4\})$: computes the list of all ideals of norm less or equal to *bound* in the number field nf. The result is a row vector with exactly *bound* components. Each component is itself a row vector containing the information about ideals of a given norm, in no specific order, depending on the value of *flag*:

The possible values of *flag* are:

0: give the *bid* associated to the ideals, without generators.

1: as 0, but include the generators in the *bid*.

2: in this case, *nf* must be a *bnf* with units. Each component is of the form $[bid, U]$, where *bid* is as case 0 and U is a vector of discrete logarithms of the units. More precisely, it gives the `ideallog`s with respect to *bid* of `bnf.tufu`. This structure is technical, and only meant to be used in conjunction with `bnrclassnolist` or `bnrdisclist`.

3: as 2, but include the generators in the *bid*.

4: give only the HNF of the ideal.

```
? nf = nfinit(x^2+1);
? L = ideallist(nf, 100);
? L[1]
%3 = [[1, 0; 0, 1]]  \\ A single ideal of norm 1
? #L[65]
%4 = 4               \\ There are 4 ideals of norm 4 in Z[i]
```

If one wants more information, one could do instead:

```
? nf = nfinit(x^2+1);
```

```
? L = ideallist(nf, 100, 0);
? l = L[25]; vector(#l, i, l[i].clgp)
%3 = [[20, [20]], [16, [4, 4]], [20, [20]]]
? l[1].mod
%4 = [[25, 18; 0, 1], []]
? l[2].mod
%5 = [[5, 0; 0, 5], []]
? l[3].mod
%6 = [[25, 7; 0, 1], []]
```

where we ask for the structures of the $(\mathbf{Z}[i]/I)^*$ for all three ideals of norm 25. In fact, for all moduli with finite part of norm 25 and trivial archimedean part, as the last 3 commands show. See `ideallistarch` to treat general moduli.

The library syntax is **ideallist0**(*nf*, *bound*, *flag*), where *bound* must be a C long integer. Also available is **ideallist**(*nf*, *bound*), corresponding to the case *flag* = 4.

3.6.59 ideallistarch(*nf*, *list*, *arch*): *list* is a vector of vectors of bid's, as output by `ideallist` with flag 0 to 3. Return a vector of vectors with the same number of components as the original *list*. The leaves give information about moduli whose finite part is as in original list, in the same order, and archimedean part is now *arch* (it was originally trivial). The information contained is of the same kind as was present in the input; see `ideallist`, in particular the meaning of *flag*.

```
? bnf = bnfinit(x^2-2);
? bnf.sign
%2 = [2, 0]                          \\ two places at infinity
? L = ideallist(bnf, 100, 0);
? l = L[98]; vector(#l, i, l[i].clgp)
%4 = [[42, [42]], [36, [6, 6]], [42, [42]]]
? La = ideallistarch(bnf, L, [1,1]); \\ add them to the modulus
? l = La[98]; vector(#l, i, l[i].clgp)
%6 = [[168, [42, 2, 2]], [144, [6, 6, 2, 2]], [168, [42, 2, 2]]]
```

Of course, the results above are obvious: adding t places at infinity will add t copies of $\mathbf{Z}/2\mathbf{Z}$ to the ray class group. The following application is more typical:

```
? L = ideallist(bnf, 100, 2);        \\ units are required now
? La = ideallistarch(bnf, L, [1,1]);
? H = bnrclassnolist(bnf, La);
? H[98];
%6 = [2, 12, 2]
```

The library syntax is **ideallistarch**(*nf*, *list*, *arch*).

3.6.60 ideallog(nf, x, bid): nf is a number field, bid a "big ideal" as output by `idealstar` and x a non-necessarily integral element of nf which must have valuation equal to 0 at all prime ideals dividing $I = bid[1]$. This function computes the "discrete logarithm" of x on the generators given in $bid[2]$. In other words, if g_i are these generators, of orders d_i respectively, the result is a column vector of integers (x_i) such that $0 \le x_i < d_i$ and

$$x \equiv \prod_i g_i^{x_i} \pmod{{}^*I} \ .$$

Note that when I is a module, this implies also sign conditions on the embeddings.

The library syntax is **zideallog**(nf, x, bid).

3.6.61 idealmin$(nf, x, \{vdir\})$: computes a minimum of the ideal x in the direction $vdir$ in the number field nf.

The library syntax is **minideal**$(nf, x, vdir, prec)$, where an omitted $vdir$ is coded as `NULL`.

3.6.62 idealmul$(nf, x, y, \{flag = 0\})$: ideal multiplication of the ideals x and y in the number field nf. The result is a generating set for the ideal product with at most n elements, and is in Hermite normal form if either x or y is in HNF or is a prime ideal as output by `idealprimedec`, and this is given together with the sum of the Archimedean information in x and y if both are given.

If $flag$ is non-zero, reduce the result using `idealred`.

The library syntax is **idealmul**(nf, x, y) ($flag = 0$) or **idealmulred**$(nf, x, y, prec)$ ($flag \neq 0$), where as usual, $prec$ is a C long integer representing the precision.

3.6.63 idealnorm(nf, x): computes the norm of the ideal x in the number field nf.

The library syntax is **idealnorm**(nf, x).

3.6.64 idealpow$(nf, x, k, \{flag = 0\})$: computes the k-th power of the ideal x in the number field nf. k can be positive, negative or zero. The result is NOT reduced, it is really the k-th ideal power, and is given in HNF.

If $flag$ is non-zero, reduce the result using `idealred`. Note however that this is NOT the same as as `idealpow`(nf, x, k) followed by reduction, since the reduction is performed throughout the powering process.

The library syntax corresponding to $flag = 0$ is **idealpow**(nf, x, k). If k is a `long`, you can use **idealpows**(nf, x, k). Corresponding to $flag = 1$ is **idealpowred**$(nf, vp, k, prec)$, where $prec$ is a `long`.

3.6.65 idealprimedec(nf, p): computes the prime ideal decomposition of the prime number p in the number field nf. p must be a (positive) prime number. Note that the fact that p is prime is not checked, so if a non-prime p is given the result is undefined.

The result is a vector of *pr* structures, each representing one of the prime ideals above p in the number field nf. The representation $P = [p, a, e, f, b]$ of a prime ideal means the following. The prime ideal is equal to $p\mathbf{Z}_K + \alpha\mathbf{Z}_K$ where $\mathbf{Z}_K$ is the ring of integers of the field and $\alpha = \sum_i a_i\omega_i$ where the ω_i form the integral basis *nf*`.zk`, e is the ramification index, f is the residual index, and b represents a $\beta \in \mathbf{Z}_K$ such that $P^{-1} = \mathbf{Z}_K + \beta/p\mathbf{Z}_K$ which will be useful for computing valuations, but which the user can ignore. The number α is guaranteed to have a valuation equal to 1 at the prime ideal (this is automatic if $e > 1$).

The components of `P` should be accessed by member functions: `P.p`, `P.e`, `P.f`, and `P.gen` (returns the vector $[p, a]$).

The library syntax is **primedec**(nf, p).

3.6.66 idealprincipal(nf, x): creates the principal ideal generated by the algebraic number x (which must be of type integer, rational or polmod) in the number field nf. The result is a one-column matrix.

The library syntax is **principalideal**(nf, x).

3.6.67 idealred$(nf, I, \{vdir = 0\})$: LLL reduction of the ideal I in the number field nf, along the direction *vdir*. If *vdir* is present, it must be an $r1 + r2$-component vector ($r1$ and $r2$ number of real and complex places of *nf* as usual).

This function finds a "small" a in I (it is an LLL pseudo-minimum along direction *vdir*). The result is the Hermite normal form of the LLL-reduced ideal rI/a, where r is a rational number such that the resulting ideal is integral and primitive. This is often, but not always, a reduced ideal in the sense of Buchmann. If I is an idele, the logarithmic embeddings of a are subtracted to the Archimedean part.

More often than not, a principal ideal will yield the identity matrix. This is a quick and dirty way to check if ideals are principal without computing a full `bnf` structure, but it's not a necessary condition; hence, a non-trivial result doesn't prove the ideal is non-trivial in the class group.

Note that this is *not* the same as the LLL reduction of the lattice I since ideal operations are involved.

The library syntax is **idealllred**$(nf, x, vdir, prec)$, where an omitted *vdir* is coded as `NULL`.

3.6.68 idealstar$(nf, I, \{flag = 1\})$: outputs a *bid* structure, necessary for computing in the finite abelian group $G = (\mathbf{Z}_K/I)^*$. Here, *nf* is a number field and I is a *modulus*: either an ideal in any form, or a row vector whose first component is an ideal and whose second component is a row vector of r_1 0 or 1.

This *bid* is used in `ideallog` to compute discrete logarithms. It also contains useful information which can be conveniently retrieved as *bid*`.mod` (the modulus), *bid*`.clgp` (G as a finite abelian group), *bid*`.no` (the cardinality of G), *bid*`.cyc` (elementary divisors) and *bid*`.gen` (generators).

If *flag* = 1 (default), the result is a *bid* structure without generators.

If *flag* = 2, as *flag* = 1, but including generators, which wastes some time.

If *flag* = 0, *deprecated*. Only outputs $(\mathbf{Z}_K/I)^*$ as an abelian group, i.e as a 3-component vector $[h, d, g]$: h is the order, d is the vector of SNF cyclic components and g the corresponding generators. This flag is deprecated: it is in fact slightly faster to compute a true *bid* structure, which contains much more information.

The library syntax is **idealstar0**$(nf, I, flag)$.

3.6.69 idealtwoelt$(nf, x, \{a\})$: computes a two-element representation of the ideal x in the number field nf, using a straightforward (exponential time) search. x can be an ideal in any form, (including perhaps an Archimedean part, which is ignored) and the result is a row vector $[a, \alpha]$ with two components such that $x = a\mathbf{Z}_K + \alpha\mathbf{Z}_K$ and $a \in \mathbf{Z}$, where a is the one passed as argument if any. If x is given by at least two generators, a is chosen to be the positive generator of $x \cap \mathbf{Z}$.

Note that when an explicit a is given, we use an asymptotically faster method, however in practice it is usually slower.

The library syntax is **ideal_two_elt0**(nf, x, a), where an omitted a is entered as `NULL`.

3.6.70 idealval(nf, x, vp): gives the valuation of the ideal x at the prime ideal vp in the number field nf, where vp must be a 5-component vector as given by `idealprimedec`.

The library syntax is **idealval**(nf, x, vp), and the result is a `long` integer.

3.6.71 ideleprincipal(nf, x): creates the principal idele generated by the algebraic number x (which must be of type integer, rational or polmod) in the number field nf. The result is a two-component vector, the first being a one-column matrix representing the corresponding principal ideal, and the second being the vector with $r_1 + r_2$ components giving the complex logarithmic embedding of x.

The library syntax is **principalidele**(nf, x).

3.6.72 matalgtobasis(nf, x): nf being a number field in `nfinit` format, and x a matrix whose coefficients are expressed as polmods in nf, transforms this matrix into a matrix whose coefficients are expressed on the integral basis of nf. This is the same as applying `nfalgtobasis` to each entry, but it would be dangerous to use the same name.

The library syntax is **matalgtobasis**(nf, x).

3.6.73 matbasistoalg(nf, x): nf being a number field in `nfinit` format, and x a matrix whose coefficients are expressed as column vectors on the integral basis of nf, transforms this matrix into a matrix whose coefficients are algebraic numbers expressed as polmods. This is the same as applying `nfbasistoalg` to each entry, but it would be dangerous to use the same name.

The library syntax is **matbasistoalg**(nf, x).

3.6.74 modreverse(a): a being a polmod $A(X)$ modulo $T(X)$, finds the "reverse polmod" $B(X)$ modulo $Q(X)$, where Q is the minimal polynomial of a, which must be equal to the degree of T, and such that if θ is a root of T then $\theta = B(\alpha)$ for a certain root α of Q.

This is very useful when one changes the generating element in algebraic extensions.

The library syntax is **polmodrecip**(x).

3.6.75 newtonpoly(x, p): gives the vector of the slopes of the Newton polygon of the polynomial x with respect to the prime number p. The n components of the vector are in decreasing order, where n is equal to the degree of x. Vertical slopes occur iff the constant coefficient of x is zero and are denoted by `VERYBIGINT`, the biggest single precision integer representable on the machine ($2^{31} - 1$ (resp. $2^{63} - 1$) on 32-bit (resp. 64-bit) machines), see Section 3.2.49.

The library syntax is **newtonpoly**(x, p).

3.6.76 nfalgtobasis(nf, x): this is the inverse function of `nfbasistoalg`. Given an object x whose entries are expressed as algebraic numbers in the number field nf, transforms it so that the entries are expressed as a column vector on the integral basis nf`.zk`.

The library syntax is **algtobasis**(nf, x).

3.6.77 nfbasis$(x, \{flag = 0\}, \{fa\})$: integral basis of the number field defined by the irreducible, preferably monic, polynomial x, using a modified version of the round 4 algorithm by default, due to David Ford, Sebastian Pauli and Xavier Roblot. The binary digits of *flag* have the following meaning:

1: assume that no square of a prime greater than the default `primelimit` divides the discriminant of x, i.e. that the index of x has only small prime divisors.

2: use round 2 algorithm. For small degrees and coefficient size, this is sometimes a little faster. (This program is the translation into C of a program written by David Ford in Algeb.)

Thus for instance, if $flag = 3$, this uses the round 2 algorithm and outputs an order which will be maximal at all the small primes.

If *fa* is present, we assume (without checking!) that it is the two-column matrix of the factorization of the discriminant of the polynomial x. Note that it does *not* have to be a complete factorization. This is especially useful if only a local integral basis for some small set of places is desired: only factors with exponents greater or equal to 2 will be considered.

The library syntax is **nfbasis0**$(x, flag, fa)$. An extended version is **nfbasis**$(x, \&d, flag, fa)$, where d receives the discriminant of the number field (*not* of the polynomial x), and an omitted *fa* is input as `NULL`. Also available are **base**$(x, \&d)$ ($flag = 0$), **base2**$(x, \&d)$ ($flag = 2$) and **factoredbase**$(x, fa, \&d)$.

3.6.78 nfbasistoalg(nf, x): this is the inverse function of `nfalgtobasis`. Given an object x whose entries are expressed on the integral basis nf`.zk`, transforms it into an object whose entries are algebraic numbers (i.e. polmods).

The library syntax is **basistoalg**(nf, x).

3.6.79 nfdetint(nf, x): given a pseudo-matrix x, computes a non-zero ideal contained in (i.e. multiple of) the determinant of x. This is particularly useful in conjunction with `nfhnfmod`.

The library syntax is **nfdetint**(nf, x).

3.6.80 nfdisc$(x, \{flag = 0\}, \{fa\})$: field discriminant of the number field defined by the integral, preferably monic, irreducible polynomial x. *flag* and fa are exactly as in `nfbasis`. That is, fa provides the matrix of a partial factorization of the discriminant of x, and binary digits of *flag* are as follows:

1: assume that no square of a prime greater than `primelimit` divides the discriminant.

2: use the round 2 algorithm, instead of the default round 4. This should be slower except maybe for polynomials of small degree and coefficients.

The library syntax is **nfdiscf0**$(x, flag, fa)$ where an omitted fa is input as `NULL`. You can also use **discf**(x) $(flag = 0)$.

3.6.81 nfeltdiv(nf, x, y): given two elements x and y in nf, computes their quotient x/y in the number field nf.

The library syntax is **element_div**(nf, x, y).

3.6.82 nfeltdiveuc(nf, x, y): given two elements x and y in nf, computes an algebraic integer q in the number field nf such that the components of $x - qy$ are reasonably small. In fact, this is functionally identical to `round(nfeltdiv(`nf`,x,y))`.

The library syntax is **nfdiveuc**(nf, x, y).

3.6.83 nfeltdivmodpr(nf, x, y, pr): given two elements x and y in nf and pr a prime ideal in `modpr` format (see `nfmodprinit`), computes their quotient x/y modulo the prime ideal pr.

The library syntax is **element_divmodpr**(nf, x, y, pr).

3.6.84 nfeltdivrem(nf, x, y): given two elements x and y in nf, gives a two-element row vector $[q, r]$ such that $x = qy + r$, q is an algebraic integer in nf, and the components of r are reasonably small.

The library syntax is **nfdivrem**(nf, x, y).

3.6.85 nfeltmod(nf, x, y): given two elements x and y in nf, computes an element r of nf of the form $r = x - qy$ with q and algebraic integer, and such that r is small. This is functionally identical to

$$\texttt{x} - \texttt{nfeltmul}(nf, \texttt{round}(\texttt{nfeltdiv}(nf, \texttt{x}, \texttt{y})), \texttt{y}).$$

The library syntax is **nfmod**(nf, x, y).

3.6.86 nfeltmul(nf, x, y): given two elements x and y in nf, computes their product $x * y$ in the number field nf.

The library syntax is **element_mul**(nf, x, y).

3.6.87 nfeltmulmodpr(nf, x, y, pr): given two elements x and y in nf and pr a prime ideal in `modpr` format (see `nfmodprinit`), computes their product $x * y$ modulo the prime ideal pr.

The library syntax is **element_mulmodpr**(nf, x, y, pr).

3.6.88 nfeltpow(nf, x, k): given an element x in nf, and a positive or negative integer k, computes x^k in the number field nf.

The library syntax is **element_pow**(nf, x, k).

3.6.89 nfeltpowmodpr(nf, x, k, pr): given an element x in nf, an integer k and a prime ideal pr in `modpr` format (see `nfmodprinit`), computes x^k modulo the prime ideal pr.

The library syntax is **element_powmodpr**(nf, x, k, pr).

3.6.90 nfeltreduce$(nf, x, ideal)$: given an ideal in Hermite normal form and an element x of the number field nf, finds an element r in nf such that $x - r$ belongs to the ideal and r is small.

The library syntax is **element_reduce**$(nf, x, ideal)$.

3.6.91 nfeltreducemodpr(nf, x, pr): given an element x of the number field nf and a prime ideal pr in `modpr` format compute a canonical representative for the class of x modulo pr.

The library syntax is **nfreducemodpr**(nf, x, pr).

3.6.92 nfeltval(nf, x, pr): given an element x in nf and a prime ideal pr in the format output by `idealprimedec`, computes their the valuation at pr of the element x. The same result could be obtained using `idealval(`*nf*`,x,`*pr*`)` (since x would then be converted to a principal ideal), but it would be less efficient.

The library syntax is **element_val**(nf, x, pr), and the result is a `long`.

3.6.93 nffactor(nf, x): factorization of the univariate polynomial x over the number field nf given by `nfinit`. x has coefficients in nf (i.e. either scalar, polmod, polynomial or column vector). The main variable of nf must be of *lower* priority than that of x (see Section 2.5.4). However if the polynomial defining the number field occurs explicitly in the coefficients of x (as modulus of a `t_POLMOD`), its main variable must be *the same* as the main variable of x. For example,

```
? nf = nfinit(y^2 + 1);
? nffactor(nf, x^2 + y); \\ OK
? nffactor(nf, x^2 + Mod(y, y^2+1)); \\  OK
? nffactor(nf, x^2 + Mod(z, z^2+1)); \\  WRONG
```

The library syntax is **nffactor**(nf, x).

3.6.94 nffactormod(nf, x, pr): factorization of the univariate polynomial x modulo the prime ideal pr in the number field nf. x can have coefficients in the number field (scalar, polmod, polynomial, column vector) or modulo the prime ideal (intmod modulo the rational prime under pr, polmod or polynomial with intmod coefficients, column vector of intmod). The prime ideal pr *must* be in the format output by `idealprimedec`. The main variable of nf must be of lower priority than that of x (see Section 2.5.4). However if the coefficients of the number field occur explicitly (as polmods) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t (see `nffactor`).

The library syntax is **nffactormod**(nf, x, pr).

3.6.95 nfgaloisapply(nf, aut, x): nf being a number field as output by `nfinit`, and aut being a Galois automorphism of nf expressed either as a polynomial or a polmod (such automorphisms being found using for example one of the variants of `nfgaloisconj`), computes the action of the automorphism aut on the object x in the number field. x can be an element (scalar, polmod, polynomial or column vector) of the number field, an ideal (either given by $\mathbf{Z}_K$-generators or by a $\mathbf{Z}$-basis), a prime ideal (given as a 5-element row vector) or an idele (given as a 2-element row vector). Because of possible confusion with elements and ideals, other vector or matrix arguments are forbidden.

The library syntax is **galoisapply**(nf, aut, x).

3.6.96 nfgaloisconj$(nf, \{flag = 0\}, \{d\})$: nf being a number field as output by `nfinit`, computes the conjugates of a root r of the non-constant polynomial $x = nf[1]$ expressed as polynomials in r. This can be used even if the number field nf is not Galois since some conjugates may lie in the field.

nf can simply be a polynomial if $flag \neq 1$.

If no flags or $flag = 0$, if nf is a number field use a combination of flag 4 and 1 and the result is always complete, else use a combination of flag 4 and 2 and the result is subject to the restriction of $flag = 2$, but a warning is issued when it is not proven complete.

If $flag = 1$, use `nfroots` (require a number field).

If $flag = 2$, use complex approximations to the roots and an integral LLL. The result is not guaranteed to be complete: some conjugates may be missing (no warning issued), especially so if the corresponding polynomial has a huge index. In that case, increasing the default precision may help.

If $flag = 4$, use Allombert's algorithm and permutation testing. If the field is Galois with "weakly" super solvable Galois group, return the complete list of automorphisms, else only the identity element. If present, d is assumed to be a multiple of the least common denominator of the conjugates expressed as polynomial in a root of pol.

A group G is "weakly" super solvable (WKSS) if it contains a super solvable normal subgroup H such that $G = H$, or $G/H \simeq A_4$, or $G/H \simeq S_4$. Abelian and nilpotent groups are WKSS. In practice, almost all groups of small order are WKSS, the exceptions having order 36(1 exception), 48(2), 56(1), 60(1), 72(5), 75(1), 80(1), 96(10) and ≥ 108.

Hence $flag = 4$ permits to quickly check whether a polynomial of order strictly less than 36 is Galois or not. This method is much faster than `nfroots` and can be applied to polynomials of degree larger than 50.

This routine can only compute $\mathbf{Q}$-automorphisms, but it may be used to get K-automorphism for any base field K as follows:

```
rnfgaloisconj(nfK, R) = \\ K-automorphisms of L = K[X] / (R)
{ local(polabs, N, H);
  R *= Mod(1, nfK.pol);             \\ convert coeffs to polmod elts of K
  polabs = rnfequation(nfK, R);
  N = nfgaloisconj(polabs) % R;     \\ Q-automorphisms of L
  H = [];
  for(i=1, #N,                      \\ select the ones that fix K
    if (subst(R, variable(R), Mod(N[i],R)) == 0,
```

```
      H = concat(H,N[i])
    )
  ); H
}
K  = nfinit(y^2 + 7);
polL = x^4 - y*x^3 - 3*x^2 + y*x + 1;
rnfgaloisconj(K, polL)            \\ K-automorphisms of L
```

The library syntax is **galoisconj0**$(nf, flag, d, prec)$. Also available are **galoisconj**(nf) for $flag = 0$, **galoisconj2**$(nf, n, prec)$ for $flag = 2$ where n is a bound on the number of conjugates, and **galoisconj4**(nf, d) corresponding to $flag = 4$.

3.6.97 nfhilbert$(nf, a, b, \{pr\})$: if pr is omitted, compute the global Hilbert symbol (a, b) in nf, that is 1 if $x^2 - ay^2 - bz^2$ has a non trivial solution (x, y, z) in nf, and -1 otherwise. Otherwise compute the local symbol modulo the prime ideal pr (as output by `idealprimedec`).

The library syntax is **nfhilbert**(nf, a, b, pr), where an omitted pr is coded as `NULL`.

3.6.98 nfhnf(nf, x): given a pseudo-matrix (A, I), finds a pseudo-basis in Hermite normal form of the module it generates.

The library syntax is **nfhermite**(nf, x).

3.6.99 nfhnfmod$(nf, x, detx)$: given a pseudo-matrix (A, I) and an ideal $detx$ which is contained in (read integral multiple of) the determinant of (A, I), finds a pseudo-basis in Hermite normal form of the module generated by (A, I). This avoids coefficient explosion. $detx$ can be computed using the function `nfdetint`.

The library syntax is **nfhermitemod**$(nf, x, detx)$.

3.6.100 nfinit$(pol, \{flag = 0\})$: pol being a non-constant, preferably monic, irreducible polynomial in $\mathbf{Z}[X]$, initializes a *number field* structure (`nf`) associated to the field K defined by pol. As such, it's a technical object passed as the first argument to most `nf`*xxx* functions, but it contains some information which may be directly useful. Access to this information via *member functions* is preferred since the specific data organization specified below may change in the future. Currently, `nf` is a row vector with 9 components:

$nf[1]$ contains the polynomial pol (nf`.pol`).

$nf[2]$ contains $[r1, r2]$ (nf`.sign`, nf`.r1`, nf`.r2`), the number of real and complex places of K.

$nf[3]$ contains the discriminant $d(K)$ (nf`.disc`) of K.

$nf[4]$ contains the index of $nf[1]$ (nf`.index`), i.e. $[\mathbf{Z}_K : \mathbf{Z}[\theta]]$, where θ is any root of $nf[1]$.

$nf[5]$ is a vector containing 7 matrices M, G, $T2$, T, MD, TI, MDI useful for certain computations in the number field K.

• M is the $(r1+r2)\times n$ matrix whose columns represent the numerical values of the conjugates of the elements of the integral basis.

• G is such that $T2 = {}^tGG$, where $T2$ is the quadratic form $T_2(x) = \sum |\sigma(x)|^2$, σ running over the embeddings of K into $\mathbf{C}$.

• The $T2$ component is deprecated and currently unused.

• T is the $n \times n$ matrix whose coefficients are $\mathrm{Tr}(\omega_i\omega_j)$ where the ω_i are the elements of the integral basis. Note also that $\det(T)$ is equal to the discriminant of the field K.

• The columns of MD (*nf*`.diff`) express a $\mathbf{Z}$-basis of the different of K on the integral basis.

• TI is equal to $d(K)T^{-1}$, which has integral coefficients. Note that, understood as as ideal, the matrix T^{-1} generates the codifferent ideal.

• Finally, MDI is a two-element representation (for faster ideal product) of $d(K)$ times the codifferent ideal (*nf*`.disc`$*$*nf*`.codiff`, which is an integral ideal). MDI is only used in `idealinv`.

nf[6] is the vector containing the $r1+r2$ roots (*nf*`.roots`) of *nf*[1] corresponding to the $r1+r2$ embeddings of the number field into $\mathbf{C}$ (the first $r1$ components are real, the next $r2$ have positive imaginary part).

nf[7] is an integral basis for $\mathbf{Z}_K$ (*nf*`.zk`) expressed on the powers of θ. Its first element is guaranteed to be 1. This basis is LLL-reduced with respect to T_2 (strictly speaking, it is a permutation of such a basis, due to the condition that the first element be 1).

nf[8] is the $n \times n$ integral matrix expressing the power basis in terms of the integral basis, and finally

nf[9] is the $n \times n^2$ matrix giving the multiplication table of the integral basis.

