User's Guide

to

PARI / GP

(version 2.5.5)

The PARI Group

Institut de Mathématiques de Bordeaux, UMR 5251 du CNRS.
Université Bordeaux 1, 351 Cours de la Libération
F-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 has been put to good use in many other different fields, from topology or numerical analysis to physics.

Although quite an amount of symbolic manipulation is possible, PARI does badly compared to systems like Axiom, Magma, Maple, Mathematica, Maxima, 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 (from the above-mentioned systems, only Magma provides similar features).

Non-mathematical strong points include the possibility to program either in high-level scripting languages or with the PARI library, a mature system (development started in the mid eighties) that was used to conduct and disseminate original mathematical research, while building a large user community, linked by helpful mailing lists and a tradition of great user support from the developers. And, of course, PARI/GP is Free Software, covered by the GNU General Public License, either version 2 of the License or (at your option) any later version.

PARI is used in three different ways:

1) as a library `libpari`, 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 `libpari` 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`.

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 information, 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 Multiprecision kernels / Portability.

The PARI multiprecision kernel comes in three non exclusive flavors. 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 old 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.

A 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, use the GMP kernel!

1.3 The PARI types.

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

The crucial word is recursiveness: most of the PARI types 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, it *is* allowed to have components of different, but compatible, types, which can be freely mixed in basic ring operations $+$ or $\times$. For example, taking again complex numbers, the real part could be an integer, and the imaginary part a rational number. On the other hand, if the real part is a real number, the imaginary part cannot be an integer modulo 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_FFELT`	$\mathbf{F}_q$	Finite field element
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`	$T[X]/(P)$	Polmods (polynomials modulo $P \in T[X]$)
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
type `t_CLOSURE`		Functions

and where the types T in recursive types can be different in each component. The first nine basic types, from `t_INT` to `t_POLMOD`, are called scalar types because they essentially occur as coefficients of other more complicated objects. Type `t_POLMOD` is used to define algebraic extensions of a base ring, and as such is a scalar type.

In addition, there exist types `t_QFR` and `t_QFI` for integral binary quadratic forms, and the internal type `t_VECSMALL`. The latter 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.

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

1.3.1 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 nonexistent): integers must be in absolute value less than $2^{536870815}$ (i.e. roughly 161614219 decimal digits). The precision of real numbers is also at most 161614219 significant decimal digits, and the binary exponent must be in absolute value less than 2^{29}.

Integers and real numbers are non-recursive types.

1.3.2 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.

1.3.3 Finite field elements:

The exact internal format depends of the finite field size, but it includes the field characteristic p, an irreducible polynomial $T \in \mathbf{F}_p[X]$ defining the finite field $\mathbf{F}_p[X]/(T)$ and the element expressed as a polynomial in (the class of) X.

1.3.4 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.

1.3.5 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 2.5.5 of PARI, it is not possible to handle conveniently 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 is a difficult design problem: the mathematical problem itself contains some amount of imprecision, and it is not easy to design an intuitive generic interface for such beasts.

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

1.3.7 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: there are infinitely many distinct zeros of each of these types! For p-adics and power series the answer is as follows: every such object, including 0, has an exponent e. This p-adic or X-adic zero is understood to be equal to $O(p^e)$ or $O(X^e)$ respectively.

Real numbers also have exponents and a real zero is in fact $O(2^e)$ where e is now usually a negative binary exponent. This of course is printed as usual for a floating point number ($0.00\cdots$ or $0.Exx$ depending on the output 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.

1.4 The PARI philosophy.

The basic principles which govern 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.

In this respect, we make an important distinction between exact and inexact objects: by definition, types `t_REAL`, `t_PADIC` or `t_SER` are imprecise. A PARI object having one of these imprecise types anywhere in its tree is *inexact*, and *exact* otherwise. No loss of accuracy (rounding error) is involved when dealing with exact objects. Specifically, an exact operation between exact objects will yield an exact object. For example, dividing 1 by 3 does not give $0.333\cdots$, but the rational number $(1/3)$. To get the result as a floating point real number, evaluate `1./3` or `0.+1/3`.

Conversely, the result of operations between imprecise objects, although inexact by nature, 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, its value is only used to convert exact types to inexact ones.

In particular, 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 one wants to apply a given function to all elements in a vector. This is easily done using a loop, or using the `apply` built-in function, but in fact PARI assumes that this is exactly what you want to

do when you apply a scalar function to a vector. Taking the exponential of a vector will do just that, so no work is necessary. Most transcendental functions work in the same way*.

In the same spirit, when objects of different types are combined they are first automatically mapped to a suitable ring, where the computation becomes meaningful:

```
? 1/3 + Mod(1,5)
%1 = Mod(3, 5)
? I + O(5^9)
%2 = 2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + O(5^9)
? Mod(1,15) + Mod(1,10)
%3 = Mod(2, 5)
```

The first example is straightforward: since 3 is invertible mod 5, (1/3) is easily mapped to $\mathbf{Z}/5\mathbf{Z}$. In the second example, `I` stands for the customary square root of -1; we obtain a 5-adic number, 5-adically close to a square root of -1. The final example is more problematic, but there are natural maps from $\mathbf{Z}/15\mathbf{Z}$ and $\mathbf{Z}/10\mathbf{Z}$ to $\mathbf{Z}/5\mathbf{Z}$, and the computation takes place there.

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. We emphasize once more 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; 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.

1.5.3 Transcendental functions

They usually operate on any complex number, power series, and some also on p-adics. The list is ever-expanding and of course contains all the elementary functions (exp/log, trigonometric functions), plus many others (modular functions, Bessel functions, polylogarithms...). Recall that by extension, PARI usually allows a transcendental function to operate componentwise on vectors or matrices.

* An ambiguity arises with square matrices. PARI always considers that you want to do componentwise function evaluation in this context, hence to get for example the standard exponential of a square matrix you would need to implement a different function.

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, Galois and 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.

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

Chapter 2:
The gp Calculator

2.1 Introduction.

Originally, `gp` was designed as a debugging device for the PARI system library. Over the years, it has become a powerful user-friendly stand-alone calculator. The mathematical 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 and use the PariEmacs package, you can work in a special Emacs shell, described in Section 2.16. 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` *parisize*`] [-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 2.12 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 (to be discussed in Section 2.14) is found, `gp` then reads it and executes the commands it contains. This provides an easy way to customize `gp`. The *files* argument is processed right after the `gprc`.

A copyright banner 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 2.13.1) 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.

More generally, commands starting with the symbols `\` 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 2.13).

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`: a final semicolon means the result should not be printed. (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

```
***   at top-level: gcd(a,b)
***                 ^--------
*** gcd: user interrupt after 236 ms.
```

telling you how much time elapsed 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, and has been strongly influenced by C and Perl since then.

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.

More precisely, the input is case-sensitive and, outside of character strings, blanks are completely ignored. Inputs are either metacommands or sequences of expressions. Metacommands are shortcuts designed to alter gp's internal state, such as the working precision or general verbosity level; we shall describe them in Section 2.13, and ignore them for the time being.

The evaluation of a sequence of instructions proceeds in two phases: your input is first digested (byte-compiled) to a bytecode suitable for fast evaluation, in particular loop bodies are compiled only once but a priori evaluated many times; then the bytecode is evaluated.

An expression is formed by combining constants, variables, operator symbols, functions and control statements. It is evaluated using the conventions about operator priorities and left to right associativity. An expression always has a value, which can be any PARI object:

```
? 1 + 1
%1 = 2              \\ an ordinary integer
? x
%2 = x              \\ a polynomial of degree 1 in the unknown x
? print("Hello")
Hello               \\ void return value
? f(x) = x^2
%3 = (x)->x^2       \\ a user function
```

In the third example, `Hello` is printed as a side effect, but is not the return value. The `print` command is a *procedure*, which conceptually returns nothing. But in fact procedures return a special `void` object, meant to be ignored; in particular, it does not clutter the history (but evaluates to 0 in a numeric context). The final example assigns to the variable `f` the function $x \mapsto x^2$, the alternative form `f = x->x^2` achieving the same effect; the return value of a function definition is, unsurprisingly, a function object (of type `t_CLOSURE`).

Several expressions are combined on a single line by separating them with semicolons (';'). Such an expression sequence will be called 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 `;`.

* 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.

2.2.2 The gp history of results

This is not to be confused with the history of your *commands*, maintained by `readline`. The `gp` history contains the *results* they produced, in sequence. More precisely, several inputs act through side effects and produce a `void` result, for instance a `print` statement or a `for` loop. The `gp` history consists exactly of the non-`void` results.

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, it is still stored in the history, but its history number will not appear either. It is 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, you may just use `%`.

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 your commands are executed as soon as you have finished typing them, there must be a way to tell `gp` to wait for the next line or lines of input before doing anything. There are three ways of doing this.

The first one is 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 variation on the first, and is mostly useful when defining a user function (see Section 2.7): since an equal sign can never end a valid expression, `gp` disregards a newline immediately following an `=`.

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

The third one is in general much more useful, and uses braces `{` and `}`. An opening brace `{` signals that you are typing a multi-line command, and newlines are ignored until you type a closing brace `}`. There are two important, but easily obeyed, restrictions: first, braces do not nest; second, inside an open brace-close brace pair, all input lines are concatenated, suppressing any newlines. Thus, all newlines should occur after a semicolon (`;`), a comma (`,`) or an operator (for clarity's sake, never split 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`. It should have been written

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

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 quite 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 (`t_INT`): after an (optional) leading `+` or `-`, type in the decimal digits of your integer. No decimal point!

```
? 1234567
%1 = 1234567
? -3
%2 = -3
? 1.          \\ oops, not an integer
%3 = 1.0000000000000000000000000000
```

2.3.2 Real numbers (`t_REAL`): after an (optional) leading `+` or `-`, type a number with a decimal point. Leading zeroes may be omitted, up to the decimal point, but trailing zeroes are important: your `t_REAL` is assigned an internal precision, which is the supremum of the input precision and the default precision, expressed in decimal digits. For example, if the default precision is 28 digits, typing `2.` yields a precision of 28 digits, but `2.0...0` with 45 zeros 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`. As usual, `e`n is interpreted as $\times 10^n$ for all integers n. Since the result is converted to a `t_REAL`, you may often omit the decimal point in this case: `6.02 E 23` or `1e-5` are fine, but `e10` is not.

By definition, `0.E` n returns a real 0 of exponent n, whereas `0.` returns a real 0 "of default precision" (of exponent $-$`realprecision`), see Section 1.3.7, behaving like the machine epsilon for the current default accuracy: any float of smaller absolute value is indistinguishable from 0.

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 1.3.7). 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.

2.3.3 Intmods (`t_INTMOD`): to create the image of the integer a in $\mathbf{Z}/b\mathbf{Z}$ (for some non-zero integer b), type `Mod(a,b)`; *not* `a%b`. Internally, all operations are done on integer representatives belonging to $[0, b-1]$.

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

If x is a `t_INTMOD` `Mod(a,b)`, the following member function is defined:

`x.mod`: return the modulus `b`.

2.3.4 Rational numbers (`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, floating point division is *not* performed, only reduction to lowest terms.

Note that rational computation are almost never the fastest method to proceed: in the PARI implementation, each elementary operation involves computing a gcd. It is generally a little more efficient to cancel denominators and work with integers only:

```
? P = Pol( vector(10^3,i, 1/i) ); \\ big polynomial with small rational coeffs
? P^2
time = 1,392 ms.
? c = content(P); c^2 * (P/c)^2;  \\ same computation in integers
time = 1,116 ms.
```

And much more efficient (but harder to setup) to use homomorphic imaging schemes and modular computations. As the simple example below indicates, if you only need modular information, it is very worthwhile to work with `t_INTMOD`s directly, rather than deal with `t_FRAC`s all the way through:

```
? p = nextprime(10^7);
? sum(i=1, 10^5, 1/i) % p
time = 13,288 ms.
%1 = 2759492
? sum(i=1, 10^5, Mod(1/i, p))
time = 60 ms.
%2 = Mod(2759492, 10000019)
```

2.3.5 Finite field elements (`t_FFELT`): let $T \in \mathbf{F}_p[X]$ be a monic irreducible polynomial defining your finite field over $\mathbf{F}_p$, for instance obtained using `ffinit`. Then the `ffgen` function creates a generator of the finite field as an $\mathbf{F}_p$-algebra, namely the class of X in $\mathbf{F}_p[X]/(T)$, from which you can build all other elements. For instance, to create the field $\mathbf{F}_{2^8}$, we write

```
? T = ffinit(2, 8);
? y = ffgen(T, 'y);
? y^0    \\ the unit element in the field
%3 = 1
? y^8
%4 = y^6 + y^5 + y^4 + y^3 + y + 1
```

The second (optional) parameter to `ffgen` is only used to display the result; it is customary to use the name of the variable we assign the generator to. If `f` is a `t_FFELT`, the following member functions are defined:

`f.pol`: the polynomial (with reduced integer coefficients) expressing `f` in term of the field generator.

`f.p`: the characteristic of the finite field.

`f.mod`: the minimal polynomial (with reduced integer coefficients) of the field generator.

2.3.6 Complex numbers (`t_COMPLEX`): to enter $x + iy$, type `x + I*y`. (That's `I`, *not* `i`!) The letter `I` stands for $\sqrt{-1}$. The "real" and "imaginary" parts x and y can be of type `t_INT`, `t_REAL`, `t_INTMOD`, `t_FRAC`, or `t_PADIC`.

2.3.7 p-adic numbers (`t_PADIC`): Typing `O(p^k)`, where p and k are integers, yields a p-adic 0 of accuracy k, representing any p-adic number whose valuation is $\geq k$. To input a general non-0 p-adic number, write a suitably precise rational or integer approximation and add `O(p^k)` to it.

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)`.

If a is a `t_PADIC`, the following member functions are defined:

`a.mod`: returns the modulus p^k.

`a.p`: returns p.

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=1,10^6, n++)
time = 841 ms.
? n = Mod(1,2^20); for (i=1,10^6, n++)
time = 441 ms.
```

```
? n = 1;            for (i=1,10^6, n++)
time = 328 ms.
```

The penalty associated with maintaining p^k decreases steeply as p increases (and updates become very rare). But `t_INTMOD`s remain at least 25% more efficient. (But they do not have denominators!)

2.3.8 Quadratic numbers (`t_QUAD`): This type is used to work in the quadratic order of *discriminant* `d`, where `d` is a non-square integer congruent to 0 or 1 (modulo 4). The command

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

assigns to `w` the "canonical" generator for the integer basis of the order of discriminant d, i.e. $w = \sqrt{d}/2$ if $d \equiv 0 \bmod 4$, and $w = (1+\sqrt{d})/2$ if $d \equiv 1 \bmod 4$. The name `w` is of course just a suggestion, but corresponds to traditional usage. You can use any variable name that you like, but `quadgen(d)` is always printed as `w`, regardless of the discriminant. So beware, two `t_QUAD`s can be printed in the same way and not be equal; however, `gp` will refuse to add or multiply them for example.

Since the order is $\mathbf{Z} + \mathtt{w}\mathbf{Z}$, any other element can be input as x`+`y`*w` for some integers x and y. In fact, you may work in its fraction field $\mathbf{Q}(\sqrt{d})$ and use `t_FRAC` values for x and y.

2.3.9 Polmods (`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. If p is a `t_POLMOD`, the following member functions are defined:

`p.pol`: return a representative of the polynomial class of minimal degree.

`p.mod`: return the modulus.

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 2.5.3 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)
```

2.3.10 Polynomials (`t_POL`): type the polynomial in a natural way, not forgetting to put a "$*$" between a coefficient and a formal variable;

```
? 1 + 2*x + 3*x^2
%1 = 3*x^2 + 2*x + 1
```

This assumes that `x` is still a "free variable".

```
? x = 1; 1 + 2*x + 3*x^2
%2 = 6
```

generates an integer, not a polynomial! It is good practice to never assign values to polynomial variables to avoid the above problem, but a foolproof construction is available using `'x` instead of `x`: `'x` is a constant evaluating to the free variable with name `x`, independently of the current value of `x`.

```
? x = 1; 1 + 2*'x + 3*'x^2
%3 = 1 + 2*x + 3*x^2
? x = 'x; 1 + 2*x + 3*x^2
%4 = 1 + 2*x + 3*x^2
```

You may also use the functions `Pol` or `Polrev`:

```
? Pol([1,2,3])          \\ Pol creates a polynomial in x by default
%1 = x^2 + 2*x + 3
? Polrev([1,2,3])
%2 = 3*x^2 + 2*x + 1
? Pol([1,2,3], 'y)      \\ we use 'y, safer than y
%3 = y^2 + 2*y + 3
```

The latter two are much more efficient constructors than an explicit summation (the latter is quadratic in the degree, the former linear):

```
? for (i=1, 10^4, Polrev( vector(100, i,i) ) )
time = 124ms
? for (i=1, 10^4, sum(i = 1, 100, (i+1) * 'x^i) )
time = 3,985ms
```

Polynomials are always printed as *univariate* polynomials, with monomials sorted by decreasing degree:

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

(Univariate polynomial in `x` whose coefficients are polynomials in `y`.) See Section 2.5 for valid variable names, and a discussion of multivariate polynomial rings.

2.3.11 Power series (`t_SER`): Typing `O(X^k)`, where k is an integer, yields an X-adic 0 of accuracy k, representing any power series in `X` whose valuation is $\geq k$. Of course, `X` can be replaced by any other variable name! To input a general non-0 power series, type in a polynomial or rational function (in `X`, say), and add `O(X^k)` to it. The discussion in the `t_POL` section about variables remains valid; a constructor `Ser` replaces `Pol` and `Polrev`.

Caveat. Power series with inexact coefficients sometimes have a non-intuitive behavior: 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 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 output could be construed as a bug since it is a priori impossible to deduce such a result from the input (0. represents 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.

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
  *** _/_: Warning: normalizing a series with 0 leading term.
%3 = 1.0000000000000000000000000000*y^-1 + O(1)
? A*B
  *** _*_: Warning: normalizing a series with 0 leading term.
%4 = 1.0000000000000000000000000000*y^-1 + O(1)
```

2.3.12 Rational functions (`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.

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

2.3.14 Row and column vectors (`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).

If the variable x contains a (row or column) vector, $x[m]$ refers to its m-th entry. You can assign a result to $x[m]$ (i.e. write something like $x[k] = expr$).

2.3.15 Matrices (`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 an empty (0×0) matrix. The function `Mat` transforms any object into a matrix, and `matrix` creates matrices whose (i, j)-th entry is described by a function $f(i, j)$:

```
? Mat(1)
%1 =
[1]
? matrix(2,2, i,j, 2*i+j)
%2 =
[3 4]

[5 6]
```

If `M` is a matrix, `M[i,j]` refers to its (i, j)-th entry, `M[i,]` to its i-th row, and `M[,j]` to its j-th column; but `M[i]` is meaningless and triggers an error. You can assign a result to `M[i,j]`, `M[i,]` or `M[,j]`, provided that the expression is a row or column vector of the right dimension in the latter two cases. This process is recursive, so if `M` is a matrix of matrices of ..., an expression such as `M[1,1][,3][4] = 1` is perfectly valid (and actually identical to `M[1,1][4,3] = 1`), assuming that all matrices along the way have compatible dimensions.

Technical note (design flaw). Matrices are internally represented as a vector of columns. All matrices with 0 columns are thus represented by the same object (internally, an empty vector), and there is no way to distinguish between them. Thus it is not possible to create or represent matrices with zero columns and an actual nonzero number of rows. The empty matrix `[;]` is handled as though it had an arbitrary number of rows, exactly as many as needed for the current computation to make sense:

```
? [1,2,3; 4,5,6] * [;]
%1 = [;]
```

The empty matrix on the first line is understood as a 3×0 matrix, and the result as a 2×0 matrix. On the other hand, it is possible to create matrices with a given positive number of columns, each of which has zero rows, e.g. using `Mat` as above or using the `matrix` function.

Note that although the internal representation is essentially the same, a row vector of column vectors is *not* a matrix; for example, multiplication will not work in the same way. It is easy to go from one representation to the other using `Vec` / `Mat`, though:

```
? [1,2,3;4,5,6]
%1 =
[1 2 3]

[4 5 6]

? Vec(%)
```

```
%2 = [[1, 4]~, [2, 5]~, [3, 6]~]
? Mat(%)
%3 =
[1 2 3]

[4 5 6]
```

2.3.16 Lists (`t_LIST`): lists can be input directly, as in `List([1,2,3,4])`; but in most cases, one creates an empty list, then appends elements using `listput`:

```
? a = List(); listput(a,1); listput(a,2);
? a
%2 = List([1, 2])
```

Elements can be accessed directly as with the vector types described above.

2.3.17 Strings (`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.

2.3.18 Small vectors (`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.

2.3.19 Functions (`t_CLOSURE`): we will explain this at length in Section 2.7. For the time being, suffice it to say that functions can be assigned to variables, as any other object, and the following equivalent basic forms are available to create new ones

```
f = (x,y) -> x^2 + y^2

f(x,y) = x^2 + y^2
```

2.4 GP operators.

Loosely speaking, an operator is a function, usually associated to basic arithmetic operations, whose name contains only non-alphanumeric characters. For instance `+` or `-`, but also `=` or `+=`, or even `[ ]` (the selection operator). As all functions, operators take arguments, and return a value; *assignment* operators also have side effects: besides returning a value, they change the value of some variable.

Each operator has a fixed and unchangeable priority, which means that, in a given expression, the operations with the highest priority is performed first. Unless mentioned otherwise, operators at the same priority level are left-associative (performed from left to right), unless they are assignments, in which case they are right-associative. Anything enclosed between parenthesis is considered a complete subexpression, and is resolved recursively, independently of the surrounding context. For instance,

```
a + b + c     -->    (a + b) + c      \\ left-associative
a = b = c     -->    a = (b = c)      \\ right-associative
```

Assuming that op_1, op_2, op_3 are binary operators with increasing priorities (think of `+`, `*`, `^`),

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

is equivalent to

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

GP contains many different operators, either unary (having only one argument) or binary, plus a few special selection operators. Unary operators are defined as either *prefix* or *postfix*, meaning that they respectively precede (*op* x) and follow (x *op*) their single argument. Some symbols are syntactically correct in both positions, like `!`, but then represent different operators: the `!` symbol represents the negation and factorial operators when in prefix and postfix position respectively. Binary operators all use the (infix) syntax x *op* y.

Most operators are standard (`+`, `%`, `=`), some are borrowed from the C language (`++`, `<<`), and a few are specific to GP (`\`, `#`). Beware that some GP operators 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 list of available operators, ordered by decreasing priority, binary and left-associative unless mentioned otherwise. An expression is an *lvalue* if something can be assigned to it. (The name comes from left-value, to the left of a `=` operator; e.g. `x`, or `v[1]` are lvalues, but `x + 1` is not.)

• Priority 14

`:` as in `x:small`, is used to indicate to the GP2C compiler that the variable on the left-hand side always contains objects of the type specified on the right hand-side (here, a small integer) in order to produce more efficient or more readable C code. This is ignored by GP.

• Priority 13

`( )` is the function call operator. If f is a closure and *args* is a comma-separated list of arguments (possibly empty), f(*args*) evaluates f on those arguments.

• Priority 12

`++` and `--` (unary, postfix): if x is an `lvalue`, 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 $x - 1$. These operators are not associative, i.e. `x++++` is invalid, since `x++` is not an lvalue.

• Priority 11

`.`*member* (unary, postfix): x`.`*member* extracts *member* from structure x (see Section 2.8).

`[ ]` 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 10

`'` (unary, postfix): derivative with respect to the main variable. If f is a function (`t_CLOSURE`), f' is allowed and defines a new function, which will perform numerical derivation when evaluated at a scalar x; this is defined as $(f(x + \varepsilon) - f(x - \varepsilon))/2\varepsilon$ for a suitably small epsilon depending on current precision.

```
? (x^2 + y*x + y^2)'  \\ derivation with respect to main variable x
%1 = 2*x + y
? SIN = cos'
%2 = cos'
```

```
? SIN(Pi/6)          \\ numerical derivation
%3 = -0.50000000000000000000000000000
? cos'(Pi/6)         \\ works directly: no need for the intermediate SIN
%4 = -0.50000000000000000000000000000
```

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

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

! (unary, prefix): logical *not*. !x returns 1 if x is equal to 0 (specifically, if `gequal0(`x`)==1`), and 0 otherwise.

• Priority 9

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

• Priority 8

^: powering. This operator is right associative: `2 ^3^4` is understood as `2 ^(3^4)`.

• Priority 7

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

• Priority 6

*: multiplication.

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

\, %: Euclidean quotient and remainder, i.e. if $x = qy + r$, then `x\y` $= q$, `x%y` $= r$. If x and y are scalars, then q is an integer and r satisfies $0 \leq r < y$; if x and y are polynomials, then q and r are polynomials such that $\deg r < \deg y$ and the leading terms of r and x have the same sign.

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

<<, >>: left and right binary shift. By definition, `x<<n` $= x * 2^n$ if $n > 0$, and `truncate`$(x2^{-n})$ otherwise. Right shift is defined by `x>>n` = `x<<(-n)`.

• Priority 5

+, -: addition/subtraction.

• Priority 4

<, >, <=, >=: 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. `t_QFR` having the same coefficients but a different distance component are tested as equal.

===: test whether two objects are identical component-wise. This is stricter than == : for instance, the integer 0, a 0 polynomial or a vector with 0 entries, are all tested equal by ==, but they are not identical.

• Priority 3

&, &&: 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 == 0 || test(1/x)
```

will never produce an error since `test(1/x)` is not even evaluated when the first test is true (hence the final truth value is true). Similarly

```
type(p) == "t_INT" && isprime(p)
```

does not evaluate `isprime(p)` if `p` is not an integer.

• Priority 2

`=` (assignment, *lvalue* `=` *expr*). The result of `x = y` is the value of the expression y, which is also assigned to the variable `x`. This assignment operator is right-associative. 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 equality test would be `x == 1`. The right hand side of the assignment operator is evaluated before the left hand side.

It is crucial that the left hand-side be an *lvalue* there, it avoids ambiguities in expressions like `1 + x = 1`. The latter evaluates as `1 + (x = 1)`, not as `(1 + x) = 1`, even though the priority of `=` is lower than the priority of `+` : `1 + x` is not an lvalue.

If the expression cannot be parsed in a way where the left hand side is an lvalue, raise an error.

```
? x + 1 = 1
  ***   unused characters: x+1=1
  ***                          ^--
```

op`=`, where *op* is any binary operator among `+`, `-`, `*`, `%`, `/`, `\`, `\/`, `<<`, or `>>` (composed assignment *lvalue op*`=` *expr*). The expression `x` *op*`=` y assigns (`x` *op* y) to `x`, and returns the new value of `x`. The result is *not* an *lvalue*; thus

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

is invalid. These assignment operators are right-associative:

```
? x = 'x; x += x *= 2
%1 = 3*x
```

• Priority 0

`->` (function definition): (*vars*)`->`*expr* returns a function object, of type `t_CLOSURE`.