If a non monic polynomial is input, `nfinit` will transform it into a monic one, then reduce it (see *flag* = 3). It is allowed, though not very useful given the existence of `nfnewprec`, to input a `nf` or a `bnf` instead of a polynomial.

```
? nf = nfinit(x^3 - 12); \\ initialize number field Q[X] / (X^3 - 12)
? nf.pol   \\ defining polynomial
%2 = x^3 - 12
? nf.disc  \\ field discriminant
%3 = -972
? nf.index \\ index of power basis order in maximal order
%4 = 2
? nf.zk    \\ integer basis, lifted to Q[X]
%5 = [1, x, 1/2*x^2]
? nf.sign  \\ signature
%6 = [1, 1]
? factor(abs(nf.disc ))  \\ determines ramified primes
%7 =
[2 2]

[3 5]
? idealfactor(nf, 2)
%8 =
[[2, [0, 0, -1]~, 3, 1, [0, 1, 0]~] 3]  \\  P_2^3
```

In case *pol* has a huge discriminant which is difficult to factor, the special input format $[pol, B]$ is also accepted where *pol* is a polynomial as above and B is the integer basis, as would be computed by `nfbasis`. This is useful if the integer basis is known in advance, or was computed conditionnally.

```
? pol = polcompositum(x^5 - 101, polcyclo(7))[1];
? B = nfbasis(pol, 1);   \\ faster than nfbasis(pol), but conditional
```

```
? nf = nfinit( [pol, B] );
? factor( abs(nf.disc) )
[5 18]

[7 25]

[101 24]
```

`B` is conditional when its discriminant, which is `nf.disc`, can't be factored. In this example, the above factorization proves the correctness of the computation.

If *flag* = 2: *pol* is changed into another polynomial P defining the same number field, which is as simple as can easily be found using the `polred` algorithm, and all the subsequent computations are done using this new polynomial. In particular, the first component of the result is the modified polynomial.

If *flag* = 3, does a `polred` as in case 2, but outputs $[nf, \texttt{Mod}(a, P)]$, where nf is as before and $\texttt{Mod}(a, P) = \texttt{Mod}(x, pol)$ gives the change of variables. This is implicit when *pol* is not monic: first a linear change of variables is performed, to get a monic polynomial, then a `polred` reduction.

If *flag* = 4, as 2 but uses a partial `polred`.

If *flag* = 5, as 3 using a partial `polred`.

The library syntax is **nfinit0**$(x, flag, prec)$.

3.6.101 nfisideal(nf, x): returns 1 if x is an ideal in the number field nf, 0 otherwise.

The library syntax is **isideal**(x).

3.6.102 nfisincl(x, y): tests whether the number field K defined by the polynomial x is conjugate to a subfield of the field L defined by y (where x and y must be in $\mathbf{Q}[X]$). If they are not, the output is the number 0. If they are, the output is a vector of polynomials, each polynomial a representing an embedding of K into L, i.e. being such that $y \mid x \circ a$.

If y is a number field (nf), a much faster algorithm is used (factoring x over y using `nffactor`). Before version 2.0.14, this wasn't guaranteed to return all the embeddings, hence was triggered by a special flag. This is no more the case.

The library syntax is **nfisincl**$(x, y, flag)$.

3.6.103 nfisisom(x, y): as `nfisincl`, but tests for isomorphism. If either x or y is a number field, a much faster algorithm will be used.

The library syntax is **nfisisom**$(x, y, flag)$.

3.6.104 nfnewprec(nf): transforms the number field nf into the corresponding data using current (usually larger) precision. This function works as expected if nf is in fact a *bnf* (update *bnf* to current precision) but may be quite slow (many generators of principal ideals have to be computed).

The library syntax is **nfnewprec**$(nf, prec)$.

3.6.105 nfkermodpr(nf, a, pr): kernel of the matrix a in $\mathbf{Z}_K/pr$, where pr is in **modpr** format (see `nfmodprinit`).

The library syntax is **nfkermodpr**(nf, a, pr).

3.6.106 nfmodprinit(nf, pr): transforms the prime ideal pr into `modpr` format necessary for all operations modulo pr in the number field nf.

The library syntax is **nfmodprinit**(nf, pr).

3.6.107 nfsubfields$(pol, \{d = 0\})$: finds all subfields of degree d of the number field defined by the (monic, integral) polynomial pol (all subfields if d is null or omitted). The result is a vector of subfields, each being given by $[g, h]$, where g is an absolute equation and h expresses one of the roots of g in terms of the root x of the polynomial defining nf. This routine uses J. Klüners's algorithm in the general case, and B. Allombert's `galoissubfields` when nf is Galois (with weakly supersolvable Galois group).

The library syntax is **subfields**(nf, d).

3.6.108 nfroots$(\{nf\}, x)$: roots of the polynomial x in the number field nf given by `nfinit` without multiplicity (in $\mathbf{Q}$ if nf is omitted). x has coefficients in the number field (scalar, polmod, polynomial, column vector). The main variable of nf must be of lower priority than that of x (see Section 2.5.4). However if the coefficients of the number field occur explicitly (as polmods) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t (see `nffactor`).

The library syntax is **nfroots**(nf, x).

3.6.109 nfrootsof1(nf): computes the number of roots of unity w and a primitive w-th root of unity (expressed on the integral basis) belonging to the number field nf. The result is a two-component vector $[w, z]$ where z is a column vector expressing a primitive w-th root of unity on the integral basis nf`.zk`.

The library syntax is **rootsof1**(nf).

3.6.110 nfsnf(nf, x): given a torsion module x as a 3-component row vector $[A, I, J]$ where A is a square invertible $n \times n$ matrix, I and J are two ideal lists, outputs an ideal list $d_1, \ldots, d_n$ which is the Smith normal form of x. In other words, x is isomorphic to $\mathbf{Z}_K/d_1 \oplus \cdots \oplus \mathbf{Z}_K/d_n$ and d_i divides d_{i-1} for $i \geq 2$. The link between x and $[A, I, J]$ is as follows: if e_i is the canonical basis of K^n, $I = [b_1, \ldots, b_n]$ and $J = [a_1, \ldots, a_n]$, then x is isomorphic to

$$(b_1 e_1 \oplus \cdots \oplus b_n e_n)/(a_1 A_1 \oplus \cdots \oplus a_n A_n) \ ,$$

where the A_j are the columns of the matrix A. Note that every finitely generated torsion module can be given in this way, and even with $b_i = Z_K$ for all i.

The library syntax is **nfsmith**(nf, x).

3.6.111 nfsolvemodpr(nf, a, b, pr): solution of $a \cdot x = b$ in $\mathbf{Z}_K/pr$, where a is a matrix and b a column vector, and where pr is in **modpr** format (see `nfmodprinit`).

The library syntax is **nfsolvemodpr**(nf, a, b, pr).

3.6.112 polcompositum$(P, Q, \{flag = 0\})$: P and Q being squarefree polynomials in $\mathbf{Z}[X]$ in the same variable, outputs the simple factors of the étale $\mathbf{Q}$-algebra $A = \mathbf{Q}(X, Y)/(P(X), Q(Y))$. The factors are given by a list of polynomials R in $\mathbf{Z}[X]$, associated to the number field $\mathbf{Q}(X)/(R)$, and sorted by increasing degree (with respect to lexicographic ordering for factors of equal degrees). Returns an error if one of the polynomials is not squarefree.

Note that it is more efficient to reduce to the case where P and Q are irreducible first. The routine will not perform this for you, since it may be expensive, and the inputs are irreducible in most applications anyway. Assuming P is irreducible (of smaller degree than Q for efficiency), it is in general *much* faster to proceed as follows

```
nf = nfinit(P); L = nffactor(nf, Q)[,1];
vector(#L, i, rnfequation(nf, L[i]))
```

to obtain the same result. If you are only interested in the degrees of the simple factors, the `rnfequation` instruction can be replaced by a trivial `poldegree(P) * poldegree(L[i])`.

If $flag = 1$, outputs a vector of 4-component vectors $[R, a, b, k]$, where R ranges through the list of all possible compositums as above, and a (resp. b) expresses the root of P (resp. Q) as an element of $\mathbf{Q}(X)/(R)$. Finally, k is a small integer such that $b + ka = X$ modulo R.

A compositum is quite often defined by a complicated polynomial, which it is advisable to reduce before further work. Here is a simple example involving the field $\mathbf{Q}(\zeta_5, 5^{1/5})$:

```
? z = polcompositum(x^5 - 5, polcyclo(5), 1)[1];
? pol = z[1]                    \\ pol defines the compositum
%2 = x^20 + 5*x^19 + 15*x^18 + 35*x^17 + 70*x^16 + 141*x^15 + 260*x^14 \
  + 355*x^13 + 95*x^12 - 1460*x^11 - 3279*x^10 - 3660*x^9 - 2005*x^8    \
  + 705*x^7 + 9210*x^6 + 13506*x^5 + 7145*x^4 - 2740*x^3 + 1040*x^2     \
  - 320*x + 256
? a = z[2]; a^5 - 5             \\ a is a fifth root of 5
%3 = 0
? z = polredabs(pol, 1);        \\ look for a simpler polynomial
? pol = z[1]
%5 = x^20 + 25*x^10 + 5
? a = subst(a.pol, x, z[2])     \\ a in the new coordinates
%6 = Mod(-5/22*x^19 + 1/22*x^14 - 123/22*x^9 + 9/11*x^4, x^20 + 25*x^10 + 5)
```

The library syntax is **polcompositum0**$(P, Q, flag)$.

3.6.113 polgalois(x): Galois group of the non-constant polynomial $x \in \mathbf{Q}[X]$. In the present version 2.3.5, x must be irreducible and the degree of x must be less than or equal to 7. On certain versions for which the data file of Galois resolvents has been installed (available in the Unix distribution as a separate package), degrees 8, 9, 10 and 11 are also implemented.

The output is a 4-component vector $[n, s, k, name]$ with the following meaning: n is the cardinality of the group, s is its signature ($s = 1$ if the group is a subgroup of the alternating group A_n, $s = -1$ otherwise) and name is a character string containing name of the transitive group according to the GAP 4 transitive groups library by Alexander Hulpke.

k is more arbitrary and the choice made up to version 2.2.3 of PARI is rather unfortunate: for $n > 7$, k is the numbering of the group among all transitive subgroups of S_n, as given in "The transitive groups of degree up to eleven", G. Butler and J. McKay, *Communications in Algebra*,

vol. 11, 1983, pp. 863–911 (group k is denoted T_k there). And for $n \leq 7$, it was ad hoc, so as to ensure that a given triple would design a unique group. Specifically, for polynomials of degree ≤ 7, the groups are coded as follows, using standard notations

In degree 1: $S_1 = [1, 1, 1]$.

In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1]$, $S_3 = [6, -1, 1]$.

In degree 4: $C_4 = [4, -1, 1]$, $V_4 = [4, 1, 1]$, $D_4 = [8, -1, 1]$, $A_4 = [12, 1, 1]$, $S_4 = [24, -1, 1]$.

In degree 5: $C_5 = [5, 1, 1]$, $D_5 = [10, 1, 1]$, $M_{20} = [20, -1, 1]$, $A_5 = [60, 1, 1]$, $S_5 = [120, -1, 1]$.

In degree 6: $C_6 = [6, -1, 1]$, $S_3 = [6, -1, 2]$, $D_6 = [12, -1, 1]$, $A_4 = [12, 1, 1]$, $G_{18} = [18, -1, 1]$, $S_4^- = [24, -1, 1]$, $A_4 \times C_2 = [24, -1, 2]$, $S_4^+ = [24, 1, 1]$, $G_{36}^- = [36, -1, 1]$, $G_{36}^+ = [36, 1, 1]$, $S_4 \times C_2 = [48, -1, 1]$, $A_5 = PSL_2(5) = [60, 1, 1]$, $G_{72} = [72, -1, 1]$, $S_5 = PGL_2(5) = [120, -1, 1]$, $A_6 = [360, 1, 1]$, $S_6 = [720, -1, 1]$.

In degree 7: $C_7 = [7, 1, 1]$, $D_7 = [14, -1, 1]$, $M_{21} = [21, 1, 1]$, $M_{42} = [42, -1, 1]$, $PSL_2(7) = PSL_3(2) = [168, 1, 1]$, $A_7 = [2520, 1, 1]$, $S_7 = [5040, -1, 1]$.

This is deprecated and obsolete, but for reasons of backward compatibility, we cannot change this behaviour yet. So you can use the default `new_galois_format` to switch to a consistent naming scheme, namely k is always the standard numbering of the group among all transitive subgroups of S_n. If this default is in effect, the above groups will be coded as:

In degree 1: $S_1 = [1, 1, 1]$.

In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1]$, $S_3 = [6, -1, 2]$.

In degree 4: $C_4 = [4, -1, 1]$, $V_4 = [4, 1, 2]$, $D_4 = [8, -1, 3]$, $A_4 = [12, 1, 4]$, $S_4 = [24, -1, 5]$.

In degree 5: $C_5 = [5, 1, 1]$, $D_5 = [10, 1, 2]$, $M_{20} = [20, -1, 3]$, $A_5 = [60, 1, 4]$, $S_5 = [120, -1, 5]$.

In degree 6: $C_6 = [6, -1, 1]$, $S_3 = [6, -1, 2]$, $D_6 = [12, -1, 3]$, $A_4 = [12, 1, 4]$, $G_{18} = [18, -1, 5]$, $A_4 \times C_2 = [24, -1, 6]$, $S_4^+ = [24, 1, 7]$, $S_4^- = [24, -1, 8]$, $G_{36}^- = [36, -1, 9]$, $G_{36}^+ = [36, 1, 10]$, $S_4 \times C_2 = [48, -1, 11]$, $A_5 = PSL_2(5) = [60, 1, 12]$, $G_{72} = [72, -1, 13]$, $S_5 = PGL_2(5) = [120, -1, 14]$, $A_6 = [360, 1, 15]$, $S_6 = [720, -1, 16]$.

In degree 7: $C_7 = [7, 1, 1]$, $D_7 = [14, -1, 2]$, $M_{21} = [21, 1, 3]$, $M_{42} = [42, -1, 4]$, $PSL_2(7) = PSL_3(2) = [168, 1, 5]$, $A_7 = [2520, 1, 6]$, $S_7 = [5040, -1, 7]$.

Warning: The method used is that of resolvent polynomials and is sensitive to the current precision. The precision is updated internally but, in very rare cases, a wrong result may be returned if the initial precision was not sufficient.

The library syntax is **polgalois**$(x, prec)$. To enable the new format in library mode, set the global variable `new_galois_format` to 1.

3.6.114 polred$(x, \{flag = 0\}, \{fa\})$: finds polynomials with reasonably small coefficients defining subfields of the number field defined by x. One of the polynomials always defines $\mathbf{Q}$ (hence is equal to $x - 1$), and another always defines the same number field as x if x is irreducible. All x accepted by `nfinit` are also allowed here (e.g. non-monic polynomials, `nf`, `bnf`, `[x,Z_K_basis]`).

The following binary digits of *flag* are significant:

1: possibly use a suborder of the maximal order. The primes dividing the index of the order chosen are larger than `primelimit` or divide integers stored in the `addprimes` table.

2: gives also elements. The result is a two-column matrix, the first column giving the elements defining these subfields, the second giving the corresponding minimal polynomials.

If fa is given, it is assumed that it is the two-column matrix of the factorization of the discriminant of the polynomial x.

The library syntax is **polred0**$(x, flag, fa)$, where an omitted fa is coded by `NULL`. Also available are **polred**(x) and **factoredpolred**(x, fa), both corresponding to $flag = 0$.

3.6.115 polredabs$(x, \{flag = 0\})$: finds one of the polynomial defining the same number field as the one defined by x, and such that the sum of the squares of the modulus of the roots (i.e. the T_2-norm) is minimal. All x accepted by `nfinit` are also allowed here (e.g. non-monic polynomials, `nf`, `bnf`, `[x,Z_K_basis]`).

Warning: this routine uses an exponential-time algorithm to enumerate all potential generators, and may be exceedingly slow when the number field has many subfields, hence a lot of elements of small T_2-norm. E.g. do not try it on the compositum of many quadratic fields, use `polred` instead.

The binary digits of *flag* mean

1: outputs a two-component row vector $[P, a]$, where P is the default output and a is an element expressed on a root of the polynomial P, whose minimal polynomial is equal to x.

4: gives *all* polynomials of minimal T_2 norm (of the two polynomials $P(x)$ and $P(-x)$, only one is given).

16: possibly use a suborder of the maximal order. The primes dividing the index of the order chosen are larger than `primelimit` or divide integers stored in the `addprimes` table. In that case it may happen that the output polynomial does not have minimal T_2 norm.

The library syntax is **polredabs0**$(x, flag)$.

3.6.116 polredord(x): finds polynomials with reasonably small coefficients and of the same degree as that of x defining suborders of the order defined by x. One of the polynomials always defines $\mathbf{Q}$ (hence is equal to $(x - 1)^n$, where n is the degree), and another always defines the same order as x if x is irreducible.

The library syntax is **ordred**(x).

3.6.117 poltschirnhaus(x): applies a random Tschirnhausen transformation to the polynomial x, which is assumed to be non-constant and separable, so as to obtain a new equation for the étale algebra defined by x. This is for instance useful when computing resolvents, hence is used by the `polgalois` function.

The library syntax is **tschirnhaus**(x).

3.6.118 rnfalgtobasis(rnf, x): expresses x on the relative integral basis. Here, *rnf* is a relative number field extension L/K as output by `rnfinit`, and x an element of L in absolute form, i.e. expressed as a polynomial or polmod with polmod coefficients, *not* on the relative integral basis.

The library syntax is **rnfalgtobasis**(rnf, x).

3.6.119 rnfbasis(bnf, M): let K the field represented by *bnf*, as output by `bnfinit`. M is a projective $\mathbf{Z}_K$-module given by a pseudo-basis, as output by `rnfhnfbasis`. The routine returns either a true $\mathbf{Z}_K$-basis of M if it exists, or an $n+1$-element generating set of M if not, where n is the rank of M over K. (Note that n is the size of the pseudo-basis.)

It is allowed to use a polynomial P with coefficients in K instead of M, in which case, M is defined as the ring of integers of $K[X]/(P)$ (P is assumed irreducible over K), viewed as a $\mathbf{Z}_K$-module.

The library syntax is **rnfbasis**(bnf, x).

3.6.120 rnfbasistoalg(rnf, x): computes the representation of x as a polmod with polmods coefficients. Here, *rnf* is a relative number field extension L/K as output by `rnfinit`, and x an element of L expressed on the relative integral basis.

The library syntax is **rnfbasistoalg**(rnf, x).

3.6.121 rnfcharpoly$(nf, T, a, \{v = x\})$: characteristic polynomial of a over nf, where a belongs to the algebra defined by T over nf, i.e. $nf[X]/(T)$. Returns a polynomial in variable v (x by default).

The library syntax is **rnfcharpoly**(nf, T, a, v), where v is a variable number.

3.6.122 rnfconductor$(bnf, pol, \{flag = 0\})$: given *bnf* as output by `bnfinit`, and *pol* a relative polynomial defining an Abelian extension, computes the class field theory conductor of this Abelian extension. The result is a 3-component vector $[conductor, rayclgp, subgroup]$, where *conductor* is the conductor of the extension given as a 2-component row vector $[f_0, f_\infty]$, *rayclgp* is the full ray class group corresponding to the conductor given as a 3-component vector [h,cyc,gen] as usual for a group, and *subgroup* is a matrix in HNF defining the subgroup of the ray class group on the given generators gen. If *flag* is non-zero, check that *pol* indeed defines an Abelian extension, return 0 if it does not.

The library syntax is **rnfconductor**$(rnf, pol, flag)$.

3.6.123 rnfdedekind(nf, pol, pr): given a number field nf as output by `nfinit` and a polynomial pol with coefficients in nf defining a relative extension L of nf, evaluates the relative Dedekind criterion over the order defined by a root of pol for the prime ideal pr and outputs a 3-component vector as the result. The first component is a flag equal to 1 if the enlarged order could be proven to be pr-maximal and to 0 otherwise (it may be maximal in the latter case if pr is ramified in L), the second component is a pseudo-basis of the enlarged order and the third component is the valuation at pr of the order discriminant.

The library syntax is **rnfdedekind**(nf, pol, pr).

3.6.124 rnfdet(nf, M): given a pseudo-matrix M over the maximal order of nf, computes its determinant.

The library syntax is **rnfdet**(nf, M).

3.6.125 rnfdisc(nf, pol): given a number field nf as output by `nfinit` and a polynomial pol with coefficients in nf defining a relative extension L of nf, computes the relative discriminant of L. This is a two-element row vector $[D, d]$, where D is the relative ideal discriminant and d is the relative discriminant considered as an element of $nf^*/{nf^*}^2$. The main variable of nf *must* be of lower priority than that of pol, see Section 2.5.4.

The library syntax is **rnfdiscf**(bnf, pol).

3.6.126 rnfeltabstorel(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being an element of L expressed as a polynomial modulo the absolute equation rnf`.pol`, computes x as an element of the relative extension L/K as a polmod with polmod coefficients.

The library syntax is **rnfelementabstorel**(rnf, x).

3.6.127 rnfeltdown(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being an element of L expressed as a polynomial or polmod with polmod coefficients, computes x as an element of K as a polmod, assuming x is in K (otherwise an error will occur). If x is given on the relative integral basis, apply `rnfbasistoalg` first, otherwise PARI will believe you are dealing with a vector.

The library syntax is **rnfelementdown**(rnf, x).

3.6.128 rnfeltreltoabs(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being an element of L expressed as a polynomial or polmod with polmod coefficients, computes x as an element of the absolute extension $L/\mathbf{Q}$ as a polynomial modulo the absolute equation rnf`.pol`. If x is given on the relative integral basis, apply `rnfbasistoalg` first, otherwise PARI will believe you are dealing with a vector.

The library syntax is **rnfelementreltoabs**(rnf, x).

3.6.129 rnfeltup(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being an element of K expressed as a polynomial or polmod, computes x as an element of the absolute extension $L/\mathbf{Q}$ as a polynomial modulo the absolute equation rnf`.pol`. If x is given on the integral basis of K, apply `nfbasistoalg` first, otherwise PARI will believe you are dealing with a vector.

The library syntax is **rnfelementup**(rnf, x).

3.6.130 rnfequation$(nf, pol, \{flag = 0\})$: given a number field nf as output by `nfinit` (or simply a polynomial) and a polynomial pol with coefficients in nf defining a relative extension L of nf, computes the absolute equation of L over $\mathbf{Q}$.

If $flag$ is non-zero, outputs a 3-component row vector $[z, a, k]$, where z is the absolute equation of L over $\mathbf{Q}$, as in the default behaviour, a expresses as an element of L a root α of the polynomial defining the base field nf, and k is a small integer such that $\theta = \beta + k\alpha$ where θ is a root of z and β a root of pol.

The main variable of nf *must* be of lower priority than that of pol (see Section 2.5.4). Note that for efficiency, this does not check whether the relative equation is irreducible over nf, but only if it is squarefree. If it is reducible but squarefree, the result will be the absolute equation of the étale algebra defined by pol. If pol is not squarefree, an error message will be issued.

The library syntax is **rnfequation0**$(nf, pol, flag)$.

3.6.131 rnfhnfbasis(bnf, x): given bnf as output by `bnfinit`, and either a polynomial x with coefficients in bnf defining a relative extension L of bnf, or a pseudo-basis x of such an extension, gives either a true bnf-basis of L in upper triangular Hermite normal form, if it exists, and returns 0 otherwise.

The library syntax is **rnfhnfbasis**(nf, x).

3.6.132 rnfidealabstorel(rnf, x): let rnf be a relative number field extension L/K as output by `rnfinit`, and x an ideal of the absolute extension $L/\mathbf{Q}$ given by a $\mathbf{Z}$-basis of elements of L. Returns the relative pseudo-matrix in HNF giving the ideal x considered as an ideal of the relative extension L/K.

If x is an ideal in HNF form, associated to an nf structure, for instance as output by `idealhnf`$(nf, \ldots)$, use `rnfidealabstorel(rnf, nf.zk * x)` to convert it to a relative ideal.

The library syntax is **rnfidealabstorel**(rnf, x).

3.6.133 rnfidealdown(rnf, x): let rnf be a relative number field extension L/K as output by `rnfinit`, and x an ideal of L, given either in relative form or by a $\mathbf{Z}$-basis of elements of L (see Section 3.6.132), returns the ideal of K below x, i.e. the intersection of x with K.

The library syntax is **rnfidealdown**(rnf, x).

3.6.134 rnfidealhnf(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being a relative ideal (which can be, as in the absolute case, of many different types, including of course elements), computes the HNF pseudo-matrix associated to x, viewed as a $\mathbf{Z}_K$-module.

The library syntax is **rnfidealhermite**(rnf, x).

3.6.135 rnfidealmul(rnf, x, y): rnf being a relative number field extension L/K as output by `rnfinit` and x and y being ideals of the relative extension L/K given by pseudo-matrices, outputs the ideal product, again as a relative ideal.

The library syntax is **rnfidealmul**(rnf, x, y).

3.6.136 rnfidealnormabs(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being a relative ideal (which can be, as in the absolute case, of many different types, including of course elements), computes the norm of the ideal x considered as an ideal of the absolute extension $L/\mathbf{Q}$. This is identical to `idealnorm(rnfidealnormrel(`rnf`,x))`, but faster.

The library syntax is **rnfidealnormabs**(rnf, x).

3.6.137 rnfidealnormrel(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being a relative ideal (which can be, as in the absolute case, of many different types, including of course elements), computes the relative norm of x as a ideal of K in HNF.

The library syntax is **rnfidealnormrel**(rnf, x).

3.6.138 rnfidealreltoabs(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being a relative ideal, gives the ideal $x\mathbf{Z}_L$ as an absolute ideal of $L/\mathbf{Q}$, in the form of a $\mathbf{Z}$-basis, given by a vector of polynomials (modulo `rnf.pol`). The following routine might be useful:

```
\\ return y = rnfidealreltoabs(rnf,...) as an ideal in HNF form
\\ associated to nf = nfinit( rnf.pol );
idealgentoHNF(nf, y) = mathnf( Mat( nfalgtobasis(nf, y) ) );
```

The library syntax is **rnfidealreltoabs**(rnf, x).

3.6.139 rnfidealtwoelt(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being an ideal of the relative extension L/K given by a pseudo-matrix, gives a vector of two generators of x over $\mathbf{Z}_L$ expressed as polmods with polmod coefficients.

The library syntax is **rnfidealtwoelement**(rnf, x).

3.6.140 rnfidealup(rnf, x): rnf being a relative number field extension L/K as output by `rnfinit` and x being an ideal of K, gives the ideal $x\mathbf{Z}_L$ as an absolute ideal of $L/\mathbf{Q}$, in the form of a $\mathbf{Z}$-basis, given by a vector of polynomials (modulo `rnf.pol`). The following routine might be useful:

```
\\ return y = rnfidealup(rnf,...) as an ideal in HNF form
\\ associated to nf = nfinit( rnf.pol );
idealgentoHNF(nf, y) = mathnf( Mat( nfalgtobasis(nf, y) ) );
```

The library syntax is **rnfidealup**(rnf, x).

3.6.141 rnfinit(nf, pol): nf being a number field in `nfinit` format considered as base field, and pol a polynomial defining a relative extension over nf, this computes all the necessary data to work in the relative extension. The main variable of pol must be of higher priority (see Section 2.5.4) than that of nf, and the coefficients of pol must be in nf.

The result is a row vector, whose components are technical. In the following description, we let K be the base field defined by nf, m the degree of the base field, n the relative degree, L the large field (of relative degree n or absolute degree nm), r_1 and r_2 the number of real and complex places of K.

$rnf[1]$ contains the relative polynomial pol.

$rnf[2]$ is currently unused.

rnf[3] is a two-component row vector $[\mathfrak{d}(L/K), s]$ where $\mathfrak{d}(L/K)$ is the relative ideal discriminant of L/K and s is the discriminant of L/K viewed as an element of $K^*/(K^*)^2$, in other words it is the output of `rnfdisc`.

rnf[4] is the ideal index $\mathfrak{f}$, i.e. such that $d(pol)\mathbf{Z}_K = \mathfrak{f}^2\mathfrak{d}(L/K)$.

rnf[5] is currently unused.

rnf[6] is currently unused.

rnf[7] is a two-component row vector, where the first component is the relative integral pseudo basis expressed as polynomials (in the variable of *pol*) with polmod coefficients in *nf*, and the second component is the ideal list of the pseudobasis in HNF.

rnf[8] is the inverse matrix of the integral basis matrix, with coefficients polmods in *nf*.

rnf[9] is currently unused.

rnf[10] is *nf*.

rnf[11] is the output of `rnfequation(nf, pol, 1)`. Namely, a vector *vabs* with 3 entries describing the *absolute* extension $L/\mathbf{Q}$. *vabs*[1] is an absolute equation, more conveniently obtained as `rnf.pol`. *vabs*[2] expresses the generator α of the number field *nf* as a polynomial modulo the absolute equation *vabs*[1]. *vabs*[3] is a small integer k such that, if β is an abstract root of *pol* and α the generator of *nf*, the generator whose root is *vabs* will be $\beta + k\alpha$. Note that one must be very careful if $k \neq 0$ when dealing simultaneously with absolute and relative quantities since the generator chosen for the absolute extension is not the same as for the relative one. If this happens, one can of course go on working, but we strongly advise to change the relative polynomial so that its root will be $\beta + k\alpha$. Typically, the GP instruction would be

```
pol = subst(pol, x, x - k*Mod(y,nf.pol))
```

rnf[12] is by default unused and set equal to 0. This field is used to store further information about the field as it becomes available (which is rarely needed, hence would be too expensive to compute during the initial `rnfinit` call).

The library syntax is **rnfinitalg**(*nf*, *pol*, *prec*).

3.6.142 rnfisfree(*bnf*, *x*): given *bnf* as output by `bnfinit`, and either a polynomial x with coefficients in *bnf* defining a relative extension L of *bnf*, or a pseudo-basis x of such an extension, returns true (1) if L/bnf is free, false (0) if not.

The library syntax is **rnfisfree**(*bnf*, *x*), and the result is a `long`.

3.6.143 rnfisnorm($T, a, \{flag = 0\}$): similar to `bnfisnorm` but in the relative case. T is as output by `rnfisnorminit` applied to the extension L/K. This tries to decide whether the element a in K is the norm of some x in the extension L/K.

The output is a vector $[x, q]$, where $a = \mathrm{Norm}(x) * q$. The algorithm looks for a solution x which is an S-integer, with S a list of places of K containing at least the ramified primes, the generators of the class group of L, as well as those primes dividing a. If L/K is Galois, then this is enough; otherwise, *flag* is used to add more primes to S: all the places above the primes $p \leq flag$ (resp. $p|flag$) if $flag > 0$ (resp. $flag < 0$).

The answer is guaranteed (i.e. a is a norm iff $q = 1$) if the field is Galois, or, under GRH, if S contains all primes less than $12\log^2|\mathrm{disc}(M)|$, where M is the normal closure of L/K.

If `rnfisnorminit` has determined (or was told) that L/K is Galois, and *flag* $\neq 0$, a Warning is issued (so that you can set *flag* $= 1$ to check whether L/K is known to be Galois, according to T). Example:

```
bnf = bnfinit(y^3 + y^2 - 2*y - 1);
p = x^2 + Mod(y^2 + 2*y + 1, bnf.pol);
T = rnfisnorminit(bnf, p);
rnfisnorm(T, 17)
```

checks whether 17 is a norm in the Galois extension $\mathbf{Q}(\beta)/\mathbf{Q}(\alpha)$, where $\alpha^3 + \alpha^2 - 2\alpha - 1 = 0$ and $\beta^2 + \alpha^2 + 2\alpha + 1 = 0$ (it is).

The library syntax is **rnfisnorm**($T, x,$ *flag*).

3.6.144 rnfisnorminit(*pol*, *polrel*, {*flag* $= 2$}): let K be defined by a root of *pol*, and L/K the extension defined by the polynomial *polrel*. As usual, *pol* can in fact be an *nf*, or *bnf*, etc; if *pol* has degree 1 (the base field is $\mathbf{Q}$), polrel is also allowed to be an *nf*, etc. Computes technical data needed by `rnfisnorm` to solve norm equations $Nx = a$, for x in L, and a in K.

If *flag* $= 0$, do not care whether L/K is Galois or not.

If *flag* $= 1$, L/K is assumed to be Galois (unchecked), which speeds up `rnfisnorm`.

If *flag* $= 2$, let the routine determine whether L/K is Galois.

The library syntax is **rnfisnorminit**(*pol*, *polrel*, *flag*).

3.6.145 rnfkummer(*bnr*, {*subgroup*}, {*deg* $= 0$}): *bnr* being as output by `bnrinit`, finds a relative equation for the class field corresponding to the module in *bnr* and the given congruence subgroup (the full ray class field if *subgroup* is omitted). If *deg* is positive, outputs the list of all relative equations of degree *deg* contained in the ray class field defined by *bnr*, with the *same* conductor as (*bnr*, *subgroup*).

Warning: this routine only works for subgroups of prime index. It uses Kummer theory, adjoining necessary roots of unity (it needs to compute a tough `bnfinit` here), and finds a generator via Hecke's characterization of ramification in Kummer extensions of prime degree. If your extension does not have prime degree, for the time being, you have to split it by hand as a tower / compositum of such extensions.

The library syntax is **rnfkummer**(*bnr*, *subgroup*, *deg*, *prec*), where *deg* is a `long` and an omitted *subgroup* is coded as `NULL`

3.6.146 rnflllgram(*nf*, *pol*, *order*): given a polynomial *pol* with coefficients in *nf* defining a relative extension L and a suborder *order* of L (of maximal rank), as output by `rnfpseudobasis`(*nf*, *pol*) or similar, gives [[*neworder*], U], where *neworder* is a reduced order and U is the unimodular transformation matrix.

The library syntax is **rnflllgram**(*nf*, *pol*, *order*, *prec*).

3.6.147 rnfnormgroup(bnr, pol): bnr being a big ray class field as output by `bnrinit` and pol a relative polynomial defining an Abelian extension, computes the norm group (alias Artin or Takagi group) corresponding to the Abelian extension of $bnf = bnr[1]$ defined by pol, where the module corresponding to bnr is assumed to be a multiple of the conductor (i.e. pol defines a subextension of bnr). The result is the HNF defining the norm group on the given generators of $bnr[5][3]$. Note that neither the fact that pol defines an Abelian extension nor the fact that the module is a multiple of the conductor is checked. The result is undefined if the assumption is not correct.

The library syntax is **rnfnormgroup**(bnr, pol).

3.6.148 rnfpolred(nf, pol): relative version of `polred`. Given a monic polynomial pol with coefficients in nf, finds a list of relative polynomials defining some subfields, hopefully simpler and containing the original field. In the present version 2.3.5, this is slower and less efficient than `rnfpolredabs`.

The library syntax is **rnfpolred**$(nf, pol, prec)$.

3.6.149 rnfpolredabs$(nf, pol, \{flag = 0\})$: relative version of `polredabs`. Given a monic polynomial pol with coefficients in nf, finds a simpler relative polynomial defining the same field. The binary digits of $flag$ mean

1: returns $[P, a]$ where P is the default output and a is an element expressed on a root of P whose characteristic polynomial is pol

2: returns an absolute polynomial (same as `rnfequation(`nf`,rnfpolredabs(`nf`,`pol`))` but faster).

16: possibly use a suborder of the maximal order. This is slower than the default when the relative discriminant is smooth, and much faster otherwise. See Section 3.6.115.

Remark. In the present implementation, this is both faster and much more efficient than `rnfpolred`, the difference being more dramatic than in the absolute case. This is because the implementation of `rnfpolred` is based on (a partial implementation of) an incomplete reduction theory of lattices over number fields, the function `rnflllgram`, which deserves to be improved.

The library syntax is **rnfpolredabs**$(nf, pol, flag, prec)$.

3.6.150 rnfpseudobasis(nf, pol): given a number field nf as output by `nfinit` and a polynomial pol with coefficients in nf defining a relative extension L of nf, computes a pseudo-basis (A, I) for the maximal order $\mathbf{Z}_L$ viewed as a $\mathbf{Z}_K$-module, and the relative discriminant of L. This is output as a four-element row vector $[A, I, D, d]$, where D is the relative ideal discriminant and d is the relative discriminant considered as an element of $nf^*/{nf^*}^2$.

The library syntax is **rnfpseudobasis**(nf, pol).

3.6.151 rnfsteinitz(nf, x): given a number field nf as output by `nfinit` and either a polynomial x with coefficients in nf defining a relative extension L of nf, or a pseudo-basis x of such an extension as output for example by `rnfpseudobasis`, computes another pseudo-basis (A, I) (not in HNF in general) such that all the ideals of I except perhaps the last one are equal to the ring of integers of nf, and outputs the four-component row vector $[A, I, D, d]$ as in `rnfpseudobasis`. The name of this function comes from the fact that the ideal class of the last ideal of I, which is well defined, is the Steinitz class of the $\mathbf{Z}_K$-module $\mathbf{Z}_L$ (its image in $SK_0(\mathbf{Z}_K)$).

The library syntax is **rnfsteinitz**(nf, x).

3.6.152 subgrouplist$(bnr, \{bound\}, \{flag = 0\})$: *bnr* being as output by `bnrinit` or a list of cyclic components of a finite Abelian group G, outputs the list of subgroups of G. Subgroups are given as HNF left divisors of the SNF matrix corresponding to G.

Warning: the present implementation cannot treat a group G where any cyclic factor has more than 2^{31}, resp. 2^{63} elements on a 32-bit, resp. 64-bit architecture. `forsubgroup` is a bit more general and can handle G if all p-Sylow subgroups of G satisfy the condition above.

If *flag* $= 0$ (default) and *bnr* is as output by `bnrinit`, gives only the subgroups whose modulus is the conductor. Otherwise, the modulus is not taken into account.

If *bound* is present, and is a positive integer, restrict the output to subgroups of index less than *bound*. If *bound* is a vector containing a single positive integer B, then only subgroups of index exactly equal to B are computed. For instance

```
? subgrouplist([6,2])
%1 = [[6, 0; 0, 2], [2, 0; 0, 2], [6, 3; 0, 1], [2, 1; 0, 1], [3, 0; 0, 2],
      [1, 0; 0, 2], [6, 0; 0, 1], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],3)    \\ index less than 3
%2 = [[2, 1; 0, 1], [1, 0; 0, 2], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],[3])  \\ index 3
%3 = [[3, 0; 0, 1]]
? bnr = bnrinit(bnfinit(x), [120,[1]], 1);
? L = subgrouplist(bnr, [8]);
```

In the last example, L corresponds to the 24 subfields of $\mathbf{Q}(\zeta_{120})$, of degree 8 and conductor 120∞ (by setting *flag*, we see there are a total of 43 subgroups of degree 8).

```
? vector(#L, i, galoissubcyclo(bnr, L[i]))
```

will produce their equations. (For a general base field, you would have to rely on `bnrstark`, or `rnfkummer`.)

The library syntax is **subgrouplist0**$(bnr, bound, flag)$, where *flag* is a long integer, and an omitted *bound* is coded by `NULL`.

3.6.153 zetak$(znf, x, \{flag = 0\})$: *znf* being a number field initialized by `zetakinit` (*not* by `nfinit`), computes the value of the Dedekind zeta function of the number field at the complex number x. If *flag* $= 1$ computes Dedekind Λ function instead (i.e. the product of the Dedekind zeta function by its gamma and exponential factors).

CAVEAT. This implementation is not satisfactory and must be rewritten. In particular

• The accuracy of the result depends in an essential way on the accuracy of both the `zetakinit` program and the current accuracy. Be wary in particular that x of large imaginary part or, on the contrary, very close to an ordinary integer will suffer from precision loss, yielding fewer significant digits than expected. Computing with 28 eight digits of relative accuracy, we have

```
? zeta(3)
    %1 = 1.2020569031595942853997381615
    ? zeta(3-1e-20)
    %2 = 1.2020569031595942854017194240
    ? zetak(zetakinit(x), 3-1e-20)
    %3 = 1.2020569031595952919  \\ 5 digits are wrong
    ? zetak(zetakinit(x), 3-1e-28)
    %4 = -25.33411749           \\ junk
```

• As the precision increases, results become unexpectedly completely wrong:

```
? \p100
? zetak(zetakinit(x^2-5), -1) - 1/30
%1 = 7.26691813 E-108    \\ perfect
? \p150
? zetak(zetakinit(x^2-5), -1) - 1/30
%2 = -2.486113578 E-156  \\ perfect
? \p200
? zetak(zetakinit(x^2-5), -1) - 1/30
%3 = 4.47... E-75        \\ more than half of the digits are wrong
? \p250
? zetak(zetakinit(x^2-5), -1) - 1/30
%4 = 1.6 E43             \\ junk
```

The library syntax is **glambdak**($znf, x, prec$) or **gzetak**($znf, x, prec$).

3.6.154 zetakinit(x): computes a number of initialization data concerning the number field defined by the polynomial x so as to be able to compute the Dedekind zeta and lambda functions (respectively `zetak`(x) and `zetak`($x, 1$)). This function calls in particular the `bnfinit` program. The result is a 9-component vector v whose components are very technical and cannot really be used by the user except through the `zetak` function. The only component which can be used if it has not been computed already is $v[1][4]$ which is the result of the `bnfinit` call.

This function is very inefficient and should be rewritten. It needs to computes millions of coefficients of the corresponding Dirichlet series if the precision is big. Unless the discriminant is small it will not be able to handle more than 9 digits of relative precision. For instance, `zetakinit(x^8 - 2)` needs 440MB of memory at default precision.

The library syntax is **initzeta**(x).

3.7 Polynomials and power series.

We group here all functions which are specific to polynomials or power series. Many other functions which can be applied on these objects are described in the other sections. Also, some of the functions described here can be applied to other types.

3.7.1 O(pˆe): if p is an integer greater than 2, returns a p-adic 0 of precision e. In all other cases, returns a power series zero with precision given by ev, where v is the X-adic valuation of p with respect to its main variable.

The library syntax is **zeropadic**(p, e) for a p-adic and **zeroser**(v, e) for a power series zero in variable v, which is a `long`. The precision e is a `long`.

3.7.2 deriv$(x, \{v\})$: derivative of x with respect to the main variable if v is omitted, and with respect to v otherwise. The derivative of a scalar type is zero, and the derivative of a vector or matrix is done componentwise. One can use x' as a shortcut if the derivative is with respect to the main variable of x.

By definition, the main variable of a `t_POLMOD` is the main variable among the coefficients from its two polynomial components (representative and modulus); in other words, assuming a polmod represents an element of $R[X]/(T(X))$, the variable X is a mute variable and the derivative is taken with respect to the main variable used in the base ring R.

The library syntax is **deriv**(x, v), where v is a `long`, and an omitted v is coded as -1. When x is a `t_POL`, `derivpol`(x) is a shortcut for `deriv`$(x, -1)$.

3.7.3 eval(x): replaces in x the formal variables by the values that have been assigned to them after the creation of x. This is mainly useful in GP, and not in library mode. Do not confuse this with substitution (see `subst`).

If x is a character string, `eval(`x`)` executes x as a GP command, as if directly input from the keyboard, and returns its output. For convenience, x is evaluated as if `strictmatch` was off. In particular, unused characters at the end of x do not prevent its evaluation:

```
? eval("1a")
% 1 = 1
```

The library syntax is **geval**(x). The more basic functions **poleval**(q, x), **qfeval**(q, x), and **hqfeval**(q, x) evaluate q at x, where q is respectively assumed to be a polynomial, a quadratic form (a symmetric matrix), or an Hermitian form (an Hermitian complex matrix).

3.7.4 factorpadic$(pol, p, r, \{flag = 0\})$: p-adic factorization of the polynomial pol to precision r, the result being a two-column matrix as in `factor`. The factors are normalized so that their leading coefficient is a power of p. r must be strictly larger than the p-adic valuation of the discriminant of pol for the result to make any sense. The method used is a modified version of the round 4 algorithm of Zassenhaus.

If $flag = 1$, use an algorithm due to Buchmann and Lenstra, which is usually less efficient.

The library syntax is **factorpadic4**(pol, p, r), where r is a `long` integer.

3.7.5 intformal$(x, \{v\})$: formal integration of x with respect to the main variable if v is omitted, with respect to the variable v otherwise. Since PARI does not know about "abstract" logarithms (they are immediately evaluated, if only to a power series), logarithmic terms in the result will yield an error. x can be of any type. When x is a rational function, it is assumed that the base ring is an integral domain of characteristic zero.

The library syntax is **integ**(x, v), where v is a `long` and an omitted v is coded as -1.

3.7.6 padicappr(pol, a): vector of p-adic roots of the polynomial *pol* congruent to the p-adic number a modulo p, and with the same p-adic precision as a. The number a can be an ordinary p-adic number (type `t_PADIC`, i.e. an element of $\mathbf{Z}_p$) or can be an integral element of a finite extension of $\mathbf{Q}_p$, given as a `t_POLMOD` at least one of whose coefficients is a `t_PADIC`. In this case, the result is the vector of roots belonging to the same extension of $\mathbf{Q}_p$ as a.

The library syntax is **padicappr**(pol, a).

3.7.7 polcoeff$(x, s, \{v\})$: coefficient of degree s of the polynomial x, with respect to the main variable if v is omitted, with respect to v otherwise. Also applies to power series, scalars (polynomial of degree 0), and to rational functions provided the denominator is a monomial.

The library syntax is **polcoeff0**(x, s, v), where v is a `long` and an omitted v is coded as -1. Also available is **truecoeff**(x, v).

3.7.8 poldegree$(x, \{v\})$: degree of the polynomial x in the main variable if v is omitted, in the variable v otherwise.

The degree of 0 is a fixed negative number, whose exact value should not be used. The degree of a non-zero scalar is 0. Finally, when x is a non-zero polynomial or rational function, returns the ordinary degree of x. Raise an error otherwise.

The library syntax is **poldegree**(x, v), where v and the result are `long`s (and an omitted v is coded as -1). Also available is **degree**(x), which is equivalent to `poldegree(`x`,-1)`.

3.7.9 polcyclo$(n, \{v = x\})$: n-th cyclotomic polynomial, in variable v (x by default). The integer n must be positive.

The library syntax is **cyclo**(n, v), where n and v are `long` integers (v is a variable number, usually obtained through `varn`).

3.7.10 poldisc$(pol, \{v\})$: discriminant of the polynomial *pol* in the main variable is v is omitted, in v otherwise. The algorithm used is the subresultant algorithm.

The library syntax is **poldisc0**(x, v). Also available is **discsr**(x), equivalent to `poldisc0(x,-1)`.

3.7.11 poldiscreduced(f): reduced discriminant vector of the (integral, monic) polynomial f. This is the vector of elementary divisors of $\mathbf{Z}[\alpha]/f'(\alpha)\mathbf{Z}[\alpha]$, where α is a root of the polynomial f. The components of the result are all positive, and their product is equal to the absolute value of the discriminant of f.

The library syntax is **reduceddiscsmith**(x).

3.7.12 polhensellift(x, y, p, e): given a prime p, an integral polynomial x whose leading coefficient is a p-unit, a vector y of integral polynomials that are pairwise relatively prime modulo p, and whose product is congruent to x modulo p, lift the elements of y to polynomials whose product is congruent to x modulo p^e.

The library syntax is **polhensellift**(x, y, p, e) where e must be a `long`.

3.7.13 polinterpolate$(xa, \{ya\}, \{v = x\}, \{\&e\})$: given the data vectors xa and ya of the same length n (xa containing the x-coordinates, and ya the corresponding y-coordinates), this function finds the interpolating polynomial passing through these points and evaluates it at v. If ya is omitted, return the polynomial interpolating the $(i, xa[i])$. If present, e will contain an error estimate on the returned value.

The library syntax is **polint**$(xa, ya, v, \&e)$, where e will contain an error estimate on the returned value.

3.7.14 polisirreducible(pol): *pol* being a polynomial (univariate in the present version 2.3.5), returns 1 if *pol* is non-constant and irreducible, 0 otherwise. Irreducibility is checked over the smallest base field over which *pol* seems to be defined.

The library syntax is **gisirreducible**(pol).

3.7.15 pollead$(x, \{v\})$: leading coefficient of the polynomial or power series x. This is computed with respect to the main variable of x if v is omitted, with respect to the variable v otherwise.

The library syntax is **pollead**(x, v), where v is a `long` and an omitted v is coded as -1. Also available is **leading_term**(x).

3.7.16 pollegendre$(n, \{v = x\})$: creates the n^{th} Legendre polynomial, in variable v.

The library syntax is **legendre**(n), where x is a `long`.

3.7.17 polrecip(pol): reciprocal polynomial of *pol*, i.e. the coefficients are in reverse order. *pol* must be a polynomial.

The library syntax is **polrecip**(x).

3.7.18 polresultant$(x, y, \{v\}, \{flag = 0\})$: resultant of the two polynomials x and y with exact entries, with respect to the main variables of x and y if v is omitted, with respect to the variable v otherwise. The algorithm assumes the base ring is a domain.

If $flag = 0$, uses the subresultant algorithm.

If $flag = 1$, uses the determinant of Sylvester's matrix instead (here x and y may have non-exact coefficients).

If $flag = 2$, uses Ducos's modified subresultant algorithm. It should be much faster than the default if the coefficient ring is complicated (e.g multivariate polynomials or huge coefficients), and slightly slower otherwise.

The library syntax is **polresultant0**$(x, y, v, flag)$, where v is a `long` and an omitted v is coded as -1. Also available are **subres**(x, y) ($flag = 0$) and **resultant2**(x, y) ($flag = 1$).

3.7.19 polroots$(pol, \{flag = 0\})$: complex roots of the polynomial *pol*, given as a column vector where each root is repeated according to its multiplicity. The precision is given as for transcendental functions: in GP it is kept in the variable `realprecision` and is transparent to the user, but it must be explicitly given as a second argument in library mode.

The algorithm used is a modification of A. Schönhage's root-finding algorithm, due to and implemented by X. Gourdon. Barring bugs, it is guaranteed to converge and to give the roots to the required accuracy.

If $flag = 1$, use a variant of the Newton-Raphson method, which is *not* guaranteed to converge, but is rather fast. If you get the messages "too many iterations in roots" or "INTERNAL ERROR: incorrect result in roots", use the default algorithm. This used to be the default root-finding function in PARI until version 1.39.06.

The library syntax is **roots**$(pol, prec)$ or **rootsold**$(pol, prec)$.

3.7.20 polrootsmod$(pol, p, \{flag = 0\})$: row vector of roots modulo p of the polynomial *pol*. The particular non-prime value $p = 4$ is accepted, mainly for 2-adic computations. Multiple roots are *not* repeated.

If p is very small, you may try setting $flag = 1$, which uses a naive search.

The library syntax is **rootmod**(pol, p) ($flag = 0$) or **rootmod2**(pol, p) ($flag = 1$).

3.7.21 polrootspadic(pol, p, r): row vector of p-adic roots of the polynomial *pol*, given to p-adic precision r. Multiple roots are *not* repeated. p is assumed to be a prime, and *pol* to be non-zero modulo p. Note that this is not the same as the roots in $\mathbf{Z}/p^r\mathbf{Z}$, rather it gives approximations in $\mathbf{Z}/p^r\mathbf{Z}$ of the true roots living in $\mathbf{Q}_p$.

If *pol* has inexact `t_PADIC` coefficients, this is not always well-defined; in this case, the equation is first made integral, then lifted to $\mathbf{Z}$. Hence the roots given are approximations of the roots of a polynomial which is p-adically close to the input.

The library syntax is **rootpadic**(pol, p, r), where r is a `long`.

3.7.22 polsturm$(pol, \{a\}, \{b\})$: number of real roots of the real polynomial *pol* in the interval $]a, b]$, using Sturm's algorithm. a (resp. b) is taken to be $-\infty$ (resp. $+\infty$) if omitted.

The library syntax is **sturmpart**(pol, a, b). Use `NULL` to omit an argument. **sturm**(pol) is equivalent to **sturmpart**$(pol, \texttt{NULL}, \texttt{NULL})$. The result is a `long`.

3.7.23 polsubcyclo$(n, d, \{v = x\})$: gives polynomials (in variable v) defining the sub-Abelian extensions of degree d of the cyclotomic field $\mathbf{Q}(\zeta_n)$, where $d \mid \phi(n)$.

If there is exactly one such extension the output is a polynomial, else it is a vector of polynomials, eventually empty.

To be sure to get a vector, you can use `concat([],polsubcyclo(n,d))`

The function `galoissubcyclo` allows to specify more closely which sub-Abelian extension should be computed.

The library syntax is **polsubcyclo**(n, d, v), where n, d and v are `long` and v is a variable number. When $(\mathbf{Z}/n\mathbf{Z})^*$ is cyclic, you can use **subcyclo**(n, d, v), where n, d and v are `long` and v is a variable number.

3.7.24 polsylvestermatrix(x, y): forms the Sylvester matrix corresponding to the two polynomials x and y, where the coefficients of the polynomials are put in the columns of the matrix (which is the natural direction for solving equations afterwards). The use of this matrix can be essential when dealing with polynomials with inexact entries, since polynomial Euclidean division doesn't make much sense in this case.