Remark. Use the *op*`=` operators as often as possible since they make complex assignments more legible: one needs not parse complicated expressions twice to make sure they are indeed identical. Compare

```
v[i+j-1] = v[i+j-1] + 1    -->    v[i+j-1]++
M[i,i+j] = M[i,i+j] * 2    -->    M[i,i+j] *= 2
```

Remark. Less important but still interesting. The `++`, `--` and *op*`=` operators are slightly more efficient:

```
? a = 10^6;
? i = 0; while(i<a, i=i+1)
time = 365 ms.
? i = 0; while(i<a, i++)
time = 352ms.
```

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

```
? a = 1<<(10^5);
? i = 1; while(i<a, i=i*2);
time = 1,052 ms.
? i = 1; while(i<a, i<<=1);
time = 617 ms.
```

2.5 Variables and symbolic expressions.

In this section we use *variable* in the standard mathematical sense, symbols representing algebraically independent elements used to build rings of polynomials and power series, and explain the all-important concept of *variable priority*. In the next Section 2.6, we shall no longer consider only free variables, but adopt the viewpoint of computer programming and assign values to these symbols: (bound) variables are names associated to values in a given scope.

2.5.1 Variable names A valid name starts with a letter, followed by any number of keyword characters: `_` or alphanumeric characters ([`A-Za-z0-9`]). The built-in function names are reserved and cannot be used; see the list with `\c`, including the constants `Pi`, `Euler` and `I` $= \sqrt{-1}$.

GP names are case sensitive. For instance, the symbol `i` is perfectly safe to use, and will not be mistaken for `I` $= \sqrt{-1}$; analogously, `o` is not synonymous to `0`.

In GP you can use up to 16383 variable names (up to 65535 on 64-bit machines). If you ever need thousands of variables and this becomes a serious limitation, you should probably be using vectors instead: e.g. instead of variables `X1`, `X2`, `X3`, ..., you might equally well store their values in `X[1]`, `X[2]`, `X[3]`, ...

2.5.2 Variables and polynomials What happens when you use a valid variable name,`t` say, for the first time *before* assigning a value into it? This registers a new *free variable* with the interpreter (which will be written as `t`), and evaluates to a monomial of degree 1 in the said variable `t`. It is important to understand that PARI/GP is *not* a symbolic manipulation package: even free variables already have default values*, there is no such thing as an "unbound" variable in GP. You have access to this default value using the quote operator: `'t` always evaluates to the above monomial of degree 1, independently of assignments made since then (e.g. `t = 1`).

```
? t^2 + 1
%1 = t^2 + 1
? t = 2; t^2 + 1
```

* More generally, any expression has a value, and is *replaced* by its value as soon as it is read; it never stays in an abstract form.

```
%2 = 5
? %1
%3 = t^2 + 1
? eval(%1)
%4 = 5
```

In the above, `t` is initially a free variable, later bound to 2. We see that assigning a value to a variable does not affect previous expressions involving it; to take into account the new variable's value, one must force a new evaluation, using the function `eval` (see Section 3.7.4). It is preferable to leave alone your "polynomial variables", never assigning values to them, and to use `subst` and its more powerful variants rather than `eval`. You will avoid the following kind of problems:

```
? p = t^2 + 1; subst(p, t, 2)
%1 = 5
? t = 2;
? subst(p, t, 3)     \\ t is no longer free: it evaluates to 2
  ***   at top-level: subst(p,t,3)
  ***                           ^----
  ***   variable name expected.
? subst(p, 't, 3)   \\ OK
%3 = 10
```

A statement like `x = 'x` in effect restores `x` as a free variable.

2.5.3 Variable priorities, multivariate objects 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. (PARI currently has no sparse representation for polynomials, listing only non-zero monomials.) All computations are then 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

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

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 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 `variable()`.

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 `libpari` programming and low-level constructors) 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.

* This is not strictly true: the variable x is predefined and always has the highest possible priority.

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;
```

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 become 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
```

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.

2.5.4 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 Variables and Scope.

This section is rather technical, and strives to explain potentially confusing concepts. Skip to the last subsection for practical advice, if the next discussion does not make sense to you. After learning about user functions, study the example in Section 2.7.3 then come back.

Definitions.

A *scope* is an enclosing context where names and values are associated. A user's function body, the body of a loop, an individual command line, all define scopes; the whole program defines the *global* scope. The argument of `eval` is evaluated in the enclosing scope.

Variables are bound to values within a given scope. This is traditionally implemented in two different ways:

• lexical (or static) scoping: the binding makes sense within a given block of program text. The value is private to the block and may not be accessed from outside. Where to find the value is determined at compile time.

• dynamic scoping: introducing a local variable, say `x`, pushes a new value on a stack associated to the name `x` (possibly empty at this point), which is popped out when the control flow leaves the scope. Evaluating `x` in any context, possibly outside of the given block, always yields the top value on this dynamic stack.

GP implements both lexical and dynamic scoping, using the keywords* `my` (lexical) and `local` (dynamic):

```
x = 0;
f() = x
g() =    my(x = 1); f()
h() = local(x = 1); f()
```

The function `g` returns 0 since the global `x` binding is unaffected by the introduction of a private variable of the same name in `g`. On the other hand, `h` returns 1; when it calls `f()`, the binding stack for the `x` identifier contains two items: the global binding to 0, and the binding to 1 introduced in `h`, which is still present on the stack since the control flow has not left `h` yet.

2.6.1 Scoping rules

Named parameters in a function definition, as well as all loop indices*, have lexical scope within the function body and the loop body respectively.

```
p = 0;
forprime (p = 2, 11, print(p)); p   \\ prints 0 at the end
```

```
x = 0;
f(x) = x++;
f(1)  \\ returns 2, and leave global x unaffected (= 0)
```

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));
```

* The names are borrowed from the `Perl` scripting language.

* More generally, in all iterative constructs which use a variable name (`for`, `prod`, `sum`, `vector`, `matrix`, `plot`, etc.) the given variable is lexically scoped to the construct's body.

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

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

But it is usually more natural to wrap the loop in a user function and use `return` instead of `break`:

```
try() =
{
  for(i = 1, n,
    if (ok(i), return (i));
  );
  0 \\ failure
}
```

A list of variables can be lexically or dynamically scoped (to the block between the declaration and the end of the innermost enclosing scope) using a `my` or `local` declaration:

```
for (i = 1, 10,
  my(x, y, z, i2 = i^2); \\ temps needed within the loop body
  ...
)
```

Note how the declaration can include (optional) initial values, `i2 = i^2` in the above. Variables for which no explicit default value is given in the declaration are initialized to 0. It would be more natural to initialize them to free variables, but this would break backward compatibility. To obtain this behavior, you may explicitly use the quoting operator:

```
my(x = 'x, y = 'y, z = 'z);
```

A more complicated example:

```
for (i = 1, 3,
  print("main loop");
  my(x = i);          \\ local to the outermost loop
  for (j = 1, 3,
    my (y = x^2);     \\ local to the innermost loop
    print (y + y^2);
    x++;
  )
)
```

When we leave the loops, the values of `x`, `y`, `i`, `j` are the same as before they were started.

Note that `eval` is evaluated in the given scope, and can access values of lexical variables:

```
? x = 1;
? my(x = 0); eval("x")
%2 = 0     \\ we see the local x scoped to this command line, not the global one
```

Variables dynamically scoped using `local` 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 practice, you almost certainly want true private variables, hence should use almost exclusively `my`.

We strongly recommended to explicitly scope (lexically) all variables to the smallest possible block. Should you forget this, in expressions involving such "rogue" variables, the value used will be the one which happens to be on top of the value stack at the time of the call; which depends on the whole calling context in a non-trivial way. This is in general *not* what you want.

2.7 User defined functions.

The most important thing to understand about user-defined functions is that they are ordinary GP objects, bound to variables just like any other object. Those variables are subject to scoping rules as any other: while you can define all your functions in global scope, it is usually possible and cleaner to lexically scope your private helper functions to the block of text where they will be needed.

Whenever gp meets a construction of the form `expr(`*argument list*`)` and the expression `expr` evaluates to a function (an object of type `t_CLOSURE`), the function is called with the proper arguments. For instance, constructions like `funcs[i](x)` are perfectly valid, assuming `funcs` is an array of functions.

2.7.1 Defining a function

A user function is defined as follows:

`(`*list of formal variables*`) -> `*seq*.

The list of formal variables is a comma-separated list of *distinct* variable names and allowed to be empty. It there is a single formal variable, the parentheses are optional. This list corresponds to the list of parameters you will supply to your function when calling it.

In most cases you want to assign a function to a variable immediately, as in

```
R = (x,y) -> sqrt( x^2+y^2 );
sq = x -> x^2;  \\ or equivalently (x) -> x^2
```

but it is quite possible to define (a priori short-lived) anonymous functions. The trailing semicolon is not part of the definition, but as usual prevents `gp` from printing the result of the evaluation, i.e. the function object. The construction

`f(`*list of formal variables*`) = `*seq*

is available as an alias for

`f = (`*list of formal variables*`) -> `*seq*

Using that syntax, it is not possible to define anonymous functions (obviously), and the above two examples become:

```
R(x,y) = sqrt( x^2+y^2 );
sq(x) = x^2;
```

The semicolon serves the same purpose as above: preventing the printing of the resulting function object; compare

```
? sq(x) = x^2;  \\ no output
```

```
? sq(x) = x^2    \\ print the result: a function object
%2 = (x)->x^2
```

Of course, the sequence *seq* can be arbitrarily complicated, in which case it will look better written on consecutive lines, with properly scoped variables:

```
{
f(x_0, x_1, ...) =
  my(t_0, t_1, ...); \\ variables lexically scoped to the function body
  ...
}
```

Note that the following variant would also work:

```
f(x_0, x_1, ...) =
{
  my(t_0, t_1, ...); \\ variables lexically scoped to the function body
  ...
}
```

(the first newline is disregarded due to the preceding `=` sign, and the others because of the enclosing braces). The `my` statements can actually occur anywhere within the function body, scoping the variables to more restricted blocks than the whole function body.

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 scope. In fact, a *copy* of the actual parameter is assigned to the formal parameter when the function is called. Formal parameters are lexically scoped to the function body. It is not allowed to use the same variable name for different parameters of your function:

```
? f(x,x) = 1
  ***   variable declared twice: f(x,x)=1
  ***                                 ^----
```

Finishing touch. You can add a specific help message for your function using `addhelp`, but the online help system already handles it. By default `?`*name* will print the definition of the function *name*: the list of arguments, as well as their default values, the text of *seq* as you input it. Just as `\c` prints the list of all built-in commands, `\u` outputs the list of all user-defined functions.

Backward compatibility (lexical scope). Lexically scoped variables were introduced in version 2.4.2. Before that, the formal parameters were dynamically scoped. If your script depends on this behavior, you may use the following trick: replace the initial `f(x) =` by

```
f(x_orig) = local(x = x_orig)
```

Backward compatibility (disjoint namespaces). Before version 2.4.2, variables and functions lived in disjoint namespaces and it was not possible to have a variable and a function share the same name. Hence the need for a `kill` function allowing to reuse symbols. This is no longer the case.

There is now no distinction between variable and function names: we have PARI objects (functions of type `t_CLOSURE`, or more mundane mathematical entities, like `t_INT`, etc.) and variables bound to them. There is nothing wrong with the following sequence of assignments:

```
? f = 1        \\ assigns the integer 1 to f
%1 = 1;
? f() = 1      \\ a function with a constant value
%2 = ()->1
? f = x^2      \\ f now holds a polynomial
%3 = x^2
? f(x) = x^2   \\ ... and now a polynomial function
%4 = (x)->x^2
? g(fun) = fun(Pi);\\ a function taking a function as argument
? g(cos)
%6 = -1.0000000000000000000000000000
```

Previously used names can be recycled as above: you are just redefining the variable. The previous definition is lost of course.

Important technical note. Built-in functions are a special case since they are read-only (you cannot overwrite their default meaning), and they use features not available to user functions, in particular pointer arguments. In the present version 2.5.5, it is possible to assign a built-in function to a variable, or to use a built-in function name to create an anonymous function, but some special argument combinations may not be available:

```
? issquare(9, &e)
%1 = 1
? e
%2 = 3
? g = issquare;
? g(9)
%4 = 1
? g(9, &e)  \\ pointers are not implemented for user functions
  ***   unexpected &: g(9,&e)
  ***                      ^---
```

2.7.2 Function call, Default arguments

You may now call your function, as in `f(1,2)`, supplying values for the formal variables. The number of parameters actually supplied may be *less* than the number of formal variables in the function definition. An uninitialized formal variable is given an implicit default value of (the integer) 0, i.e. after the definition

```
f(x, y) = ...
```

you may call `f(1, 2)`, supplying values for the two formal parameters, or for example
`f(2)` equivalent to `f(2,0)`,

`f()`	`f(0,0)`,
`f(,3)`	`f(0,3)`. ("Empty argument" trick)

More generally, the argument list is filled with user supplied values, in order. A comma or closing parenthesis, where a value should have been, signals we must use a default value. When no input arguments are left, the defaults are used instead to fill in remaining formal parameters.

Of course, you can change these default values to something more useful than 0. In the function definition, you can append `=`*expr* to a formal parameter, to give that variable an explicit default value. The expression gets evaluated the moment the function is called, and may involve the preceding function parameters: a default value for x_i may involve x_j for $j < i$. For instance, after

```
f(x = 1, y = 2, z = y+1) = ....
```

typing in `f(3,4)` would give you `f(3,4,5)`. In the rare case when you want to set some far away argument, and leave the defaults in between as they stand, use the "empty argument" trick: `f(6,,1)` would yield `f(6,2,1)`. Of course, `f()` by itself yields `f(1,2,3)` as was to be expected.

More specifically,

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

defines a function which prints its arguments (at most three of them), separated by colons.

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

Example. We conclude with an amusing example, intended to illustrate both user-defined functions and the power of the `sumalt` function. Although the Riemann zeta-function is included (as `zeta`) among the standard functions, let us assume that we want to check other implementations. Since we are highly interested in the critical strip, we use the classical formula

$$(2^{1-s}-1)\zeta(s) = \sum_{n\geq 1}(-1)^n n^{-s}, \qquad \Re s > 0.$$

The implementation is obvious:

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

Note that `n` is automatically lexically scoped to the `sumalt` "loop", so that it is unnecessary to add a `my(n)` declaration to the function body. Surprisingly, this gives very good accuracy in a larger region than expected:

```
? check = z -> ZETA(z) / zeta(z);
? check(2)
%1 = 1.0000000000000000000000000000
? check(200)
%2 = 1.0000000000000000000000000000
? check(0)
%3 = 0.99999999999999999999999999994
```

```
? check(-5)
%4 = 1.0000000000000000007549266557
? check(-11)
%5 = 0.99997526410478249026608477 45
? check(1/2+14.134*I)  \\ very close to a non-trivial zero
%6 = 1.0000000000000000000003747432 + 7.62329066 E-21*I
? check(-1+10*I)
%7 = 1.0000000000000000000000002511 + 2.989950968 E-24*I
```

Now wait a minute; not only are we summing a series which is certainly no longer alternating (it has complex coefficients), but we are also way outside of the region of convergence, and still get decent results! No programming mistake this time: `sumalt` is a "magic" function*, providing very good convergence acceleration; in effect, we are computing the analytic continuation of our original function. To convince ourselves that `sumalt` is a non-trivial implementation, let us try a simpler example:

```
? sum(n=1, 10^7, (-1)^n/n, 0.) / (-log(2)) \\ approximates the well-known formula
time = 7,417 ms.
%1 = 0.99999992786525156228934054 57
? sumalt(n=1, (-1)^n/n) / (-log(2))        \\ accurate and fast
time = 0 ms.
%2 = 1.0000000000000000000000000000
```

No, we are not using a powerful simplification tool here, only numerical computations. Remember, PARI is not a computer algebra system!

2.7.3 Beware scopes! Be extra careful with the scopes of variables. What is wrong with the following definition?

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

Hint. The function body is equivalent to

```
{ my(newp = 0);
  forprime(p=2, x, if (x%p == 0, break));
  newp
}
```

* `sumalt` is heuristic, but its use can be rigorously justified for a given function, in particular our $\zeta(s)$ formula. Indeed, Peter Borwein (*An efficient algorithm for the Riemann zeta function*, CMS Conf. Proc. **27** (2000), pp. 29–34) proved that the formula used in `sumalt` with n terms computes $(1-2^{1-s})\zeta(s)$ with a relative error of the order of $(3+\sqrt{8})^{-n}|\Gamma(s)|^{-1}$.

Detailed explanation. The index `p` in the `forprime` loop is lexically scoped to the loop and is not visible to the outside world. Hence, it will not survive the `break` statement. More precisely, at this point the loop index is restored to its preceding value. The initial `my(p)`, although well-meant, adds to the confusion: it indeed scopes `p` to the function body, with initial value 0, but the `forprime` loop introduces *another* variable, unfortunately also called `p`, scoped to the loop body, which shadows the one we wanted. So we always return 0, since the value of the `p` scoped to the function body never changes and is initially 0.

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)))
```

2.7.4 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) = dive(n+1)
? dive(0);
  ***   [...] at: dive(n+1)
  ***             ^---------
  ***   in function dive: dive(n+1)
  ***                     ^---------
  \\ (last 2 lines repeated 19 times)
  ***   deep recursion.
```

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`).

2.7.5 Function which take functions as parameters ? Very easy:

```
? calc(f, x) = f(x)
? calc(sin, Pi)
%2 = -5.04870979 E-29
? g(x) = x^2;
? calc(g, 3)
%4 = 9
```

If we do not need `g` elsewhere, we should use an anonymous function here, `calc(x->x^2, 3)`. Here is a variation:

```
? funs = [cos, sin, tan, x->x^3+1]; \\ an array of functions
? call(i, x) = funs[i](x)
```

evaluates the appropriate function on argument `x`, provided $1 \leq i \leq 4$. Finally, a more useful example:

```
APPLY(f, v) = vector(#v, i, f(v[i]))
```

applies the function `f` to every element in the vector `v`. (The built-in function `apply` is more powerful since it also applies to lists and matrices.)

2.7.6 Defining functions within a function ? Defining a single function is easy:

```
init(x) = (add = y -> x+y);
```

Basically, we are defining a global variable `add` whose value is the function `y->x+y`. The parentheses were added for clarity and are not mandatory.

```
? init(5);
? add(2)
%2 = 7
```

A more refined approach is to avoid global variables and *return* the function:

```
init(x) = y -> x+y
add = init(5)
```

Then `add(2)` still returns 7, as expected! Of course, if `add` is in global scope, there is no gain, but we can lexically scope it to the place where it is useful:

```
  my ( add = init(5) );
```

How about multiple functions then ? We can use the last idea and return a vector of functions, but if we insist on global variables ? 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` (itself defining a function `mul`). The way out is to use parentheses for grouping, so that enclosed subexpressions will be evaluated independently:

```
? init(x) = ( add(y) = x+y ); ( mul(y) = x*y );
? init(5);
? add(2)
%3 = 7
```

```
? mul(3)
%4 = 15
```

This defines two global functions which have access to the lexical variables private to `init`! The following would work in exactly the same way:

```
? init5() = my(x = 5); ( add(y) = x+y ); ( mul(y) = x*y );
```

2.7.7 Closures as Objects? Contrary to what you might think after the preceding examples, GP's closures may not be used to simulate true "objects", with private and public parts and methods to access and manipulate them. In fact, closures indeed incorporate an existing context (they may access lexical variables that existed at the time of their definition), but then may not change it. More precisely, they access a copy, which they are welcome to change, but a further function call still accesses the original context, as it existed at the time the function was defined:

```
init() =
{ my(count = 0);
  inc()=count++;
  dec()=count--;
}
? inc()
%1 = 1
? inc()
%2 = 1
? inc()
%3 = 1
```

2.8 Member functions.

Member functions use the 'dot' notation to retrieve information from complicated structures. The built-in structures are *bid*, *ell*, *galois*, *ff*, *nf*, *bnf*, *bnr* and *prid*, which will be described at length in Chapter 3. The syntax `structure.member` is taken to mean: retrieve `member` from `structure`, e.g. `E.j` returns the j-invariant of the elliptic curve `E`, or outputs an error message if `E` is not a proper *ell* structure. To define your own member functions, use the syntax

var`.`*member* `=` *seq*,

where the formal variable *var* is scoped to the function body *seq*. This is of course reminiscent of a user function with a single formal variable *var*. For instance, the current implementation of the `ell` type is a vector, the j-invariant being the thirteenth component. It could be implemented as

```
x.j =
{
  if (type(x) != "t_VEC" || #x < 14, error("not an elliptic curve: " x));
  x[13]
}
```

As for user functions, you can redefine your member functions simply by typing new definitions. On the other hand, as a safety measure, you cannot redefine the built-in member functions, so attempting to redefine `x.j` as above would in fact produce an error; you would have to call it e.g. `x.myj` in order for `gp` to accept it.

Rationale. In most cases, member functions are simple accessors of the form

```
x.a = x[1];
x.b = x[2];
x.c = x[3];
```

where `x` is a vector containing relevant data. There are at least three alternative approaches to the above member functions: 1) hardcode `x[1]`, etc. in the program text, 2) define constant global variables `AINDEX = 1`, `BINDEX = 2` and hardcode `x[AINDEX]`, 3) user functions `a(x) = x[1]` and so on.

Even if 2) improves on 1), these solutions are neither elegant nor flexible, and they scale badly. 3) is a genuine possibility, but the main advantage of member functions is that their namespace is independent from the variables (and functions) namespace, hence we can use very short identifiers without risk. The j-invariant is a good example: it would clearly not be a good idea to define `j(E) = E[13]`, because clashes with loop indices are likely.

Note. Typing `\um` will output all user-defined member functions.

Member function names. A valid name starts with a letter followed by any number of keyword characters: _ or alphanumeric characters ([`A-Za-z0-9`]). The built-in member function names are reserved and cannot be used (see the list with `?.`). Finally, names starting with `e` or `E` followed by a digit are forbidden, due to a clash with the floating point exponent notation : we understand `1.e2` as 100.000..., not as extracting member `e2` of object `1`.

2.9 Strings and Keywords.

2.9.1 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).

2.9.2 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)
`whatnow`

2.9.3 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)))
```

2.10 Errors and error recovery.

2.10.1 Errors Your input program is first compiled to a more efficient bytecode; then the latter is evaluated, calling appropriate functions from the PARI library. Accordingly, there are two kind of errors: syntax errors occurring during the compilation phase, and runtime 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 output its usual prompt.

When reporting a *syntax error*, `gp` gives meaningful context by copying (part of) the expression it was trying to compile, indicating where the error with a little caret, as in

```
? factor()
  ***   too few arguments: factor()
  ***                              ^-
```

```
? 1+
  ***   syntax error, unexpected $end: 1+
  ***                                    ^-
```

possibly enlarged to a full arrow given enough trailing context

```
? if (isprime(1+, do_something())
  ***   syntax error, unexpected ',': if(isprime(1+,do_something()))
  ***                                                ^----------------
```

These error messages may be mysterious, because gp cannot guess what you were trying to do, and the error usually occurs once gp has been sidetracked. The first error is straightforward: `factor` has one mandatory argument, which is missing there.

The next two are simple typos involving an ill-formed addition `1 +` missing its second operand. The error messages differ because the parsing context is slightly different: in the first case we reach the end of input (`$end`) while still expecting a token, and in the second one, we received an unexpected token (the comma).

Here is a more complicated one:

```
? factor(x
  ***   syntax error, unexpected $end, expecting )-> or ',' or ')': factor(x
  ***                                                                    ^-
```

The 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 useful, see the description of the function). In fact gp expected either a closing parenthesis, or a second argument separated from the first by a comma. And this is essentially what the error message says: we reached the end of the input (`$end`) while expecting a `')'` or a `','`.

Actually, a third possibility is mentioned in the error message `)->`, which could never be valid in the above context, but a subexpression like `(x)->sin(x)`, defining an inline closure would be valid, and the parser is not clever enough to rule that out, so we get the same message as in

```
? (x
  ***   syntax error, unexpected $end, expecting )-> or ',' or ')': (x
  ***                                                               ^-
```

where all three proposed continuations would be valid.

Runtime errors from the evaluator are nicer because they answer a correctly worded query, otherwise the bytecode compiler would have protested first; here is a slightly pathological case:

```
? if (siN(x) < eps, do_something())
  ***   at top-level: if(siN(x)<eps,do_someth
  ***                    ^--------------------
  ***   not a function: 'siN'.
```

(no arrow!) The code is syntactically correct and compiled correctly, even though the `siN` function, a typo for `sin`, was not defined at this point. When trying to evaluate the bytecode, however, it turned out that `siN` is still undefined so we cannot evaluate the function call `siN(x)`.

Library runtime errors are even nicer because they have more mathematical content, which is easier to grasp than a parser's logic:

```
? 1/0
```

```
  ***   at top-level: 1/0
  ***                  ^--
  *** _/_: division by zero
```

telling us that a runtime error occurred while evaluating the binary `/` operator (the _ surrounding the operator are placeholders). More context is provided if the error occurs deep in the call chain:

```
? f(x) = 1/x;
? g(N) = for(i = -N, N, f(i));
? g(10)
  ***   at top-level: g(10)
  ***                 ^-----
  ***   in function g: for(i=-N,N,f(i))
  ***                               ^-----
  ***   in function f: 1/x
  ***                   ^--
  *** _/_: division by zero
```

2.10.2 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 obvious; 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 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.

2.10.3 Break loop

A *break loop* is a special debugging mode that you enter whenever a user interrupt (`Control-C`) or runtime error occurs, freezing the `gp` state, and preventing cleanup until you get out of the loop. By runtime error, we mean an error from the library or a user error (from `error`), but *not* syntax errors. 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 do *not* terminate the loop).

To get out of a break loop, you can use `next`, `break`, `return`, or type `C-d` (`EOF`), any of which will let `gp` perform its usual cleanup, and send you back to the `gp` prompt. Note that `C-d` is slightly dangerous, since typing it *twice* will not only send you back to the `gp` prompt, but to your shell prompt ! (Since `C-d` at the `gp` prompt exits the gp session.)

If the break loop was started by a user interrupt (and not by an error), inputting an empty line, i.e hitting the `<Return>` key at the `break>` prompt, resumes the temporarily interrupted computation. A single empty line has no effect in case of a fatal error, to avoid getting get out of the loop prematurely, thereby losing valuable debugging data. Any of `next`, `break`, `return`, or `C-d` will abort the computation and send you back to the `gp` prompt as above.

Break loops are useful as a debugging tool. You may inspect the values of `gp` variables to understand why an error occurred, or change `gp`'s state in the middle of a computation (increase debugging level, start storing results in a log file, 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:

```
? v = 0; 1/v
  ***   at top-level: v=0;1/v
  ***                       ^--
  *** _/_: division by zero
  ***   Break loop (type 'break' or Control-d to go back to GP)
break>
```

So the standard error message is printed first. The `break>` at the bottom 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.