The library syntax is **sylvestermatrix**(x, y).

3.7.25 polsym(x, n): creates the vector of the symmetric powers of the roots of the polynomial x up to power n, using Newton's formula.

The library syntax is **polsym**(x).

3.7.26 poltchebi$(n, \{v = x\})$: creates the n^{th} Chebyshev polynomial T_n of the first kind in variable v.

The library syntax is **tchebi**(n, v), where n and v are `long` integers (v is a variable number).

3.7.27 polzagier(n, m): creates Zagier's polynomial $P_n^{(m)}$ used in the functions `sumalt` and `sumpos` (with *flag* $= 1$). One must have $m \leq n$. The exact definition can be found in "Convergence acceleration of alternating series", Cohen et al., Experiment. Math., vol. 9, 2000, pp. 3–12.

The library syntax is **polzagreel**$(n, m, prec)$ if the result is only wanted as a polynomial with real coefficients to the precision *prec*, or **polzag**(n, m) if the result is wanted exactly, where n and m are `long`s.

3.7.28 serconvol(x, y): convolution (or Hadamard product) of the two power series x and y; in other words if $x = \sum a_k * X^k$ and $y = \sum b_k * X^k$ then `serconvol`$(x, y) = \sum a_k * b_k * X^k$.

The library syntax is **convol**(x, y).

3.7.29 serlaplace(x): x must be a power series with non-negative exponents. If $x = \sum (a_k/k!) * X^k$ then the result is $\sum a_k * X^k$.

The library syntax is **laplace**(x).

3.7.30 serreverse(x): reverse power series (i.e. x^{-1}, not $1/x$) of x. x must be a power series whose valuation is exactly equal to one.

The library syntax is **recip**(x).

3.7.31 subst(x, y, z): replace the simple variable y by the argument z in the "polynomial" expression x. Every type is allowed for x, but if it is not a genuine polynomial (or power series, or rational function), the substitution will be done as if the scalar components were polynomials of degree zero. In particular, beware that:

```
? subst(1, x, [1,2; 3,4])
%1 =
[1 0]

[0 1]

? subst(1, x, Mat([0,1]))
  ***   forbidden substitution by a non square matrix
```

If x is a power series, z must be either a polynomial, a power series, or a rational function.

The library syntax is **gsubst**(x, y, z), where y is the variable number.

3.7.32 substpol(x, y, z): replace the "variable" y by the argument z in the "polynomial" expression x. Every type is allowed for x, but the same behaviour as `subst` above apply.

The difference with `subst` is that y is allowed to be any polynomial here. The substitution is done as per the following script:

```
subst_poly(pol, from, to) =
{ local(t = 'subst_poly_t, M = from - t);
  subst(lift(Mod(pol,M), variable(M)), t, to)
}
```

For instance

```
? substpol(x^4 + x^2 + 1, x^2, y)
%1 = y^2 + y + 1
? substpol(x^4 + x^2 + 1, x^3, y)
%2 = x^2 + y*x + 1
? substpol(x^4 + x^2 + 1, (x+1)^2, y)
%3 = (-4*y - 6)*x + (y^2 + 3*y - 3)
```

The library syntax is **gsubstpol**(x, y, z).

3.7.33 substvec(x, v, w): v being a vector of monomials (variables), w a vector of expressions of the same length, replace in the expression x all occurences of v_i by w_i. The substitutions are done simultaneously; more precisely, the v_i are first replaced by new variables in x, then these are replaced by the w_i:

```
? substvec([x,y], [x,y], [y,x])
%1 = [y, x]
? substvec([x,y], [x,y], [y,x+y])
%2 = [y, x + y]     \\ not [y, 2*y]
```

The library syntax is **gsubstvec**(x, v, w).

3.7.34 taylor(x, y): Taylor expansion around 0 of x with respect to the simple variable y. x can be of any reasonable type, for example a rational function. The number of terms of the expansion is transparent to the user in GP, but must be given as a second argument in library mode.

The library syntax is **tayl**(x, y, n), where the `long` integer n is the desired number of terms in the expansion.

3.7.35 thue$(tnf, a, \{sol\})$: solves the equation $P(x, y) = a$ in integers x and y, where tnf was created with `thueinit`(P). sol, if present, contains the solutions of $\mathrm{Norm}(x) = a$ modulo units of positive norm in the number field defined by P (as computed by `bnfisintnorm`). If the result is conditional (on the GRH or some heuristic strenghtening), a Warning is printed. Otherwise, the result is unconditional, barring bugs. For instance, here's how to solve the Thue equation $x^{13} - 5y^{13} = -4$:

```
? tnf = thueinit(x^13 - 5);
? thue(tnf, -4)
%1 = [[1, 1]]
```

Hence, the only solution is $x = 1$, $y = 1$ and the result is unconditional. On the other hand:

```
? tnf = thueinit(x^3-2*x^2+3*x-17);
? thue(tnf, -15)
  *** thue: Warning: Non trivial conditional class group.
  *** May miss solutions of the norm equation.
%2 = [[1, 1]]
```

This time the result is conditional. All results computed using this tnf are likewise conditional, *except* for a right-hand side of ± 1.

The library syntax is **thue**(*tnf*, *a*, *sol*), where an omitted *sol* is coded as `NULL`.

3.7.36 thueinit$(P, \{flag = 0\})$: initializes the *tnf* corresponding to P. It is meant to be used in conjunction with `thue` to solve Thue equations $P(x, y) = a$, where a is an integer. If *flag* is non-zero, certify the result unconditionnally. Otherwise, assume GRH, this being much faster of course.

If the conditional computed class group is trivial *or* you are only interested in the case $a = \pm 1$, then results are unconditional anyway. So one should only use the flag is `thue` prints a Warning (see the example there).

The library syntax is **thueinit**$(P, flag, prec)$.

3.8 Vectors, matrices, linear algebra and sets.

Note that most linear algebra functions operating on subspaces defined by generating sets (such as `mathnf`, `qflll`, etc.) take matrices as arguments. As usual, the generating vectors are taken to be the *columns* of the given matrix.

Since PARI does not have a strong typing system, scalars live in unspecified commutative base rings. It is very difficult to write robust linear algebra routines in such a general setting. The developpers's choice has been to assume the base ring is a domain and work over its field of fractions. If the base ring is *not* a domain, one gets an error as soon as a non-zero pivot turns out to be non-invertible. Some functions, e.g. `mathnf` or `mathnfmod`, specifically assume the base ring is **Z**.

3.8.1 algdep$(x, k, \{flag = 0\})$: x being real/complex, or p-adic, finds a polynomial of degree at most k with integer coefficients having x as approximate root. Note that the polynomial which is obtained is not necessarily the "correct" one. In fact it is not even guaranteed to be irreducible. One can check the closeness either by a polynomial evaluation (use `subst`), or by computing the roots of the polynomial given by `algdep` (use `polroots`).

Internally, `lindep`$([1, x, \ldots, x^k], flag)$ is used. If `lindep` is not able to find a relation and returns a lower bound for the sup norm of the smallest relation, `algdep` returns that bound instead. A suitable non-zero value of *flag* may improve on the default behaviour:

```
\\\\\\\\\ LLL
? \p200
? algdep(2^(1/6)+3^(1/5), 30);      \\ wrong in 3.8s
? algdep(2^(1/6)+3^(1/5), 30, 100); \\ wrong in 1s
? algdep(2^(1/6)+3^(1/5), 30, 170); \\ right in 3.3s
? algdep(2^(1/6)+3^(1/5), 30, 200); \\ wrong in 2.9s
? \p250
```

```
? algdep(2^(1/6)+3^(1/5), 30);      \\ right in 2.8s
? algdep(2^(1/6)+3^(1/5), 30, 200); \\ right in 3.4s
\\\\\\\\\ PSLQ
? \p200
? algdep(2^(1/6)+3^(1/5), 30, -3);  \\ failure in 14s.
? \p250
? algdep(2^(1/6)+3^(1/5), 30, -3);  \\ right in 18s
```

Proceeding by increments of 5 digits of accuracy, `algdep` with default flag produces its first correct result at 205 digits, and from then on a steady stream of correct results. Interestingly enough, our PSLQ also reliably succeeds from 205 digits on (and is 5 times slower at that accuracy).

The above example is the testcase studied in a 2000 paper by Borwein and Lisonek, Applications of integer relation algorithms, *Discrete Math.*, **217**, p. 65–82. The paper conludes in the superiority of the PSLQ algorithm, which either shows that PARI's implementation of PSLQ is lacking, or that its LLL is extremely good. The version of PARI tested there was 1.39, which succeeded reliably from precision 265 on, in about 60 as much time as the current version.

The library syntax is **algdep0**$(x, k, flag, prec)$, where k and *flag* are `long`s. Also available is **algdep**$(x, k, prec)$ ($flag = 0$).

3.8.2 charpoly$(A, \{v = x\}, \{flag = 0\})$: characteristic polynomial of A with respect to the variable v, i.e. determinant of $v * I - A$ if A is a square matrix. If A is not a square matrix, it returns the characteristic polynomial of the map "multiplication by A" if A is a scalar, in particular a polmod. E.g. `charpoly(I) = x^2+1`.

The value of *flag* is only significant for matrices.

If $flag = 0$, the method used is essentially the same as for computing the adjoint matrix, i.e. computing the traces of the powers of A.

If $flag = 1$, uses Lagrange interpolation which is almost always slower.

If $flag = 2$, uses the Hessenberg form. This is faster than the default when the coefficients are intmod a prime or real numbers, but is usually slower in other base rings.

The library syntax is **charpoly0**$(A, v, flag)$, where v is the variable number. Also available are the functions **caract**(A, v) ($flag = 1$), **carhess**(A, v) ($flag = 2$), and **caradj**(A, v, pt) where, in this last case, pt is a `GEN*` which, if not equal to `NULL`, will receive the address of the adjoint matrix of A (see `matadjoint`), so both can be obtained at once.

3.8.3 concat$(x, \{y\})$: concatenation of x and y. If x or y is not a vector or matrix, it is considered as a one-dimensional vector. All types are allowed for x and y, but the sizes must be compatible. Note that matrices are concatenated horizontally, i.e. the number of rows stays the same. Using transpositions, it is easy to concatenate them vertically.

To concatenate vectors sideways (i.e. to obtain a two-row or two-column matrix), use `Mat` instead (see the example there). Concatenating a row vector to a matrix having the same number of columns will add the row to the matrix (top row if the vector is x, i.e. comes first, and bottom row otherwise).

The empty matrix `[;]` is considered to have a number of rows compatible with any operation, in particular concatenation. (Note that this is definitely *not* the case for empty vectors `[ ]` or `[ ]~`.)

If y is omitted, x has to be a row vector or a list, in which case its elements are concatenated, from left to right, using the above rules.

```
? concat([1,2], [3,4])
%1 = [1, 2, 3, 4]
? a = [[1,2]~, [3,4]~]; concat(a)
%2 =
[1 3]

[2 4]

? concat([1,2; 3,4], [5,6]~)
%3 =
[1 2 5]

[3 4 6]
? concat([%, [7,8]~, [1,2,3,4]])
%5 =
[1 2 5 7]

[3 4 6 8]

[1 2 3 4]
```

The library syntax is **concat**(x, y).

3.8.4 lindep$(x, \{flag = 0\})$:x being a vector with p-adic or real/complex coefficients, finds a small integral linear combination among these coefficients.

If x is p-adic, *flag* is meaningless and the algorithm LLL-reduces a suitable (dual) lattice.

Otherwise, the value of *flag* determines the algorithm used; in the current version of PARI, we suggest to use *non-negative* values, since it is by far the fastest and most robust implementation. See the detailed example in Section 3.8.1 (`algdep`).

If $flag \geq 0$, uses a floating point (variable precision) LLL algorithm. This is in general much faster than the other variants. If $flag = 0$ the accuracy is chosen internally using a crude heuristic. If $flag > 0$ the computation is done with an accuracy of *flag* decimal digits. In that case, the parameter *flag* should be between 0.6 and 0.9 times the number of correct decimal digits in the input.

If $flag = -1$, uses a variant of the LLL algorithm due to Hastad, Lagarias and Schnorr (STACS 1986). If the precision is too low, the routine may enter an infinite loop.

If $flag = -2$, x is allowed to be (and in any case interpreted as) a matrix. Returns a non trivial element of the kernel of x, or 0 if x has trivial kernel. The element is defined over the field of coefficients of x, and is in general not integral.

If $flag = -3$, uses the PSLQ algorithm. This may return a real number B, indicating that the input accuracy was exhausted and that no relation exist whose sup norm is less than B.

If $flag = -4$, uses an experimental 2-level PSLQ, which does not work at all. (Should be rewritten.)

The library syntax is **lindep0**$(x, flag, prec)$. Also available is **lindep**$(x, prec)$ ($flag = 0$).

3.8.5 listcreate(n): creates an empty list of maximal length n.

This function is useless in library mode.

3.8.6 listinsert($list, x, n$): inserts the object x at position n in *list* (which must be of type `t_LIST`). All the remaining elements of *list* (from position $n+1$ onwards) are shifted to the right. This and `listput` are the only commands which enable you to increase a list's effective length (as long as it remains under the maximal length specified at the time of the `listcreate`).

This function is useless in library mode.

3.8.7 listkill($list$): kill *list*. This deletes all elements from *list* and sets its effective length to 0. The maximal length is not affected.

This function is useless in library mode.

3.8.8 listput($list, x, \{n\}$): sets the n-th element of the list *list* (which must be of type `t_LIST`) equal to x. If n is omitted, or greater than the list current effective length, just appends x. This and `listinsert` are the only commands which enable you to increase a list's effective length (as long as it remains under the maximal length specified at the time of the `listcreate`).

If you want to put an element into an occupied cell, i.e. if you don't want to change the effective length, you can consider the list as a vector and use the usual `list[n] = x` construct.

This function is useless in library mode.

3.8.9 listsort($list, \{flag = 0\}$): sorts *list* (which must be of type `t_LIST`) in place. If *flag* is non-zero, suppresses all repeated coefficients. This is much faster than the `vecsort` command since no copy has to be made.

This function is useless in library mode.

3.8.10 matadjoint(x): adjoint matrix of x, i.e. the matrix y of cofactors of x, satisfying $x * y = \det(x) * \mathrm{Id}$. x must be a (non-necessarily invertible) square matrix.

The library syntax is **adj**(x).

3.8.11 matcompanion(x): the left companion matrix to the polynomial x.

The library syntax is **assmat**(x).

3.8.12 matdet($x, \{flag = 0\}$): determinant of x. x must be a square matrix.

If *flag* $= 0$, uses Gauss-Bareiss.

If *flag* $= 1$, uses classical Gaussian elimination, which is better when the entries of the matrix are reals or integers for example, but usually much worse for more complicated entries like multivariate polynomials.

The library syntax is **det**(x) (*flag* $= 0$) and **det2**(x) (*flag* $= 1$).

3.8.13 matdetint(x): x being an $m \times n$ matrix with integer coefficients, this function computes a *multiple* of the determinant of the lattice generated by the columns of x if it is of rank m, and returns zero otherwise. This function can be useful in conjunction with the function `mathnfmod` which needs to know such a multiple. To obtain the exact determinant (assuming the rank is maximal), you can compute `matdet(mathnfmod(x, matdetint(x)))`.

Note that as soon as one of the dimensions gets large (m or n is larger than 20, say), it will often be much faster to use `mathnf(x, 1)` or `mathnf(x, 4)` directly.

The library syntax is **detint**(x).

3.8.14 matdiagonal(x): x being a vector, creates the diagonal matrix whose diagonal entries are those of x.

The library syntax is **diagonal**(x).

3.8.15 mateigen(x): gives the eigenvectors of x as columns of a matrix.

The library syntax is **eigen**(x).

3.8.16 matfrobenius($M, \{flag = 0\}, \{v = x\}$): returns the Frobenius form of the square matrix `M`. If *flag* $= 1$, returns only the elementary divisors as a vectr of polynomials in the variable `v`. If *flag* $= 2$, returns a two-components vector [F,B] where `F` is the Frobenius form and `B` is the basis change so that $M = B^{-1}FB$.

The library syntax is **matfrobenius**($M, flag, v$), where v is the variable number.

3.8.17 mathess(x): Hessenberg form of the square matrix x.

The library syntax is **hess**(x).

3.8.18 mathilbert(x): x being a `long`, creates the Hilbert matrixof order x, i.e. the matrix whose coefficient (i,j) is $1/(i + j - 1)$.

The library syntax is **mathilbert**(x).

3.8.19 mathnf($x, \{flag = 0\}$): if x is a (not necessarily square) matrix with integer entries, finds the *upper triangular* Hermite normal form of x. If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x.

If *flag* $= 0$, uses the naive algorithm. This should never be used if the dimension is at all large (larger than 10, say). It is recommanded to use either `mathnfmod(x, matdetint(x))` (when x has maximal rank) or `mathnf(x, 1)`. Note that the latter is in general faster than `mathnfmod`, and also provides a base change matrix.

If *flag* $= 1$, uses Batut's algorithm, which is much faster than the default. Outputs a two-component row vector $[H, U]$, where H is the *upper triangular* Hermite normal form of x defined as above, and U is the unimodular transformation matrix such that $xU = [0|H]$. U has in general huge coefficients, in particular when the kernel is large.

If *flag* $= 3$, uses Batut's algorithm, but outputs $[H, U, P]$, such that H and U are as before and P is a permutation of the rows such that P applied to xU gives H. The matrix U is smaller than with *flag* $= 1$, but may still be large.

If *flag* $= 4$, as in case 1 above, but uses a heuristic variant of LLL reduction along the way. The matrix U is in general close to optimal (in terms of smallest L_2 norm), but the reduction is slower than in case 1.

The library syntax is **mathnf0**$(x, flag)$. Also available are **hnf**(x) (*flag* $= 0$) and **hnfall**(x) (*flag* $= 1$). To reduce *huge* (say 400×400 and more) relation matrices (sparse with small entries), you can use the pair `hnfspec` / `hnfadd`. Since this is rather technical and the calling interface may change, they are not documented yet. Look at the code in `basemath/alglin1.c`.

3.8.20 mathnfmod(x, d): if x is a (not necessarily square) matrix of maximal rank with integer entries, and d is a multiple of the (non-zero) determinant of the lattice spanned by the columns of x, finds the *upper triangular* Hermite normal form of x.

If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x. This is much faster than `mathnf` when d is known.

The library syntax is **hnfmod**(x, d).

3.8.21 mathnfmodid(x, d): outputs the (upper triangular) Hermite normal form of x concatenated with d times the identity matrix. Assumes that x has integer entries.

The library syntax is **hnfmodid**(x, d).

3.8.22 matid(n): creates the $n \times n$ identity matrix.

The library syntax is **matid**(n) where n is a `long`.

Related functions are **gscalmat**(x, n), which creates x times the identity matrix (x being a `GEN` and n a `long`), and **gscalsmat**(x, n) which is the same when x is a `long`.

3.8.23 matimage$(x, \{flag = 0\})$: gives a basis for the image of the matrix x as columns of a matrix. A priori the matrix can have entries of any type. If *flag* $= 0$, use standard Gauss pivot. If *flag* $= 1$, use `matsupplement`.

The library syntax is **matimage0**$(x, flag)$. Also available is **image**(x) (*flag* $= 0$).

3.8.24 matimagecompl(x): gives the vector of the column indices which are not extracted by the function `matimage`. Hence the number of components of `matimagecompl(x)` plus the number of columns of `matimage(x)` is equal to the number of columns of the matrix x.

The library syntax is **imagecompl**(x).

3.8.25 matindexrank(x): x being a matrix of rank r, gives two vectors y and z of length r giving a list of rows and columns respectively (starting from 1) such that the extracted matrix obtained from these two vectors using `vecextract`(x, y, z) is invertible.

The library syntax is **indexrank**(x).

3.8.26 matintersect(x, y): x and y being two matrices with the same number of rows each of whose columns are independent, finds a basis of the **Q**-vector space equal to the intersection of the spaces spanned by the columns of x and y respectively. See also the function `idealintersect`, which does the same for free **Z**-modules.

The library syntax is **intersect**(x, y).

3.8.27 matinverseimage(M, y): gives a column vector belonging to the inverse image z of the column vector or matrix y by the matrix M if one exists (i.e such that $Mz = y$), the empty vector otherwise. To get the complete inverse image, it suffices to add to the result any element of the kernel of x obtained for example by `matker`.

The library syntax is **inverseimage**(x, y).

3.8.28 matisdiagonal(x): returns true (1) if x is a diagonal matrix, false (0) if not.

The library syntax is **isdiagonal**(x), and this returns a `long` integer.

3.8.29 matker$(x, \{flag = 0\})$: gives a basis for the kernel of the matrix x as columns of a matrix. A priori the matrix can have entries of any type.

If x is known to have integral entries, set $flag = 1$.

Note: The library function `FpM_ker`(x, p), where x has integer entries *reduced mod* p and p is prime, is equivalent to, but orders of magnitude faster than, `matker(x*Mod(1,p))` and needs much less stack space. To use it under `gp`, type `install(FpM_ker, GG)` first.

The library syntax is **matker0**$(x, flag)$. Also available are **ker**(x) ($flag = 0$), **keri**(x) ($flag = 1$).

3.8.30 matkerint$(x, \{flag = 0\})$: gives an LLL-reduced **Z**-basis for the lattice equal to the kernel of the matrix x as columns of the matrix x with integer entries (rational entries are not permitted).

If $flag = 0$, uses a modified integer LLL algorithm.

If $flag = 1$, uses `matrixqz`$(x, -2)$. If LLL reduction of the final result is not desired, you can save time using `matrixqz(matker(x),-2)` instead.

The library syntax is **matkerint0**$(x, flag)$. Also available is **kerint**(x) ($flag = 0$).

3.8.31 matmuldiagonal(x, d): product of the matrix x by the diagonal matrix whose diagonal entries are those of the vector d. Equivalent to, but much faster than $x * $`matdiagonal`$(d)$.

The library syntax is **matmuldiagonal**(x, d).

3.8.32 matmultodiagonal(x, y): product of the matrices x and y assuming that the result is a diagonal matrix. Much faster than $x * y$ in that case. The result is undefined if $x * y$ is not diagonal.

The library syntax is **matmultodiagonal**(x, y).

3.8.33 matpascal$(x, \{q\})$: creates as a matrix the lower triangular Pascal triangle of order $x + 1$ (i.e. with binomial coefficients up to x). If q is given, compute the q-Pascal triangle (i.e. using q-binomial coefficients).

The library syntax is **matqpascal**(x, q), where x is a `long` and $q =$ `NULL` is used to omit q. Also available is **matpascal**(x).

3.8.34 matrank(x): rank of the matrix x.

The library syntax is **rank**(x), and the result is a `long`.

3.8.35 matrix$(m, n, \{X\}, \{Y\}, \{expr = 0\})$: creation of the $m \times n$ matrix whose coefficients are given by the expression *expr*. There are two formal parameters in *expr*, the first one (X) corresponding to the rows, the second (Y) to the columns, and X goes from 1 to m, Y goes from 1 to n. If one of the last 3 parameters is omitted, fill the matrix with zeroes.

The library syntax is **matrice**(`GEN nlig,GEN ncol,entree *e1,entree *e2,char *expr`).

3.8.36 matrixqz(x, p): x being an $m \times n$ matrix with $m \geq n$ with rational or integer entries, this function has varying behaviour depending on the sign of p:

If $p \geq 0$, x is assumed to be of maximal rank. This function returns a matrix having only integral entries, having the same image as x, such that the GCD of all its $n \times n$ subdeterminants is equal to 1 when p is equal to 0, or not divisible by p otherwise. Here p must be a prime number (when it is non-zero). However, if the function is used when p has no small prime factors, it will either work or give the message "impossible inverse modulo" and a non-trivial divisor of p.

If $p = -1$, this function returns a matrix whose columns form a basis of the lattice equal to $\mathbf{Z}^n$ intersected with the lattice generated by the columns of x.

If $p = -2$, returns a matrix whose columns form a basis of the lattice equal to $\mathbf{Z}^n$ intersected with the $\mathbf{Q}$-vector space generated by the columns of x.

The library syntax is **matrixqz0**(x, p).

3.8.37 matsize(x): x being a vector or matrix, returns a row vector with two components, the first being the number of rows (1 for a row vector), the second the number of columns (1 for a column vector).

The library syntax is **matsize**(x).

3.8.38 matsnf$(X, \{flag = 0\})$: if X is a (singular or non-singular) matrix outputs the vector of elementary divisors of X (i.e. the diagonal of the Smith normal form of X).

The binary digits of *flag* mean:

1 (complete output): if set, outputs $[U, V, D]$, where U and V are two unimodular matrices such that UXV is the diagonal matrix D. Otherwise output only the diagonal of D.

2 (generic input): if set, allows polynomial entries, in which case the input matrix must be square. Otherwise, assume that X has integer coefficients with arbitrary shape.

4 (cleanup): if set, cleans up the output. This means that elementary divisors equal to 1 will be deleted, i.e. outputs a shortened vector D' instead of D. If complete output was required, returns $[U', V', D']$ so that $U'XV' = D'$ holds. If this flag is set, X is allowed to be of the form D or $[U, V, D]$ as would normally be output with the cleanup flag unset.

The library syntax is **matsnf0**$(X, flag)$. Also available is **smith**(X) $(flag = 0)$.

3.8.39 matsolve(x, y): x being an invertible matrix and y a column vector, finds the solution u of $x * u = y$, using Gaussian elimination. This has the same effect as, but is a bit faster, than $x^{-1} * y$.

The library syntax is **gauss**(x, y).

3.8.40 matsolvemod$(m, d, y, \{flag = 0\})$: m being any integral matrix, d a vector of positive integer moduli, and y an integral column vector, gives a small integer solution to the system of congruences $\sum_i m_{i,j}x_j \equiv y_i \pmod{d_i}$ if one exists, otherwise returns zero. Shorthand notation: y (resp. d) can be given as a single integer, in which case all the y_i (resp. d_i) above are taken to be equal to y (resp. d).

```
? m = [1,2;3,4];
? matsolvemod(m, [3,4], [1,2]~)
%2 = [-2, 0]~
? matsolvemod(m, 3, 1) \\ m X = [1,1]~ over F_3
%3 = [-1, 1]~
```

If $flag = 1$, all solutions are returned in the form of a two-component row vector $[x, u]$, where x is a small integer solution to the system of congruences and u is a matrix whose columns give a basis of the homogeneous system (so that all solutions can be obtained by adding x to any linear combination of columns of u). If no solution exists, returns zero.

The library syntax is **matsolvemod0**$(m, d, y, flag)$. Also available are **gaussmodulo**(m, d, y) ($flag = 0$) and **gaussmodulo2**(m, d, y) ($flag = 1$).

3.8.41 matsupplement(x): assuming that the columns of the matrix x are linearly independent (if they are not, an error message is issued), finds a square invertible matrix whose first columns are the columns of x, i.e. supplement the columns of x to a basis of the whole space.

The library syntax is **suppl**(x).

3.8.42 mattranspose(x) or x~: transpose of x. This has an effect only on vectors and matrices.

The library syntax is **gtrans**(x).

3.8.43 minpoly$(A, \{v = x\}, \{flag = 0\})$: minimal polynomial of A with respect to the variable v., i.e. the monic polynomial P of minimal degree (in the variable v) such that $P(A) = 0$.

The library syntax is **minpoly**(A, v), where v is the variable number.

3.8.44 qfgaussred(q): decomposition into squares of the quadratic form represented by the symmetric matrix q. The result is a matrix whose diagonal entries are the coefficients of the squares, and the non-diagonal entries represent the bilinear forms. More precisely, if (a_{ij}) denotes the output, one has

$$q(x) = \sum_i a_{ii}(x_i + \sum_{j>i} a_{ij}x_j)^2$$

The library syntax is **sqred**(x).

3.8.45 qfjacobi(x): x being a real symmetric matrix, this gives a vector having two components: the first one is the vector of eigenvalues of x, the second is the corresponding orthogonal matrix of eigenvectors of x. The method used is Jacobi's method for symmetric matrices.

The library syntax is **jacobi**(x).

3.8.46 qflll$(x, \{flag = 0\})$: LLL algorithm applied to the *columns* of the matrix x. The columns of x must be linearly independent, unless specified otherwise below. The result is a unimodular transformation matrix T such that $x \cdot T$ is an LLL-reduced basis of the lattice generated by the column vectors of x.

If $flag = 0$ (default), the computations are done with floating point numbers, using Householder matrices for orthogonalization. If x has integral entries, then computations are nonetheless approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr).

If $flag = 1$, it is assumed that x is integral. The computation is done entirely with integers. In this case, x needs not be of maximal rank, but if it is not, T will not be square. This is slower and no more accurate than $flag = 0$ above if x has small dimension (say 100 or less).

If $flag = 2$, x should be an integer matrix whose columns are linearly independent. Returns a partially reduced basis for x, using an unpublished algorithm by Peter Montgomery: a basis is said to be *partially reduced* if $|v_i \pm v_j| \geq |v_i|$ for any two distinct basis vectors v_i, v_j.

This is significantly faster than $flag = 1$, esp. when one row is huge compared to the other rows. Note that the resulting basis is *not* LLL-reduced in general.

If $flag = 4$, x is assumed to have integral entries, but needs not be of maximal rank. The result is a two-component vector of matrices: the columns of the first matrix represent a basis of the integer kernel of x (not necessarily LLL-reduced) and the second matrix is the transformation matrix T such that $x \cdot T$ is an LLL-reduced $\mathbf{Z}$-basis of the image of the matrix x.

If $flag = 5$, case as case 4, but x may have polynomial coefficients.

If $flag = 8$, same as case 0, but x may have polynomial coefficients.

The library syntax is **qflll0**$(x, flag, prec)$. Also available are **lll**$(x, prec)$ ($flag = 0$), **lllint**(x) ($flag = 1$), and **lllkerim**(x) ($flag = 4$).

3.8.47 qflllgram$(G, \{flag = 0\})$: same as `qflll`, except that the matrix $G =$ `x~` $*$ `x` is the Gram matrix of some lattice vectors x, and not the coordinates of the vectors themselves. In particular, G must now be a square symmetric real matrix, corresponding to a positive definite quadratic form. The result is a unimodular transformation matrix T such that $x \cdot T$ is an LLL-reduced basis of the lattice generated by the column vectors of x.

If $flag = 0$ (default): the computations are done with floating point numbers, using Householder matrices for orthogonalization. If G has integral entries, then computations are nonetheless approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr).

If $flag = 1$: G has integer entries, still positive but not necessarily definite (i.e x needs not have maximal rank). The computations are all done in integers and should be slower than the default, unless the latter triggers accuracy problems.

$flag = 4$: G has integer entries, gives the kernel and reduced image of x.

$flag = 5$: same as case 4, but G may have polynomial coefficients.

The library syntax is **qflllgram0**$(G, flag, prec)$. Also available are **lllgram**$(G, prec)$ ($flag = 0$), **lllgramint**(G) ($flag = 1$), and **lllgramkerim**(G) ($flag = 4$).

3.8.48 qfminim$(x, \{b\}, \{m\}, \{flag = 0\})$: x being a square and symmetric matrix representing a positive definite quadratic form, this function deals with the vectors of x whose norm is less than or equal to b, enumerated using the Fincke-Pohst algorithm. The function searches for the minimal non-zero vectors if b is omitted. The precise behaviour depends on *flag*.

If $flag = 0$ (default), seeks at most $2m$ vectors. The result is a three-component vector, the first component being the number of vectors found, the second being the maximum norm found, and the last vector is a matrix whose columns are the vectors found, only one being given for each pair $\pm v$ (at most m such pairs). The vectors are returned in no particular order. In this variant, an explicit m must be provided.

If $flag = 1$, ignores m and returns the first vector whose norm is less than b. In this variant, an explicit b must be provided.

In both these cases, x is assumed to have integral entries. The implementation uses low precision floating point computations for maximal speed, which gives incorrect result when x has large entries. (The condition is checked in the code and the routine will raise an error if large rounding errors occur.) A more robust, but much slower, implementation is chosen if the following flag is used:

If $flag = 2$, x can have non integral real entries. In this case, if b is omitted, the "minimal" vectors only have approximately the same norm. If b is omitted, m is an upper bound for the number of vectors that will be stored and returned, but all minimal vectors are nevertheless enumerated. If m is omitted, all vectors found are stored and returned; note that this may be a huge vector!

The library syntax is **qfminim0**$(x, b, m, flag, prec)$, also available are **minim**(x, b, m) ($flag = 0$), **minim2**(x, b, m) ($flag = 1$). In all cases, an omitted b or m is coded as `NULL`.

3.8.49 qfperfection(x): x being a square and symmetric matrix with integer entries representing a positive definite quadratic form, outputs the perfection rank of the form. That is, gives the rank of the family of the s symmetric matrices $v_i v_i^t$, where s is half the number of minimal vectors and the v_i ($1 \leq i \leq s$) are the minimal vectors.

As a side note to old-timers, this used to fail bluntly when x had more than 5000 minimal vectors. Beware that the computations can now be very lengthy when x has many minimal vectors.

The library syntax is **perf**(x).

3.8.50 qfrep$(q, B, \{flag = 0\})$: q being a square and symmetric matrix with integer entries representing a positive definite quadratic form, outputs the vector whose i-th entry, $1 \leq i \leq B$ is half the number of vectors v such that $q(v) = i$. This routine uses a naive algorithm based on `qfminim`, and will fail if any entry becomes larger than 2^{31}.

The binary digits of *flag* mean:

- 1: count vectors of even norm from 1 to $2B$.
- 2: return a `t_VECSMALL` instead of a `t_GEN`

The library syntax is **qfrep0**$(q, B, flag)$.

3.8.51 qfsign(x): signature of the quadratic form represented by the symmetric matrix x. The result is a two-component vector.

The library syntax is **signat**(x).

3.8.52 setintersect(x, y): intersection of the two sets x and y.

The library syntax is **setintersect**(x, y).

3.8.53 setisset(x): returns true (1) if x is a set, false (0) if not. In PARI, a set is simply a row vector whose entries are strictly increasing. To convert any vector (and other objects) into a set, use the function `Set`.

The library syntax is **setisset**(x), and this returns a `long`.

3.8.54 setminus(x, y): difference of the two sets x and y, i.e. set of elements of x which do not belong to y.

The library syntax is **setminus**(x, y).

3.8.55 setsearch$(x, y, \{flag = 0\})$: searches if y belongs to the set x. If it does and *flag* is zero or omitted, returns the index j such that $x[j] = y$, otherwise returns 0. If *flag* is non-zero returns the index j where y should be inserted, and 0 if it already belongs to x (this is meant to be used in conjunction with `listinsert`).

This function works also if x is a *sorted* list (see `listsort`).

The library syntax is **setsearch**$(x, y, flag)$ which returns a `long` integer.

3.8.56 setunion(x, y): union of the two sets x and y.

The library syntax is **setunion**(x, y).

3.8.57 trace(x): this applies to quite general x. If x is not a matrix, it is equal to the sum of x and its conjugate, except for polmods where it is the trace as an algebraic number.

For x a square matrix, it is the ordinary trace. If x is a non-square matrix (but not a vector), an error occurs.

The library syntax is **gtrace**(x).

3.8.58 vecextract$(x, y, \{z\})$: extraction of components of the vector or matrix x according to y. In case x is a matrix, its components are as usual the *columns* of x. The parameter y is a component specifier, which is either an integer, a string describing a range, or a vector.

If y is an integer, it is considered as a mask: the binary bits of y are read from right to left, but correspond to taking the components from left to right. For example, if $y = 13 = (1101)_2$ then the components 1,3 and 4 are extracted.

If y is a vector, which must have integer entries, these entries correspond to the component numbers to be extracted, in the order specified.

If y is a string, it can be

• a single (non-zero) index giving a component number (a negative index means we start counting from the end).

• a range of the form "$a..b$", where a and b are indexes as above. Any of a and b can be omitted; in this case, we take as default values $a = 1$ and $b = -1$, i.e. the first and last components respectively. We then extract all components in the interval $[a, b]$, in reverse order if $b < a$.

In addition, if the first character in the string is `^`, the complement of the given set of indices is taken.

If z is not omitted, x must be a matrix. y is then the *line* specifier, and z the *column* specifier, where the component specifier is as explained above.

```
? v = [a, b, c, d, e];
? vecextract(v, 5)          \\ mask
%1 = [a, c]
? vecextract(v, [4, 2, 1])  \\ component list
%2 = [d, b, a]
? vecextract(v, "2..4")     \\ interval
%3 = [b, c, d]
? vecextract(v, "-1..-3")   \\ interval + reverse order
%4 = [e, d, c]
? vecextract(v, "^2")       \\ complement
%5 = [a, c, d, e]
? vecextract(matid(3), "2..", "..")
%6 =
[0 1 0]

[0 0 1]
```

The library syntax is **extract**(x, y) or **matextract**(x, y, z).

3.8.59 vecsort$(x, \{k\}, \{flag = 0\})$: sorts the vector x in ascending order, using a mergesort method. x must be a vector, and its components integers, reals, or fractions.

If k is present and is an integer, sorts according to the value of the k-th subcomponents of the components of x. Note that mergesort is stable, hence is the initial ordering of "equal" entries (with respect to the sorting criterion) is not changed.

k can also be a vector, in which case the sorting is done lexicographically according to the components listed in the vector k. For example, if $k = [2, 1, 3]$, sorting will be done with respect to the second component, and when these are equal, with respect to the first, and when these are equal, with respect to the third.

The binary digits of *flag* mean:

• 1: indirect sorting of the vector x, i.e. if x is an n-component vector, returns a permutation of $[1, 2, \ldots, n]$ which applied to the components of x sorts x in increasing order. For example, `vecextract(x, vecsort(x,,1))` is equivalent to `vecsort(x)`.

• 2: sorts x by ascending lexicographic order (as per the `lex` comparison function).

• 4: use descending instead of ascending order.

The library syntax is **vecsort0**$(x, k, flag)$. To omit k, use `NULL` instead. You can also use the simpler functions

sort(x) (= **vecsort0**$(x, \texttt{NULL}, 0)$).

indexsort(x) (= **vecsort0**$(x, \texttt{NULL}, 1)$).

lexsort(x) (= **vecsort0**$(x, \texttt{NULL}, 2)$).

Also available are **sindexsort**(x) and **sindexlexsort**(x) which return a `t_VECSMALL` v, where $v[1] \ldots v[n]$ contain the indices.

3.8.60 vector$(n, \{X\}, \{expr = 0\})$: creates a row vector (type `t_VEC`) with n components whose components are the expression *expr* evaluated at the integer points between 1 and n. If one of the last two arguments is omitted, fill the vector with zeroes.

Avoid modifying X within *expr*; if you do, the formal variable still runs from 1 to n. In particular, `vector(n,i,expr)` is not equivalent to

```
v = vector(n)
for (i = 1, n, v[i] = expr)
```

as the following example shows:

```
n = 3
v = vector(n); vector(n, i, i++)            ----> [2, 3, 4]
v = vector(n); for (i = 1, n, v[i] = i++)   ----> [2, 0, 4]
```

The library syntax is **vecteur**(`GEN nmax, entree *ep, char *expr`).

3.8.61 vectorsmall$(n, \{X\}, \{expr = 0\})$: creates a row vector of small integers (type `t_VECSMALL`) with n components whose components are the expression *expr* evaluated at the integer points between 1 and n. If one of the last two arguments is omitted, fill the vector with zeroes.

The library syntax is **vecteursmall**(`GEN nmax, entree *ep, char *expr`).

3.8.62 vectorv$(n, X, expr)$: as `vector`, but returns a column vector (type `t_COL`).

The library syntax is **vvecteur**(`GEN nmax, entree *ep, char *expr`).

3.9 Sums, products, integrals and similar functions.

Although the `gp` calculator is programmable, it is useful to have preprogrammed a number of loops, including sums, products, and a certain number of recursions. Also, a number of functions from numerical analysis like numerical integration and summation of series will be described here.

One of the parameters in these loops must be the control variable, hence a simple variable name. In the descriptions, the letter X will always denote any simple variable name, and represents the formal parameter used in the function. The expression to be summed, integrated, etc. is any legal PARI expression, including of course expressions using loops.

Library mode. Since it is easier to program directly the loops in library mode, these functions are mainly useful for GP programming. Using them in library mode is tricky and we will not give any details, although the reader can try and figure it out by himself by checking the example given for `sum`.

On the other hand, numerical routines code a function (to be integrated, summed, etc.) with two parameters named

```
GEN (*eval)(GEN,void*)
void *E;
```

The second is meant to contain all auxilliary data needed by your function. The first is such that `eval(x, E)` returns your function evaluated at `x`. For instance, one may code the family of functions $f_t : x \to (x+t)^2$ via

```
GEN f(GEN x, void *t) { return gsqr(gadd(x, (GEN)t)); }
```

One can then integrate f_1 between a and b with the call

```
intnum((void*)stoi(1), &fun, a, b, NULL, prec);
```

Since you can set `E` to a pointer to any `struct` (typecast to `void*`) the above mechanism handles arbitrary functions. For simple functions without extra parameters, you may set `E = NULL` and ignore that argument in your function definition.

Numerical integration. Starting with version 2.2.9 the powerful "double exponential" univariate integration method is implemented in `intnum` and its variants. Romberg integration is still available under the name `intnumromb`, but superseded. It is possible to compute numerically integrals to thousands of decimal places in reasonable time, as long as the integrand is regular. It is also reasonable to compute numerically integrals in several variables, although more than two becomes lengthy. The integration domain may be non-compact, and the integrand may have reasonable singularities at endpoints. To use `intnum`, the user must split the integral into a sum of subintegrals where the function has (possible) singularities only at the endpoints. Polynomials in logarithms are not considered singular, and neglecting these logs, singularities are assumed to be algebraic (in other words asymptotic to $C(x-a)^{-\alpha}$ for some α such that $\alpha > -1$ when x is close to a), or to correspond to simple discontinuities of some (higher) derivative of the function. For instance, the point 0 is a singularity of abs(x).

See also the discrete summation methods below (sharing the prefix `sum`).

3.9.1 intcirc($X = a, R, expr, \{tab\}$): numerical integration of *expr* with respect to X on the circle $|X - a| = R$, divided by $2i\pi$. In other words, when *expr* is a meromorphic function, sum of the residues in the corresponding disk. *tab* is as in `intnum`, except that if computed with `intnuminit` it should be with the endpoints `[-1, 1]`.

```
? \p105
? intcirc(s=1, 0.5, zeta(s)) - 1
time = 3,460 ms.
%1 = -2.40... E-104 - 2.7... E-106*I
```

The library syntax is **intcirc**(`void *E, GEN (*eval)(GEN,void*), GEN a,GEN R,GEN tab, long prec`).

3.9.2 intfouriercos$(X = a, b, z, expr, \{tab\})$: numerical integration of $expr(X)\cos(2\pi zX)$ from a to b, in other words Fourier cosine transform (from a to b) of the function represented by *expr*. a and b are coded as in `intnum`, and are not necessarily at infinity, but if they are, oscillations (i.e. $[[\pm 1], \alpha I]$) are forbidden.

The library syntax is **intfouriercos**(`void *E, GEN (*eval)(GEN,void*), GEN a, GEN b, GEN z, GEN tab, long prec`).

3.9.3 intfourierexp$(X = a, b, z, expr, \{tab\})$: numerical integration of $expr(X)\exp(-2\pi zX)$ from a to b, in other words Fourier transform (from a to b) of the function represented by *expr*. Note the minus sign. a and b are coded as in `intnum`, and are not necessarily at infinity but if they are, oscillations (i.e. $[[\pm 1], \alpha I]$) are forbidden.

The library syntax is **intfourierexp**(`void *E, GEN (*eval)(GEN,void*), GEN a, GEN b, GEN z, GEN tab, long prec`).

3.9.4 intfouriersin$(X = a, b, z, expr, \{tab\})$: numerical integration of $expr(X)\sin(2\pi zX)$ from a to b, in other words Fourier sine transform (from a to b) of the function represented by *expr*. a and b are coded as in `intnum`, and are not necessarily at infinity but if they are, oscillations (i.e. $[[\pm 1], \alpha I]$) are forbidden.

The library syntax is **intfouriersin**(`void *E, GEN (*eval)(GEN,void*), GEN a, GEN b, GEN z, GEN tab, long prec`).

3.9.5 intfuncinit$(X = a, b, expr, \{flag = 0\}, \{m = 0\})$: initalize tables for use with integral transforms such as `intmellininv`, etc., where a and b are coded as in `intnum`, *expr* is the function $s(X)$ to which the integral transform is to be applied (which will multiply the weights of integration) and m is as in `intnuminit`. If *flag* is nonzero, assumes that $s(-X) = \overline{s(X)}$, which makes the computation twice as fast. See `intmellininvshort` for examples of the use of this function, which is particularly useful when the function $s(X)$ is lengthy to compute, such as a gamma product.

The library syntax is **intfuncinit**(`void *E, GEN (*eval)(GEN,void*), GEN a,GEN b,long m, long flag, long prec`). Note that the order of m and *flag* are reversed compared to the `GP` syntax.

3.9.6 intlaplaceinv$(X = sig, z, expr, \{tab\})$: numerical integration of $expr(X)e^{Xz}$ with respect to X on the line $\Re(X) = sig$, divided by $2i\pi$, in other words, inverse Laplace transform of the function corresponding to *expr* at the value z.

sig is coded as follows. Either it is a real number σ, equal to the abcissa of integration, and then the function to be integrated is assumed to be slowly decreasing when the imaginary part of the variable tends to $\pm\infty$. Or it is a two component vector $[\sigma, \alpha]$, where σ is as before, and either $\alpha = 0$ for slowly decreasing functions, or $\alpha > 0$ for functions decreasing like $\exp(-\alpha t)$. Note that it is not necessary to choose the exact value of α. *tab* is as in `intnum`.

It is often a good idea to use this function with a value of m one or two higher than the one chosen by default (which can be viewed thanks to the function `intnumstep`), or to increase the abcissa of integration σ. For example:

```
? \p 105
? intlaplaceinv(x=2, 1, 1/x) - 1
time = 350 ms.
%1 = 7.37... E-55 + 1.72... E-54*I \\ not so good
```

```
? m = intnumstep()
%2 = 7
? intlaplaceinv(x=2, 1, 1/x, m+1) - 1
time = 700 ms.
%3 = 3.95... E-97 + 4.76... E-98*I \\ better
? intlaplaceinv(x=2, 1, 1/x, m+2) - 1
time = 1400 ms.
%4 = 0.E-105 + 0.E-106*I \\ perfect but slow.
? intlaplaceinv(x=5, 1, 1/x) - 1
time = 340 ms.
%5 = -5.98... E-85 + 8.08... E-85*I \\ better than %1
? intlaplaceinv(x=5, 1, 1/x, m+1) - 1
time = 680 ms.
%6 = -1.09... E-106 + 0.E-104*I \\ perfect, fast.
? intlaplaceinv(x=10, 1, 1/x) - 1
time = 340 ms.
%7 = -4.36... E-106 + 0.E-102*I \\ perfect, fastest, but why sig = 10?
? intlaplaceinv(x=100, 1, 1/x) - 1
time = 330 ms.
%7 = 1.07... E-72 + 3.2... E-72*I \\ too far now...
```

The library syntax is **intlaplaceinv**(`void *E, GEN (*eval)(GEN,void*), GEN sig,GEN z, GEN tab, long prec`).

3.9.7 intmellininv$(X = sig, z, expr, \{tab\})$: numerical integration of $expr(X)z^{-X}$ with respect to X on the line $\Re(X) = sig$, divided by $2i\pi$, in other words, inverse Mellin transform of the function corresponding to *expr* at the value z.

sig is coded as follows. Either it is a real number σ, equal to the abcissa of integration, and then the function to be integrated is assumed to decrease exponentially fast, of the order of $\exp(-t)$ when the imaginary part of the variable tends to $\pm\infty$. Or it is a two component vector $[\sigma, \alpha]$, where σ is as before, and either $\alpha = 0$ for slowly decreasing functions, or $\alpha > 0$ for functions decreasing like $\exp(-\alpha t)$, such as gamma products. Note that it is not necessary to choose the exact value of α, and that $\alpha = 1$ (equivalent to *sig* alone) is usually sufficient. *tab* is as in `intnum`.

As all similar functions, this function is provided for the convenience of the user, who could use `intnum` directly. However it is in general better to use `intmellininvshort`.

```
? \p 105
? intmellininv(s=2,4, gamma(s)^3);
time = 1,190 ms. \\ reasonable.
? \p 308
? intmellininv(s=2,4, gamma(s)^3);
time = 51,300 ms. \\ slow because of Γ(s)^3.
```

The library syntax is **intmellininv**(`void *E, GEN (*eval)(GEN,void*), GEN sig, GEN z, GEN tab, long prec`).

3.9.8 intmellininvshort(sig, z, tab): numerical integration of $s(X)z^{-X}$ with respect to X on the line $\Re(X) = sig$, divided by $2i\pi$, in other words, inverse Mellin transform of $s(X)$ at the value z. Here $s(X)$ is implicitly contained in *tab* in `intfuncinit` format, typically

```
tab = intfuncinit(T = [-1], [1], s(sig + I*T))
```

or similar commands. Take the example of the inverse Mellin transform of $\Gamma(s)^3$ given in `intmellininv`:

```
? \p 105
? oo = [1]; \\ for clarity
? A = intmellininv(s=2,4, gamma(s)^3);
time = 2,500 ms. \\ not too fast because of Γ(s)^3.
\\  function of real type, decreasing as exp(-3π/2 · |t|)
? tab = intfuncinit(t=[-oo, 3*Pi/2],[oo, 3*Pi/2], gamma(2+I*t)^3, 1);
time = 1,370 ms.
? intmellininvshort(2,4, tab) - A
time = 50 ms.
%4 = -1.26... - 3.25...E-109*I \\ 50 times faster than A and perfect.
? tab2 = intfuncinit(t=-oo, oo, gamma(2+I*t)^3, 1);
? intmellininvshort(2,4, tab2)
%6 = -1.2...E-42 - 3.2...E-109*I  \\ 63 digits lost
```

In the computation of *tab*, it was not essential to include the *exact* exponential decrease of $\Gamma(2+it)^3$. But as the last example shows, a rough indication *must* be given, otherwise slow decrease is assumed, resulting in catastrophic loss of accuracy.

The library syntax is **intmellininvshort**(`GEN sig, GEN z, GEN tab, long prec`).

3.9.9 intnum$(X = a, b, expr, \{tab\})$: numerical integration of *expr* on $[a, b]$ (possibly infinite interval) with respect to X, where a and b are coded as explained below. The integrand may have values belonging to a vector space over the real numbers; in particular, it can be complex-valued or vector-valued.

If *tab* is omitted, necessary integration tables are computed using `intnuminit` according to the current precision. It may be a positive integer m, and tables are computed assuming the integration step is $1/2^m$. Finally *tab* can be a table output by `intnuminit`, in which case it is used directly. This is important if several integrations of the same type are performed (on the same kind of interval and functions, and the same accuracy), since it saves expensive precomputations.

If *tab* is omitted the algorithm guesses a reasonable value for m depending on the current precision. That value may be obtained as

```
intnumstep()
```

However this value may be off from the optimal one, and this is important since the integration time is roughly proportional to 2^m. One may try consecutive values of m until they give the same value up to an accepted error.

The endpoints a and b are coded as follows. If a is not at $\pm\infty$, it is either coded as a scalar (real or complex), or as a two component vector $[a, \alpha]$, where the function is assumed to have a singularity of the form $(x - a)^{\alpha+\epsilon}$ at a, where ϵ indicates that powers of logarithms are neglected. In particular, $[a, \alpha]$ with $\alpha \geq 0$ is equivalent to a. If a wrong singularity exponent is used, the result

will lose a catastrophic number of decimals, for instance approximately half the number of digits will be correct if $\alpha = -1/2$ is omitted.

The endpoints of integration can be $\pm\infty$, which is coded as $[\pm 1]$ or as $[[\pm 1], \alpha]$. Here α codes the behaviour of the function at $\pm\infty$ as follows.

• $\alpha = 0$ (or no α at all, i.e. simply $[\pm 1]$) assumes that the function to be integrated tends to zero, but not exponentially fast, and not oscillating such as $\sin(x)/x$.

• $\alpha > 0$ assumes that the function tends to zero exponentially fast approximately as $\exp(-\alpha x)$, including reasonably oscillating functions such as $\exp(-x)\sin(x)$. The precise choice of α, while useful in extreme cases, is not critical, and may be off by a *factor* of 10 or more from the correct value.

• $\alpha < -1$ assumes that the function tends to 0 slowly, like x^α. Here it is essential to give the correct α, if possible, but on the other hand $\alpha \leq -2$ is equivalent to $\alpha = 0$, in other words to no α at all.

The last two codes are reserved for oscillating functions. Let $k > 0$ real, and $g(x)$ a nonoscillating function tending to 0, then

• $\alpha = kI$ assumes that the function behaves like $\cos(kx)g(x)$.

• $\alpha = -kI$ assumes that the function behaves like $\sin(kx)g(x)$.

Here it is critical to give the exact value of k. If the oscillating part is not a pure sine or cosine, one must expand it into a Fourier series, use the above codings, and sum the resulting contributions. Otherwise you will get nonsense. Note that $\cos(kx)$ (and similarly $\sin(kx)$) means that very function, and not a translated version such as $\cos(kx + a)$.