Even lexically-scoped variables are accessible to the evaluator during the break loop:

```
? for(v = -2, 2, print(1/v))
-1/2
-1
  ***   at top-level: for(v=-2,2,print(1/v))
  ***                                   ^----
  *** _/_: division by zero
  ***   Break loop (type 'break' or Control-d to go back to GP)
break> v
0
```

Even though loop indices are automatically lexically scoped and no longer exist when the break loop is run, enough debugging information is retained in the bytecode to reconstruct the evaluation context.

Note. If you do not like this behavior, you may disable it by setting the default `breakloop` to 0 in for `gprc`. A runtime error will send you back to the prompt. Note that the break loop is automatically disabled when running `gp` in non interactive mode, i.e. when the program's standard input is not attached to a terminal.

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.

2.10.4 Protecting code Finally `trap` can define a temporary handler used within the scope of a code fragment, 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.

2.10.5 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, see the documentation of `trap` for the complete list. For instance, we have

`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.

2.11 Interfacing GP with other languages.

The PARI library was meant to be interfaced with C programs. This specific use is dealt with extensively in the *User's guide to the PARI library*. Of course, `gp` itself provides a convenient interpreter 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 if speed is paramount. The GP2C compiler often eases this part.

The `install` command (see Section 3.12.17) efficiently imports foreign functions for use under `gp`, which can of course be written using other libraries than PARI. Thus you may code only critical parts of your program in C, and still maintain most of the program as a GP script.

We are aware of three PARI-related Free Software packages to embed PARI in other languages. We *neither endorse nor support* any of them, but you may want to give them a try if you are familiar with the languages they are based on. The first is William Stein's Python-based SAGE* system. The second is the `Math::Pari` Perl module (see any CPAN mirror), written by Ilya Zakharevich. 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 `python`, `perl`, or `Lisp` programs, respectively.

2.12 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 2.14), 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 3.12.7. The basic syntax is

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

which sets the default *def* to *value*. In interactive use, most of these can be abbreviated using `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 factory settings using command-line switches or a `gprc`.

* see `http://sagemath.org/`

*** see `http://clisp.cons.org/`

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 `parisize` which typically involve a lot of trailing zeroes.

(somewhat technical) Note. As we saw in Section 2.9, the second argument to `default` is subject to string context expansion, which means you can use run-time values. In other words, something like

```
a = 3;
default(logfile, "file" a ".log")
```

logs the output in `file3.log`.

Some special defaults, corresponding to file names and prompts, expand further the resulting value at the time they are set. Two kinds of expansions may be 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,

```
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 file names.

Available defaults are described in the reference guide, Section 3.13.

2.13 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 file name whose first character is a digit, and with `\r` or `\w`, if the file name itself contains a space. In such cases, just use the underlying `read` or `write` function; see Section 3.12.34).

2.13.1 `?` {*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 functionality:

`??` *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. This functionality is 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`.

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

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

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

2.13.5 `\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`.

2.13.6 `\d`: prints the defaults as described in the previous section (shortcut for `default()`, see Section 3.12.7).

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

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

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

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

2.13.11 `\h` $\{m\text{-}n\}$: 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.

2.13.12 `\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.

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

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

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

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

2.13.17 `\q`: quits the `gp` session and returns to the system. Shortcut for `quit()` (see Section 3.12.23).

2.13.18 `\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 3.12.24); 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 3.12.36) 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.

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

2.13.20 `\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.

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

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

2.13.23 `\v`: prints the version number and implementation architecture (680x0, Sparc, Alpha, other) of the `gp` executable you are using.

2.13.24 `\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 file names).

2.13.25 `\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.

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

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

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

2.14 The preferences file.

This file, called `gprc` in the sequel, is used to modify or extend `gp` default behavior, 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.

2.14.1 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).

2.14.1.1 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 behavior as `=` under `gp`). Only three tests can be performed:

`EMACS`: `true` if `gp` is running in an Emacs or TeXmacs shell (see Section 2.16).

`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 version 1.0.0). This is `true` if `gp`'s version number satisfies the required inequality.

2.14.1.2 Commands:. After 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 2.12), 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";
}
```

2.14.2 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.

2.15 Using readline.

This very useful library provides line editing and contextual completion to `gp`. You are encouraged to read the `readline` user manual, but we describe basic usage here.

2.15.1 A (too) short introduction to readline: In the following, `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 at the `gp` prompt are stored in a history, a multiline command being saved as a single concatenated line. The Up and Down arrows (or `C-p`/`C-n`) will move you through the history, `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 behavior) 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 behavior 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 Hitting `<TAB>` will complete words for you. This mechanism is 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.

2.16 GNU Emacs and PariEmacs.

If you install the PariEmacs package (see Appendix A), you may use `gp` as a subprocess in Emacs. You then need to 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 the PariEmacs package and ensures that file with the `.gp` suffix are edited in gp-script mode.

Once this is done, under GNU Emacs if you type `M-x gp` (where as usual `M` is the `Meta` key), a special shell will be started launching `gp` with the default stack size and prime limit. You can then

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

work as usual under gp, but with all the facilities of an advanced text editor. See the PariEmacs documentation for customizations, menus, etc.

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 ever-expanding. Here is a description of the ones available in version 2.5.5. 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 `GEN foo(GEN x, long fl = 0)`.

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

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

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, instructs it to change its default behavior, 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 mentioned 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 ...
```

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). People used to the bit-or notation in languages like C may also use the form `"Parametric | no_Recursive"`.

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 2.5.5, 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 function seen in the present chapter has been implemented so as to accept a variable number of arguments, so all arguments are mandatory when programming with the library (usually, variants are provided corresponding to the various flag values). We include an `= default value` token in the prototype to signal how a missing argument should be encoded. Most of the time, it will be a `NULL` pointer, or -1 for a variable number. Refer to the *User's Guide to the PARI library* for general background and 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 `GEN gneg(GEN x)` for `-`x.

3.1.2 `+`, `-`: The expression x `+` y is the sum and x `-` y is the difference of x and y. Addition/subtraction between a scalar type x and a `t_COL` or `t_MAT` y returns respectively $[y[1] \pm x, y[2], \ldots]$ and $y \pm x\mathrm{Id}$. Other addition/subtraction between a scalar type and a vector or a matrix, or between vector/matrices of incompatible sizes are forbidden.

The library syntax is `GEN gadd(GEN x, GEN y)` for x `+` y, `GEN gsub(GEN x, GEN 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 a `t_INTMOD` or `t_PADIC` Restricted to scalars, `*` is commutative; because of vector and matrix operations, it is not commutative in general.

Multiplication between two `t_VEC`s or two `t_COL`s is not allowed; to take the scalar product of two vectors of the same length, transpose one of the vectors (using the operator `~` or the function `mattranspose`, see Section 3.8) and multiply a line vector by a column vector:

```
? a = [1,2,3];
? a * a
  ***   at top-level: a*a
  ***                  ^--
  *** _*_: forbidden multiplication t_VEC * t_VEC.
? a * a~
%2 = 14
```

If x, y are binary quadratic forms, compose them; see also `qfbnucomp` and `qfbnupow`. If x, y are `t_VECSMALL` of the same length, understand them as permutations and compose them.

The library syntax is `GEN gmul(GEN x, GEN y)` for x `*` y. Also available is `GEN gsqr(GEN x)` for x `*` x.

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 3.7.38).

The library syntax is `GEN gdiv(GEN x, GEN 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 `GEN gdivent(GEN x, GEN 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 `GEN gdivround(GEN x, GEN 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. When x or y is a non-integral real number, x`%`y is defined as x `-` `(`x`\`y`)*`y. Otherwise, if y is an integer, this is the smallest non-negative integer congruent to x modulo y. (This actually coincides with the previous definition if and only if x is an integer.) If y is a polynomial, this is the polynomial of smallest degree congruent to x modulo y. For instance:

```
? (1/2) % 3
%1 = 2
? 0.5 % 3
%2 = 0.50000000000000000000000000000
? (1/2) % 3.0
%3 = 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 `GEN gmod(GEN x, GEN y)` for x `%` y.

3.1.8 ^: 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)
  ***   at top-level: 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)
  ***   at top-level: 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);
  ***   at top-level: Mod(a,b)^(-1)
  ***                           ^-----
  *** _^_: impossible inverse modulo: Mod(0, 4).
```

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) =
{ my(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 `GEN gpow(GEN x, GEN n, long prec)` for $x\hat{}n$.

3.1.9 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; no GP operator corresponds to it:

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

The library syntax is `GEN divrem(GEN x, GEN y, long v = -1)`, where `v` is a variable number. Also available is `GEN gdiventres(GEN x, GEN y)` when v is not needed.

3.1.10 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 `GEN lexcmp(GEN x, GEN y)`.

3.1.11 max(x, y): creates the maximum of x and y when they can be compared.

The library syntax is `GEN gmax(GEN x, GEN y)`.

3.1.12 min(x, y): creates the minimum of x and y when they can be compared.

The library syntax is `GEN gmin(GEN x, GEN y)`.

3.1.13 shift(x, n): shifts x componentwise left by n bits if $n \geq 0$ and right by $|n|$ bits if $n < 0$. May be abbreviated as x `<<` n or x `>>` $(-n)$. 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 `GEN gshift(GEN x, long n)`.

3.1.14 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 `GEN gmul2n(GEN x, long n)`.

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 `GEN gsigne(GEN x)`.

3.1.16 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 `GEN vecmax(GEN x)`.

3.1.17 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 `GEN vecmin(GEN x)`.

3.1.18 Comparison and Boolean operators The six standard comparison operators `<=`, `<`, `>=`, `>`, `==`, `!=` are available in GP. The result is 1 if the comparison is true, 0 if it is false. The operator `==` is quite liberal : for instance, the integer 0, a 0 polynomial, and a vector with 0 entries are all tested equal.

The extra operator `===` tests whether two objects are identical and is much stricter than `==` : objects of different type or length are never identical.

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.

GP accepts the following synonyms for some of the above functions: `|` 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 `==`: it is the assignment statement. (We do not use the customary C operators for bitwise `and` or bitwise `or`; use `bitand` or `bitor`.)

The standard boolean operators `||` (inclusive or), `&&` (and) and `!` (not) are also available.

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 has a single component , 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),

• 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. In this last case, `Col` is the reciprocal function of `Pol` and `Ser` respectively.

Note that the function `Colrev` does not exist, use `Vecrev`.

The library syntax is `GEN gtocol(GEN x = NULL)`.

3.2.2 List($\{x = []\}$): transforms a (row or column) vector x into a list, whose components are the entries of x. Similarly for a list, but rather useless in this case. For other types, creates a list with the single element x. Note that, except when x is omitted, this function creates a small memory leak; so, either initialize all lists to the empty list, or use them sparingly.

The library syntax is `GEN gtolist(GEN x = NULL)`. The variant `GEN listcreate(void)` creates an empty list.

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 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. If x is a binary quadratic form, creates the associated 2×2 matrix. Otherwise, this creates a 1×1 matrix containing x.

```
? 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]
? Mat(Qfb(1,2,3))
%6 =
[1 1]

[1 3]
```

The library syntax is `GEN gtomat(GEN x = NULL)`.

3.2.4 Mod(x, y): 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.

The library syntax is `GEN gmodulo(GEN x, GEN 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 with non-negative valuation or a rational function, the effect is similar to `truncate`, i.e. we chop off the $O(X^k)$ or compute the Euclidean quotient of the numerator by the denominator, then change the main variable of the result to v.

The main use of this function is when x is a vector: it creates the polynomial whose coefficients are given by x, with $x[1]$ being the leading coefficient (which can be zero). It is much faster to evaluate `Pol` on a vector of coefficients in this way, than the corresponding formal expression $a_n X^n + \ldots + a_0$, which is evaluated naively exactly as written (linear versus quadratic time in n). `Polrev` can be used if one wants $x[1]$ to be the constant coefficient:

```
? Pol([1,2,3])
%1 = x^2 + 2*x + 3
? Polrev([1,2,3])
%2 = 3*x^2 + 2*x + 1
```

The reciprocal function of `Pol` (resp. `Polrev`) is `Vec` (resp. `Vecrev`).

```
? Vec(Pol([1,2,3]))
%1 = [1, 2, 3]
? Vecrev( Polrev([1,2,3]) )
%2 = [1, 2, 3]
```

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

```
? Pol(x + y, y)
  ***   at top-level: Pol(x+y,y)
  ***                 ^----------
  *** Pol: variable must have higher priority in gtopoly.
```

The library syntax is `GEN gtopoly(GEN x, long v = -1)`, 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`, i.e. it chops off the $O(X^k)$.

The main use of this function is when x is a vector: it creates the polynomial whose coefficients are given by x, with $x[1]$ being the constant term. `Pol` can be used if one wants $x[1]$ to be the leading coefficient:

```
? Polrev([1,2,3])
%1 = 3*x^2 + 2*x + 1
? Pol([1,2,3])
%2 = x^2 + 2*x + 3
```

The reciprocal function of `Pol` (resp. `Polrev`) is `Vec` (resp. `Vecrev`).

The library syntax is `GEN gtopolyrev(GEN x, long v = -1)`, 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 `GEN Qfb0(GEN a, GEN b, GEN c, GEN D = NULL, long prec)`. Also available are `GEN qfi(GEN a, GEN b, GEN c)` (assumes $b^2 - 4ac < 0$) and `GEN qfr(GEN a, GEN b, GEN c, GEN D)` (assumes $b^2 - 4ac > 0$).

3.2.8 Ser$(s, \{v = x\}, \{d = \textit{seriesprecision}\})$: transforms the object s into a power series with main variable v (x by default) and precision (number of significant terms) equal to d (= the default `seriesprecision` by default). If s is a scalar, this gives a constant power series with precision `d`. If s is a polynomial, the precision is the maximum of `d` and the degree of the polynomial. If s is a vector, the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (as in `Polrev`(x)), and the precision d is ignored.

```
? Ser(x^2,, 5)
%1 = x^2 + O(x^7)
? Ser([1,2,3], t)
%2 = 1 + 2*t + 3*t^2 + O(t^3)
```

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

The library syntax is `GEN gtoser(GEN s, long v = -1, long precdl)`, where `v` is a variable number.

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.

Note that most set functions also accept ordinary vectors, provided their components can be compared with `<`. Sets as created by this function are only useful when e.g. polynomial or vector entries are involved.

The library syntax is `GEN gtoset(GEN x = NULL)`.

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 2.9.

```
? 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"
```

The library syntax is `GEN Strchr(GEN x)`.

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 perform environment expansion, see Section 2.12. 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 2.9.

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 2.9.

3.2.14 Vec($\{x = []\}$): transforms the object x into a row vector. That vector has a single component, 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),

• 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. In this last case, `Vec` is the reciprocal function of `Pol` and `Ser` respectively.

The library syntax is `GEN gtovec(GEN x = NULL)`.

3.2.15 Vecrev($\{x = []\}$): as `Vec`(x), then reverse the result. In particular In this case, `Vecrev` is the reciprocal function of `Polrev`: the coefficients of the vector start with the constant coefficient of the polynomial and the others follow by increasing degree.

The library syntax is `GEN gtovecrev(GEN x = NULL)`.

3.2.16 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 `GEN gtovecsmall(GEN x = NULL)`.

3.2.17 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 `GEN binaire(GEN x)`.

3.2.18 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 `GEN gbitand(GEN x, GEN y)`. Also available is `GEN ibitand(GEN x, GEN y)`, which returns the bitwise *and* of $|x|$ and $|y|$, two integers.

3.2.19 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 3.2.18 for the behavior for negative arguments.

The library syntax is `GEN gbitneg(GEN x, long n)`.

3.2.20 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 3.2.18 for the behavior for negative arguments.

The library syntax is `GEN gbitnegimply(GEN x, GEN y)`. Also available is `GEN ibitnegimply(GEN x, GEN y)`, which returns the bitwise negated imply of $|x|$ and $|y|$, two integers.

3.2.21 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 3.2.18 for the behavior for negative arguments.

The library syntax is `GEN gbitor(GEN x, GEN y)`. Also available is `GEN ibitor(GEN x, GEN y)`, which returns the bitwise *ir* of $|x|$ and $|y|$, two integers.

3.2.22 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.

```
? bittest(7, 3)
%1 = 1 \\ the 3rd bit is 1
? bittest(7, 4)
%2 = 0 \\ the 4th bit is 0
```

See Section 3.2.18 for the behavior at negative arguments.

The library syntax is `GEN gbittest(GEN x, long n)`. For a `t_INT` x, the variant `long bittest(GEN x, long n)` is generally easier to use.

3.2.23 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 3.2.18 for the behavior for negative arguments.

The library syntax is `GEN gbitxor(GEN x, GEN y)`. Also available is `GEN ibitxor(GEN x, GEN y)`, which returns the bitwise *xor* of $|x|$ and $|y|$, two integers.

3.2.24 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 Euclidean quotient of the numerator by the denominator.

The library syntax is `GEN gceil(GEN x)`.

3.2.25 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 $-n/2 < y \leq n/2$. A `t_PADIC` is lifted as above if its valuation v is non-negative; if not, returns the fraction p^v `centerlift`(x^{-v}); in particular, note that rational reconstruction is not attempted.

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 `GEN centerlift0(GEN x, long v = -1)`, where `v` is a variable number. Also available is `GEN centerlift(GEN x)` corresponding to `centerlift0(x,-1)`.

3.2.26 component(x, n): 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.

For polynomials and power series, one should rather use `polcoeff`, and for vectors and matrices, the `[]` operator. Namely, if x is a vector, then `x[n]` represents the n^{th} component of x. 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.

Using of this function requires detailed knowledge of the structure of the different PARI types, and thus it should almost never be used directly. Some useful exceptions:

```
? x = 3 + O(3^5);
? component(x, 2)
%2 = 81   \\ p^(p-adic acurracy)
? component(x, 1)
%3 = 3    \\ p
? q = Qfb(1,2,3);
? component(q, 1)
%5 = 1
```

The library syntax is `GEN compo(GEN x, long n)`.

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 `GEN gconj(GEN x)`.

3.2.28 conjvec(z): conjugate vector representation of z. If z is a polmod, equal to `Mod`(a, T), this gives a vector of length degree(T) containing:

• the complex embeddings of z if T has rational coefficients, i.e. the $a(r[i])$ where $r =$ `polroots`(T);

• the conjugates of z if T has some intmod coefficients;

if z is a finite field element, the result is the vector of conjugates $[z, z^p, z^{p^2}, \ldots, z^{p^{n-1}}]$ where $n =$ degree(T).

If z is an integer or a rational number, the result is z. If z is a (row or column) vector, the result is a matrix whose columns are the conjugate vectors of the individual elements of z.

The library syntax is `GEN conjvec(GEN z, long 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 or 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 2.5.3.

The library syntax is `GEN denom(GEN x)`.

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

The library syntax is `GEN gfloor(GEN 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 `GEN gfrac(GEN 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 `GEN gimag(GEN x)`.

3.2.33 length(x): length of x; `#`x is a shortcut for `length`(x). This is mostly useful for

• vectors: dimension (0 for empty vectors),

• lists: number of entries (0 for empty lists),

• matrices: number of columns,

• character strings: number of actual characters (without trailing `\0`, should you expect it from *C* `char*`).

```
? #"a string"
%1 = 8
? #[3,2,1]
%2 = 3
? #[]
%3 = 0
? #matrix(2,5)
%4 = 5
? L = List([1,2,3,4]); #L
%5 = 4
```

The routine is in fact defined for arbitrary GP types, but is awkward and useless in other cases: it returns the number of non-code words in x, e.g. the effective length minus 2 for integers since the `t_INT` type has two code words.

The library syntax is `long glength(GEN x)`.

3.2.34 lift($x, \{v\}$): 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 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 `GEN lift0(GEN x, long v = -1)`, where `v` is a variable number. Also available is `GEN lift(GEN x)` corresponding to `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 `GEN gnorm(GEN 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 polynomial, 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 polynomial or vector (resp. matrix) with complex components.

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

The library syntax is `GEN gnorml2(GEN 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 or 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 2.5.3.

The library syntax is `GEN numer(GEN 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 `GEN numtoperm(long n, GEN k)`.

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 `LONG_MAX` ($2^{31} - 1$ for 32-bit machines or $2^{63} - 1$ for 64-bit machines) if x is an exact object.

The library syntax is `long padicprec(GEN x, GEN p)`.

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 `GEN permtonum(GEN 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.

```
? precision(exp(1e-100))
%1 = 134                \\ 134 significant decimal digits
? precision(2 + x)
%2 = 2147483647         \\ exact object
? precision(0.5 + O(x))
%3 = 28                 \\ floating point accuracy, NOT series precision
? precision( [ exp(1e-100), 0.5 ] )
%4 = 28                 \\ minimal accuracy among components
```

The return value for exact objects is meaningless since it is not even the same on 32 and 64-bit machines. The proper way to test whether an object is exact is

```
? isexact(x) = precision(x) == precision(0)
```

If n is not omitted, creates a new object equal to x with a new "precision" n. (This never changes the type of the result. In particular it is not possible to use it to obtain a polynomial from a power series; tor that, see `truncate`.) Now the meaning of precision is different from the above (floating point accuracy), and depends on the type of x:

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: yes, this is inconsistent.) Note that the precision is a priori distinct from the exponent k appearing in $O(*^k)$; it is indeed equal to k if and only if x is a p-adic or X-adic *unit*.

```
? precision(1 + O(x), 10)
%1 = 1 + O(x^10)
? precision(x^2 + O(x^10), 3)
%2 = x^2 + O(x^5)
? precision(7^2 + O(7^10), 3)
%3 = 7^2 + O(7^5)
```

For the last two examples, note that $x^2+O(x^5) = x^2(1+O(x^3))$ indeed has 3 significant coefficients

The library syntax is `GEN precision0(GEN x, long n)`. Also available are `GEN gprec(GEN x, long n)` and `long precision(GEN x)`. In both, the accuracy is expressed in *words* (32-bit or 64-bit depending on the architecture).

3.2.42 random($\{N = 2^{31}\}$): returns a random element in various natural sets depending on the argument N.

• `t_INT`: returns an integer uniformly distributed between 0 and $N-1$. Omitting the argument is equivalent to `random(2^31)`.

• `t_REAL`: returns a real number in $[0, 1[$ with the same accuracy as N (whose mantissa has the same number of significant words).

• `t_INTMOD`: returns a random intmod for the same modulus.

• `t_FFELT`: returns a random element in the same finite field.

• `t_VEC` generated by `ellinit` over a finite field k (coefficients are `t_INTMOD`s modulo a prime or `t_FFELT`s): returns a random k-rational *affine* point on the curve. More precisely an abscissa is drawn uniformly at random until `ellordinate` succeeds. In particular, the curves over $\mathbf{F}_2$ with a single point (at infinity!) will trigger an infinite loop. Note that this is definitely not a uniform distribution over $E(k)$.

• `t_POL` return a random polynomial of degree at most the degree of N. The coefficients are drawn by applying `random` to the leading coefficient of N.

```
? random(10)
%1 = 9
? random(Mod(0,7))
%2 = Mod(1, 7)
? a = ffgen(ffinit(3,7), 'a); random(a)
%3 = a^6 + 2*a^5 + a^4 + a^3 + a^2 + 2*a
? E = ellinit([0,0,0,3,7]*Mod(1,109)); random(E)
%4 = [Mod(103, 109), Mod(10, 109)]
? E = ellinit([0,0,0,1,7]*a^0); random(E)
%5 = [a^6 + a^5 + 2*a^4 + 2*a^2, 2*a^6 + 2*a^4 + 2*a^3 + a^2 + 2*a]
? random(Mod(1,7)*x^4)
%6 = Mod(5, 7)*x^4 + Mod(6, 7)*x^3 + Mod(2, 7)*x^2 + Mod(2, 7)*x + Mod(5, 7)
```

These variants all depend on a single internal generator, and are independent from the system's random number generator. A random seed may be obtained via `getrand`, and reset using `setrand`: from a given seed, and given sequence of `random`s, the exact same values will be generated. The same seed is used at each startup, reseed the generator yourself if this is a problem.

Technical note. Up to version 2.4 included, the internal generator produced pseudo-random numbers by means of linear congruences, which were not well distributed in arithmetic progressions. We now use Brent's XORGEN algorithm, based on Feedback Shift Registers, see `http://wwwmaths.anu.edu.au/~brent/random.html`. The generator has period $2^{4096} - 1$, passes the Crush battery of statistical tests of L'Ecuyer and Simard, but is not suitable for cryptographic purposes: one can reconstruct the state vector from a small sample of consecutive values, thus predicting the entire sequence.

The library syntax is `GEN genrand(GEN N = NULL)`.

Also available: `GEN ellrandom(GEN E)` and `GEN ffrandom(GEN a)`.

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 `GEN greal(GEN x)`.

3.2.44 round$(x, \{\&e\})$: If x is in $\mathbf{R}$, rounds x to the nearest integer (rounding to $+\infty$ in case of ties), then 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. 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 an approximate computation, when theory tells you that the coefficients must be integers.

The library syntax is `GEN round0(GEN x, GEN *e = NULL)`. Also available are `GEN grndtoi(GEN x, long *e)` and `GEN ground(GEN 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 the integer 0 (i.e. not `Mod(0,2)` 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 = 2 + y - y
%1 = 2
? isprime(x)
  ***   at top-level: isprime(x)
  ***                 ^----------
  *** isprime: 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 `GEN simplify(GEN 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 `long gsizebyte(GEN x)`. Also available is `long gsizeword(GEN x)` returning a number of *words*.

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 `long sizedigit(GEN x)`.

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 `GEN trunc0(GEN x, GEN *e = NULL)`. The following functions are also available: `GEN gtrunc(GEN x)` and `GEN gcvtoi(GEN x, long *e)`.

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 `LONG_MAX` ($2^{31} - 1$ 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 `long ggval(GEN x, GEN p)`.

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. If x is omitted, returns the list of user variables known to the interpreter, by order of decreasing priority. (Highest priority is x, which always come first.)

The library syntax is `GEN gpolvar(GEN x = NULL)`. However, in library mode, this function should not be used for x non-`NULL`, since `gvar` is more appropriate. Instead, for x a p-adic (type `t_PADIC`), p is $gel(x, 2)$; otherwise, use `long gvar(GEN x)` which returns the variable number of x if it exists, `NO_VARIABLE` otherwise, which satisfies the property `varncmp`(`NO_VARIABLE`, v) > 0 for all valid variable number v, i.e. it has lower priority than any variable.

3.3 Transcendental functions.

Since the values of transcendental functions cannot be exactly represented, these functions will always return an inexact object: a real number, a complex number, a p-adic number or a power series. All these objects have a certain finite precision.

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 a real number or an inexact complex number (like `1.0 + I` or `Pi*I` but not `2 - 3*I`), then the computation is done with the precision of the argument. In the example below, we see that changing the precision to 50 digits does not matter, because x only had a precision of 19 digits.

```
? \p 15
   realprecision = 19 significant digits (15 digits displayed)
? x = Pi/4
%1 = 0.785398163397448
? \p 50
   realprecision = 57 significant digits (50 digits displayed)
? sin(x)
%2 = 0.70710678118654752440
```

Note that even if the argument is real, the result may be complex (e.g. acos(2.0) or acosh(0.0)). See each individual function help for the definition of the branch cuts and choice of principal value.

• If the argument is either an integer, a rational, an exact complex number or a quadratic number, it is first converted to a real or complex number using the current precision held in the default `realprecision`. This precision (the number of decimal digits) can be changed using `\p` or `default(realprecision,...)`). After this conversion, the computation proceeds as above for real or complex arguments.

In library mode, the `realprecision` does not matter; instead the precision is taken from the `prec` parameter which every transcendental function has. As in `gp`, this `prec` is not used when the argument to a function is already inexact. Note that the argument *prec* stands for the length in words of a real number, including codewords. Hence we must have $prec \geq 3$.

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.

• If the argument is a polmod (representing an algebraic number), then the function is evaluated for every possible complex embedding of that algebraic number. A column vector of results is returned, with one component for each complex embedding. Therefore, the number of components equals the degree of the `t_POLMOD` modulus.

• 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. If we wanted to be strictly consistent with the PARI philosophy, we should have $x * y = (4 \bmod 8)$ and $\texttt{sqr}(x) = (4 \bmod 32)$ 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, a power series or a rational function, it is, if necessary, first converted to a power series using the current series precision, held in the default `seriesprecision`. This precision (the number of significant terms) can be changed using `\ps` or `default(seriesprecision,...)`. 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.

Under `gp` this again is transparent to the user. When programming in library mode, however, it is *strongly* advised to perform an explicit conversion to a power series first, as in `x = gtoser(x, seriesprec)`, where the number of significant terms `seriesprec` can be specified explicitly. If you do not do this, a global variable `precdl` is used instead, to convert polynomials and rational functions to a power series with a reasonable number of terms; tampering with the value of this global variable is *deprecated* and strongly discouraged.

• 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 2.5.5, 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 `GEN gpow(GEN x, GEN n, long prec)` for $x\hat{}n$.

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 `GEN mpeuler(long prec)`.

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

The library syntax is `GEN gen_I()`.

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

The library syntax is `GEN mppi(long prec)`.

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 = 1.4142135623730950488801688724
```

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 `GEN gabs(GEN x, long prec)`.

3.3.6 acos(x): principal branch of $\cos^{-1}(x) = -i\log(x + i\sqrt{1-x^2})$. In particular, $\mathrm{Re}(\mathrm{acos}(x)) \in [0,\pi]$ and if $x \in \mathbf{R}$ and $|x| > 1$, then $\mathrm{acos}(x)$ is complex. The branch cut is in two pieces: $]-\infty,-1]$, continuous with quadrant II, and $[1,+\infty[$, continuous with quadrant IV. We have $\mathrm{acos}(x) = \pi/2 - \mathrm{asin}(x)$ for all x.

The library syntax is `GEN gacos(GEN x, long prec)`.

3.3.7 acosh(x): principal branch of $\cosh^{-1}(x) = 2\log(\sqrt{(x+1)/2} + \sqrt{(x-1)/2})$. In particular, $\mathrm{Re}(\mathrm{acosh}(x)) \geq 0$ and $\mathrm{In}(\mathrm{acosh}(x)) \in]-\pi,\pi]0$; if $x \in \mathbf{R}$ and $x < 1$, then $\mathrm{acosh}(x)$ is complex.

The library syntax is `GEN gach(GEN x, long 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 `GEN agm(GEN x, GEN y, long prec)`.

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

The library syntax is `GEN garg(GEN x, long prec)`.

3.3.10 asin(x): principal branch of $\sin^{-1}(x) = -i\log(ix + \sqrt{1-x^2})$. In particular, $\mathrm{Re}(\mathrm{asin}(x)) \in [-\pi/2,\pi/2]$ and if $x \in \mathbf{R}$ and $|x| > 1$ then $\mathrm{asin}(x)$ is complex. The branch cut is in two pieces: $]-\infty,-1]$, continuous with quadrant II, and $[1,+\infty[$ continuous with quadrant IV. The function satisfies $i\mathrm{asin}(x) = \mathrm{asinh}(ix)$.

The library syntax is `GEN gasin(GEN x, long prec)`.

3.3.11 asinh(x): principal branch of $\sinh^{-1}(x) = \log(x + \sqrt{1+x^2})$. In particular $\mathrm{Im}(\mathrm{asinh}(x)) \in [-\pi/2,\pi/2]$. The branch cut is in two pieces: [-i oo ,-i], continuous with quadrant III and [i,+i oo [continuous with quadrant I.

The library syntax is `GEN gash(GEN x, long prec)`.

3.3.12 atan(x): principal branch of $\tan^{-1}(x) = \log((1+ix)/(1-ix))/2i$. In particular the real part of atan(x)) belongs to $]-\pi/2,\pi/2[$. The branch cut is in two pieces: $]-i\infty,-i[$, continuous with quadrant IV, and $]i,+i\infty[$ continuous with quadrant II. The function satisfies $i\mathrm{atan}(x) = -i\mathrm{atanh}(ix)$ for all $x \neq \pm i$.

The library syntax is `GEN gatan(GEN x, long prec)`.

3.3.13 atanh(x): principal branch of $\tanh^{-1}(x) = log((1+x)/(1-x))/2$. In particular the imaginary part of $\mathrm{atanh}(x)$ belongs to $[-\pi/2,\pi/2]$; if $x \in \mathbf{R}$ and $|x| > 1$ then $\mathrm{atanh}(x)$ is complex.

The library syntax is `GEN gath(GEN x, long prec)`.

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

The library syntax is `GEN bernfrac(long 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 `GEN bernreal(long x, long 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 `GEN bernvec(long x)`.

3.3.17 besselh1(nu, x): H^1-Bessel function of index nu and argument x.

The library syntax is `GEN hbessel1(GEN nu, GEN x, long prec)`.

3.3.18 besselh2(nu, x): H^2-Bessel function of index nu and argument x.

The library syntax is `GEN hbessel2(GEN nu, GEN x, long 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 `GEN ibessel(GEN nu, GEN x, long 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 `GEN jbessel(GEN nu, GEN x, long 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 2.5.5, this function is not very accurate when x is small.

The library syntax is `GEN jbesselh(GEN n, GEN x, long prec)`.

3.3.22 besselk(nu, x): K-Bessel function of index nu and argument x.

The library syntax is `GEN kbessel(GEN nu, GEN x, long prec)`.

3.3.23 besseln(nu, x): N-Bessel function of index nu and argument x.

The library syntax is `GEN nbessel(GEN nu, GEN x, long prec)`.

3.3.24 cos(x): cosine of x.

The library syntax is `GEN gcos(GEN x, long prec)`.

3.3.25 cosh(x): hyperbolic cosine of x.

The library syntax is `GEN gch(GEN x, long prec)`.

3.3.26 cotan(x): cotangent of x.

The library syntax is `GEN gcotan(GEN x, long 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 `GEN dilog(GEN x, long 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 `GEN veceint1(GEN x, GEN n = NULL, long prec)`. Also available is `GEN eint1(GEN x, long prec)`.

3.3.29 erfc(x): complementary error function, analytic continuation of $(2/\sqrt{\pi})\int_x^\infty e^{-t^2}\,dt$ $(x \in \mathbf{R})$.

The library syntax is `GEN gerfc(GEN x, long prec)`.

3.3.30 eta$(z, \{\mathit{flag} = 0\})$: Variants of Dedekind's η function. If $\mathit{flag} = 0$, return $\prod_{n=1}^\infty (1-q^n)$, where q depends on x in the following way:

• $q = e^{2i\pi x}$ if x is a *complex number* (which must then have positive imaginary part); notice that the factor $q^{1/24}$ is missing!

• $q = x$ if x is a `t_PADIC`, or can be converted to a *power series* (which must then have positive valuation).

If *flag* is non-zero, x is converted to a complex number and we return the true η function, $q^{1/24}\prod_{n=1}^\infty (1-q^n)$, where $q = e^{2i\pi x}$.

The library syntax is `GEN eta0(GEN z, long flag, long prec)`.

Also available is `GEN trueeta(GEN x, long prec)` ($\mathit{flag} = 1$).

3.3.31 exp(x): exponential of x. p-adic arguments with positive valuation are accepted.

The library syntax is `GEN gexp(GEN x, long prec)`. For a `t_PADIC` x, the function `GEN Qp_exp(GEN x)` is also available.

3.3.32 gamma(s): For s a complex number, evaluates Euler's gamma function

$$\Gamma(s) = \int_0^\infty t^{s-1} \exp(-t)\, dt.$$

Error if s is a non-positive integer, where Γ has a pole.

For s a p-adic number, evaluates the Morita gamma function at s, that is the unique continuous p-adic function on the p-adic integers extending $\Gamma_p(k) = (-1)^k \prod'_{j<k} j$, where the prime means that p does not divide j.

```
? gamma(1/4 + O(5^10))
%1= 1 + 4*5 + 3*5^4 + 5^6 + 5^7 + 4*5^9 + O(5^10)
? algdep(%,4)
%2 = x^4 + 4*x^2 + 5
```

The library syntax is `GEN ggamma(GEN s, long prec)`. For a `t_PADIC` x, the function `GEN Qp_gamma(GEN x)` is also available.

3.3.33 gammah(x): gamma function evaluated at the argument $x + 1/2$.

The library syntax is `GEN ggamd(GEN x, long 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 `GEN hyperu(GEN a, GEN b, GEN x, long 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 `GEN incgam0(GEN s, GEN x, GEN y = NULL, long prec)`. Also available is `GEN incgam(GEN s, GEN x, long prec)`.

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 `GEN incgamc(GEN s, GEN x, long prec)`.

3.3.37 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 `GEN glngamma(GEN x, long prec)`.

3.3.38 log(x): principal branch of the natural logarithm of $x \in \mathbf{C}^*$, i.e. such that $\mathrm{Im}(\log(x)) \in]-\pi, \pi]$. The branch cut lies along the negative real axis, continuous with quadrant 2, i.e. such that $\lim_{b \to 0^+} \log(a+bi) = \log a$ for $a \in \mathbf{R}^*$. 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 `GEN glog(GEN x, long prec)`. For a `t_PADIC` x, the function `GEN Qp_log(GEN x)` is also available.

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 $\Re_m$ denote $\Re$ or $\Im$ depending on 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 `GEN polylog0(long m, GEN x, long flag, long prec)`. Also available is `GEN gpolylog(long m, GEN x, long prec)` (*flag*$= 0$).

3.3.40 psi(x): the ψ-function of x, i.e. the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

The library syntax is `GEN gpsi(GEN x, long prec)`.

3.3.41 sin(x): sine of x.

The library syntax is `GEN gsin(GEN x, long prec)`.

3.3.42 sinh(x): hyperbolic sine of x.

The library syntax is `GEN gsh(GEN x, long 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 `GEN gsqr(GEN x)`.

3.3.44 sqrt(x): principal branch of the square root of x, defined as $\sqrt{x} = \exp(\log x/2)$. In particular, we have $\mathrm{Arg}(\mathrm{sqrt}(x)) \in]-\pi/2, \pi/2]$, and if $x \in \mathbf{R}$ and $x < 0$, then the result is complex with positive imaginary part.

Intmod a prime p 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 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 `GEN gsqrt(GEN x, long prec)`. For a `t_PADIC` x, the function `GEN Qp_sqrt(GEN x)` is also available.

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)
  ***   at top-level: 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)=
{ my(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 `GEN gsqrtn(GEN x, GEN n, GEN *z = NULL, long prec)`. If x is a `t_PADIC`, the function `GEN Qp_sqrt(GEN x, GEN n, GEN *z)` is also available.

3.3.46 tan(x): tangent of x.

The library syntax is `GEN gtan(GEN x, long prec)`.

3.3.47 tanh(x): hyperbolic tangent of x.

The library syntax is `GEN gth(GEN x, long 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 `GEN teich(GEN x)`.

3.3.49 theta(q,z): Jacobi sine theta-function

$$\theta_1(z,q) = 2q^{1/4}\sum_{n\geq 0}(-1)^n q^{n(n+1)}\sin((2n+1)z).$$

The library syntax is `GEN theta(GEN q, GEN z, long prec)`.

3.3.50 thetanullk(q,k): k-th derivative at $z=0$ of `theta`(q,z).

The library syntax is `GEN thetanullk(GEN q, long k, long prec)`.

`GEN vecthetanullk(GEN q, long k, long prec)` returns the vector of all $\frac{d^i\theta}{dz^i}(q,0)$ for all odd $i = 1, 3, \ldots, 2k-1$.

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 $ff_1f_2 = \sqrt{2}$.

The library syntax is `GEN weber0(GEN x, long flag, long prec)`. Also available are `GEN weberf(GEN x, long prec)`, `GEN weberf1(GEN x, long prec)` and `GEN weberf2(GEN x, long 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 `GEN gzeta(GEN s, long prec)`.

3.4 Arithmetic functions.

These functions are by definition functions whose natural domain of definition is either $\mathbf{Z}$ (or $\mathbf{Z}_{>0}$). The way these functions are used is completely different from transcendental functions in that there are no automatic type conversions: in general only integers are accepted as arguments.

3.4.1 Arithmetic functions and factorization 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**. This machinery relies on a fairly strong probabilistic primality test, see `ispseudoprime`, but you may also set

```
default(factor_proven, 1)
```

to ensure that all tentative factorizations are fully proven. This should not slow down PARI too much, unless prime numbers with hundreds of decimal digits occur frequently in your application.

3.4.2 Orders in finite groups and Discrete Logarithm functions

The following functions compute the order of an element in a finite group: `ellorder` (the rational points on an elliptic curve defined over a finite field), `fforder` (the multiplicative group of a finite field), `znorder` (the invertible elements in $\mathbf{Z}/n\mathbf{Z}$). The following functions compute discrete logarithms in the same groups (whenever this is meaningful) `elllog`, `fflog`, `znlog`.

All such functions allow an optional argument specifying an integer N, representing the order of the group. (The *order* functions also allows any non-zero multiple of the order, with a minor loss of efficiency.) The meaning of that optional argument is as follows and depends on its type

- `t_INT`: the integer N,
- `t_MAT`: the factorization `fa = factor(`N`)`,
- `t_VEC`: this is the preferred format and provides both the integer N and its factorization in a two-component vector `[`N`, fa]`.

When the group is fixed and many orders or discrete logarithms will be computed, it is much more efficient to initialize this data once and for all and pass it to the relevant functions, as in

```
? p = nextprime(10^40);
? v = [p-1, factor(p-1)]; \\ data for discrete log & order computations
? znorder(Mod(2,p), v)
%3 = 500000000000000000000000000028
? g = znprimroot(p);
? znlog(2, g, v)
%5 = 54303807090401490880187861137 4
```

3.4.3 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 divide by the elements in this table. If x is empty or omitted, just returns the current list of extra primes.

The entries in x must be primes: there is no internal check, even if the `factor_proven` default is set. To remove primes from the list use `removeprimes`.

The library syntax is `GEN addprimes(GEN x = NULL)`.

3.4.4 bestappr($x, \{A\}, \{B\}$): if B is omitted, finds the best rational approximation to $x \in \mathbf{R}$ using continued fractions. If A is omitted, return the best approximation affordable given the input accuracy; otherwise make sure that denominator is at most equal to A.

If B is present perform rational modular reconstruction (see below). In both cases, the function applies recursively to components of complex objects (polynomials, vectors, ...).

```
? bestappr(Pi, 100)
%1 = 22/7
? bestappr(0.142857142857142857142857142857142857142857142857142857142857142857142857142857142857142857142857142857)
%2 = 1/7
? bestappr([Pi, sqrt(2) + 'x], 10^3)
%3 = [355/113, x + 1393/985]
```

By definition, n/d is the best rational approximation to x if $|dx - n| < |vx - u|$ for all integers (u, v) with $v \leq A$. (Which implies that n/d is a convergent of the continued fraction of x.)

If x is an `t_INTMOD`, (or a recursive combination of those), modulo N say, B must be present. The routine then returns the unique rational number a/b in coprime integers $a \leq A$ and $b \leq B$ which is congruent to x modulo N. If $N \leq 2AB$, 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.

```
? bestappr(Mod(18526731858, 11^10), 10^10, 10^10)
 ***   at top-level: bestappr(Mod(1852673
 ***                 ^--------------------
 *** bestappr: ratlift: must have 2*amax*bmax < m, found
       amax=10000000000
       bmax=10000000000
       m=25937424601
? bestappr(Mod(18526731858, 11^10), 10^5, 10^5)
%1 = 1/7
? bestappr(Mod(18526731858, 11^20), 10^10, 10^10)
%2 = -1
```

In most concrete uses, B is a prime power and we performed Hensel lifting to obtain x.

If x is a `t_POLMOD`, modulo T say, B must be present. The routine then returns the unique rational function P/Q with $\deg P \leq A$ and $\deg Q \leq B$ which is congruent to x modulo T. If $\deg T \leq A + B$, uniqueness is not guaranteed and the function fails with an error message. If rational reconstruction is not possible, returns -1.

The library syntax is `GEN bestappr0(GEN x, GEN A = NULL, GEN B = NULL)`. Also available is `GEN bestappr(GEN x, GEN A)`.

3.4.5 bezout(x, y): Returns $[u, v, d]$ such that d is the gcd of x, y, $x * u + y * v = \gcd(x, y)$, and u and v minimal in a natural sense. The arguments must be integers or polynomials.

If x, y are polynomials in the same variable and *inexact* coefficients, then compute u, v, d such that $x * u + y * v = d$, where d approximately divides both and x and y; in particular, we do not obtain `gcd(x,y)` which is *defined* to be a scalar in this case:

```
? a = x + 0.0; gcd(a,a)
%1 = 1

? bezout(a,a)
%2 = [0, 1, x + 0.E-28]

? bezout(x-Pi,6*x^2-zeta(2))
%3 = [-6*x - 18.8495559, 1, 57.5726923]
```

For inexact inputs, the output is thus not well defined mathematically, but you obtain explicit polynomials to check whether the approximation is close enough for your needs.

The library syntax is `GEN vecbezout(GEN x, GEN y)`.

3.4.6 bezoutres(x, y): finds u and v such that $x * u + y * v = d$, where d is the resultant of x and y. The result is the row vector $[u, v, d]$. The algorithm used (subresultant) assumes that the base ring is a domain.

The library syntax is `GEN vecbezoutres(GEN x, GEN y)`.

3.4.7 bigomega(x): number of prime divisors of the integer $|x|$ counted with multiplicity:

```
? factor(392)
%1 =
[2 3]

[7 2]

? bigomega(392)
%2 = 5;  \\ = 3+2
? omega(392)
%3 = 2;  \\ without multiplicity
```

The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gbigomega(GEN x)`. For a `t_INT` x, the variant `long bigomega(GEN n)` is generally easier to use.

3.4.8 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 `GEN binomial(GEN x, long y)`. The function `GEN binomialuu(ulong n, ulong k)` is also available, and so is `GEN vecbinome(long n)`, which returns a vector v with $n + 1$ components such that $v[k + 1] =$ `binomial`(n, k) for k from 0 up to n.

3.4.9 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 `GEN chinese(GEN x, GEN y = NULL)`. `GEN chinese1(GEN x)` is also available.

3.4.10 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 `GEN content(GEN x)`.

3.4.11 contfrac$(x, \{b\}, \{nmax\})$: returns the row vector whose components are the partial quotients of the continued fraction expansion of x. In other words, a result $[a_0, \ldots, a_n]$ means that $x \approx a_0 + 1/(a_1 + \ldots + 1/a_n)$. The output is normalized so that $a_n \neq 1$ (unless we also have $n = 0$).

The number of partial quotients $n + 1$ is limited by `nmax`. If `nmax` is omitted, the expansion stops at the last significant partial quotient.

```
? \p19
  realprecision = 19 significant digits
? contfrac(Pi)
%1 = [3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1, 2, 2]
? contfrac(Pi,, 3)  \\ n = 2
%2 = [3, 7, 15]
```

x can also be a rational function or a power series.

If a vector b is supplied, the numerators are equal to the coefficients of b, instead of all equal to 1 as above; more precisely, $x \approx (1/b_0)(a_0 + b_1/(a_1 + \ldots + b_n/a_n))$; for a numerical continued fraction (x real), the a_i are integers, as large as possible; if x is a rational function, they are polynomials with $\deg a_i = \deg b_i + 1$. The length of the result is then equal to the length of b, unless the next partial quotient cannot be reliably computed, in which case the expansion stops. This happens when a partial remainder is equal to zero (or too small compared to the available significant digits for x a `t_REAL`).

A direct implementation of the numerical continued fraction `contfrac(x,b)` described above would be

```
\\ "greedy" generalized continued fraction
cf(x, b) =
{ my( a= vector(#b), t );

  x *= b[1];
  for (i = 1, #b,
    a[i] = floor(x);
    t = x - a[i]; if (!t || i == #b, break);
```

```
    x = b[i+1] / t;
  ); a;
}
```

There is some degree of freedom when choosing the a_i; the program above can easily be modified to derive variants of the standard algorithm. In the same vein, although no builtin function implements the related Engel expansion (a special kind of Egyptian fraction decomposition: $x = 1/a_1 + 1/(a_1a_2) + \dots$), it can be obtained as follows:

```
\\ n terms of the Engel expansion of x
engel(x, n = 10) =
{ my( u = x, a = vector(n) );
  for (k = 1, n,
    a[k] = ceil(1/u);
    u = u*a[k] - 1;
    if (!u, break);
  ); a
}
```

Obsolete hack. (don't use this): If b is an integer, *nmax* is ignored and the command is understood as `contfrac`$(x, , b)$.

The library syntax is `GEN contfrac0(GEN x, GEN b = NULL, long nmax)`. Also available are `GEN gboundcf(GEN x, long nmax)`, `GEN gcf(GEN x)` and `GEN gcf2(GEN b, GEN x)`.

3.4.12 contfracpnqn(x): when x is a vector or a one-row matrix, x is considered as the list of partial quotients $[a_0, a_1, \dots, a_n]$ 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 $p_n/q_n = a_0 + 1/(a_1 + \dots + 1/a_n)$. If x is a matrix with two rows $[b_0, b_1, \dots, b_n]$ and $[a_0, a_1, \dots, a_n]$, this is then considered as a generalized continued fraction and we have similarly $p_n/q_n = (1/b_0)(a_0 + b_1/(a_1 + \dots + b_n/a_n))$. Note that in this case one usually has $b_0 = 1$.

The library syntax is `GEN pnqn(GEN x)`.

3.4.13 core$(n, \{flag = 0\})$: if n is an integer written as $n = df^2$ with d squarefree, returns d. If *flag* is non-zero, returns the two-element row vector $[d, f]$. By convention, we write $0 = 0 \times 1^2$, so `core(0, 1)` returns $[0, 1]$.

The library syntax is `GEN core0(GEN n, long flag)`. Also available are `GEN core(GEN n)` (*flag* $= 0$) and `GEN core2(GEN n)` (*flag* $= 1$)

3.4.14 coredisc$(n, \{flag = 0\})$: a *fundamental discriminant* is an integer of the form $t \equiv 1 \bmod 4$ or $4t \equiv 8, 12 \bmod 16$, with t squarefree (i.e. 1 or the discriminant of a quadratic number field). Given a non-zero integer n, this routine returns the (unique) fundamental discriminant d such that $n = df^2$, f a positive rational number. If *flag* is non-zero, returns the two-element row vector $[d, f]$. If n is congruent to 0 or 1 modulo 4, f is an integer, and a half-integer otherwise.

By convention, `coredisc(0, 1))` returns $[0, 1]$.

Note that `quaddisc`(n) returns the same value as `coredisc`(n), and also works with rational inputs $n \in \mathbf{Q}^*$.

The library syntax is `GEN coredisc0(GEN n, long flag)`. Also available are `GEN coredisc(GEN n)` (*flag* $= 0$) and `GEN coredisc2(GEN n)` (*flag* $= 1$)

3.4.15 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 `GEN dirdiv(GEN x, GEN y)`.

3.4.16 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)(void*,GEN), GEN a, GEN b)`

3.4.17 dirmul(x, y): x and y being vectors of perhaps different lengths representing the Dirichlet series $\sum_n x_n n^{-s}$ and $\sum_n y_n n^{-s}$, computes the product of x by y, again as a vector.

```
? dirmul(vector(10,n,1), vector(10,n,moebius(n)))
%1 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The product length is the minimum of $\#x*v(y)$ and $\#y*v(x)$, where $v(x)$ is the index of the first non-zero coefficient.

```
? dirmul([0,1], [0,1]);
%2 = [0, 0, 0, 1]
```

The library syntax is `GEN dirmul(GEN x, GEN y)`.

3.4.18 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 `GEN divisors(GEN x)`.

3.4.19 eulerphi(x): Euler's ϕ (totient) function of $|x|$, in other words $|(\mathbf{Z}/x\mathbf{Z})^*|$. Normally, x must be of type integer, but the function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN geulerphi(GEN x)`. For a `t_INT` x, the variant `GEN eulerphi(GEN n)` is also available.

3.4.20 factor$(x, \{lim\})$: general factorization function, where x is a rational (including integers), a complex number with rational real and imaginary parts, or a rational function (including polynomials). The result is a two-column matrix: the first contains the irreducibles dividing x (rational or Gaussian primes, irreducible polynomials), and the second the exponents. By convention, 0 is factored as 0^1.

Q and Q(i). See `factorint` for more information about the algorithms used. The rational or Gaussian primes are in fact *pseudoprimes* (see `ispseudoprime`), a priori not rigorously proven primes. In fact, any factor which is $\leq 10^{15}$ (whose norm is $\leq 10^{15}$ for an irrational Gaussian prime) is a genuine prime. Use `isprime` to prove primality of other factors, as in

```
? fa = factor(2^2^7 + 1)
%1 =
[59649589127497217 1]

[5704689200685129054721 1]

? isprime( fa[,1] )
%2 = [1, 1]~   \\ both entries are proven primes
```

Another possibility is to set the global default `factor_proven`, which will perform a rigorous primality proof for each pseudoprime factor.

A `t_INT` argument *lim* can be added, meaning that we look only for prime factors $p < lim$. The limit *lim* must be non-negative and satisfy $lim \leq$ `primelimit` $+ 1$; setting $lim = 0$ is the same as setting it to `primelimit` $+ 1$. In this case, all but the last factor are proven primes, but the remaining factor may actually be a proven composite! If the remaining factor is less than lim^2, then it is prime.

```
? factor(2^2^7 +1, 10^5)
%3 =
[340282366920938463463374607431768211457 1]
```

This routine uses trial division and perfect power tests, and should not be used for huge values of *lim* (at most 10^9, say): `factorint(, 1 + 8)` will in general be faster. The latter does not guarantee that all small prime factors are found, but it also finds larger factors, and in a much more efficient way.