If for instance $f(x) = \cos(kx)g(x)$ where $g(x)$ tends to zero exponentially fast as $\exp(-\alpha x)$, it is up to the user to choose between $[[\pm 1], \alpha]$ and $[[\pm 1], kI]$, but a good rule of thumb is that if the oscillations are much weaker than the exponential decrease, choose $[[\pm 1], \alpha]$, otherwise choose $[[\pm 1], kI]$, although the latter can reasonably be used in all cases, while the former cannot. To take a specific example, in the inverse Mellin transform, the function to be integrated is almost always exponentially decreasing times oscillating. If we choose the oscillating type of integral we perhaps obtain the best results, at the expense of having to recompute our functions for a different value of the variable z giving the transform, preventing us to use a function such as `intmellininvshort`. On the other hand using the exponential type of integral, we obtain less accurate results, but we skip expensive recomputations. See `intmellininvshort` and `intfuncinit` for more explanations.

Note. If you do not like the code $[\pm 1]$ for $\pm\infty$, you are welcome to set, e.g `oo = [1]` or `INFINITY = [1]`, then using `+oo`, `-oo`, `-INFINITY`, etc. will have the expected behaviour.

We shall now see many examples to get a feeling for what the various parameters achieve. All examples below assume precision is set to 105 decimal digits. We first type

```
? \p 105
? oo = [1]   \\ for clarity
```

Apparent singularities. Even if the function $f(x)$ represented by *expr* has no singularities, it may be important to define the function differently near special points. For instance, if $f(x) = 1/(\exp(x)-1) - \exp(-x)/x$, then $\int_0^\infty f(x)\,dx = \gamma$, Euler's constant `Euler`. But

```
? f(x) = 1/(exp(x)-1) - exp(-x)/x
? intnum(x = 0, [oo,1],  f(x)) - Euler
%1 = 6.00... E-67
```

thus only correct to 76 decimal digits. This is because close to 0 the function f is computed with an enormous loss of accuracy. A better solution is

```
? f(x) = 1/(exp(x)-1)-exp(-x)/x
? F = truncate( f(t + O(t^7)) ); \\ expansion around t = 0
? g(x) = if (x > 1e-18, f(x), subst(F,t,x))  \\ note that 6 · 18 > 105
? intnum(x = 0, [oo,1],  g(x)) - Euler
%2 = 0.E-106 \\ perfect
```

It is up to the user to determine constants such as the 10^{-18} and 7 used above.

True singularities. With true singularities the result is much worse. For instance

```
? intnum(x = 0, 1,  1/sqrt(x)) - 2
%1 = -1.92... E-59 \\ only 59 correct decimals
? intnum(x = [0,-1/2], 1,  1/sqrt(x)) - 2
%2 = 0.E-105 \\ better
```

Oscillating functions.

```
? intnum(x = 0, oo, sin(x) / x) - Pi/2
%1 = 20.78.. \\ nonsense
? intnum(x = 0, [oo,1], sin(x)/x) - Pi/2
%2 = 0.004.. \\ bad
? intnum(x = 0, [oo,-I], sin(x)/x) - Pi/2
%3 = 0.E-105 \\ perfect
? intnum(x = 0, [oo,-I], sin(2*x)/x) - Pi/2  \\ oops, wrong k
%4 = 0.07...
? intnum(x = 0, [oo,-2*I], sin(2*x)/x) - Pi/2
%5 = 0.E-105 \\ perfect
? intnum(x = 0, [oo,-I], sin(x)^3/x) - Pi/4
%6 = 0.0092... \\ bad
? sin(x)^3 - (3*sin(x)-sin(3*x))/4
%7 = O(x^17)
```

We may use the above linearization and compute two oscillating integrals with "infinite endpoints" `[oo, -I]` and `[oo, -3*I]` respectively, or notice the obvious change of variable, and reduce to the single integral $\frac{1}{2}\int_0^\infty \sin(x)/x\,dx$. We finish with some more complicated examples:

```
? intnum(x = 0, [oo,-I], (1-cos(x))/x^2) - Pi/2
%1 = -0.0004... \\ bad
? intnum(x = 0, 1, (1-cos(x))/x^2) \
+ intnum(x = 1, oo, 1/x^2) - intnum(x = 1, [oo,I], cos(x)/x^2) - Pi/2
%2 = -2.18... E-106 \\ OK
```

```
? intnum(x = 0, [oo, 1], sin(x)^3*exp(-x)) - 0.3
%3 = 5.45... E-107 \\ OK
? intnum(x = 0, [oo,-I], sin(x)^3*exp(-x)) - 0.3
%4 = -1.33... E-89 \\ lost 16 decimals. Try higher m:
? m = intnumstep()
%5 = 7 \\ the value of m actually used above.
? tab = intnuminit(0,[oo,-I], m+1); \\ try m one higher.
? intnum(x = 0, oo, sin(x)^3*exp(-x), tab) - 0.3
%6 = 5.45... E-107 \\ OK this time.
```

Warning. Like `sumalt`, `intnum` often assigns a reasonable value to diverging integrals. Use these values at your own risk! For example:

```
? intnum(x = 0, [oo, -I], x^2*sin(x))
%1 = -2.0000000000...
```

Note the formula

$$\int_0^\infty \sin(x)/x^s\,dx = \cos(\pi s/2)\Gamma(1-s)\ ,$$

a priori valid only for $0 < \Re(s) < 2$, but the right hand side provides an analytic continuation which may be evaluated at $s = -2\ldots$

Multivariate integration. Using successive univariate integration with respect to different formal parameters, it is immediate to do naive multivariate integration. But it is important to use a suitable `intnuminit` to precompute data for the *internal* integrations at least!

For example, to compute the double integral on the unit disc $x^2 + y^2 \leq 1$ of the function $x^2 + y^2$, we can write

```
? tab = intnuminit(-1,1);
? intnum(x=-1,1, intnum(y=-sqrt(1-x^2),sqrt(1-x^2), x^2+y^2, tab), tab)
```

The first *tab* is essential, the second optional. Compare:

```
? tab = intnuminit(-1,1);
time = 30 ms.
? intnum(x=-1,1, intnum(y=-sqrt(1-x^2),sqrt(1-x^2), x^2+y^2));
time = 54,410 ms. \\ slow
? intnum(x=-1,1, intnum(y=-sqrt(1-x^2),sqrt(1-x^2), x^2+y^2, tab), tab);
time = 7,210 ms.  \\ faster
```

However, the `intnuminit` program is usually pessimistic when it comes to choosing the integration step 2^{-m}. It is often possible to improve the speed by trial and error. Continuing the above example:

```
? test(M) =
{
  tab = intnuminit(-1,1, M);
  intnum(x=-1,1, intnum(y=-sqrt(1-x^2),sqrt(1-x^2), x^2+y^2,tab), tab) - Pi/2
}
? m = intnumstep() \\ what value of m did it take ?
%1 = 7
? test(m - 1)
```

```
time = 1,790 ms.
%2 = -2.05... E-104 \\ 4 = 2^2 times faster and still OK.
? test(m - 2)
time = 430 ms.
%3 = -1.11... E-104 \\ 16 = 2^4 times faster and still OK.
? test(m - 3)
time = 120 ms.
%3 = -7.23... E-60 \\ 64 = 2^6 times faster, lost 45 decimals.
```

The library syntax is **intnum**(`void *E, GEN (*eval)(GEN,void*), GEN a,GEN b,GEN tab, long prec`), where an omitted *tab* is coded as `NULL`.

3.9.10 intnuminit$(a, b, \{m = 0\})$: initialize tables for integration from a to b, where a and b are coded as in `intnum`. Only the compactness, the possible existence of singularities, the speed of decrease or the oscillations at infinity are taken into account, and not the values. For instance `intnuminit(-1,1)` is equivalent to `intnuminit(0,Pi)`, and `intnuminit([0,-1/2],[1])` is equivalent to `intnuminit([-1],[-1,-1/2])`. If m is not given, it is computed according to the current precision. Otherwise the integration step is $1/2^m$. Reasonable values of m are $m = 6$ or $m = 7$ for 100 decimal digits, and $m = 9$ for 1000 decimal digits.

The result is technical, but in some cases it is useful to know the output. Let $x = \phi(t)$ be the change of variable which is used. $tab[1]$ contains the integer m as above, either given by the user or computed from the default precision, and can be recomputed directly using the function `intnumstep`. $tab[2]$ and $tab[3]$ contain respectively the abcissa and weight corresponding to $t = 0$ ($\phi(0)$ and $\phi'(0)$). $tab[4]$ and $tab[5]$ contain the abcissas and weights corresponding to positive $t = nh$ for $1 \le n \le N$ and $h = 1/2^m$ ($\phi(nh)$ and $\phi'(nh)$). Finally $tab[6]$ and $tab[7]$ contain either the abcissas and weights corresponding to negative $t = nh$ for $-N \le n \le -1$, or may be empty (but not always) if $\phi(t)$ is an odd function (implicitly we would have $tab[6] = -tab[4]$ and $tab[7] = tab[5]$).

The library syntax is **intnuminit**(`GEN a, GEN b, long m, long prec`).

3.9.11 intnumromb$(X = a, b, expr, \{flag = 0\})$: numerical integration of *expr* (smooth in $]a, b[$), with respect to X. This function is deprecated, use `intnum` instead.

Set *flag* $= 0$ (or omit it altogether) when a and b are not too large, the function is smooth, and can be evaluated exactly everywhere on the interval $[a, b]$.

If *flag* $= 1$, uses a general driver routine for doing numerical integration, making no particular assumption (slow).

flag $= 2$ is tailored for being used when a or b are infinite. One *must* have $ab > 0$, and in fact if for example $b = +\infty$, then it is preferable to have a as large as possible, at least $a \ge 1$.

If *flag* $= 3$, the function is allowed to be undefined (but continuous) at a or b, for example the function $\sin(x)/x$ at $x = 0$.

The user should not require too much accuracy: 18 or 28 decimal digits is OK, but not much more. In addition, analytical cleanup of the integral must have been done: there must be no singularities in the interval or at the boundaries. In practice this can be accomplished with a simple change of variable. Furthermore, for improper integrals, where one or both of the limits of integration are plus or minus infinity, the function must decrease sufficiently rapidly at infinity. This can often be accomplished through integration by parts. Finally, the function to be integrated

should not be very small (compared to the current precision) on the entire interval. This can of course be accomplished by just multiplying by an appropriate constant.

Note that infinity can be represented with essentially no loss of accuracy by 1e1000. However beware of real underflow when dealing with rapidly decreasing functions. For example, if one wants to compute the $\int_0^\infty e^{-x^2}\,dx$ to 28 decimal digits, then one should set infinity equal to 10 for example, and certainly not to 1e1000.

The library syntax is **intnumromb**(`void *E, GEN (*eval)(GEN,void*), GEN a, GEN b, long flag, long prec`), where `eval`(x, E) returns the value of the function at x. You may store any additional information required by `eval` in E, or set it to `NULL`.

3.9.12 intnumstep(): give the value of m used in all the `intnum` and `sumnum` programs, hence such that the integration step is equal to $1/2^m$.

The library syntax is **intnumstep**(`long prec`).

3.9.13 prod$(X = a, b, expr, \{x = 1\})$: product of expression *expr*, initialized at x, the formal parameter X going from a to b. As for `sum`, the main purpose of the initialization parameter x is to force the type of the operations being performed. For example if it is set equal to the integer 1, operations will start being done exactly. If it is set equal to the real 1., they will be done using real numbers having the default precision. If it is set equal to the power series $1 + O(X^k)$ for a certain k, they will be done using power series of precision at most k. These are the three most common initializations.

As an extreme example, compare

```
? prod(i=1, 100, 1 - X^i);  \\ this has degree 5050 !!
time = 3,335 ms.
? prod(i=1, 100, 1 - X^i, 1 + O(X^101))
time = 43 ms.
%2 = 1 - X - X^2 + X^5 + X^7 - X^12 - X^15 + X^22 + X^26 - X^35 - X^40 + \
  X^51 + X^57 - X^70 - X^77 + X^92 + X^100 + O(X^101)
```

The library syntax is **produit**(`entree *ep, GEN a, GEN b, char *expr, GEN x`).

3.9.14 prodeuler$(X = a, b, expr)$: product of expression *expr*, initialized at 1. (i.e. to a *real* number equal to 1 to the current `realprecision`), the formal parameter X ranging over the prime numbers between a and b.

The library syntax is **prodeuler**(`void *E, GEN (*eval)(GEN,void*), GEN a,GEN b, long prec`).

3.9.15 prodinf$(X = a, expr, \{flag = 0\})$: infinite product of expression *expr*, the formal parameter X starting at a. The evaluation stops when the relative error of the expression minus 1 is less than the default precision. The expressions must always evaluate to an element of **C**.

If *flag* $= 1$, do the product of the $(1 +$ *expr*$)$ instead.

The library syntax is **prodinf**(`void *E, GEN (*eval)(GEN, void*), GEN a, long prec`) (*flag* $= 0$), or **prodinf1** with the same arguments (*flag* $= 1$).

3.9.16 solve$(X = a, b, expr)$: find a real root of expression *expr* between a and b, under the condition $expr(X = a) * expr(X = b) \leq 0$. This routine uses Brent's method and can fail miserably if *expr* is not defined in the whole of $[a, b]$ (try `solve(x=1, 2, tan(x))`).

The library syntax is **zbrent**(`void *E,GEN (*eval)(GEN,void*),GEN a,GEN b,long prec`).

3.9.17 sum$(X = a, b, expr, \{x = 0\})$: sum of expression *expr*, initialized at x, the formal parameter going from a to b. As for `prod`, the initialization parameter x may be given to force the type of the operations being performed.

As an extreme example, compare

```
? sum(i=1, 5000, 1/i); \\ rational number: denominator has 2166 digits.
time = 1,241 ms.
? sum(i=1, 5000, 1/i, 0.)
time = 158 ms.
%2 = 9.094508852984436967261245533
```

The library syntax is **somme**(`entree *ep, GEN a, GEN b, char *expr, GEN x`). This is to be used as follows: `ep` represents the dummy variable used in the expression `expr`

```
/* compute a^2 + ... + b^2 */
{
  /* define the dummy variable "i" */
  entree *ep = is_entry("i");
  /* sum for a <= i <= b */
  return somme(ep, a, b, "i^2", gen_0);
}
```

3.9.18 sumalt$(X = a, expr, \{flag = 0\})$: numerical summation of the series *expr*, which should be an alternating series, the formal variable X starting at a. Use an algorithm of F. Villegas as modified by D. Zagier (improves on Euler-Van Wijngaarden method).

If $flag = 1$, use a variant with slightly different polynomials. Sometimes faster.

Divergent alternating series can sometimes be summed by this method, as well as series which are not exactly alternating (see for example Section 2.6). If the series already converges geometrically, `suminf` is often a better choice:

```
? \p28
? sumalt(i = 1, -(-1)^i / i)  - log(2)
time = 0 ms.
%1 = -2.524354897 E-29
? suminf(i = 1, -(-1)^i / i)
  *** suminf: user interrupt after 10min, 20,100 ms.
? \p1000
? sumalt(i = 1, -(-1)^i / i)  - log(2)
time = 90 ms.
%2 = 4.459597722 E-1002
? sumalt(i = 0, (-1)^i / i!) - exp(-1)
time = 670 ms.
%3 = -4.03698781490633483156497361352190615794353338591897830587 E-944
? suminf(i = 0, (-1)^i / i!) - exp(-1)
```

```
time = 110 ms.
%4 = -8.39147638 E-1000    \\  faster and more accurate
```

The library syntax is **sumalt**(`void *E, GEN (*eval)(GEN,void*),GEN a,long prec`). Also available is `sumalt2` with the same arguments ($flag = 1$).

3.9.19 sumdiv($n, X, expr$): sum of expression *expr* over the positive divisors of n.

Arithmetic functions like `sigma` use the multiplicativity of the underlying expression to speed up the computation. In the present version 2.3.5, there is no way to indicate that *expr* is multiplicative in n, hence specialized functions should be preferred whenever possible.

The library syntax is **divsum**(`entree *ep, GEN num, char *expr`).

3.9.20 suminf($X = a, expr$): infinite sum of expression *expr*, the formal parameter X starting at a. The evaluation stops when the relative error of the expression is less than the default precision for 3 consecutive evaluations. The expressions must always evaluate to a complex number.

If the series converges slowly, make sure `realprecision` is low (even 28 digits may be too much). In this case, if the series is alternating or the terms have a constant sign, `sumalt` and `sumpos` should be used instead.

```
? \p28
? suminf(i = 1, -(-1)^i / i)
  *** suminf: user interrupt after 10min, 20,100 ms.
? sumalt(i = 1, -(-1)^i / i) - log(2)
time = 0 ms.
%1 = -2.524354897 E-29
```

The library syntax is **suminf**(`void *E, GEN (*eval)(GEN,void*), GEN a, long prec`).

3.9.21 sumnum($X = a, sig, expr, \{tab\}), \{flag = 0\}$: numerical summation of *expr*, the variable X taking integer values from ceiling of a to $+\infty$, where *expr* is assumed to be a holomorphic function $f(X)$ for $\Re(X) \geq \sigma$.

The parameter $\sigma \in \mathbf{R}$ is coded in the argument `sig` as follows: it is either

• a real number σ. Then the function f is assumed to decrease at least as $1/X^2$ at infinity, but not exponentially;

• a two-component vector $[\sigma, \alpha]$, where σ is as before, $\alpha < -1$. The function f is assumed to decrease like X^α. In particular, $\alpha \leq -2$ is equivalent to no α at all.

• a two-component vector $[\sigma, \alpha]$, where σ is as before, $\alpha > 0$. The function f is assumed to decrease like $\exp(-\alpha X)$. In this case it is essential that α be exactly the rate of exponential decrease, and it is usually a good idea to increase the default value of m used for the integration step. In practice, if the function is exponentially decreasing `sumnum` is slower and less accurate than `sumpos` or `suminf`, so should not be used.

The function uses the `intnum` routines and integration on the line $\Re(s) = \sigma$. The optional argument *tab* is as in intnum, except it must be initialized with `sumnuminit` instead of `intnuminit`.

When *tab* is not precomputed, `sumnum` can be slower than `sumpos`, when the latter is applicable. It is in general faster for slowly decreasing functions.

Finally, if *flag* is nonzero, we assume that the function f to be summed is of real type, i.e. satisfies $\overline{f(z)} = f(\overline{z})$, which speeds up the computation.

```
? \p 308
? a = sumpos(n=1, 1/(n^3+n+1));
time = 1,410 ms.
? tab = sumnuminit(2);
time = 1,620 ms. \\ slower but done once and for all.
? b = sumnum(n=1, 2, 1/(n^3+n+1), tab);
time = 460 ms. \\ 3 times as fast as sumpos
? a - b
%4 = -1.0... E-306 + 0.E-320*I \\ perfect.
? sumnum(n=1, 2, 1/(n^3+n+1), tab, 1) - a; \\ function of real type
time = 240 ms.
%2 = -1.0... E-306 \\ twice as fast, no imaginary part.
? c = sumnum(n=1, 2, 1/(n^2+1), tab, 1);
time = 170 ms. \\ fast
? d = sumpos(n=1, 1 / (n^2+1));
time = 2,700 ms. \\ slow.
? d - c
time = 0 ms.
%5 = 1.97... E-306 \\ perfect.
```

For slowly decreasing function, we must indicate singularities:

```
? \p 308
? a = sumnum(n=1, 2, n^(-4/3));
time = 9,930 ms. \\ slow because of the computation of n^(-4/3).
? a - zeta(4/3)
time = 110 ms.
%1 = -2.42... E-107 \\ lost 200 decimals because of singularity at oo
? b = sumnum(n=1, [2,-4/3], n^(-4/3), /*omitted*/, 1); \\ of real type
time = 12,210 ms.
? b - zeta(4/3)
%3 = 1.05... E-300 \\ better
```

Since the *complex* values of the function are used, beware of determination problems. For instance:

```
? \p 308
? tab = sumnuminit([2,-3/2]);
time = 1,870 ms.
? sumnum(n=1,[2,-3/2], 1/(n*sqrt(n)), tab,1) - zeta(3/2)
time = 690 ms.
%1 = -1.19... E-305 \\ fast and correct
? sumnum(n=1,[2,-3/2], 1/sqrt(n^3), tab,1) - zeta(3/2)
time = 730 ms.
%2 = -1.55... \\ nonsense. However
? sumnum(n=1,[2,-3/2], 1/n^(3/2), tab,1) - zeta(3/2)
time = 8,990 ms.
%3 = -1.19... E-305 \\ perfect, as 1/(n*sqrt(n)) above but much slower
```

For exponentially decreasing functions, `sumnum` is given for completeness, but one of `suminf` or `sumpos` should always be preferred. If you experiment with such functions and `sumnum` anyway, indicate the exact rate of decrease and increase m by 1 or 2:

```
? suminf(n=1, 2^(-n)) - 1
time = 10 ms.
%1 = -1.11... E-308 \\ fast and perfect
? sumpos(n=1, 2^(-n)) - 1
time = 10 ms.
%2 = -2.78... E-308 \\ also fast and perfect
? sumnum(n=1,2, 2^(-n)) - 1
   *** sumnum: precision too low in mpsc1 \\ nonsense
? sumnum(n=1, [2,log(2)], 2^(-n), /*omitted*/, 1) - 1 \\ of real type
time = 5,860 ms.
%3 = -1.5... E-236 \\ slow and lost 70 decimals
? m = intnumstep()
%4 = 9
? sumnum(n=1,[2,log(2)], 2^(-n), m+1, 1) - 1
time = 11,770 ms.
%5 = -1.9... E-305 \\ now perfect, but slow.
```

The library syntax is **sumnum**(`void *E, GEN (*eval)(GEN,void*), GEN a,GEN sig,GEN tab,long flag, long prec`).

3.9.22 sumnumalt$(X = a, sig, expr, \{tab\}, \{flag = 0\})$: numerical summation of $(-1)^X expr(X)$, the variable X taking integer values from ceiling of a to $+\infty$, where *expr* is assumed to be a holomorphic function for $\Re(X) \geq sig$ (or $sig[1]$).

Warning. This function uses the `intnum` routines and is orders of magnitude slower than `sumalt`. It is only given for completeness and should not be used in practice.

Warning2. The expression *expr* must *not* include the $(-1)^X$ coefficient. Thus `sumalt`$(n = a, (-1)^n f(n))$ is (approximately) equal to `sumnumalt`$(n = a, sig, f(n))$.

sig is coded as in `sumnum`. However for slowly decreasing functions (where *sig* is coded as $[\sigma, \alpha]$ with $\alpha < -1$), it is not really important to indicate α. In fact, as for `sumalt`, the program will often give meaningful results (usually analytic continuations) even for divergent series. On the other hand the exponential decrease must be indicated.
tab

is as in `intnum`, but if used must be initialized with `sumnuminit`. If *flag* is nonzero, assumes that the function f to be summed is of real type, i.e. satisfies $\overline{f(z)} = f(\overline{z})$, and then twice faster when *tab* is precomputed.

```
? \p 308
? tab = sumnuminit(2, /*omitted*/, -1); \\ abcissa σ = 2, alternating sums.
time = 1,620 ms. \\ slow, but done once and for all.
? a = sumnumalt(n=1, 2, 1/(n^3+n+1), tab, 1);
time = 230 ms. \\ similar speed to sumnum
? b = sumalt(n=1, (-1)^n/(n^3+n+1));
time = 0 ms. \\ infinitely faster!
? a - b
```

```
time = 0 ms.
%1 = -1.66... E-308 \\ perfect
```

The library syntax is **sumnumalt**(`void *E, GEN (*eval)(GEN,void*), GEN a, GEN sig, GEN tab, long flag, long prec`).

3.9.23 sumnuminit($sig, m = 0, sgn = 1$): initialize tables for numerical summation using `sumnum` (with $sgn = 1$) or `sumnumalt` (with $sgn = -1$), sig is the abcissa of integration coded as in `sumnum`, and m is as in `intnuminit`.

The library syntax is **sumnuminit**(`GEN sig, long m, long sgn, long prec`).

3.9.24 sumpos($X = a, expr, \{flag = 0\}$): numerical summation of the series $expr$, which must be a series of terms having the same sign, the formal variable X starting at a. The algorithm used is Van Wijngaarden's trick for converting such a series into an alternating one, and is quite slow. For regular functions, the function `sumnum` is in general much faster once the initializations have been made using `sumnuminit`.

If $flag = 1$, use slightly different polynomials. Sometimes faster.

The library syntax is **sumpos**(`void *E, GEN (*eval)(GEN,void*),GEN a,long prec`). Also available is `sumpos2` with the same arguments ($flag = 1$).

3.10 Plotting functions.

Although plotting is not even a side purpose of PARI, a number of plotting functions are provided. Moreover, a lot of people suggested ideas or submitted patches for this section of the code. Among these, special thanks go to Klaus-Peter Nischke who suggested the recursive plotting and the forking/resizing stuff under X11, and Ilya Zakharevich who undertook a complete rewrite of the graphic code, so that most of it is now platform-independent and should be easy to port or expand. There are three types of graphic functions.

3.10.1 High-level plotting functions (all the functions starting with `ploth`) in which the user has little to do but explain what type of plot he wants, and whose syntax is similar to the one used in the preceding section.

3.10.2 Low-level plotting functions (called *rectplot* functions, sharing the prefix `plot`), where every drawing primitive (point, line, box, etc.) is specified by the user. These low-level functions work as follows. You have at your disposal 16 virtual windows which are filled independently, and can then be physically ORed on a single window at user-defined positions. These windows are numbered from 0 to 15, and must be initialized before being used by the function `plotinit`, which specifies the height and width of the virtual window (called a *rectwindow* in the sequel). At all times, a virtual cursor (initialized at $[0, 0]$) is associated to the window, and its current value can be obtained using the function `plotcursor`.

A number of primitive graphic objects (called *rect* objects) can then be drawn in these windows, using a default color associated to that window (which can be changed under X11, using the `plotcolor` function, black otherwise) and only the part of the object which is inside the window will be drawn, with the exception of polygons and strings which are drawn entirely. The ones sharing the prefix `plotr` draw relatively to the current position of the virtual cursor, the others use

absolute coordinates. Those having the prefix `plotrecth` put in the rectwindow a large batch of rect objects corresponding to the output of the related `ploth` function.

Finally, the actual physical drawing is done using the function `plotdraw`. The rectwindows are preserved so that further drawings using the same windows at different positions or different windows can be done without extra work. To erase a window (and free the corresponding memory), use the function `plotkill`. It is not possible to partially erase a window. Erase it completely, initialize it again and then fill it with the graphic objects that you want to keep.

In addition to initializing the window, you may use a scaled window to avoid unnecessary conversions. For this, use the function `plotscale` below. As long as this function is not called, the scaling is simply the number of pixels, the origin being at the upper left and the y-coordinates going downwards.

Note that in the present version 2.3.5 all plotting functions (both low and high level) are written for the X11-window system (hence also for GUI's based on X11 such as Openwindows and Motif) only, though little code remains which is actually platform-dependent. It is also possible to compile `gp` with either of the Qt or FLTK graphical libraries. A Suntools/Sunview, Macintosh, and an Atari/Gem port were provided for previous versions, but are now obsolete.

Under X11, the physical window (opened by `plotdraw` or any of the `ploth*` functions) is completely separated from `gp` (technically, a `fork` is done, and the non-graphical memory is immediately freed in the child process), which means you can go on working in the current `gp` session, without having to kill the window first. Under X11, this window can be closed, enlarged or reduced using the standard window manager functions. No zooming procedure is implemented though (yet).

3.10.3 Functions for PostScript output: in the same way that `printtex` allows you to have a TEX output corresponding to printed results, the functions starting with `ps` allow you to have `PostScript` output of the plots. This will not be absolutely identical with the screen output, but will be sufficiently close. Note that you can use PostScript output even if you do not have the plotting routines enabled. The PostScript output is written in a file whose name is derived from the `psfile` default (`./pari.ps` if you did not tamper with it). Each time a new PostScript output is asked for, the PostScript output is appended to that file. Hence you probably want to remove this file, or change the value of `psfile`, in between plots. On the other hand, in this manner, as many plots as desired can be kept in a single file.

3.10.4 And library mode ? *None of the graphic functions are available within the PARI library, you must be under* `gp` *to use them.* The reason for that is that you really should not use PARI for heavy-duty graphical work, there are better specialized alternatives around. This whole set of routines was only meant as a convenient, but simple-minded, visual aid. If you really insist on using these in your program (we warned you), the source (`plot*.c`) should be readable enough for you to achieve something.

3.10.5 plot$(X = a, b, expr, \{Ymin\}, \{Ymax\})$: crude ASCII plot of the function represented by expression *expr* from a to b, with Y ranging from *Ymin* to *Ymax*. If *Ymin* (resp. *Ymax*) is not given, the minima (resp. the maxima) of the computed values of the expression is used instead.

3.10.6 plotbox$(w, x2, y2)$: let $(x1, y1)$ be the current position of the virtual cursor. Draw in the rectwindow w the outline of the rectangle which is such that the points $(x1, y1)$ and $(x2, y2)$ are opposite corners. Only the part of the rectangle which is in w is drawn. The virtual cursor does *not* move.

3.10.7 plotclip(w): 'clips' the content of rectwindow w, i.e remove all parts of the drawing that would not be visible on the screen. Together with `plotcopy` this function enables you to draw on a scratchpad before commiting the part you're interested in to the final picture.

3.10.8 plotcolor(w, c): set default color to c in rectwindow w. In present version 2.3.5, this is only implemented for the X11 window system, and you only have the following palette to choose from:

1=black, 2=blue, 3=sienna, 4=red, 5=green, 6=grey, 7=gainsborough.

Note that it should be fairly easy for you to hardwire some more colors by tweaking the files `rect.h` and `plotX.c`. User-defined colormaps would be nice, and *may* be available in future versions.

3.10.9 plotcopy($w1, w2, dx, dy$): copy the contents of rectwindow $w1$ to rectwindow $w2$, with offset (dx, dy).

3.10.10 plotcursor(w): give as a 2-component vector the current (scaled) position of the virtual cursor corresponding to the rectwindow w.

3.10.11 plotdraw($list$): physically draw the rectwindows given in *list* which must be a vector whose number of components is divisible by 3. If $list = [w1, x1, y1, w2, x2, y2, \ldots]$, the windows $w1$, $w2$, etc. are physically placed with their upper left corner at physical position $(x1, y1)$, $(x2, y2)$,... respectively, and are then drawn together. Overlapping regions will thus be drawn twice, and the windows are considered transparent. Then display the whole drawing in a special window on your screen.

3.10.12 ploth($X = a, b, expr, \{flag = 0\}, \{n = 0\}$): high precision plot of the function $y = f(x)$ represented by the expression *expr*, x going from a to b. This opens a specific window (which is killed whenever you click on it), and returns a four-component vector giving the coordinates of the bounding box in the form $[xmin, xmax, ymin, ymax]$.

Important note: Since this may involve a lot of function calls, it is advised to keep the current precision to a minimum (e.g. 9) before calling this function.

n specifies the number of reference point on the graph (0 means use the hardwired default values, that is: 1000 for general plot, 1500 for parametric plot, and 15 for recursive plot).

If no *flag* is given, *expr* is either a scalar expression $f(X)$, in which case the plane curve $y = f(X)$ will be drawn, or a vector $[f_1(X), \ldots, f_k(X)]$, and then all the curves $y = f_i(X)$ will be drawn in the same window.

The binary digits of *flag* mean:

• 1 = `Parametric`: *parametric plot*. Here *expr* must be a vector with an even number of components. Successive pairs are then understood as the parametric coordinates of a plane curve. Each of these are then drawn.

For instance:

`ploth(X=0,2*Pi,[sin(X),cos(X)],1)` will draw a circle.

`ploth(X=0,2*Pi,[sin(X),cos(X)])` will draw two entwined sinusoidal curves.

`ploth(X=0,2*Pi,[X,X,sin(X),cos(X)],1)` will draw a circle and the line $y = x$.

• 2 = `Recursive`: *recursive plot*. If this flag is set, only *one* curve can be drawn at a time, i.e. *expr* must be either a two-component vector (for a single parametric curve, and the parametric flag *has* to be set), or a scalar function. The idea is to choose pairs of successive reference points, and if their middle point is not too far away from the segment joining them, draw this as a local approximation to the curve. Otherwise, add the middle point to the reference points. This is fast, and usually more precise than usual plot. Compare the results of

$$\texttt{ploth}(\texttt{X} = -1, 1, \texttt{sin}(1/\texttt{X}), 2) \quad \text{and} \quad \texttt{ploth}(\texttt{X} = -1, 1, \texttt{sin}(1/\texttt{X}))$$

for instance. But beware that if you are extremely unlucky, or choose too few reference points, you may draw some nice polygon bearing little resemblance to the original curve. For instance you should *never* plot recursively an odd function in a symmetric interval around 0. Try

```
ploth(x = -20, 20, sin(x), 2)
```

to see why. Hence, it's usually a good idea to try and plot the same curve with slightly different parameters.

The other values toggle various display options:

• 4 = `no_Rescale`: do not rescale plot according to the computed extrema. This is meant to be used when graphing multiple functions on a rectwindow (as a `plotrecth` call), in conjunction with `plotscale`.

• 8 = `no_X_axis`: do not print the x-axis.

• 16 = `no_Y_axis`: do not print the y-axis.

• 32 = `no_Frame`: do not print frame.

• 64 = `no_Lines`: only plot reference points, do not join them.

• 128 = `Points_too`: plot both lines and points.

• 256 = `Splines`: use splines to interpolate the points.

• 512 = `no_X_ticks`: plot no x-ticks.

• 1024 = `no_Y_ticks`: plot no y-ticks.

• 2048 = `Same_ticks`: plot all ticks with the same length.

3.10.13 plothraw(*listx*, *listy*, {*flag* = 0}): given *listx* and *listy* two vectors of equal length, plots (in high precision) the points whose (x, y)-coordinates are given in *listx* and *listy*. Automatic positioning and scaling is done, but with the same scaling factor on x and y. If *flag* is 1, join points, other non-0 flags toggle display options and should be combinations of bits 2^k, $k \geq 3$ as in `ploth`.

3.10.14 plothsizes(): return data corresponding to the output window in the form of a 6-component vector: window width and height, sizes for ticks in horizontal and vertical directions (this is intended for the `gnuplot` interface and is currently not significant), width and height of characters.

3.10.15 plotinit$(w, x, y, \{flag\})$: initialize the rectwindow w, destroying any rect objects you may have already drawn in w. The virtual cursor is set to $(0, 0)$. The rectwindow size is set to width x and height y. If $flag = 0$, x and y represent pixel units. Otherwise, x and y are understood as fractions of the size of the current output device (hence must be between 0 and 1) and internally converted to pixels.

The plotting device imposes an upper bound for x and y, for instance the number of pixels for screen output. These bounds are available through the `plothsizes` function. The following sequence initializes in a portable way (i.e independent of the output device) a window of maximal size, accessed through coordinates in the $[0, 1000] \times [0, 1000]$ range:

```
s = plothsizes();
plotinit(0, s[1]-1, s[2]-1);
plotscale(0, 0,1000, 0,1000);
```

3.10.16 plotkill(w): erase rectwindow w and free the corresponding memory. Note that if you want to use the rectwindow w again, you have to use `plotinit` first to specify the new size. So it's better in this case to use `plotinit` directly as this throws away any previous work in the given rectwindow.

3.10.17 plotlines$(w, X, Y, \{flag = 0\})$: draw on the rectwindow w the polygon such that the (x,y)-coordinates of the vertices are in the vectors of equal length X and Y. For simplicity, the whole polygon is drawn, not only the part of the polygon which is inside the rectwindow. If *flag* is non-zero, close the polygon. In any case, the virtual cursor does not move.

X and Y are allowed to be scalars (in this case, both have to). There, a single segment will be drawn, between the virtual cursor current position and the point (X, Y). And only the part thereof which actually lies within the boundary of w. Then *move* the virtual cursor to (X, Y), even if it is outside the window. If you want to draw a line from $(x1, y1)$ to $(x2, y2)$ where $(x1, y1)$ is not necessarily the position of the virtual cursor, use `plotmove(w,x1,y1)` before using this function.

3.10.18 plotlinetype$(w, type)$: change the type of lines subsequently plotted in rectwindow w. *type* -2 corresponds to frames, -1 to axes, larger values may correspond to something else. $w = -1$ changes highlevel plotting. This is only taken into account by the `gnuplot` interface.

3.10.19 plotmove(w, x, y): move the virtual cursor of the rectwindow w to position (x, y).

3.10.20 plotpoints(w, X, Y): draw on the rectwindow w the points whose (x, y)-coordinates are in the vectors of equal length X and Y and which are inside w. The virtual cursor does *not* move. This is basically the same function as `plothraw`, but either with no scaling factor or with a scale chosen using the function `plotscale`.

As was the case with the `plotlines` function, X and Y are allowed to be (simultaneously) scalar. In this case, draw the single point (X, Y) on the rectwindow w (if it is actually inside w), and in any case *move* the virtual cursor to position (x, y).

3.10.21 plotpointsize$(w, size)$: changes the "size" of following points in rectwindow w. If $w = -1$, change it in all rectwindows. This only works in the `gnuplot` interface.

3.10.22 plotpointtype$(w, type)$: change the type of points subsequently plotted in rectwindow w. *type* $= -1$ corresponds to a dot, larger values may correspond to something else. $w = -1$ changes highlevel plotting. This is only taken into account by the `gnuplot` interface.

3.10.23 plotrbox(w, dx, dy): draw in the rectwindow w the outline of the rectangle which is such that the points $(x1, y1)$ and $(x1 + dx, y1 + dy)$ are opposite corners, where $(x1, y1)$ is the current position of the cursor. Only the part of the rectangle which is in w is drawn. The virtual cursor does *not* move.

3.10.24 plotrecth$(w, X = a, b, expr, \{flag = 0\}, \{n = 0\})$: writes to rectwindow w the curve output of `ploth`$(w, X = a, b, expr, flag, n)$.

3.10.25 plotrecthraw$(w, data, \{flag = 0\})$: plot graph(s) for *data* in rectwindow w. *flag* has the same significance here as in `ploth`, though recursive plot is no more significant.
data

is a vector of vectors, each corresponding to a list a coordinates. If parametric plot is set, there must be an even number of vectors, each successive pair corresponding to a curve. Otherwise, the first one contains the x coordinates, and the other ones contain the y-coordinates of curves to plot.

3.10.26 plotrline(w, dx, dy): draw in the rectwindow w the part of the segment $(x1, y1) - (x1 + dx, y1 + dy)$ which is inside w, where $(x1, y1)$ is the current position of the virtual cursor, and move the virtual cursor to $(x1 + dx, y1 + dy)$ (even if it is outside the window).

3.10.27 plotrmove(w, dx, dy): move the virtual cursor of the rectwindow w to position $(x1 + dx, y1 + dy)$, where $(x1, y1)$ is the initial position of the cursor (i.e. to position (dx, dy) relative to the initial cursor).

3.10.28 plotrpoint(w, dx, dy): draw the point $(x1 + dx, y1 + dy)$ on the rectwindow w (if it is inside w), where $(x1, y1)$ is the current position of the cursor, and in any case move the virtual cursor to position $(x1 + dx, y1 + dy)$.

3.10.29 plotscale$(w, x1, x2, y1, y2)$: scale the local coordinates of the rectwindow w so that x goes from $x1$ to $x2$ and y goes from $y1$ to $y2$ ($x2 < x1$ and $y2 < y1$ being allowed). Initially, after the initialization of the rectwindow w using the function `plotinit`, the default scaling is the graphic pixel count, and in particular the y axis is oriented downwards since the origin is at the upper left. The function `plotscale` allows to change all these defaults and should be used whenever functions are graphed.

3.10.30 plotstring$(w, x, \{flag = 0\})$: draw on the rectwindow w the String x (see Section 2.8), at the current position of the cursor.
flag

is used for justification: bits 1 and 2 regulate horizontal alignment: left if 0, right if 2, center if 1. Bits 4 and 8 regulate vertical alignment: bottom if 0, top if 8, v-center if 4. Can insert additional small gap between point and string: horizontal if bit 16 is set, vertical if bit 32 is set (see the tutorial for an example).

3.10.31 psdraw$(list)$: same as `plotdraw`, except that the output is a PostScript program appended to the `psfile`.

3.10.32 psploth$(X = a, b, expr)$: same as `ploth`, except that the output is a PostScript program appended to the `psfile`.

3.10.33 psplothraw$(listx, listy)$: same as `plothraw`, except that the output is a PostScript program appended to the `psfile`.

3.11 Programming in GP.

3.11.1 Control statements.

A number of control statements are available in GP. They are simpler and have a syntax slightly different from their C counterparts, but are quite powerful enough to write any kind of program. Some of them are specific to GP, since they are made for number theorists. As usual, X will denote any simple variable name, and *seq* will always denote a sequence of expressions, including the empty sequence.

Caveat: in constructs like

```
for (X = a,b, seq)
```

the variable `X` is considered local to the loop, leading to possibly unexpected behaviour:

```
n = 5;
for (n = 1, 10,
  if (something_nice(), break);
);
\\  at this point n is 5 !
```

If the sequence `seq` modifies the loop index, then the loop is modified accordingly:

```
? for (n = 1, 10, n += 2; print(n))
3
6
9
12
```

3.11.1.1 break($\{n = 1\}$): interrupts execution of current *seq*, and immediately exits from the n innermost enclosing loops, within the current function call (or the top level loop). n must be bigger than 1. If n is greater than the number of enclosing loops, all enclosing loops are exited.

3.11.1.2 for($X = a, b, seq$): evaluates *seq*, where the formal variable X goes from a to b. Nothing is done if $a > b$. a and b must be in **R**.

3.11.1.3 fordiv(n, X, seq): evaluates *seq*, where the formal variable X ranges through the divisors of n (see `divisors`, which is used as a subroutine). It is assumed that `factor` can handle n, without negative exponents. Instead of n, it is possible to input a factorization matrix, i.e. the output of `factor(n)`.

This routine uses `divisors` as a subroutine, then loops over the divisors. In particular, if n is an integer, divisors are sorted by increasing size.

To avoid storing all divisors, possibly using a lot of memory, the following (much slower) routine loops over the divisors using essentially constant space:

```
FORDIV(N)=
{ local(P, E);
  P = factor(N); E = P[,2]; P = P[,1];
  forvec( v = vector(#E, i, [0,E[i]]),
    X = factorback(P, v)
    \\ ...
  );
```

```
}
? for(i=1,10^5, FORDIV(i))
time = 3,445 ms.
? for(i=1,10^5, fordiv(i, d, ))
time = 490 ms.
```

3.11.1.4 forell(E, a, b, seq): evaluates *seq*, where the formal variable E ranges through all elliptic curves of conductors from a to b. Th `elldata` database must be installed and contain data for the specified conductors.

3.11.1.5 forprime$(X = a, b, seq)$: evaluates *seq*, where the formal variable X ranges over the prime numbers between a to b (including a and b if they are prime). More precisely, the value of X is incremented to the smallest prime strictly larger than X at the end of each iteration. Nothing is done if $a > b$. Note that a and b must be in $\mathbf{R}$.

```
? { forprime(p = 2, 12,
      print(p);
      if (p == 3, p = 6);
    )
  }
2
3
7
11
```

3.11.1.6 forstep$(X = a, b, s, seq)$: evaluates *seq*, where the formal variable X goes from a to b, in increments of s. Nothing is done if $s > 0$ and $a > b$ or if $s < 0$ and $a < b$. s must be in $\mathbf{R}^*$ or a vector of steps $[s_1, \ldots, s_n]$. In the latter case, the successive steps are used in the order they appear in s.

```
? forstep(x=5, 20, [2,4], print(x))
5
7
11
13
17
19
```

3.11.1.7 forsubgroup$(H = G, \{B\}, seq)$: evaluates *seq* for each subgroup H of the *abelian* group G (given in SNF form or as a vector of elementary divisors), whose index is bounded by B. The subgroups are not ordered in any obvious way, unless G is a p-group in which case Birkhoff's algorithm produces them by decreasing index. A subgroup is given as a matrix whose columns give its generators on the implicit generators of G. For example, the following prints all subgroups of index less than 2 in $G = \mathbf{Z}/2\mathbf{Z}g_1 \times \mathbf{Z}/2\mathbf{Z}g_2$:

```
? G = [2,2]; forsubgroup(H=G, 2, print(H))
[1; 1]
[1; 2]
[2; 1]
[1, 0; 1, 1]
```

The last one, for instance is generated by $(g_1, g_1 + g_2)$. This routine is intended to treat huge groups, when `subgrouplist` is not an option due to the sheer size of the output.

For maximal speed the subgroups have been left as produced by the algorithm. To print them in canonical form (as left divisors of G in HNF form), one can for instance use

```
? G = matdiagonal([2,2]); forsubgroup(H=G, 2, print(mathnf(concat(G,H))))
[2, 1; 0, 1]
[1, 0; 0, 2]
[2, 0; 0, 1]
[1, 0; 0, 1]
```

Note that in this last representation, the index $[G : H]$ is given by the determinant. See `galoissubcyclo` and `galoisfixedfield` for `nfsubfields` applications to Galois theory.

Warning: the present implementation cannot treat a group G, if one of its p-Sylow subgroups has a cyclic factor with more than 2^{31}, resp. 2^{63} elements on a 32-bit, resp. 64-bit architecture.

3.11.1.8 forvec$(X = v, seq, \{flag = 0\})$: Let v be an n-component vector (where n is arbitrary) of two-component vectors $[a_i, b_i]$ for $1 \le i \le n$. This routine evaluates *seq*, where the formal variables $X[1], \ldots, X[n]$ go from a_1 to b_1,..., from a_n to b_n, i.e. X goes from $[a_1, \ldots, a_n]$ to $[b_1, \ldots, b_n]$ with respect to the lexicographic ordering. (The formal variable with the highest index moves the fastest.) If $flag = 1$, generate only nondecreasing vectors X, and if $flag = 2$, generate only strictly increasing vectors X.

3.11.1.9 if$(a, \{seq1\}, \{seq2\})$: evaluates the expression sequence *seq1* if a is non-zero, otherwise the expression *seq2*. Of course, *seq1* or *seq2* may be empty:

`if (`a`,`seq`)` evaluates *seq* if a is not equal to zero (you don't have to write the second comma), and does nothing otherwise,

`if (`a`,,`seq`)` evaluates *seq* if a is equal to zero, and does nothing otherwise. You could get the same result using the `!` (`not`) operator: `if (!`a`,`seq`)`.

Note that the boolean operators `&&` and `||` are evaluated according to operator precedence as explained in Section 2.4, but that, contrary to other operators, the evaluation of the arguments is stopped as soon as the final truth value has been determined. For instance

```
if (reallydoit && longcomplicatedfunction(), ...)%
```

is a perfectly safe statement.

Recall that functions such as `break` and `next` operate on *loops* (such as `for`*xxx*, `while`, `until`). The `if` statement is *not* a loop (obviously!).

3.11.1.10 next$(\{n = 1\})$: interrupts execution of current *seq*, resume the next iteration of the innermost enclosing loop, within the current function call (or top level loop). If n is specified, resume at the n-th enclosing loop. If n is bigger than the number of enclosing loops, all enclosing loops are exited.

3.11.1.11 return$(\{x = 0\})$: returns from current subroutine, with result x. If x is omitted, return the `(void)` value (return no result, like `print`).

3.11.1.12 until(a, seq): evaluates *seq* until a is not equal to 0 (i.e. until a is true). If a is initially not equal to 0, *seq* is evaluated once (more generally, the condition on a is tested *after* execution of the *seq*, not before as in `while`).

3.11.1.13 while(a, seq): while a is non-zero, evaluates the expression sequence *seq*. The test is made *before* evaluating the *seq*, hence in particular if a is initially equal to zero the *seq* will not be evaluated at all.

3.11.2 Specific functions used in GP programming.

In addition to the general PARI functions, it is necessary to have some functions which will be of use specifically for `gp`, though a few of these can be accessed under library mode. Before we start describing these, we recall the difference between *strings* and *keywords* (see Section 2.8): the latter don't get expanded at all, and you can type them without any enclosing quotes. The former are dynamic objects, where everything outside quotes gets immediately expanded.

3.11.2.1 addhelp(S, str): changes the help message for the symbol S. The string *str* is expanded on the spot and stored as the online help for S. If S is a function *you* have defined, its definition will still be printed before the message *str*. It is recommended that you document global variables and user functions in this way. Of course `gp` will not protest if you skip this.

Nothing prevents you from modifying the help of built-in PARI functions. (But if you do, we would like to hear why you needed to do it!)

3.11.2.2 alias($newkey, key$): defines the keyword *newkey* as an alias for keyword *key*. *key* must correspond to an existing *function* name. This is different from the general user macros in that alias expansion takes place immediately upon execution, without having to look up any function code, and is thus much faster. A sample alias file `misc/gpalias` is provided with the standard distribution. Alias commands are meant to be read upon startup from the `.gprc` file, to cope with function names you are dissatisfied with, and should be useless in interactive usage.

3.11.2.3 allocatemem($\{x = 0\}$): this is a very special operation which allows the user to change the stack size *after* initialization. x must be a non-negative integer. If $x \neq 0$, a new stack of size $16 * \lceil x/16 \rceil$ bytes is allocated, all the PARI data on the old stack is moved to the new one, and the old stack is discarded. If $x = 0$, the size of the new stack is twice the size of the old one.

Although it is a function, `allocatemem` cannot be used in loop-like constructs, or as part of a larger expression, e.g `2 + allocatemem()`. Such an attempt will raise an error. The technical reason is that this routine usually moves the stack, so objects from the current expression may not be correct anymore, e.g. loop indexes.

The library syntax is **allocatememem**(x), where x is an unsigned long, and the return type is void. `gp` uses a variant which makes sure it was not called within a loop.

3.11.2.4 default($\{key\}, \{val\}$): returns the default corresponding to keyword *key*. If *val* is present, sets the default to *val* first (which is subject to string expansion first). Typing `default()` (or `\d`) yields the complete default list as well as their current values. See Section 2.11 for a list of available defaults, and Section 2.12 for some shortcut alternatives. Note that the shortcut are meant for interactive use and usually display more information than `default`.

The library syntax is **gp_default**(key, val), where *key* and *val* are `char *`.

3.11.2.5 error($\{str\}*$): outputs its argument list (each of them interpreted as a string), then interrupts the running `gp` program, returning to the input prompt. For instance

```
error("n = ", n, " is not squarefree !")
```

UNIX: **3.11.2.6 extern**(str): the string *str* is the name of an external command (i.e. one you would type from your UNIX shell prompt). This command is immediately run and its input fed into `gp`, just as if read from a file.

The library syntax is **extern0**(str), where *str* is a `char *`.

3.11.2.7 getheap(): returns a two-component row vector giving the number of objects on the heap and the amount of memory they occupy in long words. Useful mainly for debugging purposes.

The library syntax is **getheap**().

3.11.2.8 getrand(): returns the current value of the random number seed. Useful mainly for debugging purposes.

The library syntax is **getrand**(), returns a C long.

3.11.2.9 getstack(): returns the current value of `top` − `avma`, i.e. the number of bytes used up to now on the stack. Should be equal to 0 in between commands. Useful mainly for debugging purposes.

The library syntax is **getstack**(), returns a C long.

3.11.2.10 gettime(): returns the time (in milliseconds) elapsed since either the last call to `gettime`, or to the beginning of the containing GP instruction (if inside `gp`), whichever came last.

The library syntax is **gettime**(), returns a C long.

3.11.2.11 global(*list of variables*): declares the corresponding variables to be global. From now on, you will be forbidden to use them as formal parameters for function definitions or as loop indexes. This is especially useful when patching together various scripts, possibly written with different naming conventions. For instance the following situation is dangerous:

```
p = 3    \\ fix characteristic
...
forprime(p = 2, N, ...)
f(p) = ...
```

since within the loop or within the function's body (even worse: in the subroutines called in that scope), the true global value of `p` will be hidden. If the statement `global(p = 3)` appears at the beginning of the script, then both expressions will trigger syntax errors.

Calling `global` without arguments prints the list of global variables in use. In particular, `eval(global)` will output the values of all global variables.

3.11.2.12 input(): reads a string, interpreted as a GP expression, from the input file, usually standard input (i.e. the keyboard). If a sequence of expressions is given, the result is the result of the last expression of the sequence. When using this instruction, it is useful to prompt for the string by using the `print1` function. Note that in the present version 2.19 of `pari.el`, when using `gp` under GNU Emacs (see Section 2.14) one *must* prompt for the string, with a string which ends with the same prompt as any of the previous ones (a `"?  "` will do for instance).

UNIX: **3.11.2.13 install**(*name*, *code*, {*gpname*}, {*lib*}): loads from dynamic library *lib* the function *name*. Assigns to it the name *gpname* in this `gp` session, with argument code *code* (see the Libpari Manual for an explanation of those). If *lib* is omitted, uses `libpari.so`. If *gpname* is omitted, uses *name*.

This function is useful for adding custom functions to the `gp` interpreter, or picking useful functions from unrelated libraries. For instance, it makes the function `system` obsolete:

```
? install(system, vs, sys, "libc.so")
? sys("ls gp*")
gp.c            gp.h            gp_rl.c
```

But it also gives you access to all (non static) functions defined in the PARI library. For instance, the function `GEN addii(GEN x, GEN y)` adds two PARI integers, and is not directly accessible under `gp` (it's eventually called by the `+` operator of course):

```
? install("addii", "GG")
? addii(1, 2)
%1 = 3
```

Re-installing a function will print a Warning, and update the prototype code if needed, but will reload a symbol from the library, even it the latter has been recompiled.