```
? F = (2^2^7 + 1) * 1009 * 100003; factor(F, 10^5)  \\ fast, incomplete
time = 0 ms.
%4 =
[1009 1]

[340292575391946091617278508669991164503343 71 1]

? default(primelimit,10^9)
time = 4,360 ms.
%5 = 1000000000
? factor(F, 10^9)    \\ very slow
time = 6,120 ms.
%6 =
[1009 1]

[100003 1]

[340282366920938463463374607431768211457 1]

? factorint(F, 1+8)  \\ much faster, all small primes were found
time = 40 ms.
%7 =
[1009 1]

[100003 1]
```

```
[340282366920938463463374607431768211457 1]

? factorint(F)   \\ complete factorisation
time = 260 ms.
%8 =
[1009 1]

[100003 1]

[59649589127497217 1]

[5704689200685129054721 1]
```

Rational functions. 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 $\mathbf{Q}$, van Hoeij's method is used, which is able to cope with hundreds of modular factors.

The routine guesses a sensible ring over which you want to factor: the smallest ring containing all coefficients, taking into account quotient structures induced by `t_INTMOD`s and `t_POLMOD`s (e.g. if a coefficient in $\mathbf{Z}/n\mathbf{Z}$ is known, all rational numbers encountered are first mapped to $\mathbf{Z}/n\mathbf{Z}$; different moduli will produce an error). Note 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 `nffactor`.

The library syntax is `GEN gp_factor0(GEN x, GEN lim = NULL)`. This function should only be used by the `gp` interface. Use directly `GEN factor(GEN x)` or `GEN boundfact(GEN x, long lim)`. The obsolete function `GEN factor0(GEN x, long lim)` is kept for backward compatibility.

3.4.21 factorback$(f, \{e\})$: gives back the factored object corresponding to a factorization. The integer 1 corresponds to the empty factorization.

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 1s: we return the product of the $f[i]$. Finally, f can be a regular factorization, as produced with any `factor` command. A few examples:

```
? factor(12)
%1 =
[2 2]

[3 1]

? factorback(%)
```

```
%2 = 12
? factorback([2,3], [2,1])   \\ 2^3 * 3^1
%3 = 12
? factorback([5,2,3])
%4 = 30
```

The library syntax is `GEN factorback2(GEN f, GEN e = NULL)`. Also available is `GEN factorback(GEN f)` (case $e = $ `NULL`).

3.4.22 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 `GEN factcantor(GEN x, GEN p)`.

3.4.23 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.

Either a or p can omitted (in which case both are ignored) if x has `t_FFELT` coefficients; the function then becomes identical to `factor`:

```
? factorff(x^2 + 1, 5, y^2+3)  \\ over F_5[y]/(y^2+3) ~ F_25
%1 =
[Mod(Mod(1, 5), Mod(1, 5)*y^2 + Mod(3, 5))*x
 + Mod(Mod(2, 5), Mod(1, 5)*y^2 + Mod(3, 5)) 1]

[Mod(Mod(1, 5), Mod(1, 5)*y^2 + Mod(3, 5))*x
 + Mod(Mod(3, 5), Mod(1, 5)*y^2 + Mod(3, 5)) 1]
? t = ffgen(y^2 + Mod(3,5), 't); \\ a generator for F_25 as a t_FFELT
? factorff(x^2 + 1)   \\ not enough information to determine the base field
 ***   at top-level: factorff(x^2+1)
 ***                 ^---------------
 *** factorff: incorrect type in factorff.
? factorff(x^2 + t^0) \\ make sure a coeff. is a t_FFELT
%3 =
[x + 2 1]

[x + 3 1]
? factorff(x^2 + t + 1)
%11 =
[x + (2*t + 1) 1]

[x + (3*t + 4) 1]
```

Notice that the second syntax is easier to use and much more readable.

The library syntax is `GEN factorff(GEN x, GEN p = NULL, GEN a = NULL)`.

3.4.24 factorial(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 `GEN mpfactr(long x, long prec)`. `GEN mpfact(long x)` returns $x!$ as a `t_INT`.

3.4.25 factorint($x, \{flag = 0\}$): 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. The output is a two-column matrix as for `factor`: the first column contains the "prime" divisors of n, the second one contains the (positive) exponents.

By convention 0 is factored as 0^1, and 1 as the empty factorization; also the divisors are by default not proven primes is they are larger than 2^64, they only failed the BPSW compositeness test (see `ispseudoprime`). Use `isprime` on the result if you want to guarantee primality or set the `factor_proven` default to 1. Entries of the private prime tables (see `addprimes`) are also included as is.

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 not be detected, although no example is known.

You are invited to play with the flag settings and watch the internals at work by using `gp`'s `debug` default parameter (level 3 shows just the outline, 4 turns on time keeping, 5 and above show an increasing amount of internal details).

The library syntax is `GEN factorint(GEN x, long flag)`.

3.4.26 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 `GEN factormod0(GEN x, GEN p, long flag)`.

3.4.27 ffgen($P, \{v\}$): return the generator $g = X \pmod{P(X)}$ of the finite field defined by the polynomial P (which must have `t_INTMOD` coefficients). If `v` is given, the variable name is used to display g, else the variable of the polynomial P is used.

The library syntax is `GEN ffgen(GEN P, long v = -1)`, where `v` is a variable number.

3.4.28 ffinit($p, n, \{v = x\}$): computes a monic polynomial of degree n which is irreducible over $\mathbf{F}_p$, where p is assumed to be prime. This function uses a fast variant of Adleman-Lenstra's algorithm.

It is useful in conjunction with `ffgen`; for instance if `P = ffinit(3,2)`, you can represent elements in $\mathbf{F}_{3^2}$ in term of `g = ffgen(P,g)`.

The library syntax is `GEN ffinit(GEN p, long n, long v = -1)`, where `v` is a variable number.

3.4.29 fflog$(x, g, \{o\})$: discrete logarithm of the finite field element x in base g. If present, o represents the multiplicative order of g, see Section 3.4.2; the preferred format for this parameter is `[ord, factor(ord)]`, where `ord` is the order of g. It may be set as a side effect of calling `ffprimroot`.

If no o is given, assume that g is a primitive root. See `znlog` for the limitations of the underlying discrete log algorithms.

```
? t = ffgen(ffinit(7,5));
? o = fforder(t)
%2 = 5602   \\ not a primitive root.
? fflog(t^10,t)
%3 = 11214  \\ Actually correct modulo o. We are lucky !
? fflog(t^10,t, o)
%4 = 10
? g = ffprimroot(t, &o);
? o   \\ order is 16806, bundled with its factorization matrix
%6 = [16806, [2, 1; 3, 1; 2801, 1]]
? fforder(g, o)
%7 = 16806  \\ no surprise there !
? fforder(g^10000, g, o)
? fflog(g^10000, g, o)
%9 = 10000
```

The library syntax is `GEN fflog(GEN x, GEN g, GEN o = NULL)`.

3.4.30 fforder$(x, \{o\})$: multiplicative order of the finite field element x. If o is present, it represents a multiple of the order of the element, see Section 3.4.2; the preferred format for this parameter is `[N, factor(N)]`, where `N` is the cardinality of the multiplicative group of the underlying finite field.

```
? t = ffgen(ffinit(nextprime(10^8), 5));
? g = ffprimroot(t, &o);  \\ o will be useful !
? fforder(g^1000000, o)
time = 0 ms.
%5 = 500000175000024500001715000060025000 8403
? fforder(g^1000000)
time = 16 ms. \\ noticeably slower, same result of course
%6 = 500000175000024500001715000060025000 8403
```

The library syntax is `GEN fforder(GEN x, GEN o = NULL)`.

3.4.31 ffprimroot$(x, \{\&o\})$: return a primitive root of the multiplicative group of the definition field of the finite field element x (not necessarily the same as the field generated by x). If present, o is set to a vector `[ord, fa]`, where `ord` is the order of the group and `fa` its factorisation `factor(ord)`. This last parameter is useful in `fflog` and `fforder`, see Section 3.4.2.

```
? t = ffgen(ffinit(nextprime(10^7), 5));
? g = ffprimroot(t, &o);
? o[1]
%3 = 10000095000361000685900651605247609 8
? o[2]
%4 =
[2 1]

[7 2]

[31 1]

[41 1]

[67 1]

[1523 1]

[10498781 1]

[15992881 1]

[46858913131 1]

? fflog(g^1000000, g, o)
time = 1,312 ms.
%5 = 1000000
```

The library syntax is `GEN ffprimroot(GEN x, GEN *o = NULL)`.

3.4.32 fibonacci(x): x^{th} Fibonacci number.

The library syntax is `GEN fibo(long x)`.

3.4.33 gcd$(x, \{y\})$: creates the greatest common divisor of x and y. If you also need the u and v such that $x * u + y * v = \gcd(x, y)$, use the `bezout` function. x and y can have rather 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 coefficients in the same number field (in particular rational): use modular gcd algorithm.

• general polynomials: use the subresultant algorithm if coefficient explosion is likely (non modular coefficients).

If u and v are polynomials in the same variable with *inexact* coefficients, their gcd is defined to be scalar, so that

```
? a = x + 0.0; gcd(a,a)
%1 = 1
? b = y*x + O(y); gcd(b,b)
%2 = y
? c = 4*x + O(2^3); gcd(c,c)
%2 = 4
```

A good quantitative check to decide whether such a gcd "should be" non-trivial, is to use `polresultant`: a value close to 0 means that a small deformation of the inputs has non-trivial gcd. You may also use `bezout`, which does try to compute an approximate gcd d and provides u, v to check whether $ux + vy$ is close to d.

The library syntax is `GEN ggcd0(GEN x, GEN y = NULL)`. Also available are `GEN ggcd(GEN x, GEN y)`, if `y` is not `NULL`, and `GEN content(GEN x)`, if `y` = `NULL`.

3.4.34 hilbert$(x, y, \{p\})$: Hilbert symbol of x and y modulo the prime p, $p = 0$ meaning the place at infinity (the result is undefined if $p \neq 0$ is not prime).

It is possible to omit p, in which case we take $p = 0$ if both x and y are rational, or one of them is a real number. And take $p = q$ if one of x, y is a `t_INTMOD` modulo q or a q-adic. (Incompatible types will raise an error.)

The library syntax is `long hilbert(GEN x, GEN y, GEN p = NULL)`.

3.4.35 isfundamental(x): true (1) if x is equal to 1 or to the discriminant of a quadratic field, false (0) otherwise. The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gisfundamental(GEN x)`.

3.4.36 ispower$(x, \{k\}, \{\&n\})$: if k is given, returns true (1) if x is a k-th power, false (0) if not.

If k is omitted, only integers and fractions are allowed for x 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 x is indeed a k-th power, sets n to a k-th root of x.

For a `t_FFELT x`, instead of omitting `k` (which is not allowed for this type), it may be natural to set

```
k = (x.p ^ poldegree(x.pol) - 1) / fforder(x)
```

The library syntax is `long ispower(GEN x, GEN k = NULL, GEN *n = NULL)`. Also available is `long gisanypower(GEN x, GEN *pty)` (k omitted).

3.4.37 isprime$(x, \{flag = 0\})$: true (1) if x is a prime number, false (0) otherwise. A prime number is a positive integer having exactly two distinct divisors among the natural numbers, namely 1 and itself.

This routine proves or disproves rigorously that a number is prime, which can be very slow when x is indeed prime and has more than 1000 digits, say. Use `ispseudoprime` to quickly check for compositeness. See also `factor`. It accepts vector/matrices arguments, and is then applied componentwise.

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 $x < 2^{64}$, as checked by Jan Feitsma),

• 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 prime divisors p of $x-1$ (such that $\prod p^{v_p(x-1)} > x^{1/3}$), 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. Note that if you monitor `isprime` at a high enough debug level, you may see warnings about untested integers being declared primes. This is normal: we ask for partial factorisations (sufficient to prove primality if the unfactored part is not too large), and `factor` warns us that the cofactor hasn't been tested. It may or may not be tested later, and may or may not be prime. This does not affect the validity of the whole `isprime` procedure.

If *flag* $= 2$, use APRCL.

The library syntax is `GEN gisprime(GEN x, long flag)`.

3.4.38 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. The function accepts vector/matrices arguments, and is then applied componentwise.

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, although it is expected that infinitely many such numbers exist. In particular, all composites $\leq 2^{64}$ are correctly detected (checked using `http://www.cecm.sfu.ca/Pseudoprimes/index-2-to-64.html`).

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 `GEN gispseudoprime(GEN x, long flag)`.

3.4.39 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, a square root of x is put into n.

```
? 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
```

For polynomials, either we detect that the characteristic is 2 (and check directly odd and even-power monomials) or we assume that 2 is invertible and check whether squaring the truncated power series for the square root yields the original input. The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gissquareall(GEN x, GEN *n = NULL)`. Also available is `GEN gissquare(GEN x)`.

3.4.40 issquarefree(x): true (1) if x is squarefree, false (0) if not. Here x can be an integer or a polynomial. The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gissquarefree(GEN x)`. For scalar arguments x (`t_INT` or `t_POL`), the function `long issquarefree(GEN x)` is easier to use.

3.4.41 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 `GEN gkronecker(GEN x, GEN y)`.

3.4.42 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 `GEN glcm0(GEN x, GEN y = NULL)`.

3.4.43 moebius(x): Moebius μ-function of $|x|$. x must be of type integer. The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gmoebius(GEN x)`. For a `t_INT` x, the variant `long moebius(GEN n)` is generally easier to use.

3.4.44 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 function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gnextprime(GEN x)`. For a scalar x, `long nextprime(GEN n)` is also available.

3.4.45 numbpart(n): gives the number of unrestricted partitions of n, usually called $p(n)$ in the literature; in other words the number of nonnegative integer solutions to $a + 2b + 3c + \cdots = n$. n must be of type integer and $n < 10^{15}$ (with trivial values $p(n) = 0$ for $n < 0$ and $p(0) = 1$). The algorithm uses the Hardy-Ramanujan-Rademacher formula. To explicitly enumerate them, see `partitions`.

The library syntax is `GEN numbpart(GEN n)`.

3.4.46 numdiv(x): number of divisors of $|x|$. x must be of type integer. The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gnumbdiv(GEN x)`. If x is a `t_INT`, one may use `GEN numbdiv(GEN n)` directly.

3.4.47 omega(x): number of distinct prime divisors of $|x|$. x must be of type integer.

```
? factor(392)
%1 =
[2 3]

[7 2]

? omega(392)
%2 = 2;  \\ without multiplicity
? bigomega(392)
%3 = 5;  \\ = 3+2, with multiplicity
```

The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gomega(GEN x)`. For a `t_INT` x, the variant `long omega(GEN n)` is generally easier to use.

3.4.48 partitions($n, \{restr = 0\}$): returns vector of partitions of the integer n (negative values return `[]`, $n = 0$ returns the trivial partition of the empty set). The second optional argument may be set to a non-negative number smaller than n to restrict the value of each element in the partitions to that value. The default of 0 means that this maximum is n itself.

A partition is given by a `t_VECSMALL`:

```
? partitions(4, 2)
%1 = [Vecsmall([2, 2]), Vecsmall([1, 1, 2]), Vecsmall([1, 1, 1, 1])]
```

correspond to $2+2$, $1+1+2$, $1+1+1+1$.

The library syntax is `GEN partitions(long n, long restr)`.

3.4.49 polrootsff($x, \{p\}, \{a\}$): returns the vector of distinct roots of 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 $\mathbf{Z}/p\mathbf{Z}$. Either a or p can omitted (in which case both are ignored) if x has `t_FFELT` coefficients:

```
? polrootsff(x^2 + 1, 5, y^2+3)  \\ over F_5[y]/(y^2+3) ~ F_25
%1 = [Mod(Mod(3, 5), Mod(1, 5)*y^2 + Mod(3, 5)),
      Mod(Mod(2, 5), Mod(1, 5)*y^2 + Mod(3, 5))]
? t = ffgen(y^2 + Mod(3,5), 't); \\ a generator for F_25 as a t_FFELT
? polrootsff(x^2 + 1)   \\ not enough information to determine the base field
 ***   at top-level: polrootsff(x^2+1)
 ***                 ^-----------------
 *** polrootsff: incorrect type in factorff.
? polrootsff(x^2 + t^0) \\ make sure one coeff. is a t_FFELT
%3 = [3, 2]
? polrootsff(x^2 + t + 1)
%4 = [2*t + 1, 3*t + 4]
```

Notice that the second syntax is easier to use and much more readable.

The library syntax is `GEN polrootsff(GEN x, GEN p = NULL, GEN a = NULL)`.

3.4.50 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 function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN gprecprime(GEN x)`. For a scalar x, `long precprime(GEN n)` is also available.

3.4.51 prime(n): the x^{th} prime number, which must be among the precalculated primes.

The library syntax is `GEN prime(long n)`.

3.4.52 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 `GEN primepi(GEN x)`.

3.4.53 primes(x): creates a row vector whose components are the first x prime numbers, which must be among the precalculated primes.

```
? primes(10)           \\ the first 10 primes
%1 = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
? primes(primepi(10))  \\ the primes up to 10
%2 = [2, 3, 5, 7]
```

The library syntax is `GEN primes(long x)`.

3.4.54 qfbclassno($D, \{flag = 0\}$): ordinary class number of the quadratic order of discriminant D. In the present version 2.5.5, 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 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
```

See also `qfbhclassno`.

The library syntax is `GEN qfbclassno0(GEN D, long flag)`. The following functions are also available:

`GEN classno(GEN D)` (*flag* $= 0$)

`GEN classno2(GEN D)` (*flag* $= 1$).

Finally

`GEN hclassno(GEN D)` computes the class number of an imaginary quadratic field by counting reduced forms, an $O(|D|)$ algorithm.

3.4.55 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 `GEN qfbcompraw(GEN x, GEN y)`.

3.4.56 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 `GEN hclassno(GEN x)`.

3.4.57 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 takes advantage of asymptotically fast operations and careful optimizations).

The library syntax is `GEN nucomp(GEN x, GEN y, GEN L)`. Also available is `GEN nudupl(GEN x, GEN L)` when $x = y$.

3.4.58 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 `GEN nupow(GEN x, GEN n)`.

3.4.59 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 `GEN qfbpowraw(GEN x, long n)`.

3.4.60 qfbprimeform(x, p): prime binary quadratic form of discriminant x whose first coefficient is p, where $|p|$ is a prime number. By abuse of notation, $p = \pm 1$ is also valid and returns the unit form. Returns an error if x is not a quadratic residue mod p, or if $x < 0$ and $p < 0$. (Negative definite `t_QFI` are not implemented.) In the case where $x > 0$, the "distance" component of the form is set equal to zero according to the current precision.

The library syntax is `GEN primeform(GEN x, GEN p, long prec)`.

3.4.61 qfbred$(x, \{flag = 0\}, \{d\}, \{isd\}, \{sd\})$: 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

The arguments d, isd, sd, 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 `GEN qfbred0(GEN x, long flag, GEN d = NULL, GEN isd = NULL, GEN sd = NULL)`. Also available are

`GEN redimag(GEN x)` (for definite x),

and for indefinite forms:

`GEN redreal(GEN x)`

`GEN rhoreal(GEN x)` (= `qfbred(x,1)`),

`GEN redrealnod(GEN x, GEN isd)` (= `qfbred(x,2,,isd)`),

`GEN rhorealnod(GEN x, GEN isd)` (= `qfbred(x,3,,isd)`).

3.4.62 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 `GEN qfbsolve(GEN Q, GEN p)`.

3.4.63 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.

The result is a vector v whose components should be accessed using member functions:

- v`.no`: the class number
- v`.cyc`: a vector giving the structure of the class group as a product of cyclic groups;
- v`.gen`: a vector giving generators of those cyclic groups (as binary quadratic forms).
- v`.reg`: 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 *flag* is obsolete and should be left alone. In older versions, it supposedly computed the narrow class group when $D > 0$, but this did 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, see 3.6.7. The parameter is used as a threshold to balance the relation finding phase against the final linear algebra. Increasing the default $c_1 = 0.2$ means that relations are easier to find, but more relations are needed and the linear algebra will be harder. The parameter c_2 is mostly obsolete and should not be changed, but we still document it for completeness: we compute a tentative class group by generators and relations using a factorbase of prime ideals $\leq c_1(\log|D|)^2$, then prove that ideals of norm $\leq c_2(\log|D|)^2$ do not generate a larger group. By default an optimal c_2 is chosen, so that the result is provably correct under the GRH — a famous result of Bach states that $c_2 = 6$ is fine, but it is possible to improve on this algorithmically. You may provide a smaller c_2, it will be ignored (we use the provably correct one); you may provide a larger c_2 than the default value, which results in longer computing times for equally correct outputs (under GRH).

The library syntax is `GEN quadclassunit0(GEN D, long flag, GEN tech = NULL, long prec)`. If you really need to experiment with the *tech* parameter, it is usually more convenient to use `GEN Buchquad(GEN D, double c1, double c2, long prec)`

3.4.64 quaddisc(x): discriminant of the quadratic field $\mathbf{Q}(\sqrt{x})$, where $x \in \mathbf{Q}$.

The library syntax is `GEN quaddisc(GEN x)`.

3.4.65 quadgen(D): creates the quadratic number $\omega = (a + \sqrt{D})/2$ where $a = 0$ if $D \equiv 0 \bmod 4$, $a = 1$ 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 `GEN quadgen(GEN D)`.

3.4.66 quadhilbert(D): relative equation defining the Hilbert class field of the quadratic field of discriminant D.

If $D < 0$, uses complex multiplication (Schertz's variant).

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 `GEN quadhilbert(GEN D, long prec)`.

3.4.67 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 `GEN quadpoly0(GEN D, long v = -1)`, where `v` is a variable number.

3.4.68 quadray(D, f): 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 and Schertz's method.

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 `GEN quadray(GEN D, GEN f, long prec)`.

3.4.69 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 `GEN quadregulator(GEN x, long prec)`.

3.4.70 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 3.4.65).

The library syntax is `GEN quadunit(GEN D)`.

3.4.71 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 `GEN removeprimes(GEN x = NULL)`.

3.4.72 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 function accepts vector/matrices arguments for x, and is then applied componentwise.

The library syntax is `GEN gsumdivk(GEN x, long k)`. Also available are `GEN gsumdiv(GEN n)` ($k = 1$), `GEN sumdivk(GEN n,long k)` (n a `t_INT`) and `GEN sumdiv(GEN n)` ($k = 1$, n a `t_INT`)

3.4.73 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 `GEN sqrtint(GEN x)`.

3.4.74 stirling$(n, k, \{flag = 1\})$: Stirling number of the first kind $s(n, k)$ ($flag = 1$, default) or of the second kind $S(n, k)$ ($flag$=2), where n, k are non-negative integers. The former is $(-1)^{n-k}$ times the number of permutations of n symbols with exactly k cycles; the latter is the number of ways of partitioning a set of n elements into k non-empty subsets. Note that if all $s(n, k)$ are needed, it is much faster to compute

$$\sum_k s(n, k)x^k = x(x - 1)\dots(x - n + 1).$$

Similarly, if a large number of $S(n, k)$ are needed for the same k, one should use

$$\sum_n S(n, k)x^n = \frac{x^k}{(1 - x)\dots(1 - kx)}.$$

(Should be implemented using a divide and conquer product.) Here are simple variants for n fixed:

```
/* list of s(n,k), k = 1..n */
vecstirling(n) = Vec( factorback(vector(n-1,i,1-i*'x)) )

/* list of S(n,k), k = 1..n */
vecstirling2(n) =
{ my(Q = x^(n-1), t);
  vector(n, i, t = divrem(Q, x-i); Q=t[1]; t[2]);
}
```

The library syntax is `GEN stirling(long n, long k, long flag)`. Also available are `GEN stirling1(ulong n, ulong k)` ($flag = 1$) and `GEN stirling2(ulong n, ulong k)` ($flag = 2$).

3.4.75 sumdedekind(h, k): returns the Dedekind sum associated to the integers h and k, corresponding to a fast implementation of

```
s(h,k) = sum(n = 1, k-1, (n/k)*(frac(h*n/k) - 1/2))
```

The library syntax is `GEN sumdedekind(GEN h, GEN k)`.

3.4.76 zncoppersmith$(P, N, X, \{B = N\})$: N being an integer and $P \in \mathbf{Z}[X]$, finds all integers x with $|x| \leq X$ such that

$$\gcd(N, P(x)) \geq B,$$

using Coppersmith's algorithm (a famous application of the LLL algorithm). X must be smaller than $\exp(\log^2 B/(\deg(P) \log N))$: for $B = N$, this means $X < N^{1/\deg(P)}$. Some x larger than X may be returned if you are very lucky. The smaller B (or the larger X), the slower the routine will be. The strength of Coppersmith method is the ability to find roots modulo a general *composite* N: if N is a prime or a prime power, `polrootsmod` or `polrootspadic` will be much faster.

We shall now present two simple applications. The first one is finding non-trivial factors of N, given some partial information on the factors; in that case B must obviously be smaller than the largest non-trivial divisor of N.

```
setrand(1); \\ to make the example reproducible
p = nextprime(random(10^30));
q = nextprime(random(10^30)); N = p*q;
p0 = p % 10^20; \\ assume we know 1) p > 10^29, 2) the last 19 digits of p
p1 = zncoppersmith(10^19*x + p0, N, 10^12, 10^29)

\\ result in 10ms.
%1 = [35023733690]
? gcd(p1[1] * 10^19 + p0, N) == p
%2 = 1
```

and we recovered p, faster than by trying all possibilities $< 10^{12}$.

The second application is an attack on RSA with low exponent, when the message x is short and the padding P is known to the attacker. We use the same RSA modulus N as in the first example:

```
setrand(1);
P = random(N);    \\ known padding
e = 3;            \\ small public encryption exponent
X = floor(N^0.3); \\ N^(1/e - epsilon)
x0 = random(X);   \\ unknown short message
C = lift( (Mod(x0,N) + P)^e ); \\ known ciphertext, with padding P
zncoppersmith((P + x)^3 - C, N, X)

\\ result in 3.8s.
%3 = [265174753892462432]
? %[1] == x0
%4 = 1
```

We guessed an integer of the order of 10^{18} in a couple of seconds.

The library syntax is `GEN zncoppersmith(GEN P, GEN N, GEN X, GEN B = NULL)`.

3.4.77 znlog$(x, g, \{o\})$: discrete logarithm of x in $(\mathbf{Z}/N\mathbf{Z})^*$ in base g. If present, o represents the multiplicative order of g, see Section 3.4.2; the preferred format for this parameter is `[ord, factor(ord)]`, where `ord` is the order of g. If no o is given, assume that g generate $(\mathbf{Z}/N\mathbf{Z})^*$.

This function uses a simple-minded combination of generic discrete log algorithms (index calculus methods are not yet implemented).

• Pohlig-Hellman algorithm, to reduce to groups of prime order q, where $q|p-1$ and p is an odd prime divisor of N,

• Shanks baby-step/giant-step (q small),

• Pollard rho method (q large).