Caution: This function may not work on all systems, especially when `gp` has been compiled statically. In that case, the first use of an installed function will provoke a Segmentation Fault, i.e. a major internal blunder (this should never happen with a dynamically linked executable). Hence, if you intend to use this function, please check first on some harmless example such as the ones above that it works properly on your machine.

3.11.2.14 kill(s): kills the present value of the variable, alias or user-defined function s. The corresponding identifier can now be used to name any GP object (variable or function). This is the only way to replace a variable by a function having the same name (or the other way round), as in the following example:

```
? f = 1
%1 = 1
? f(x) = 0
  ***   unused characters: f(x)=0
                             ^----
? kill(f)
? f(x) = 0
? f()
%2 = 0
```

When you kill a variable, all objects that used it become invalid. You can still display them, even though the killed variable will be printed in a funny way. For example:

```
? a^2 + 1
%1 = a^2 + 1
? kill(a)
? %1
%2 = #<1>^2 + 1
```

If you simply want to restore a variable to its "undefined" value (monomial of degree one), use the quote operator: `a = 'a`. Predefined symbols (`x` and GP function names) cannot be killed.

3.11.2.15 print({*str*}$*$): outputs its (string) arguments in raw format, ending with a newline.

3.11.2.16 print1({*str*}$*$): outputs its (string) arguments in raw format, without ending with a newline (note that you can still embed newlines within your strings, using the `\n` notation !).

3.11.2.17 printp({*str*}$*$): outputs its (string) arguments in prettyprint (beautified) format, ending with a newline.

3.11.2.18 printp1({*str*}$*$): outputs its (string) arguments in prettyprint (beautified) format, without ending with a newline.

3.11.2.19 printtex({*str*}∗): outputs its (string) arguments in TEX format. This output can then be used in a TEX manuscript. The printing is done on the standard output. If you want to print it to a file you should use `writetex` (see there).

Another possibility is to enable the `log` default (see Section 2.11). You could for instance do:

```
default(logfile, "new.tex");
default(log, 1);
printtex(result);
```

3.11.2.20 quit(): exits `gp`.

3.11.2.21 read({*filename*}): reads in the file *filename* (subject to string expansion). If *filename* is omitted, re-reads the last file that was fed into `gp`. The return value is the result of the last expression evaluated.

If a GP `binary file` is read using this command (see Section 3.11.2.32), the file is loaded and the last object in the file is returned.

3.11.2.22 readvec({*str*}): reads in the file *filename* (subject to string expansion). If *filename* is omitted, re-reads the last file that was fed into `gp`. The return value is a vector whose components are the evaluation of all sequences of instructions contained in the file. For instance, if *file* contains

```
1
2
3
```

then we will get:

```
? \r a
%1 = 1
%2 = 2
%3 = 3
? read(a)
%4 = 3
? readvec(a)
%5 = [1, 2, 3]
```

In general a sequence is just a single line, but as usual braces and `\\` may be used to enter multiline sequences.

3.11.2.23 reorder({$x = []$}): x must be a vector. If x is the empty vector, this gives the vector whose components are the existing variables in increasing order (i.e. in decreasing importance). Killed variables (see `kill`) will be shown as `0`. If x is non-empty, it must be a permutation of variable names, and this permutation gives a new order of importance of the variables, *for output only*. For example, if the existing order is `[x,y,z]`, then after `reorder([z,x])` the order of importance of the variables, with respect to output, will be `[z,y,x]`. The internal representation is unaffected.

3.11.2.24 setrand(n): reseeds the random number generator to the value n. The initial seed is $n = 1$.

The library syntax is **setrand**(n), where n is a `long`. Returns n.

UNIX: **3.11.2.25 system**(*str*): *str* is a string representing a system command. This command is executed, its output written to the standard output (this won't get into your logfile), and control returns to the PARI system. This simply calls the C `system` command.

3.11.2.26 trap({*e*}, {*rec*}, {*seq*}): tries to evaluate *seq*, trapping error *e*, that is effectively preventing it from aborting computations in the usual way; the recovery sequence *rec* is executed if the error occurs and the evaluation of *rec* becomes the result of the command. If *e* is omitted, all exceptions are trapped. Note in particular that hitting `^C` (Control-C) raises an exception. See Section 2.9.2 for an introduction to error recovery under `gp`.

```
? \\ trap division by 0
? inv(x) = trap (gdiver, INFINITY, 1/x)
? inv(2)
%1 = 1/2
? inv(0)
%2 = INFINITY
```

If *seq* is omitted, defines *rec* as a default action when catching exception *e*, provided no other trap as above intercepts it first. The error message is printed, as well as the result of the evaluation of *rec*, and control is given back to the `gp` prompt. In particular, current computation is then lost.

The following error handler prints the list of all user variables, then stores in a file their name and their values:

```
? { trap( ,
      print(reorder);
      writebin("crash")) }
```

If no recovery code is given (*rec* is omitted) a *break loop* will be started (see Section 2.9.3). In particular

```
? trap()
```

by itself installs a default error handler, that will start a break loop whenever an exception is raised.

If *rec* is the empty string `""` the default handler (for that error if *e* is present) is disabled.

Note: The interface is currently not adequate for trapping individual exceptions. In the current version 2.3.5, the following keywords are recognized, but the name list will be expanded and changed in the future (all library mode errors can be trapped: it's a matter of defining the keywords to `gp`, and there are currently far too many useless ones):

`accurer`: accuracy problem

`archer`: not available on this architecture or operating system

`errpile`: the PARI stack overflows

`gdiver`: division by 0

`invmoder`: impossible inverse modulo

`siginter`: SIGINT received (usually from Control-C)

`talker`: miscellaneous error

`typeer`: wrong type

`user`: user error (from the `error` function)

3.11.2.27 type(x): this is useful only under `gp`. Returns the internal type name of the PARI object x as a string. Check out existing type names with the metacommand `\t`. For example `type(1)` will return "`t_INT`".

The library syntax is **type0**(x), though the macro `typ` is usually simpler to use since it return an integer that can easily be matched with the symbols `t_*`. The name `type` was avoided due to the fact that `type` is a reserved identifier for some C(++) compilers.

3.11.2.28 version(): Returns the current version number as a `t_VEC` with three integer components: major version number, minor version number and patchlevel. To check against a particular version number, you can use:

```
if (lex(version(), [2,2,0]) >= 0,
  \\ code to be executed if we are running 2.2.0 or more recent.
,
  \\ compatibility code
);
```

3.11.2.29 whatnow(*key*): if keyword *key* is the name of a function that was present in GP version 1.39.15 or lower, outputs the new function name and syntax, if it changed at all (387 out of 560 did).

3.11.2.30 write(*filename*, {*str*}$*$): writes (appends) to *filename* the remaining arguments, and appends a newline (same output as `print`).

3.11.2.31 write1(*filename*, {*str*}$*$): writes (appends) to *filename* the remaining arguments without a trailing newline (same output as `print1`).

3.11.2.32 writebin(*filename*, {x}): writes (appends) to *filename* the object x in binary format. This format is not human readable, but contains the exact internal structure of x, and is much faster to save/load than a string expression, as would be produced by `write`. The binary file format includes a magic number, so that such a file can be recognized and correctly input by the regular `read` or `\r` function. If saved objects refer to (polynomial) variables that are not defined in the new session, they will be displayed in a funny way (see Section 3.11.2.14).

If x is omitted, saves all user variables from the session, together with their names. Reading such a "named object" back in a `gp` session will set the corresponding user variable to the saved value. E.g after

```
x = 1; writebin("log")
```

reading `log` into a clean session will set `x` to 1. The relative variables priorities (see Section 2.5.4) of new variables set in this way remain the same (preset variables retain their former priority, but are set to the new value). In particular, reading such a session log into a clean session will restore all variables exactly as they were in the original one.

User functions, installed functions and history objects can not be saved via this function. Just as a regular input file, a binary file can be compressed using `gzip`, provided the file name has the standard `.gz` extension.

In the present implementation, the binary files are architecture dependent and compatibility with future versions of `gp` is not guaranteed. Hence binary files should not be used for long term storage (also, they are larger and harder to compress than text files).

3.11.2.33 writetex(*filename*, {*str*}$*$): as `write`, in TEX format.

Appendix A:
Installation Guide for the UNIX Versions

1. Required tools.

Compiling PARI requires an `ANSI C` or a `C++` compiler. If you do not have one, we suggest that you obtain the `gcc/g++` compiler. As for all GNU software mentioned afterwards, you can find the most convenient site to fetch `gcc` at the address

`http://www.gnu.org/order/ftp.html`

(On Mac OS X, this is also provided in the `Xcode` tool suite.) You can certainly compile PARI with a different compiler, but the PARI kernel takes advantage of optimizations provided by `gcc`. This results in at least 20% speedup on most architectures.

Optional packages. The following programs and libraries are useful in conjunction with `gp`, but not mandatory. In any case, get them before proceeding if you want the functionalities they provide. All of them are free.

• GNU `MP` library. This provides an alternative multiprecision kernel, which is faster than PARI's native one, but unfortunately binary incompatible. To enable detection of GMP, use `Configure --with-gmp`. You should really do this if you only intend to use GP, and probably also if you will use libpari unless you have backwards compatibility requirements.

• GNU `readline` library. This provides line editing under GP, an automatic context-dependent completion, and an editable history of commands. Note that it is incompatible with SUN commandtools (yet another reason to dump Suntools for X Windows).

• GNU `gzip/gunzip/gzcat` package enables GP to read compressed data.

• GNU `emacs`. GP can be run in an Emacs buffer, with all the obvious advantages if you are familiar with this editor. Note that `readline` is still useful in this case since it provides a better automatic completion than is provided by Emacs GP-mode.

• `perl` provides extended online help (full text from this manual) about functions and concepts, which can be used under GP or independently (`http://www.perl.com` will direct you to the nearest `CPAN` archive site).

• A colour-capable `xterm`, which enables GP to use different (user configurable) colours for its output. All `xterm` programs which come with current X11 distributions satisfy this requirement.

2. Compiling the library and the GP calculator.

2.1. Basic configuration: First, have a look at the `MACHINES` file to see if anything funny applies to your architecture or operating system. Then, type

```
./Configure
```

in the toplevel directory. This attempts to configure PARI/GP without outside help. Note that if you want to install the end product in some nonstandard place, you can use the `--prefix` option, as in

```
./Configure --prefix=/an/exotic/directory
```

(the default prefix is `/usr/local`). For example, to build a package for a Linux distribution, you may want to use

```
./Configure --prefix=/usr
```

This phase extracts some files and creates a directory `O`*xxx* where the object files and executables will be built. The *xxx* part depends on your architecture and operating system, thus you can build GP for several different machines from the same source tree (the builds are independent and can be done simultaneously).

Technical note: `Configure` accepts many other flags besides `--prefix`. See `Configure --help` for a complete list. In particular, there are sets of flags related to GNU MP (`--with-gmp*`) and GNU readline library (`--with-readline*`). Note that autodetection of GMP is *disabled* by default, you probably want to enable it.

Decide whether you agree with what `Configure` printed on your screen, in particular the architecture, compiler and optimization flags. Look for messages prepended by `###`, which report genuine problems. If anything should have been found and was not, consider that `Configure` failed and follow the instructions in the next section. Look especially for the `gmp`, `readline` and `X11` libraries, and the `perl` and `gunzip` (or `zcat`) binaries.

2.2. Compilation: To compile the GP binary and build the documentation, type

```
make all
```

To only compile the GP binary, type

```
make gp
```

in the toplevel directory. If your `make` program supports parallel make, you can speed up the process by going to the `O`*xxx* directory that `Configure` created and doing a parallel make here, for instance `make -j4` with GNU make. It should even work from the toplevel directory.

2.3. Basic tests:

To test the binary, type `make bench`. This will build a static executable (the default, built by `make gp` is probably dynamic) and run a series of comparative tests on those two. To test only the default binary, use `make dobench` which starts the bench immediately.

The static binary should be slightly faster. In any case, this should not take more than a few seconds on modern machines. See the file `MACHINES` to get an idea of how much time comparable systems need. We would appreciate a short note in the same format in case your system is not listed and you nevertheless have a working GP executable.

If a `[BUG]` message shows up, something went wrong. The testing utility directs you to files containing the differences between the test output and the expected results. Have a look and decide for yourself if something is amiss. If it looks like a bug in the Pari system, we would appreciate a report (see the last section).

3. Troubleshooting and fine tuning.

In case the default `Configure` run fails miserably, try

```
./Configure -a
```

(interactive mode) and answer all the questions (there are not that many). Of course, `Configure` still provides defaults for each answer but if you accept them all, it will fail just the same, so be wary. In any case, we would appreciate a bug report (see the last section).

3.1. Installation directories: The precise default destinations are as follows: the `gp` binary, the scripts `gphelp` and `tex2mail` go to `$prefix/bin`. The pari library goes to `$prefix/lib` and include files to `$prefix/include/pari`. Other system-dependant data go to `$prefix/lib/pari`.

Architecture independent files go to various subdirectories of `$share_prefix`, which defaults to `$prefix/share`, and can be specified via the `--share-prefix` argument. Man pages go into `$share_prefix/man`, Emacs files into `$share_prefix/emacs/site-lisp/pari`, and other system-independant data to various subdirectories of `$share_prefix/pari`: documentation, sample GP scripts and C code, extra packages like `elldata` or `galdata`.

You can also set directly `--bindir` (executables), `--libdir` (library), `--includedir` (include files), `--mandir` (manual pages), `--datadir` (other architecture-independent data), and finally `--sysdatadir` (other architecture-dependent data).

3.2. Environment variables: `Configure` lets the following environment variable override the defaults if set:

`AS`: Assembler.

`CC`: C compiler.

`DLLD`: Dynamic library linker.

`LD`: Static linker.

For instance, `Configure` may avoid `/bin/cc` on some architectures due to various problems which may have been fixed in your version of the compiler. You can try

```
env CC=cc Configure
```

and compare the benches. Also, if you insist on using a `C++` compiler and run into trouble with a fussy `g++`, try to use `g++ -fpermissive`.

The contents of the following variables are *appended* to the values computed by `Configure`:

`CFLAGS`: Flags for `CC`.

`CPPFLAGS`: Flags for `CC` (preprocessor).

`LDFLAGS`: Flags for `LD`.

The contents of the following variables are *prepended* to the values computed by `Configure`:

`C_INCLUDE_PATH` is prepended to the list of directories searched for include files. Note that adding `-I` flags to `CFLAGS` is not enough since `Configure` sometimes relies on finding the include files and parsing them, and it does not parse `CFLAGS` at this time.

`LIBRARY_PATH` is prepended to the list of directories searched for libraries.

You may disable inlining by adding `-DDISABLE_INLINE` to `CFLAGS`, and prevent the use of the `volatile` keyword with `-DDISABLE_VOLATILE`.

3.3. Debugging/profiling: If you also want to debug the PARI library,

```
Configure -g
```

creates a directory `O`*xxx*`.dbg` containing a special `Makefile` ensuring that the GP and PARI library built there is suitable for debugging. If you want to profile GP or the library, using `gprof` for instance,

```
Configure -pg
```

will create an `O`*xxx*`.prf` directory where a suitable version of PARI can be built.

The GP binary built above with `make all` or `make gp` is optimized. If you have run `Configure -g` or `-pg` and want to build a special purpose binary, you can `cd` to the `.dbg` or `.prf` directory and type `make gp` there. You can also invoke `make gp.dbg` or `make gp.prf` directly from the toplevel.

3.4. Multiprecision kernel: The kernel can be fully specified via the `--kernel=`*fqkn* switch. The PARI kernel is build from two kernels, called level 0 (L0, operation on words) and level 1 (L1, operation on multi-precision integer and real).

Available kernels:

L0: `auto`, `none` and

```
alpha hppa hppa64 ia64 ix86 x86_64 m68k ppc sparcv7
sparcv8_micro sparcv8_super
```

L1: `auto`, `none` and `gmp`

`auto` means to use the auto-detected value. L0=`none` means to use the portable C kernel (no assembler), L1=`none` means to use the PARI L1 kernel.

• A fully qualified kernel name *fqkn* is of the form L_0-L_1.

• A *name* not containing a dash '-' is an alias. An alias stands for *name*-`none`, but `gmp` stand for `auto-gmp`.

• The default kernel is `auto-none`.

3.5. Problems related to readline: `Configure` does not try very hard to find the `readline` library and include files. If they are not in a standard place, it will not find them. Nonetheless, it first searches the distribution toplevel for a `readline` directory. Thus, if you just want to give `readline` a try, as you probably should, you can get the source and compile it there (you do not need to install it). You can also use this feature together with a symbolic link, named `readline`, in the PARI toplevel directory if you have compiled the readline library somewhere else, without installing it to one of its standard locations.

You can also invoke `Configure` with one of the following arguments:

`--with-readline[=`*prefix to* `lib/libreadline.`*xx and* `include/readline.h]`

`--with-readline-lib=`*path to* `libreadline.`*xx*

`--with-readline-include=`*path to* `readline.h`

Technical note: Configure can build GP on different architectures simultaneously from the same toplevel sources. Instead of the `readline` link alluded above, you can create `readline-`*osname-arch*, using the same naming conventions as for the `O`*xxx* directory, e.g `readline-linux-i686`.

Known problems:

• on Linux: Linux distributions have separate `readline` and `readline-devel` packages. You need both of them installed to compile gp with readline support. If only `readline` is installed, `Configure` will complain. `Configure` may also complain about a missing libncurses.so, in which case, you have to install the `ncurses-devel` package (some distributions let you install `readline-devel` without `ncurses-devel`, which is a bug in their package dependency handling).

• on OS X.4: Tiger comes equipped with a fake `readline`, which is not sufficient for our purpose. As a result, gp is built without readline support. Since `readline` is not trivial to install in this environment, a step by step solution can be found in the PARI FAQ, see

`http://pari.math.u-bordeaux.fr/`

3.6. Testing

3.6.1. Known problems: if `BUG` shows up in `make bench`

• `program`: the GP function `install` may not be available on your platform, triggering an error message ("not yet available for this architecture"). Have a look at the `MACHINES` files to check if your system is known not to support it, or has never been tested yet.

• If when running `gp-dyn`, you get a message of the form

`ld.so: warning: libpari.so.`*xxx* `has older revision than expected` *xxx*

(possibly followed by more errors), you already have a dynamic PARI library installed *and* a broken local configuration. Either remove the old library or unset the `LD_LIBRARY_PATH` environment variable. Try to disable this variable in any case if anything *very* wrong occurs with the `gp-dyn` binary, like an Illegal Instruction on startup. It does not affect `gp-sta`.

• Some implementations of the `diff` utility (on HPUX for instance) output `No differences encountered` or some similar message instead of the expected empty input. Thus producing a spurious `[BUG]` message.

3.6.2. Some more testing [*Optional*]

You can test GP in compatibility mode with `make test-compat`. If you want to test the graphic routines, use `make test-ploth`. You will have to click on the mouse button after seeing each image. There will be eight of them, probably shown twice (try to resize at least one of them as a further test). More generaly, typing `make` without argument will print the list of available extra tests among all available targets.

The `make bench` and `make test-compat` runs produce a Postscript file `pari.ps` in `O`*xxx* which you can send to a Postscript printer. The output should bear some similarity to the screen images.

3.6.3. Heavy-duty testing [*Optional*] There are a few extra tests which should be useful only for developpers.

`make test-kernel` checks whether the low-level kernel seems to work, and provides simple diagnostics if it does not. Only useful if `make bench` fails horribly, e.g. things like `1+1` do not work.

`make test-all` runs all available test suites. Slow.

4. Installation.

When everything looks fine, type

```
make install
```

You may have to do this with superuser privileges, depending on the target directories. (Tip for MacOS X beginners: use `sudo make install`.) In this case, it is advised to type `make all` first to avoid running unnecessary commands as `root`.

Beware that, if you chose the same installation directory as before in the `Configure` process, this will wipe out any files from version 1.39.15 and below that might already be there. Libraries and executable files from newer versions (starting with version 1.900) are not removed since they are only links to files bearing the version number (beware of that as well: if you are an avid `gp` fan, do not forget to delete the old pari libraries once in a while).

This installs in the directories chosen at `Configure` time the default GP executable (probably `gp-dyn`) under the name `gp`, the default PARI library (probably `libpari.so`), the necessary include files, the manual pages, the documentation and help scripts and emacs macros.

To save on disk space, you can manually `gzip` some of the documentation files if you wish: `usersch*.tex` and all `dvi` files (assuming your `xdvi` knows how to deal with compressed files); the online-help system can handle it.

By default, if a dynamic library `libpari.so` could be built, the static library `libpari.a` will not be created. If you want it as well, you can use the target `make install-lib-sta`. You can install a statically linked `gp` with the target `make install-bin-sta`. As a rule, programs linked statically (with `libpari.a`) may be slightly faster (about 5% gain), but use more disk space and take more time to compile. They are also harder to upgrade: you will have to recompile them all instead of just installing the new dynamic library. On the other hand, there is no risk of breaking them by installing a new pari library.

4.1. Extra packages: The following optional packages endow PARI with some extra capabilities (only two packages for now!).

• `elldata`: This package contains the elliptic curves in John Cremona's database. It is needed by the functions `ellidentify`, `ellsearch` and can be used by `ellinit` to initialize a curve given by its standard code.

• `galdata`: The default `polgalois` function can only compute Galois groups of polynomials of degree less or equal to 7. Install this package if you want to handle polynomials of degree bigger than 7 (and less than 11).

To install package *pack*, you need to fetch the separate archive: *pack*`.tgz` which you can download from the `pari` server. Copy the archive in the PARI toplevel directory, then extract its contents; these will go to `data/`*pack*`/`. Typing `make install` installs all such packages.

4.2. The `GPRC` file: Copy the file `misc/gprc.dft` (or `gprc.dos` if you are using `GP.EXE`) to `$HOME/.gprc`. Modify it to your liking. For instance, if you are not using an ANSI terminal, remove control characters from the `prompt` variable. You can also enable colors.

If desired, read `$datadir/misc/gpalias` from the `gprc` file, which provides some common shortcuts to lengthy names; fix the path in gprc first. (Unless you tampered with this via Configure, `datadir` is `$prefix/share/pari`.) If you have superuser privileges and want to provide system-wide defaults, copy your customized `.gprc` file to `/etc/gprc`.

In older versions, `gphelp` was hidden in pari lib directory and was not meant to be used from the shell prompt, but not anymore. If gp complains it cannot find `gphelp`, check whether your `.gprc` (or the system-wide `gprc`) does contain explicit paths. If so, correct them according to the current `misc/gprc.dft`.

5. Getting Started.

5.1. Printable Documentation: Building gp with `make all` also builds its documentation. You can also type directly `make doc`. In any case, you need a working (plain) TEX installation.

After that, the `doc` directory contains various `dvi` files: `libpari.dvi` (manual for the PARI library), `users.dvi` (manual for the `gp` calculator), `tutorial.dvi` (a tutorial), and `refcard.dvi` (a reference card for GP). You can send these files to your favourite printer in the usual way, probably via `dvips`. The reference card is also provided as a `PostScript` document, which may be easier to print than its `dvi` equivalent (it is in Landscape orientation and assumes A4 paper size).

If the `pdftex` package is part of your TEX setup, you can produce these documents in PDF format, which may be more convenient for online browsing (the manual is complete with hyperlinks); type

```
make docpdf
```

All these documents are available online from PARI home page (see the last section).

5.2. C programming: Once all libraries and include files are installed, you can link your C programs to the PARI library. A sample makefile `examples/Makefile` is provided to illustrate the use of the various libraries. Type `make all` in the `examples` directory to see how they perform on the `extgcd.c` program, which is commented in the manual.

This should produce a statically linked binary `extgcd-sta` (standalone), a dynamically linked binary `extgcd-dyn` (loads libpari at runtime) and a shared library `libextgcd`, which can be used from `gp` to `install` your new `extgcd` command.

The standalone binary should be bulletproof, but the other two may fail for various reasons. If when running `extgcd-dyn`, you get a message of the form "DLL not found", then stick to statically linked binaries or look at your system documentation to see how to indicate at linking time where the required DLLs may be found! (E.g. on Windows, you will need to move `libpari.dll` somewhere in your `PATH`.)

5.3. GP scripts: Several complete sample GP programs are also given in the `examples` directory, for example Shanks's SQUFOF factoring method, the Pollard rho factoring method, the Lucas-Lehmer primality test for Mersenne numbers and a simple general class group and fundamental unit algorithm. See the file `examples/EXPLAIN` for some explanations.

5.4. EMACS: If you want to use `gp` under GNU Emacs, read the file `emacs/pariemacs.txt`. If you are familiar with Emacs, we suggest that you do so.

5.5. The PARI Community: PARI's home page at the address

`http://pari.math.u-bordeaux.fr/`

maintains an archive of mailing lists dedicated to PARI, documentation (including Frequently Asked Questions), a download area and our Bug Tracking System (BTS). Bug reports should be submitted online to the BTS, which may be accessed from the navigation bar on the home page or directly at

`http://pari.math.u-bordeaux.fr/Bugs`

Further information can be found at that address but, to report a configuration problem, make sure to include the relevant `*.dif` files in the `O`*xxx* directory and the file `pari.cfg`.

There are three mailing lists devoted to PARI/GP (run courtesy of Dan Bernstein), and most feedback should be directed to those. They are:

• `pari-announce`: to announce major version changes. You cannot write to this one, but you should probably subscribe.

• `pari-dev`: for everything related to the development of PARI, including suggestions, technical questions, bug reports or patch submissions. (The BTS forwards the mail it receives to this list.)

• `pari-users`: for everything else.

You may send an email to the last two without being subscribed. (You will have to confirm that your message is not unsollicited bulk email, aka *Spam*.) To subscribe, send empty messages respectively to

`pari-announce-subscribe@list.cr.yp.to`
`pari-users-subscribe@list.cr.yp.to`

pari-dev-subscribe@list.cr.yp.to

You can also write to us at the address

pari@math.u-bordeaux.fr

but we cannot promise you will get an individual answer.

If you have used PARI in the preparation of a paper, please cite it in the following form (BibTeX format):

```
@manual{PARI2,
    organization = "{The PARI~Group}",
    title        = "{PARI/GP, Version 2.3.5}",
    year         = 2006,
    address      = "Bordeaux",
    note         = "available from {\tt http://pari.math.u-bordeaux.fr/}"
}
```

In any case, if you like this software, we would be indebted if you could send us an email message giving us some information about yourself and what you use PARI for.

Good luck and enjoy!

Index

SomeWord refers to PARI-GP concepts.
`SomeWord` is a PARI-GP keyword.
SomeWord is a generic index entry.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z