The latter two algorithms require $O(\sqrt{q})$ operations in the group on average, hence will not be able to treat cases where $q > 10^{30}$, say.

```
? g = znprimroot(101)
%1 = Mod(2,101)
? znlog(5, g)
%2 = 24
? g^24
%3 = Mod(5, 101)

? G = znprimroot(2 * 101^10)
%4 = Mod(11046221254112045 1003, 22092442508224090 2002)
? znlog(5, G)
%5 = 76210072736547066624
? G^% == 5
%6 = 1
```

The result is undefined when x is not a power of g or when x is not invertible mod N:

```
? znlog(6, Mod(2,3))
  ***   at top-level: znlog(6,Mod(2,3))
  ***                 ^-----------------
  *** znlog: impossible inverse modulo: Mod(0, 3).
```

For convenience, g is also allowed to be a p-adic number:

```
? g = 3+O(5^10); znlog(2, g)
%1 = 1015243
? g^%
%2 = 2 + O(5^10)
```

The library syntax is `GEN znlog(GEN x, GEN g, GEN o = NULL)`.

3.4.78 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. The parameter o, if present, represents a non-zero multiple of the order of x, see Section 3.4.2; the preferred format for this parameter is `[ord, factor(ord)]`, where `ord = eulerphi(n)` is the cardinality of the group.

The library syntax is `GEN znorder(GEN x, GEN o = NULL)`. Also available is `GEN order(GEN x)`.

3.4.79 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 $k \geq 0$). If the group is not cyclic, the result is undefined. If n is a prime, then the smallest positive primitive root is returned. This is no longer true for composites.

Note that this function requires factoring $p-1$ for p as above, in order to determine the exact order of elements in $(\mathbf{Z}/n\mathbf{Z})^*$: this is likely to be very costly if p is large. The function accepts vector/matrices arguments, and is then applied componentwise.

The library syntax is `GEN znprimroot0(GEN n)`. For a `t_INT` x, the special case `GEN znprimroot(GEN n)` is also available.

3.4.80 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 $i \geq 2$ 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 `GEN znstar(GEN 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`, which initializes and returns an *ell* structure by default. If a specific flag is added, a shortened *smallell* 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 *smallell*:

`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_1/ω_2 belongs to Poincaré's half-plane).
`eta` : quasi-periods $[\eta_1, \eta_2]$, such that $\eta_1\omega_2 - \eta_2\omega_1 = 2i\pi$.
`roots` : roots of the associated Weierstrass equation.
`tate` : $[u^2, u, v]$ in the notation of Tate.
`w` : Mestre's w (this is technical).

Warning: as for the orientation of the basis of the period lattice, beware that many sources use the inverse convention where ω_2/ω_1 has positive imaginary part and our ω_2 is the negative of theirs. Our convention $\tau = \omega_1/\omega_2$ ensures that the action of PSL_2 is the natural one:

$$[a, b; c, d] \cdot \tau = (a\tau + b)/(c\tau + d) = (a\omega_1 + b\omega_2)/(c\omega_1 + d\omega_2),$$

instead of a twisted one. (Our *tau* is $-1/\tau$ in the above inverse convention.)

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 a *smallell*, or as an *ell*; if a *smallell* is requested, an *ell* may equally be given.

A few routines — namely `ellgenerators`, `ellidentify`, `ellsearch`, `forell` — require the optional package `elldata` (John Cremona's database) to be installed. The function `ellinit` will also allow alternative inputs, e.g. `ellinit("11a1")`. Functions using this package need to load chunks of a large database in memory and require at least 2MB stack to avoid stack overflows.

3.5.1 ellL1(e, r): returns the value at $s = 1$ of the derivative of order r of the L-function of the elliptic curve e assuming that r is at most the order of vanishing of the L-function at $s = 1$. (The result is wrong if r is strictly larger than the order of vanishing at 1.)

```
? e = ellinit("11a1"); \\ order of vanishing is 0
? ellL1(e, 0)
%2 = 0.25384186085591068433775892335
? e = ellinit("389a1");  \\ order of vanishing is 2
? ellL1(e, 0)
%4 = -5.3840673118372180892350324140 E-29
? ellL1(e, 1)
%5 = 0
? ellL1(e, 2)
%6 = 1.5186330005768535404603852140
```

The main use of this function, after computing at *low* accuracy the order of vanishing using `ellanalyticrank`, is to compute the leading term at *high* accuracy to check (or use) the Birch and Swinnerton-Dyer conjecture:

```
? \p18
  realprecision = 18 significant digits
? ellanalyticrank(ellinit([0, 0, 1, -7, 6]))
```

```
time = 32 ms.
%1 = [3, 10.3910994007158041]
? \p200
  realprecision = 202 significant digits (200 digits displayed)
? ellL1(e, 3)
time = 23,113 ms.
%3 = 10.391099400715804138751850510360917069726356375657009279 7[...]
```

The library syntax is `GEN ellL1(GEN e, long r, long prec)`.

3.5.2 elladd$(E, z1, z2)$: sum of the points $z1$ and $z2$ on the elliptic curve corresponding to E.

The library syntax is `GEN addell(GEN E, GEN z1, GEN z2)`.

3.5.3 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 a *smallell* 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 `GEN akell(GEN E, GEN n)`.

3.5.4 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 `GEN anell(GEN E, long n)`.

3.5.5 ellanalyticrank$(e, \{eps\})$: returns the order of vanishing at $s = 1$ of the L-function of the elliptic curve e and the value of the first non-zero derivative. To determine this order, it is assumed that any value less than `eps` is zero. If no value of `eps` is given, a value of half the current precision is used.

```
? e = ellinit("11a1"); \\ rank 0
? ellanalyticrank(e)
%2 = [0, 0.25384186085591068433775892 33]
? e = ellinit("37a1"); \\ rank 1
? ellanalyticrank(e)
%4 = [1, 0.30599977383405230182048368 35]
? e = ellinit("389a1"); \\ rank 2
? ellanalyticrank(e)
%6 = [2, 1.51863300057685354046038521 4]
? e = ellinit("5077a1"); \\ rank 3
? ellanalyticrank(e)
%8 = [3, 10.391099400715804138751850 35]
```

The library syntax is `GEN ellanalyticrank(GEN e, GEN eps = NULL, long prec)`.

3.5.6 ellap$(E, \{p\})$: computes the trace of Frobenius a_p for the elliptic curve E and the prime number p. This is 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 $\mathbf{F}_p$.

No checking is done that p is indeed prime. E must be a *smallell* as output by `ellinit`, defined over $\mathbf{Q}$, $\mathbf{Q}_p$, or $\mathbf{F}_p$. The prime p may be omitted if the curve was defined over $\mathbf{F}_p$ (`t_INTMOD` coefficients) or $\mathbf{Q}_p$ (`t_PADIC` coefficients). Otherwise the curve must be defined over $\mathbf{Q}$, and p must be explicitly given. Over $\mathbf{Q}$ or $\mathbf{Q}_p$, the equation is assumed to be minimal at p.

```
? E = ellinit([0,0,0,0,1]);  \\ defined over Q
? ellap(E, 3)  \\ 3 necessary here
%2 = 0    \\ #E(F_3) = 3+1 - 0 = 4
? ellap(E, 7)
%3 = -4   \\ #E(F_7) = 12
? E = ellinit([0,0,0,0,1] * Mod(1,11));  \\ defined over F_11
? ellap(E)       \\ no need to repeat 11
%5 = 0
? ellap(E, 11)   \\ ... but it also works
%6 = 0
? ellgroup(E, 13) \\ ouch, inconsistent input !
   ***   at top-level: ellap(E,13)
   ***                 ^-----------
   *** ellap: inconsistent moduli in Rg_to_Fp: 11, 13.
```

Algorithms used. If $E/\mathbf{F}_p$ has CM by a principal imaginary quadratic order we use an explicit formula (involving essentially Kronecker symbols and Cornacchia's algorithm, hence very fast: $O(\log p)^2$). Otherwise, we use Shanks-Mestre's baby-step/giant-step method, which runs in time $O(p^{1/4})$ using $O(p^{1/4})$ storage, hence becomes unreasonable when p has about 30 digits. If the `seadata` package is installed, the `SEA` algorithm becomes available and primes of the order of 200 digits become feasible.

The library syntax is `GEN ellap(GEN E, GEN p = NULL)`.

3.5.7 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 `GEN bilhell(GEN E, GEN z1, GEN z2, long prec)`.

3.5.8 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 a *smallell* as output by `ellinit`.

The library syntax is `GEN ellchangecurve(GEN E, GEN v)`.

3.5.9 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 `GEN ellchangepoint(GEN x, GEN v)`. The reciprocal function `GEN ellchangepointinv(GEN x, GEN ch)` inverts the coordinate change.

3.5.10 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 `GEN ellconvertname(GEN name)`.

3.5.11 elldivpol$(E, n, \{v =' x\})$: n-division polynomial for the curve `E` in the variable v.

The library syntax is `GEN elldivpol(GEN E, long n, long v = -1)`, where `v` is a variable number.

3.5.12 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 $q = \exp(2i\pi\tau)$ and $\tau := \omega_1/\omega_2$ belongs to the complex upper half-plane.

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 `GEN elleisnum(GEN E, long k, long flag, long prec)`.

3.5.13 elleta(om): returns the quasi-periods $[\eta_1, \eta_2]$ associated to the lattice basis $om = [\omega_1, \omega_2]$. Alternatively, om can be an elliptic curve E as output by `ellinit`, in which case, the quasi periods associated to the period lattice basis E`.omega` (namely, E`.eta`) are returned.

```
? elleta([1, I])
%1 = [3.1415926535897932384626433832, 9.4247779607693797153879301490*I]
```

The library syntax is `GEN elleta(GEN om, long prec)`.

3.5.14 ellgenerators(E): returns a **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 **Z** of small conductors.

The library syntax is `GEN ellgenerators(GEN E)`.

3.5.15 ellglobalred(E): calculates the arithmetic conductor, the global minimal model of E and the global Tamagawa number c. E must be a *smallell* 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 `GEN ellglobalred(GEN E)`.

3.5.16 ellgroup$(E, \{p\})$: computes the structure of the group $E(\mathbf{F}_p) \sim \mathbf{Z}/d_1\mathbf{Z} \times \mathbf{Z}/d_2\mathbf{Z}$, with $d_2 \mid d_1$. The prime p may be omitted if the curve was defined over $\mathbf{F}_p$ (`t_INTMOD` coefficients) or $\mathbf{Q}_p$ (`t_PADIC` coefficients). Otherwise the curve must be defined over $\mathbf{Q}$, and p must be explicitly given. Over $\mathbf{Q}$ or $\mathbf{Q}_p$, the equation is assumed to be minimal at p.

```
? E = ellinit([0,0,0,0,1]);  \\ defined over Q
? ellgroup(E, 3)  \\ 3 necessary here
%2 = [4]    \\ cyclic
? ellgroup(E, 7)
%3 = [6, 2] \\ non-cyclic
? E = ellinit([0,0,0,0,1] * Mod(1,11));  \\ defined over F_11
? ellgroup(E)   \\ no need to repeat 11
%5 = [12]
? ellgroup(E, 11)   \\ ... but it also works
%6 = [12]
? ellgroup(E, 13) \\ ouch, inconsistent input !
   ***   at top-level: ellgroup(E,13)
   ***                 ^--------------
   *** ellgroup: inconsistent moduli in Rg_to_Fp: 11, 13.
```

The library syntax is `GEN ellgroup(GEN E, GEN p = NULL)`.

3.5.17 ellheight$(E, x, \{flag = 2\})$: 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 `GEN ellheight0(GEN E, GEN x, long flag, long prec)`. Also available is `GEN ghell(GEN E, GEN x, long prec)` ($flag = 2$).

3.5.18 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 `GEN mathell(GEN E, GEN x, long prec)`.

3.5.19 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 the 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 `GEN ellidentify(GEN E)`.

3.5.20 ellinit$(x, \{flag = 0\})$: initialize an `ell` structure, associated to the elliptic curve E. E is either 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.

```
? E = ellinit([0,0,0,0,1]); \\ y^2 = x^3 + 1
? E = ellinit("36a1");      \\ the same curve, using Cremona's notations
```

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 an *ell* structure by default, and a shorter *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, in which case the computation will be 10 (p-adic) to 200 (complex) times slower. 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, 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. E`.omega` forms a basis of the complex lattice defining E, such that $\tau = \frac{\omega_1}{\omega_2}$ has positive imaginary part.

E`.eta` is a row vector containing the quasi-periods η_1 and η_2 such that $\eta_i = 2\zeta(\omega_i/2)$, where ζ is the Weierstrass zeta function associated to the period lattice (see `ellzeta`). In particular, the Legendre relation holds: $\eta_2\omega_1 - \eta_1\omega_2 = 2i\pi$.

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 to zero. See also the description of member functions related to elliptic curves at the beginning of this section.

The library syntax is `GEN ellinit0(GEN x, long flag, long prec)`. Also available are `GEN ellinit(GEN E, long prec)` (*flag* = 0) and `GEN smallellinit(GEN E, long prec)` (*flag* = 1).

3.5.21 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 `GEN ellisoncurve(GEN E, GEN z)`. Also available is `int oncurve(GEN E, GEN z)` which does not accept vectors of points.

3.5.22 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 `GEN jell(GEN x, long prec)`.

3.5.23 elllocalred(E, p): calculates the Kodaira type of the local fiber of the elliptic curve E at the prime p. E must be a *smallell* as output by `ellinit`, and is assumed to have all its coefficients a_i in $\mathbf{Z}$. The result is a 4-component vector $[f, kod, v, c]$. 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 `GEN elllocalred(GEN E, GEN p)`.

3.5.24 elllog$(E, P, G, \{o\})$: discrete logarithm of the point P of the elliptic curve E in base G. See `znlog` for the limitations of the underlying discrete log algorithms. If present, o represents the order of G, see Section 3.4.2; the preferred format for this parameter is `[N, factor(N)]`, where N is the order of G.

If no o is given, assume that G generates the curve. The function also assumes that P is a multiple of G.

```
? a = ffgen(ffinit(2,8),'a);
? E = ellinit([a,1,0,0,1]);  \\ over F_{2^8}
? x = a^3; y = ellordinate(E,x)[1];
? P = [x,y]; G = ellpow(E, P, 113);
? ord = [242, factor(242)]; \\ P generates a group of order 242. Initialize.
```

```
? ellorder(E, G, ord)
%4 = 242
? e = elllog(E, P, G, ord)
%5 = 15
? ellpow(E,G,e) == P
%6 = 1
```

The library syntax is `GEN elllog(GEN E, GEN P, GEN G, GEN o = NULL)`.

3.5.25 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 `GEN elllseries(GEN E, GEN s, GEN A = NULL, long prec)`.

3.5.26 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 `GEN ellminimalmodel(GEN E, GEN *v = NULL)`.

3.5.27 ellmodulareqn$(l, \{x\}, \{y\})$: return a vector [`eqn`,t] where `eqn` is a modular equation of level l, for $l < 500$, l prime. This requires the package seadata to be installed. The equation is either of canonical type ($t = 0$) or of Atkin type ($t = 1$).

The library syntax is `GEN ellmodulareqn(long l, long x = -1, long y = -1)`, where `x`, `y` are variable numbers.

3.5.28 ellorder$(E, z, \{o\})$: gives the order of the point z on the elliptic curve E. If the curve is defined over $\mathbf{Q}$, return zero if the point has infinite order. The parameter o, if present, represents a non-zero multiple of the order of z, see Section 3.4.2; the preferred format for this parameter is `[ord, factor(ord)]`, where `ord` is the cardinality of the curve.

For a curve defined over $\mathbf{F}_p$, it is very important to supply o since its computation is very expensive and should only be done once, using

```
? N = p+1-ellap(E,p); o = [N, factor(N)];
```

possibly using the `seadata` package; for a curve defined over a non-prime finite field, giving the order is *mandatory* since no function is available yet to compute the cardinality or trace of Frobenius in that case.

The library syntax is `GEN ellorder(GEN E, GEN z, GEN o = NULL)`. The obsolete form `GEN orderell(GEN e, GEN z)` should no longer be used.

3.5.29 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 `GEN ellordinate(GEN E, GEN x, long prec)`.

3.5.30 ellpointtoz(E, P): 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 $0 \le \Re(t) < w1$ and $0 \le \Im(t) < \Im(w2)$.

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 `GEN zell(GEN E, GEN P, long prec)`.

3.5.31 ellpow(E, z, n): computes $[n]z$, where z is a point on the elliptic curve E. The exponent n is in $\mathbf{Z}$, or may be a complex quadratic integer if the curve E has complex multiplication by n (if not, an error message is issued).

```
? Ei = ellinit([0,0,0,1,0]); z = [0,0];
? ellpow(Ei, z, 10)
%2 = [0]     \\ unsurprising: z has order 2
? ellpow(Ei, z, I)
%3 = [0, 0]  \\ Ei has complex multiplication by Z[i]
? ellpow(Ei, z, quadgen(-4))
%4 = [0, 0]  \\ an alternative syntax for the same query
? Ej  = ellinit([0,0,0,0,1]); z = [-1,0];
? ellpow(Ej, z, I)
  ***   at top-level: ellpow(Ej,z,I)
  ***                 ^--------------
  *** ellpow: not a complex multiplication in powell.
? ellpow(Ej, z, 1+quadgen(-3))
%6 = [1 - w, 0]
```

The simple-minded algorithm for the CM case assumes that we are in characteristic 0, and that the quadratic order to which n belongs has small discriminant.

The library syntax is `GEN powell(GEN E, GEN z, GEN n)`.

3.5.32 ellrootno$(E, \{p = 1\})$: E being a *smallell* 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 `long ellrootno(GEN E, GEN p = NULL)`.

3.5.33 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"), an 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 a vector of all matching curves in the above format.

The library syntax is `GEN ellsearch(GEN N)`. Also available is `GEN ellsearchcurve(GEN N)` that only accepts complete curve names.

3.5.34 ellsigma$(E, z, \{flag = 0\})$: E being given by `ellinit`, returns the value at z of the Weierstrass σ function of the period lattice L of E:

$$\sigma(z, L) = z \prod_{\omega \in L^*} \left(1 - \frac{z}{\omega}\right) e^{\frac{z}{\omega} + \frac{z^2}{2\omega^2}}$$

Alternatively, one can input a lattice basis $[\omega_1, \omega_2]$ directly instead of E.

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 `GEN ellsigma(GEN E, GEN z, long flag, long prec)`.

3.5.35 ellsub$(E, z1, z2)$: difference of the points $z1$ and $z2$ on the elliptic curve corresponding to E.

The library syntax is `GEN subell(GEN E, GEN z1, GEN z2)`.

3.5.36 elltaniyama$(E, \{d = seriesprecision\})$: computes the modular parametrization of the elliptic curve E, where E is a *smallell* as output by `ellinit`, in the form of a two-component vector $[u, v]$ of power series, given to d significant terms (`seriesprecision` by default). 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 `GEN elltaniyama(GEN E, long precdl)`.

3.5.37 elltatepairing(E, P, Q, m): Computes the Tate pairing of the two points P and Q on the elliptic curve E. The point P must be of m-torsion.

The library syntax is `GEN elltatepairing(GEN E, GEN P, GEN Q, GEN m)`.

3.5.38 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 a *smallell*.

The library syntax is `GEN elltors0(GEN E, long flag)`. Also available is `GEN elltors(GEN E)` for `elltors(E, 0)`.

3.5.39 ellweilpairing(E, P, Q, m): Computes the Weil pairing of the two points of m-torsion P and Q on the elliptic curve E.

The library syntax is `GEN ellweilpairing(GEN E, GEN P, GEN Q, GEN m)`.

3.5.40 ellwp$(E, \{z = x\}, \{flag = 0\}, \{d = seriesprecision\})$: 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 series is given to d significant terms (`seriesprecision` by default).

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 `GEN ellwp0(GEN E, GEN z = NULL, long flag, long precdl, long prec)`. Also available is `GEN weipell(GEN E, long precdl)` for the power series.

3.5.41 ellzeta(E, z): E being given by `ellinit`, returns the value at z of the Weierstrass ζ function of the period lattice L of E:

$$\zeta(z, L) = \frac{1}{z} + z^2 \sum_{\omega \in L^*} \frac{1}{\omega^2(z - \omega)}.$$

Alternatively, one can input a lattice basis $[\omega_1, \omega_2]$ directly instead of E.

```
? e = ellinit([0,0,0,1,0]);
? ellzeta(e, e.omega[1]/2)
%2 = 0.84721308479397908660649912340 + 4.417621070 E-29*I
? 2*ellzeta([1,I], 1/2)
%3 = 3.14159265358979323846264338400 + 0.E-37*I
```

The quasi-periods of ζ, such that

$$\zeta(z + a\omega_1 + b\omega_2) = \zeta(z) + a\eta_1 + b\eta_2$$

for integers a and b are obtained directly as $\eta_i = 2\zeta(\omega_i/2)$ or using `elleta`.

The library syntax is `GEN ellzeta(GEN E, GEN z, long prec)`.

3.5.42 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 `GEN pointell(GEN E, GEN z, long 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` or `idealfactor`.

• a `t_MAT`, square and in Hermite Normal Form (or at least upper triangular with non-negative coefficients), whose columns represent a $\mathbf{Z}$-basis of the ideal.

One may use `idealhnf` to convert any ideal to the last (preferred) format.

• an *extended ideal* is a 2-component vector $[I, t]$, where I is an ideal as above and t is an algebraic number, representing the ideal $(t)I$. This is useful whenever `idealred` is involved,

implicitly working in the ideal class group, while keeping track of principal ideals. Ideal operations suitably update the principal part when it makes sense (in a multiplicative context), e.g. using `idealmul` on $[I,t]$, $[J,u]$, we obtain $[IJ,tu]$. When it does not make sense, the extended part is silently discarded, e.g. using `idealadd` with the above input produces $I+J$.

The "principal part" t in an extended ideal may be represented in any of the above forms, and *also* as a factorization matrix (in terms of number field elements, not ideals!), possibly the empty matrix `[;]` representing 1. In the latter case, elements stay in factored form, or *famat* for *fa*ctorization *mat*rix, which is a convenient way to avoid coefficient explosion. To recover the conventional expanded form, try `nffactorback`; but many functions already accept *famat*s as input, for instance `ideallog`, so expanding huge elements should never be necessary.

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, $G=\oplus_{i\le n}(\mathbf{Z}/d_i\mathbf{Z})g_i$, 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 in HNF, whose columns express generators of H on the given generators g_i. Note that the determinant of that matrix is equal to the index $(G:H)$.

3.6.4 Relative extensions

We now have a look at data structures associated to relative extensions of number fields L/K, and to projective $\mathbf{Z}_K$-modules. When defining a relative extension L/K, the *nf* associated to the base field K must be defined by a variable having a lower priority (see Section 2.5.3) than the variable defining the extension. For example, you may use the variable name y to define the base field K, and x to define the relative extension L/K.

3.6.4.1 Basic definitions.

• *rnf* denotes a relative number field, i.e. a data structure output by `rnfinit`, associated to the extension L/K. The *nf* associated to be base field K is `rnf.nf`.

• A *relative matrix* is an $m\times n$ matrix whose entries are elements of K, in any form. Its m columns A_j represent elements in K^n.

• An *ideal list* is a row vector of fractional ideals of the number field *nf*.

• A *pseudo-matrix* is a 2-component row vector (A,I) where A is a relative $m\times n$ matrix and I an ideal list of length n. If $I=\{\mathbf{a}_1,\ldots,\mathbf{a}_n\}$ and the columns of A are $(A_1,\ldots,A_n)$, this data defines the torsion-free (projective) $\mathbf{Z}_K$-module $\mathbf{a}_1A_1\oplus\mathbf{a}_nA_n$.

• An *integral pseudo-matrix* is a 3-component row vector $\mathrm{w}(A,I,J)$ where $A=(a_{i,j})$ is an $m\times n$ relative matrix and $I=(\mathbf{b}_1,\ldots,\mathbf{b}_m)$, $J=(\mathbf{a}_1,\ldots,\mathbf{a}_n)$ are ideal lists, such that $a_{i,j}\in\mathbf{b}_i\mathbf{a}_j^{-1}$ for all i,j. This data defines two abstract projective $\mathbf{Z}_K$-modules $N=\mathbf{a}_1\omega_1\oplus\cdots\oplus\mathbf{a}_n\omega_n$ in K^n, $P=\mathbf{b}_1\eta_1\oplus\cdots\oplus\mathbf{b}_m\eta_m$ in K^m, and a $\mathbf{Z}_K$-linear map $f:N\to P$ given by

$$f(\sum\alpha_j\omega_j)=\sum_i\Big(a_{i,j}\alpha_j\Big)\eta_i.$$

This data defines the $\mathbf{Z}_K$-module $M = P/f(N)$.

• Any *projective* $\mathbf{Z}_K$-module M of finite type in K^m can be given by a pseudo matrix (A, I).

• An arbitrary $\mathbf{Z}_K$ modules of finite type in K^m, with non-trivial torsion, is given by an integral pseudo-matrix (A, I, J)

3.6.4.2 Pseudo-bases, determinant.

• The pair (A, I) is a *pseudo-basis* of the module it generates if the $\mathbf{a}_j$ are non-zero, and the A_j are K-linearly independent. We call n the *size* of the pseudo-basis. If A is a relative matrix, the latter condition means it is square with non-zero determinant; 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.

• For instance, the relative integer basis `rnf.zk` is a pseudo-basis (A, I) of $\mathbf{Z}_L$, where $A =$ `rnf.zk`[1] is a vector of elements of L, which are K-linearly independent. Most *rnf* routines return and handle $\mathbf{Z}_K$-modules contained in L (e.g. $\mathbf{Z}_L$-ideals) via a pseudo-basis (A', I'), where A' is a relative matrix representing a vector of elements of L in terms of the fixed basis `rnf.zk[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, and nowhere else.

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. It 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 A, B, C, where the defining set $[A, B, C]$ can have any of the following forms: $[bnr]$, $[bnr, subgroup]$, $[bnf, mod]$, $[bnf, mod, subgroup]$. The last two forms are kept for backward compatibility, but no longer serve any real purpose (see example below); no newly written function will accept them.

• *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.

• *mod* 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.

```
? K = bnfinit(y^2+1);
```

```
? bnr = bnrinit(K, 13)
? %.clgp
%3 = [36, [12, 3]]
? bnrdisc(bnr); \\ discriminant of the full ray class field
? bnrdisc(bnr, [3,1;0,1]); \\ discriminant of cyclic cubic extension of K
```

We could have written directly

```
? bnrdisc(K, 13);
? bnrdisc(K, 13, [3,1;0,1]);
```

avoiding one `bnrinit`, but this would actually be slower since the `bnrinit` is called internally anyway. And now twice!

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*) : fundamental units.
`index` (*bnr*, *bnf*, *nf*) : index of the power order in the ring of integers.
`mod` (*bnr*) : modulus.
`nf` (*bnr*, *bnf*, *nf*) : number field.
`pol` (*bnr*, *bnf*, *nf*) : defining polynomial.
`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.
`sign` (*bnr*, *bnf*, *nf*) : signature $[r1, r2]$.
`t2` (*bnr*, *bnf*, *nf*) : the T_2 matrix (see `nfinit`).
`tu` (*bnr*, *bnf*) : a generator for the torsion units.
`zk` (*bnr*, *bnf*, *nf*) : integral basis, i.e. a $\mathbf{Z}$-basis of the maximal order.
`zkst` (*bnr*) : structure of $(\mathbf{Z}_K/m)^*$.

Deprecated. The following member functions are still available, but deprecated and should not be used in new scripts :
`futu` (*bnr*, *bnf*) : $[u_1, ..., u_r, w]$, (u_i) is a vector of fundamental units, w generates the torsion units.
`tufu` (*bnr*, *bnf*) : $[w, u_1, ..., u_r]$, (u_i) is a vector of fundamental units, w generates the torsion units.

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

$$tech = [c_1, c_2, nrpid],$$

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 *tech* interface is obsolete and you should not tamper with these parameters. Indeed, from version 2.4.0 on,

• the results are always rigorous under GRH (before that version, they relied on a heuristic strengthening, hence the need for overrides).

• the influence of these parameters on execution time and stack size is marginal. They *can* be useful to fine-tune and experiment with the `bnfinit` code, but you will be better off modifying all tuning parameters in the C code (there are many more than just those three). We nevertheless describe it for completeness.

The numbers $c_1 \leq c_2$ are positive real numbers. For $i = 1, 2$, let $B_i = c_i(\log |d_K|)^2$, and denote by $S(B)$ the set of maximal ideals of K whose norm is less than B. We want $S(B_1)$ to generate $\mathrm{Cl}(K)$ and hope that $S(B_2)$ can be *proven* to generate $\mathrm{Cl}(K)$.

More precisely, $S(B_1)$ is a factorbase used to compute a tentative $\mathrm{Cl}(K)$ by generators and relations. We then check explicitly, using essentially `bnfisprincipal`, that the elements of $S(B_2)$

belong to the span of $S(B_1)$. Under the assumption that $S(B_2)$ generates $\mathrm{Cl}(K)$, we are done. User-supplied c_i are only used to compute initial guesses for the bounds B_i, and the algorithm increases them until one can *prove* under GRH that $S(B_2)$ generates $\mathrm{Cl}(K)$. A uniform result of Bach says that $c_2 = 12$ is always suitable, but this bound is very pessimistic and a direct algorithm due to Belabas-Diaz-Friedman is used to check the condition, assuming GRH. The default values are $c_1 = c_2 = 0.3$.

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 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. You must use `bnfcertify` to certify the computations unconditionally.

Remarks.

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_1 and c_2, and provide a vector $[c_1, c_2, nrpid]$.

Note also that you can use an *nf* instead of P, which avoids recomputing the integral basis and analogous quantities.

3.6.8 bnfcertify$(bnf, \{flag = 0\})$: *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.

If flag is present, only certify that the class group is a quotient of the one computed in bnf (much simpler in general).

The library syntax is `long bnfcertify0(GEN bnf, long flag )`. Also available is `GEN bnfcertify(GEN bnf)` (*flag* $= 0$).

3.6.9 bnfcompress(bnf): computes a compressed version of *bnf* (from `bnfinit`), a "small Buchmann's number field" (or *sbnf* for short) which contains enough information to recover a full *bnf* vector very rapidly, but which is much smaller and hence easy to store and print. Calling `bnfinit` on the result recovers a true `bnf`, in general different from the original. Note that an *snbf* is useless for almost all purposes besides storage, and must be converted back to *bnf* form before use; for instance, no `nf*`, `bnf*` or member function accepts them.

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 `bnf.pol`, $v[2]$ is the number of real embeddings `bnf.sign[1]`, $v[3]$ is `bnf.disc`, $v[4]$ is `bnf.zk`, $v[5]$ is the list of roots `bnf.roots`, $v[7]$ 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.

All the components are exact (integral or rational), except for the roots in $v[5]$.

The library syntax is `GEN bnfcompress(GEN bnf)`.

3.6.10 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 `GEN decodemodule(GEN nf, GEN m)`.

3.6.11 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 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` 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.

tech is a technical vector (empty by default, see 3.6.7). Careful use of this parameter may speed up your computations, but it is mostly obsolete and you should leave it alone.

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 $(\mathfrak{p}_i)_{1\le i\le r}$.

bnf[2] contains the matrix B, i.e. the matrix containing the expressions of the prime ideal factorbase in terms of the $\mathfrak{p}_i$. 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).

The library syntax is `GEN bnfinit0(GEN P, long flag, GEN tech = NULL, long prec)`.

Also available is `GEN Buchall(GEN P, long flag, long prec)`, corresponding to `tech = NULL`, where `flag` is either 0 (default) or `nf_FORCE` (insist on finding fundamental units). The function `GEN Buchall_param(GEN P, double c1, double c2, long nrpid, long flag, long prec)` gives direct access to the technical parameters.

3.6.12 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 `GEN bnfisintnorm(GEN bnf, GEN x)`.

3.6.13 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 `GEN bnfisnorm(GEN bnf, GEN x, long flag)`.

3.6.14 bnfisprincipal$(bnf, x, \{flag = 1\})$: *bnf* being the number field data output by `bnfinit`, and x being an ideal, this function tests whether the ideal is principal or not. The result is more complete than a simple true/false answer and solves general discrete logarithm problem. Assume the class group is $\oplus(\mathbf{Z}/d_i\mathbf{Z})g_i$ (where the generators g_i and their orders d_i are respectively given by `bnf.gen` and `bnf.cyc`). The routine returns a row vector $[e, t]$, where e is a vector of exponents $0 \le e_i < d_i$, and t is a number field element such that

$$x = (t)\prod_i g_i^{e_i}.$$

For *given* g_i (i.e. for a given `bnf`), the e_i are unique, and t is unique modulo units.

In particular, x is principal if and only if e is the zero vector. Note that the empty vector, which is returned when the class number is 1, is considered to be a zero vector (of dimension 0).

```
? K = bnfinit(y^2+23);
? K.cyc
%2 = [3]
? K.gen
%3 = [[2, 0; 0, 1]]           \\ a prime ideal above 2
? P = idealprimedec(K,3)[1]; \\ a prime ideal above 3
? v = bnfisprincipal(K, P)
%5 = [[2]~, [3/4, 1/4]~]
? idealmul(K, v[2], idealfactorback(K, K.gen, v[1]))
%6 =
[3 0]

[0 1]
? % == idealhnf(K, P)
%7 = 1
```

The binary digits of *flag*mean:

• 1: If set, outputs $[e, t]$ as explained above, otherwise returns only e, which is much easier to compute. The following idiom only tests whether an ideal is principal:

```
is_principal(bnf, x) = !bnfisprincipal(bnf,x,0);
```

• 2: It may not be possible to recover t, given the initial accuracy to which `bnf` was computed. In that case, a warning is printed and t is set equal to the empty vector `[]~`. If this bit is set, increase the precision and recompute needed quantities until t can be computed. Warning: setting this may induce *very* lengthy computations.

The library syntax is `GEN bnfisprincipal0(GEN bnf, GEN x, long flag)`. Instead of the above hardcoded numerical flags, one should rather use an or-ed combination of the symbolic flags `nf_GEN` (include generators, possibly a place holder if too difficult) and `nf_FORCE` (insist on finding the generators).

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 `GEN bnfissunit(GEN bnf, GEN sfu, GEN x)`.

3.6.16 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 $i = r+1$.

Note that *bnf* need not contain the fundamental unit explicitly:

```
? setrand(1); bnf = bnfinit(x^2-x-100000);
? bnf.fu
  ***   at top-level: bnf.fu
  ***                       ^--
  *** _.fu: missing units in .fu.
? u = [11983616564425078999046283595002287166517812761131616131167, \
       379554884019013781006303254896369154068336082609238336]~;
? bnfisunit(bnf, u)
%3 = [-1, Mod(0, 2)]~
```

The given u is the inverse of the fundamental unit implicitly stored in *bnf*. In this case, the fundamental unit was not computed and stored in algebraic form since the default accuracy was too low. (Re-run the command at "g1 or higher to see such diagnostics.)

The library syntax is `GEN bnfisunit(GEN bnf, GEN x)`.

3.6.17 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 `GEN buchnarrow(GEN bnf)`.

3.6.18 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)=
{ my(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)=
{ my(vu = bnf.tufu, ex = tpuexpo(bnf));

  vector(#ex-1, i, factorback(vu, ex[,i+1]))  \\ ex[,1] is 1
```

}

The library syntax is `GEN signunits(GEN bnf)`.

3.6.19 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 `GEN bnfsunit(GEN bnf, GEN S, long prec)`.

3.6.20 bnrL1$(bnr, \{subgrp\}, \{flag = 0\})$: *bnr* being the number field data which is output by `bnrinit(,,1)` and *subgrp* 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 `GEN bnrL1(GEN bnr, GEN subgrp = NULL, long flag, long prec)`.

3.6.21 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. One can input the associated *bid* for I instead of the module itself, saving some time.

This function 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 `GEN bnrclassno(GEN bnf, GEN I)`.

3.6.22 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 `GEN bnrclassnolist(GEN bnf, GEN list)`.

3.6.23 bnrconductor$(A, \{B\}, \{C\}, \{flag = 0\})$: conductor f of the subfield of a ray class field as defined by $[A, B, C]$ (of type `[`*bnr*`]`, `[`*bnr*`,` *subgroup*`]`, `[`*bnf*`,` *modulus*`]` or `[`*bnf*`,` *modulus*`,` *subgroup*`]`, Section 3.6.5)

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 `GEN bnrconductor0(GEN A, GEN B = NULL, GEN C = NULL, long flag)`.

Also available is `GEN bnrconductor(GEN bnr, GEN H, long flag)`

3.6.24 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 `GEN bnrconductorofchar(GEN bnr, GEN chi)`.

3.6.25 bnrdisc$(A, \{B\}, \{C\}, \{flag = 0\})$: A, B, C defining a class field L over a ground field K (of type `[bnr]`, `[bnr, subgroup]`, `[bnf, modulus]` or `[bnf, modulus, subgroup]`, Section 3.6.5), outputs data $[N, r_1, D]$ giving the discriminant and signature of L, depending on the binary digits of *flag*:

• 1: if this bit is unset, output absolute data related to $L/\mathbf{Q}$: N is the absolute degree $[L : \mathbf{Q}]$, r_1 the number of real places of L, and D the discriminant of $L/\mathbf{Q}$. Otherwise, output relative data for L/K: N is the relative degree $[L : K]$, 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 N), and D is the relative discriminant ideal of L/K.

• 2: if this bit is set and if the modulus is not the conductor of L, only return 0.

The library syntax is `GEN bnrdisc0(GEN A, GEN B = NULL, GEN C = NULL, long flag)`.

3.6.26 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 awkward 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 `GEN bnrdisclist0(GEN bnf, GEN bound, GEN arch = NULL)`.

3.6.27 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`, `.gen` 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 $(\mathbf{Z}_K/f)^*$; 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 `GEN bnrinit0(GEN bnf, GEN f, long flag)`. Instead the above hard-coded numerical flags, one should rather use `GEN Buchray(GEN bnf, GEN module, long flag)` where flag is an or-ed combination of `nf_GEN` (include generators) and `nf_INIT` (if omitted, return just the cardinal of the ray class group and its structure), possibly 0.

3.6.28 bnrisconductor($A, \{B\}, \{C\}$): A, B, C represent an extension of the base field, given by class field theory (see Section 3.6.5). Outputs 1 if this modulus is the conductor, and 0 otherwise. This is slightly faster than `bnrconductor`.

The library syntax is `long bnrisconductor0(GEN A, GEN B = NULL, GEN C = NULL)`.

3.6.29 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`$(,,0)$ If x is not coprime to the modulus of *bnr* the result is undefined.

The library syntax is `GEN bnrisprincipal(GEN bnr, GEN x, long flag)`. Instead of hard-coded numerical flags, one should rather use `GEN isprincipalray(GEN bnr, GEN x)` for `flag` $= 0$, and if you want generators:

```
bnrisprincipal(bnr, x, nf_GEN)
```

3.6.30 bnrrootnumber$(bnr, chi, \{flag = 0\})$: if $\chi = chi$ is a character over bnr, not necessarily primitive, 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 - x - 57);
bnr = bnrinit(bnf, [7,[1,1]], 1);
bnrrootnumber(bnr, [2,1])
```

returns the root number of the character χ of $\mathrm{Cl}_{7\infty_1\infty_2}(\mathbf{Q}(\sqrt{229}))$ defined by $\chi(g_1^a g_2^b) = \zeta_1^{2a}\zeta_2^b$. Here g_1, g_2 are the generators of the ray-class group given by `bnr.gen` and $\zeta_1 = e^{2i\pi/N_1}, \zeta_2 = e^{2i\pi/N_2}$ where N_1, N_2 are the orders of g_1 and g_2 respectively ($N_1 = 6$ and $N_2 = 3$ as `bnr.cyc` readily tells us).

The library syntax is `GEN bnrrootnumber(GEN bnr, GEN chi, long flag, long prec)`.

3.6.31 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 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);
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 routine uses Stark units and needs to find a suitable auxiliary 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.

Even if it exists, the auxiliary conductor may be so large that later computations become unfeasible. (And of course, Stark's conjecture may simply be wrong.) In case of difficulties, try `rnfkummer`:

```
? bnr = bnrinit(bnfinit(y^8-12*y^6+36*y^4-36*y^2+9,1), 2, 1);
? bnrstark(bnr)
  ***   at top-level: bnrstark(bnr)
  ***                 ^-------------
  *** bnrstark: need 3919350809720744 coefficients in initzeta.
  *** Computation impossible.
```

```
? lift( rnfkummer(bnr) )
time = 24 ms.
%2 = x^2 + (1/3*y^6 - 11/3*y^4 + 8*y^2 - 5)
```

The library syntax is `GEN bnrstark(GEN bnr, GEN subgroup = NULL, long prec)`.

3.6.32 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 `GEN dirzetak(GEN nf, GEN b)`.

3.6.33 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 2.5.3). 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)
  ***   at top-level: factornf(x^2+Mod(z,z
  ***                 ^--------------------
  *** factornf: inconsistent data in rnf function.
? factornf(x^2 + z, y^2+1)
  ***   at top-level: factornf(x^2+z,y^2+1
  ***                 ^--------------------
  *** factornf: incorrect variable in rnf function.
```

The library syntax is `GEN polfnf(GEN x, GEN t)`.

3.6.34 galoisexport$(gal, \{flag\})$: *gal* being be a Galois group 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`.

To *import* a GAP permutation into gp (for `galoissubfields` for instance), the following GAP function may be useful :

```
PermToGP := function(p, n)
  return Permuted([1..n],p);
end;
```

```
gap> p:= (1,26)(2,5)(3,17)(4,32)(6,9)(7,11)(8,24)(10,13)(12,15)(14,27)
  (16,22)(18,28)(19,20)(21,29)(23,31)(25,30)
gap> PermToGP(p,32);
[ 26, 5, 17, 32, 2, 9, 11, 24, 6, 13, 7, 15, 10, 27, 12, 22, 3, 28, 20, 19,
  29, 16, 31, 8, 30, 1, 14, 18, 21, 25, 23, 4 ]
```

The library syntax is `GEN galoisexport(GEN gal, long flag)`.

3.6.35 galoisfixedfield$(gal, perm, \{flag\}, \{v = y\})$: *gal* being be a Galois group as output by `galoisinit` and *perm* an element of *gal.group*, a vector of such elements or a subgroup of *gal* as returned by galoissubgroups, 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 2.5.3). 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 `GEN galoisfixedfield(GEN gal, GEN perm, long flag, long v = -1)`, where `v` is a variable number.

3.6.36 galoisgetpol$(a, \{b\}, \{s\})$: Query the galpol package for a polynomial with Galois group isomorphic to GAP4(a,b), totally real if $s = 1$ (default) and totally complex if $s = 2$. The output is a vector [`pol`, `den`] where `pol` is the polynomial and `den` is the common denominator of the conjugates expressed as a polynomial in a root of `pol`, which can be passed as an optional argument to `galoisinit` and `nfgaloisconj` as follows:

```
V=galoisgetpol(8,4,1);
G=galoisinit(V[1], V[2])  \\ passing V[2] speeds up the computation
```

If b and s are omitted, return the number of isomorphic class of groups of order a.

The library syntax is `GEN galoisgetpol(long a, long b, long s)`. Also available is `GEN galoisnbpol(long a)` when b and s are omitted.

3.6.37 galoisidentify(gal): *gal* being be a Galois group 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 `GEN galoisidentify(GEN gal)`.

3.6.38 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 below; returns 0 otherwise. Hence this permits to quickly check whether a polynomial of order strictly less than 36 is Galois or not.

The algorithm used is an improved version of the paper "An efficient algorithm for the computation of Galois automorphisms", Bill Allombert, Math. Comp, vol. 73, 245, 2001, pp. 359–375.

A group G is said to be "weakly" super-solvable if there exists a normal series

$\{1\} = H_0 \lhd H_1 \lhd \cdots \lhd H_{n-1} \lhd H_n$

such that each H_i is normal in G and for $i < n$, each quotient group H_{i+1}/H_i is cyclic, and either $H_n = G$ (then G is super-solvable) or G/H_n is isomorphic to either A_4 or S_4.

In practice, almost all small groups 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.

This function 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 Vandermonde 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_n be as above, we have the following properties:

• if $G/H_n \simeq A_4$ then $[o_1, \ldots, o_g]$ ends by $[2, 2, 3]$.

• if $G/H_n \simeq S_4$ then $[o_1, \ldots, o_g]$ ends by $[2, 2, 3, 2]$.

• for $1 \leq i \leq g$ the subgroup of G generated by $[s_1, \ldots, s_g]$ is normal, with the exception of $i = g - 2$ in the A_4 case and of $i = g - 3$ in the S_A case.

• the relative order o_i of s_i is its order in the quotient group $G/\langle s_1, \ldots, s_{i-1}\rangle$, with the same exceptions.

• for any $x \in G$ there exists a unique family $[e_1, \ldots, e_g]$ such that (no exceptions):

– for $1 \leq i \leq g$ we have $0 \leq e_i < o_i$

– $x = g_1^{e_1} g_2^{e_2} \ldots g_n^{e_n}$

If present *den* must be a suitable value for *gal*[5].

The library syntax is `GEN galoisinit(GEN pol, GEN den = NULL)`.

3.6.39 galoisisabelian(*gal*, {*flag* = 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 `GEN galoisisabelian(GEN gal, long flag)`.

3.6.40 galoisisnormal(*gal*, *subgrp*): *gal* being as output by `galoisinit`, and *subgrp* a subgroup of *gal* as output by `galoissubgroups`,return 1 if *subgrp* is a normal subgroup of *gal*, else return 0.

This command also accepts subgroups returned by `galoissubgroups`.

The library syntax is `long galoisisnormal(GEN gal, GEN subgrp)`.

3.6.41 galoispermtopol(*gal*, *perm*): *gal* being a Galois group 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 `GEN galoispermtopol(GEN gal, GEN perm)`.

3.6.42 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):

```
polsubcyclo(n, d = -1)=
{ my(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 `GEN galoissubcyclo(GEN N, GEN H = NULL, long fl, long v = -1)`, where `v` is a variable number.

3.6.43 galoissubfields$(G, \{flags = 0\}, \{v\})$: outputs 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 `GEN galoissubfields(GEN G, long flags, long v = -1)`, where `v` is a variable number.

3.6.44 galoissubgroups(G): outputs 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 `GEN galoissubgroups(GEN G)`.

3.6.45 idealadd(nf, x, y): sum of the two ideals x and y in the number field nf. The result is given in HNF.

```
? K = nfinit(x^2 + 1);
? a = idealadd(K, 2, x + 1)  \\ ideal generated by 2 and 1+I
%2 =
[2 1]

[0 1]
? pr = idealprimedec(K, 5)[1];  \\ a prime ideal above 5
? idealadd(K, a, pr)     \\ coprime, as expected
%4 =
[1 0]

[0 1]
```

This function cannot be used to add arbitrary **Z**-modules, since it assumes that its arguments are ideals:

```
? b = Mat([1,0]~);
? idealadd(K, b, b)     \\ only square t_MATs represent ideals
*** idealadd: non-square t_MAT in idealtyp.
? c = [2, 0; 2, 0]; idealadd(K, c, c)   \\ non-sense
%6 =
[2 0]

[0 2]
? d = [1, 0; 0, 2]; idealadd(K, d, d)   \\ non-sense
%7 =
[1 0]

[0 1]
```

In the last two examples, we get wrong results since the matrices c and d do not correspond to an ideal : the **Z**-span of their columns (as usual interpreted as coordinates with respect to the integer basis `K.zk`) is not an O_K-module. To add arbitrary **Z**-modules generated by the columns of matrices A and B, use `mathnf(concat(A,B))`.

The library syntax is `GEN idealadd(GEN nf, GEN x, GEN y)`.

3.6.46 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 `GEN idealaddtoone0(GEN nf, GEN x, GEN y = NULL)`.

3.6.47 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 $\mathfrak{p}$ such that the valuation of x at $\mathfrak{p}$ is non-zero, we have $v_{\mathfrak{p}}(\alpha) = v_{\mathfrak{p}}(x)$, and $v_{\mathfrak{p}}(\alpha) \ge 0$ for all other $\mathfrak{p}$.

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 $\mathfrak{p}$ occurring in x, $v_{\mathfrak{p}}(\alpha)$ is equal to the exponent of $\mathfrak{p}$ in x, and for all other prime ideals, $v_{\mathfrak{p}}(\alpha) \ge 0$. 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 `GEN idealappr0(GEN nf, GEN x, long flag)`.

3.6.48 idealchinese(nf, x, y): x being a prime ideal factorization (i.e. a 2 by 2 matrix whose first column contains 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

$v_{\mathfrak{p}}(b - y_{\mathfrak{p}}) \ge v_{\mathfrak{p}}(x)$ for all prime ideals in x and $v_{\mathfrak{p}}(b) \ge 0$ for all other $\mathfrak{p}$.

The library syntax is `GEN idealchinese(GEN nf, GEN x, GEN y)`.

3.6.49 idealcoprime(nf, x, y): given two integral ideals x and y in the number field nf, returns a β in the field, such that $\beta \cdot x$ is an integral ideal coprime to y.

The library syntax is `GEN idealcoprime(GEN nf, GEN x, GEN y)`.

3.6.50 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 `GEN idealdiv0(GEN nf, GEN x, GEN y, long flag)`. Also available are `GEN idealdiv(GEN nf, GEN x, GEN y)` (*flag* $= 0$) and `GEN idealdivexact(GEN nf, GEN x, GEN y)` (*flag* $= 1$).

3.6.51 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 `GEN idealfactor(GEN nf, GEN x)`.

3.6.52 idealfactorback$(nf, f, \{e\}, \{flag = 0\})$: gives back the ideal corresponding to a factorization. The integer 1 corresponds to the empty factorization. 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 1s: we return the product of the $f[i]$. Finally, f can be a regular factorization, as produced by `idealfactor`.

```
? nf = nfinit(y^2+1); idealfactor(nf, 4 + 2*y)
%1 =
[[2, [1, 1]~, 2, 1, [1, 1]~] 2]

[[5, [2, 1]~, 1, 1, [-2, 1]~] 1]

? idealfactorback(nf, %)
%2 =
[10 4]

[0  2]

? f = %1[,1]; e = %1[,2]; idealfactorback(nf, f, e)
%3 =
[10 4]

[0  2]

? % == idealhnf(nf, 4 + 2*y)
%4 = 1
```

If `flag` is non-zero, perform ideal reductions (`idealred`) along the way. This is most useful if the ideals involved are all *extended* ideals (for instance with trivial principal part), so that the principal parts extracted by `idealred` are not lost. Here is an example:

```
? f = vector(#f, i, [f[i], [;]]);  \\ transform to extended ideals
? idealfactorback(nf, f, e, 1)
%6 = [[1, 0; 0, 1], [2, 1; [2, 1]~, 1]]
? nffactorback(nf, %[2])
%7 = [4, 2]~
```

The extended ideal returned in `%6` is the trivial ideal 1, extended with a principal generator given in factored form. We use `nffactorback` to recover it in standard form.

The library syntax is `GEN idealfactorback(GEN nf, GEN f, GEN e = NULL, long flag )`.

3.6.53 idealfrobenius(nf, gal, pr): Let K be the number field defined by nf and assume $K/\mathbf{Q}$ be a Galois extension with Galois group given `gal=galoisinit(nf)`, and that pr is the prime ideal $\mathfrak{P}$ in prid format, and that $\mathfrak{P}$ is unramified. This function returns a permutation of `gal.group` which defines the automorphism $\sigma = \left(\frac{\mathfrak{P}}{K/\mathbf{Q}}\right)$, i.e the Frobenius element associated to $\mathfrak{P}$. If p is the unique prime number in $\mathfrak{P}$, then $\sigma(x) \equiv x^p \bmod \mathbf{P}$ for all $x \in \mathbf{Z}_K$.

```
? nf = nfinit(polcyclo(31));
? gal = galoisinit(nf);
? pr = idealprimedec(nf,101)[1];
? g = idealfrobenius(nf,gal,pr);
? galoispermtopol(gal,g)
%5 = x^8
```

This is correct since $101 \equiv 8 \bmod 31$.

The library syntax is `GEN idealfrobenius(GEN nf, GEN gal, GEN pr)`.

3.6.54 idealhnf$(nf, a, \{b\})$: gives the Hermite normal form of the ideal $a\mathbf{Z}_K + b\mathbf{Z}_K$, where a and b are elements of the number field K defined by `nf`.

```
? nf = nfinit(y^3 - 2);
? idealhnf(nf, 2, y+1)
%2 =
[1 0 0]

[0 1 0]

[0 0 1]
? idealhnf(nf, y/2, [0,0,1/3]~)
%3 =
[1/3 0 0]

[0 1/6 0]

[0 0 1/6]
```

If b is omitted, returns the HNF of the ideal defined by a: a may be an algebraic number (defining a principal ideal), a maximal ideal (as given by `idealprimedec` or `idealfactor`), or a matrix whose columns give generators for the ideal. This last format is a little complicated, but useful to reduce general modules to the canonical form once in a while:

• if strictly less than $N = [K : \mathbf{Q}]$ generators are given, a is the $\mathbf{Z}_K$-module they generate,

• if N or more are given, it is *assumed* that they form a $\mathbf{Z}$-basis (that the matrix has maximal rank N). This acts as `mathnf` since the $\mathbf{Z}_K$-module structure is (taken for granted hence) not taken into account in this case.

```
? idealhnf(nf, idealprimedec(nf,2)[1])
%4 =
[2 0 0]

[0 1 0]

[0 0 1]
? idealhnf(nf, [1,2;2,3;3,4])
%5 =
[1 0 0]

[0 1 0]

[0 0 1]
```

The library syntax is `GEN idealhnf0(GEN nf, GEN a, GEN b = NULL)`. Also available is `GEN idealhnf(GEN nf, GEN a)`.

3.6.55 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) =
{ my(Ker = matkerint(concat(A,B));
  mathnf( A * vecextract(Ker, Str("..", #A), "..") )
}
```

The library syntax is `GEN idealintersect(GEN nf, GEN A, GEN B)`.

3.6.56 idealinv(nf, x): inverse of the ideal x in the number field nf, given in HNF. If x is an extended ideal, its principal part is suitably updated: i.e. inverting $[I, t]$, yields $[I^{-1}, 1/t]$.

The library syntax is `GEN idealinv(GEN nf, GEN x)`.

3.6.57 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 `GEN ideallist0(GEN nf, long bound, long flag)`.

3.6.58 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 `GEN ideallistarch(GEN nf, GEN list, GEN arch)`.

3.6.59 ideallog(nf, x, bid): *nf* is a number field, *bid* is as output by `idealstar(nf, D, ...)` and x a non-necessarily integral element of *nf* which must have valuation equal to 0 at all prime ideals in the support of `D`. This function computes the discrete logarithm of x on the generators given in *bid*`.gen`. 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

$$x \equiv \prod_i g_i^{x_i} \pmod{{}^*D} \ .$$

Note that when the support of `D` contains places at infinity, this congruence implies also sign conditions on the associated real embeddings. See `znlog` for the limitations of the underlying discrete log algorithms.

The library syntax is `GEN ideallog(GEN nf, GEN x, GEN bid)`.

3.6.60 idealmin$(nf, ix, \{vdir\})$: *This function is useless and kept for backward compatibility only, use* `idealred`. Computes a pseudo-minimum of the ideal x in the direction *vdir* in the number field *nf*.

The library syntax is `GEN idealmin(GEN nf, GEN ix, GEN vdir = NULL)`.

3.6.61 idealmul$(nf, x, y, \{flag = 0\})$: ideal multiplication of the ideals x and y in the number field *nf*; the result is the ideal product in HNF. If either x or y are extended ideals, their principal part is suitably updated: i.e. multiplying $[I, t]$, $[J, u]$ yields $[IJ, tu]$; multiplying I and $[J, u]$ yields $[IJ, u]$.

```
? nf = nfinit(x^2 + 1);
? idealmul(nf, 2, x+1)
%2 =
[4 2]

[0 2]
? idealmul(nf, [2, x], x+1)        \\ extended ideal * ideal
%4 = [[4, 2; 0, 2], x]
? idealmul(nf, [2, x], [x+1, x])   \\ two extended ideals
%5 = [[4, 2; 0, 2], [-1, 0]~]
```

If *flag* is non-zero, reduce the result using `idealred`.

The library syntax is `GEN idealmul0(GEN nf, GEN x, GEN y, long flag)`.

See also `GEN idealmul(GEN nf, GEN x, GEN y)` ($flag = 0$) and `GEN idealmulred(GEN nf, GEN x, GEN y)` ($flag \neq 0$).

3.6.62 idealnorm(nf, x): computes the norm of the ideal x in the number field *nf*.

The library syntax is `GEN idealnorm(GEN nf, GEN x)`.

3.6.63 idealpow$(nf, x, k, \{flag = 0\})$: computes the k-th power of the ideal x in the number field nf; $k \in \mathbf{Z}$. If x is an extended ideal, its principal part is suitably updated: i.e. raising $[I, t]$ to the k-th power, yields $[I^k, t^k]$.

If *flag* is non-zero, reduce the result using `idealred`, *throughout the (binary) powering process*; in particular, this is *not* the same as as `idealpow`(nf, x, k) followed by reduction.

The library syntax is `GEN idealpow0(GEN nf, GEN x, GEN k, long flag)`.

See also `GEN idealpow(GEN nf, GEN x, GEN k)` and `GEN idealpows(GEN nf, GEN x, long k)` ($flag = 0$). Corresponding to $flag = 1$ is `GEN idealpowred(GEN nf, GEN vp, GEN k)`.

3.6.64 idealprimedec(nf, p): computes the prime ideal decomposition of the (positive) prime number p in the number field K represented by nf. If a non-prime p is given the result is undefined.

The result is a vector of *prid* structures, each representing one of the prime ideals above p in the number field nf. The representation `pr` $= [p, a, e, f, b]$ of a prime ideal means the following: a and b are algebraic integers in the maximal order $\mathbf{Z}_K$; the prime ideal is equal to $\mathfrak{p} = p\mathbf{Z}_K + a\mathbf{Z}_K$; e is the ramification index; f is the residual index; and b is such that $\mathfrak{p}^{-1} = \mathbf{Z}_K + b/p\mathbf{Z}_K$, which is used internally to compute valuations. The algebraic number a is guaranteed to have a valuation equal to 1 at the prime ideal (this is automatic if $e > 1$).

The components of `pr` should be accessed by member functions: `pr.p`, `pr.e`, `pr.f`, and `pr.gen` (returns the vector $[p, a]$):

```
? K = nfinit(x^3-2);
? L = idealprimedec(K, 5);
? #L       \\ 2 primes above 5 in Q(2^(1/3))
%3 = 2
? p1 = L[1]; p2 = L[2];
? [p1.e, p1.f]    \\ the first is unramified of degree 1
%4 = [1, 1]
? [p2.e, p2.f]    \\ the second is unramified of degree 2
%5 = [1, 2]
? p1.gen
%6 = [5, [2, 1, 0]~]
? nfbasistoalg(K, %[2])  \\ a uniformizer for p1
%7 = Mod(x + 2, x^3 - 2)
```

The library syntax is `GEN idealprimedec(GEN nf, GEN p)`.

3.6.65 idealramgroups(nf, gal, pr): Let K be the number field defined by nf and assume $K/\mathbf{Q}$ be a Galois extension with Galois group G given `gal=galoisinit(nf)`, and that pr is the prime ideal $\mathfrak{P}$ in prid format. This function returns a vector g of subgroups of `gal` as follow:

- `g[1]` is the decomposition group of $\mathfrak{P}$,
- `g[2]` is $G_0(\mathfrak{P})$, the inertia group of $\mathfrak{P}$,

and for $i \geq 2$,

- `g[i]` is $G_{i-2}(\mathfrak{P})$, the $i-2$-th ramification group of $\mathfrak{P}$.

The length of g is the number of non-trivial groups in the sequence, thus is 0 if $e = 1$ and $f = 1$, and 1 if $f > 1$ and $e = 1$.

```
? nf=nfinit(x^6+108);
? gal=galoisinit(nf);
? pr=idealprimedec(nf,2)[1];
? iso=idealramgroups(nf,gal,pr)[2]
%4 = [[Vecsmall([2, 3, 1, 5, 6, 4])], Vecsmall([3])]
? nfdisc(galoisfixedfield(gal,iso,1))
%5 = -3
```

The field fixed by the inertia group of 2 is not ramified at 2.

The library syntax is `GEN idealramgroups(GEN nf, GEN gal, GEN pr)`.

3.6.66 idealred$(nf, I, \{v = 0\})$: LLL reduction of the ideal I in the number field nf, along the direction v. The v parameter is best left omitted, but if it is present, it must be an `nf.r1` + `nf.r2`-component vector of *non-negative* integers. (What counts is the relative magnitude of the entries: if all entries are equal, the effect is the same as if the vector had been omitted.)

This function finds a "small" a in I (see the end for technical details). The result is the Hermite normal form of the "reduced" ideal $J = rI/a$, where r is the unique rational number such that J is integral and primitive. (This is usually not a reduced ideal in the sense of Buchmann.)

```
? K = nfinit(y^2+1);
? P = idealprimedec(K,5)[1];
? idealred(K, P)
%3 =
[1 0]

[0 1]
```

More often than not, a principal ideal yields the unit ideal as above. This is a quick and dirty way to check if ideals are principal, but it is not a necessary condition: a non-trivial result does not prove that the ideal is non-principal. For guaranteed results, see `bnfisprincipal`, which requires the computation of a full `bnf` structure.

If the input is an extended ideal $[I, s]$, the output is $[J, sa/r]$; this way, one can keep track of the principal ideal part:

```
? idealred(K, [P, 1])
%5 = [[1, 0; 0, 1], [-2, 1]~]
```

meaning that P is generated by $[-2, 1]$. The number field element in the extended part is an algebraic number in any form *or* a factorization matrix (in terms of number field elements, not ideals!). In the latter case, elements stay in factored form, which is a convenient way to avoid coefficient explosion; see also `idealpow`.

Technical note. The routine computes an LLL-reduced basis for the lattice I equipped with the quadratic form

$$||x||_v^2 = \sum_{i=1}^{r_1+r_2} 2^{v_i}\varepsilon_i|\sigma_i(x)|^2,$$

where as usual the σ_i are the (real and) complex embeddings and $\varepsilon_i = 1$, resp. 2, for a real, resp. complex place. The element a is simply the first vector in the LLL basis. The only reason you may want to try to change some directions and set some $v_i \neq 0$ is to randomize the elements found for a fixed ideal, which is heuristically useful in index calculus algorithms like `bnfinit` and `bnfisprincipal`.

Even more technical note. In fact, the above is a white lie. We do not use $||\cdot||_v$ exactly but a rescaled rounded variant which gets us faster and simpler LLLs. There's no harm since we are not using any theoretical property of a after all, except that it belongs to I and is "expected to be small".

The library syntax is `GEN idealred0(GEN nf, GEN I, GEN v = NULL)`.

3.6.67 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* $= 0$, 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.

The library syntax is `GEN idealstar0(GEN nf, GEN I, long flag)`. Instead the above hard-coded numerical flags, one should rather use `GEN Idealstar(GEN nf, GEN ideal, long flag)`, where `flag` is an or-ed combination of `nf_GEN` (include generators) and `nf_INIT` (return a full `bid`, not a group), possibly 0. This offers one more combination: gen, but no init.

3.6.68 idealtwoelt$(nf, x, \{a\})$: computes a two-element representation of the ideal x in the number field *nf*, combining a random search and an explicit approximation theorem. x is an ideal in any form (possibly an extended ideal, whose principal part 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. Unless x was given as a principal ideal, a is chosen to be the positive generator of $x \cap \mathbf{Z}$.

Note that when an explicit a is given, we must factor it and this may be costly. When a is omitted, we use a fast lazy factorization of $x \cap \mathbf{Z}$, yielding an algorithm in randomized polynomial time (and generally much faster in practice).

The library syntax is `GEN idealtwoelt0(GEN nf, GEN x, GEN a = NULL)`. Also available are `GEN idealtwoelt(GEN nf, GEN x)` and `GEN idealtwoelt2(GEN nf, GEN x, GEN a)`.

3.6.69 idealval(nf, x, pr): gives the valuation of the ideal x at the prime ideal pr in the number field *nf*, where pr is in `idealprimedec` format.

The library syntax is `long idealval(GEN nf, GEN x, GEN pr)`.

3.6.70 matalgtobasis(nf, x): *nf* being a number field in `nfinit` format, and x a (row or column) vector or matrix, apply `nfalgtobasis` to each entry of x.

The library syntax is `GEN matalgtobasis(GEN nf, GEN x)`.

3.6.71 matbasistoalg(nf, x): nf being a number field in `nfinit` format, and x a (row or column) vector or matrix, apply `nfbasistoalg` to each entry of x.

The library syntax is `GEN matbasistoalg(GEN nf, GEN x)`.

3.6.72 modreverse(z): let $z = $ `Mod(A,T)` be a polmod, and Q be its minimal polynomial, which must satisfy $\deg(Q) = \deg(T)$. Returns a "reverse polmod" `Mod(B, Q)`, which is a root of T.

This is quite useful when one changes the generating element in algebraic extensions:

```
? u = Mod(x, x^3 - x -1); v = u^5;
? w = modreverse(v)
%2 = Mod(x^2 - 4*x + 1, x^3 - 5*x^2 + 4*x - 1)
```

which means that $x^3 - 5x^2 + 4x - 1$ is another defining polynomial for the cubic field

$$\mathbf{Q}(u) = \mathbf{Q}[x]/(x^3 - x - 1) = \mathbf{Q}[x]/(x^3 - 5x^2 + 4x - 1) = \mathbf{Q}(v),$$

and that $u \to v^2 - 4v + 1$ gives an explicit isomorphism. From this, it is easy to convert elements between the $A(u) \in \mathbf{Q}(u)$ and $B(v) \in \mathbf{Q}(v)$ representations:

```
? A = u^2 + 2*u + 3; subst(lift(A), 'x, w)
%3 = Mod(x^2 - 3*x + 3, x^3 - 5*x^2 + 4*x - 1)
? B = v^2 + v + 1;   subst(lift(B), 'x, v)
%4 = Mod(26*x^2 + 31*x + 26, x^3 - x - 1)
```

If the minimal polynomial of z has lower degree than expected, the routine fails

```
? u = Mod(-x^3 + 9*x, x^4 - 10*x^2 + 1)
? modreverse(u)
  ***   at top-level: modreverse(u)
  ***                 ^-------------
  *** modreverse: reverse polmod does not exist: Mod(-x^3+9*x, x^4-10*x^2+1).
? minpoly(u)
%6 = x^2 - 8
```

The library syntax is `GEN modreverse(GEN z)`.

3.6.73 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 `LONG_MAX`, 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 `GEN newtonpoly(GEN x, GEN p)`.

3.6.74 nfalgtobasis(nf, x): Given an algebraic number x in the number field nf, transforms it to a column vector on the integral basis nf`.zk`.

```
? nf = nfinit(y^2 + 4);
? nf.zk
%2 = [1, 1/2*y]
? nfalgtobasis(nf, [1,1]~)
%3 = [1, 1]~
? nfalgtobasis(nf, y)
%4 = [0, 2]~
? nfalgtobasis(nf, Mod(y, y^2+4))
%4 = [0, 2]~
```

This is the inverse function of `nfbasistoalg`.

The library syntax is `GEN algtobasis(GEN nf, GEN x)`.

3.6.75 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 `GEN nfbasis0(GEN x, long flag, GEN fa = NULL)`. An extended version is `GEN nfbasis(GEN x, GEN *d, long flag, GEN fa = NULL)`, where `*d` receives the discriminant of the number field (*not* of the polynomial x).

3.6.76 nfbasistoalg(nf, x): Given an algebraic number x in the number field `nf`, transforms it into `t_POLMOD` form.

```
? nf = nfinit(y^2 + 4);
? nf.zk
%2 = [1, 1/2*y]
? nfbasistoalg(nf, [1,1]~)
%3 = Mod(1/2*y + 1, y^2 + 4)
? nfbasistoalg(nf, y)
%4 = Mod(y, y^2 + 4)
? nfbasistoalg(nf, Mod(y, y^2+4))
%4 = Mod(y, y^2 + 4)
```

This is the inverse function of `nfalgtobasis`.

The library syntax is `GEN basistoalg(GEN nf, GEN x)`.

3.6.77 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 `GEN nfdetint(GEN nf, GEN x)`.

3.6.78 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 `GEN nfdisc0(GEN x, long flag, GEN fa = NULL)`. Also available is `GEN nfdisc(GEN x)` (*flag* $= 0$).

3.6.79 nfeltadd(nf, x, y): given two elements x and y in nf, computes their sum $x + y$ in the number field nf.

The library syntax is `GEN nfadd(GEN nf, GEN x, GEN y)`.

3.6.80 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 `GEN nfdiv(GEN nf, GEN x, GEN y)`.

3.6.81 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(nfdiv(`nf`,x,y))`.

The library syntax is `GEN nfdiveuc(GEN nf, GEN x, GEN y)`.

3.6.82 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 `GEN nfdivmodpr(GEN nf, GEN x, GEN y, GEN pr)`. This function is normally useless in library mode. Project your inputs to the residue field using `nf_to_Fq`, then work there.

3.6.83 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 `GEN nfdivrem(GEN nf, GEN x, GEN y)`.

3.6.84 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

$$\mathtt{x} - \mathtt{nfmul}(nf, \mathtt{round}(\mathtt{nfdiv}(nf, \mathtt{x}, \mathtt{y})), \mathtt{y}).$$

The library syntax is `GEN nfmod(GEN nf, GEN x, GEN y)`.

3.6.85 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 `GEN nfmul(GEN nf, GEN x, GEN y)`.

3.6.86 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 `GEN nfmulmodpr(GEN nf, GEN x, GEN y, GEN pr)`. This function is normally useless in library mode. Project your inputs to the residue field using `nf_to_Fq`, then work there.

3.6.87 nfeltnorm(nf, x): returns the absolute norm of x.

The library syntax is `GEN nfnorm(GEN nf, GEN x)`.

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 `GEN nfpow(GEN nf, GEN x, GEN k)`. `GEN nfinv(GEN nf, GEN x)` correspond to $k = -1$, and `GEN nfsqr(GEN nf,GEN x)` to $k = 2$.

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 `GEN nfpowmodpr(GEN nf, GEN x, GEN k, GEN pr)`. This function is normally useless in library mode. Project your inputs to the residue field using `nf_to_Fq`, then work there.

3.6.90 nfeltreduce(nf, a, id): given an ideal id in Hermite normal form and an element a of the number field nf, finds an element r in nf such that $a - r$ belongs to the ideal and r is small.

The library syntax is `GEN nfreduce(GEN nf, GEN a, GEN id)`.

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 `GEN nfreducemodpr(GEN nf, GEN x, GEN pr)`. This function is normally useless in library mode. Project your inputs to the residue field using `nf_to_Fq`, then work there.

3.6.92 nfelttrace(nf, x): returns the absolute trace of x.

The library syntax is `GEN nftrace(GEN nf, GEN x)`.

3.6.93 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 `long nfval(GEN nf, GEN x, GEN pr)`.

3.6.94 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 2.5.3). 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
```

It is possible to input a defining polynomial for nf instead, but this is in general less efficient since parts of an `nf` structure will be computed internally. This is useful in two situations: when you don't need the `nf`, or when you can't compute its discriminant due to integer factorization difficulties. In the latter case, `addprimes` is a possibility but a dangerous one: factors will probably be missed if the (true) field discriminant and an `addprimes` entry are strictly divisible by some prime. If you have such an unsafe nf, it is safer to input `nf.pol`.

The library syntax is `GEN nffactor(GEN nf, GEN x)`.

3.6.95 nffactorback$(nf, f, \{e\})$: gives back the `nf` element corresponding to a factorization. The integer 1 corresponds to the empty factorization.

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 1s: we return the product of the $f[i]$. Finally, f can be a regular factorization matrix.

```
? nf = nfinit(y^2+1);
? nffactorback(nf, [3, y+1, [1,2]~], [1, 2, 3])
%2 = [12, -66]~
? 3 * (I+1)^2 * (1+2*I)^3
%3 = 12 - 66*I
```

The library syntax is `GEN nffactorback(GEN nf, GEN f, GEN e = NULL)`.

3.6.96 nffactormod(nf, pol, 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 2.5.3). 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 `GEN nffactormod(GEN nf, GEN pol, GEN pr)`.

3.6.97 nfgaloisapply(nf, aut, x): let nf be a number field as output by `nfinit`, and let aut be a Galois automorphism of nf expressed by its image on the field generator (such automorphisms can be found using `nfgaloisconj`). The function computes the action of the automorphism aut on the object x in the number field; x can be a number field element, or an ideal (possibly extended). Because of possible confusion with elements and ideals, other vector or matrix arguments are forbidden.

```
? nf = nfinit(x^2+1);
? L = nfgaloisconj(nf)
%2 = [-x, x]~
? aut = L[1]; /* the non-trivial automorphism */
? nfgaloisapply(nf, aut, x)
%4 = Mod(-x, x^2 + 1)
? P = idealprimedec(nf,5); /* prime ideals above 5 */
? nfgaloisapply(nf, aut, P[2]) == P[1]
%7 = 0 \\ !!!!
? idealval(nf, nfgaloisapply(nf, aut, P[2]), P[1])
%8 = 1
```

The surprising failure of the equality test (`%7`) is due to the fact that although the corresponding prime ideals are equal, their representations are not. (A prime ideal is specificed by a uniformizer, and there is no guarantee that applying automorphisms yields the same elements as a direct `idealprimedec` call.)

The automorphism can also be given as a column vector, representing the image of `Mod(x, nf.pol)` as an algebraic number. This last representation is more efficient and should be preferred if a given automorphism must be used in many such calls.

```
? nf = nfinit(x^3 - 37*x^2 + 74*x - 37);
? l = nfgaloisconj(nf); aut = l[2] \\  automorphisms in basistoalg form
%2 = -31/11*x^2 + 1109/11*x - 925/11
? L = matalgtobasis(nf, l); AUT = L[2] \\  same in algtobasis form
%3 = [16, -6, 5]~
? v = [1, 2, 3]~; nfgaloisapply(nf, aut, v) == nfgaloisapply(nf, AUT, v)
%4 = 1 \\  same result...
? for (i=1,10^5, nfgaloisapply(nf, aut, v))
time = 1,451 ms.
? for (i=1,10^5, nfgaloisapply(nf, AUT, v))
time = 1,045 ms.  \\  but the latter is faster
```

The library syntax is `GEN galoisapply(GEN nf, GEN aut, GEN x)`.

3.6.98 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 also makes sense when the number field is not Galois since some conjugates may lie in the field. nf can simply be a polynomial.

If no flags or $flag = 0$, use a combination of flag 4 and 1 and the result is always complete. There is no point whatsoever in using the other flags.

If $flag = 1$, use `nfroots`: a little slow, but guaranteed to work in polynomial time.

If *flag* = 2 (OBSOLETE), use complex approximations to the roots and an integral LLL. The result is not guaranteed to be complete: some conjugates may be missing (a warning is issued if the result is not proved complete), especially so if the corresponding polynomial has a huge index, and increasing the default precision may help. This variant is slow and unreliable: don't use it.

If *flag* = 4, use `galoisinit`: very fast, but only applies to (most) Galois fields. If the field is Galois with weakly super-solvable Galois group (see `galoisinit`), 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*.

This routine can only compute **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)
{ my(polabs, N);
  R *= Mod(1, nfK.pol);              \\ convert coeffs to polmod elts of K
  polabs = rnfequation(nfK, R);
  N = nfgaloisconj(polabs) % R;      \\ Q-automorphisms of L
  \\ select the ones that fix K
  select(s->subst(R, variable(R), Mod(s,R)) == 0, N);
}
K  = nfinit(y^2 + 7);
rnfgaloisconj(K, x^4 - y*x^3 - 3*x^2 + y*x + 1)  \\ K-automorphisms of L
```

The library syntax is `GEN galoisconj0(GEN nf, long flag, GEN d = NULL, long prec)`. Use directly `GEN galoisconj(GEN nf, GEN d)`, corresponding to *flag* = 0, the others only have historical interest.

3.6.99 nfhilbert$(nf, a, b, \{pr\})$: if pr is omitted, compute the global quadratic 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 `long nfhilbert0(GEN nf, GEN a, GEN b, GEN pr = NULL)`.

Also available is `long nfhilbert(GEN bnf,GEN a,GEN b)` (global quadratic Hilbert symbol).

3.6.100 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 `GEN nfhnf(GEN nf, GEN x)`. Also available:

`GEN rnfsimplifybasis(GEN bnf, GEN x)` simplifies the pseudo-basis given by $x = (A, I)$. The ideals in the list I are integral, primitive and either trivial (equal to the full ring of integer) or non-principal.

3.6.101 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 `GEN nfhnfmod(GEN nf, GEN x, GEN detx)`.

3.6.102 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, *roundG*, 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 an $n \times n$ matrix 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 $\mathbf{C}$.

• *roundG* is a rescaled copy of G, rounded to nearest integers.

• 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. Also, when understood as an ideal, the matrix T^{-1} generates the codifferent ideal.

• The columns of MD (nf`.diff`) express a $\mathbf{Z}$-basis of the different of K on the integral basis.

• TI is equal to the primitive part of T^{-1}, which has integral coefficients.

• 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 conditionally.

```
? 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.

The library syntax is `GEN nfinit0(GEN pol, long flag, long prec)`. Also available are `GEN nfinit(GEN x, long prec)` ($flag = 0$), `GEN nfinitred(GEN x, long prec)` ($flag = 2$), `GEN nfinitred2(GEN x, long prec)` ($flag = 3$). Instead of the above hardcoded numerical flags in `nfinit0`, one should rather use

`GEN nfinitall(GEN x, long flag, long prec)`, where *flag* is an or-ed combination of

- `nf_RED`: find a simpler defining polynomial,
- `nf_ORIG`: if `nf_RED` set, also return the change of variable,
- `nf_ROUND2`: slow down the routine by using an obsolete normalization algorithm (do not use this one!),

• `nf_PARTIALFACT`: lazy factorization of the polynomial discriminant. Result is conditional unless the *field* discriminant obtained is fully factored by

```
Z_factor_limit(disc, 0)
```

Namely the "maximal order" may not be maximal at any prime bigger than `primelimit` dividing the field discriminant.

3.6.103 nfisideal(nf, x): returns 1 if x is an ideal in the number field nf, 0 otherwise.

The library syntax is `long isideal(GEN nf, GEN x)`.

3.6.104 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 `GEN nfisincl(GEN x, GEN y)`.

3.6.105 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 `GEN nfisisom(GEN x, GEN y)`.

3.6.106 nfkermodpr(nf, x, pr): kernel of the matrix a in $\mathbf{Z}_K/pr$, where pr is in **modpr** format (see `nfmodprinit`).

The library syntax is `GEN nfkermodpr(GEN nf, GEN x, GEN pr)`. This function is normally useless in library mode. Project your inputs to the residue field using `nfM_to_FqM`, then work there.

3.6.107 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 `GEN nfmodprinit(GEN nf, GEN pr)`.

3.6.108 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 `GEN nfnewprec(GEN nf, long prec)`. See also `GEN bnfnewprec(GEN bnf, long prec)` and `GEN bnrnewprec(GEN bnr, long prec)`.

3.6.109 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 2.5.3). 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`).

It is possible to input a defining polynomial for nf instead, but this is in general less efficient since parts of an `nf` structure will be computed internally. This is useful in two situations: when you don't need the `nf`, or when you can't compute its discriminant due to integer factorization difficulties. In the latter case, `addprimes` is a possibility but a dangerous one: roots will probably be missed if the (true) field discriminant and an `addprimes` entry are strictly divisible by some prime. If you have such an unsafe nf, it is safer to input `nf.pol`.

The library syntax is `GEN nfroots(GEN nf = NULL, GEN x)`. See also `GEN nfrootsQ(GEN x)`, corresponding to `nf` = `NULL`.

3.6.110 nfrootsof1(nf): Returns a two-component vector $[w, z]$ where w is the number of roots of unity in the number field nf, and z is a primitive w-th root of unity.

```
? K = nfinit(polcyclo(11));
? nfrootsof1(K)
%2 = [22, [0, 0, 0, 0, 0, -1, 0, 0, 0, 0]~]
? z = nfbasistoalg(K, %[2])   \\ in algebraic form
%3 = Mod(-x^5, x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1)
? [lift(z^11), lift(z^2)]     \\ proves that the order of z is 22
%4 = [-1, -x^9 - x^8 - x^7 - x^6 - x^5 - x^4 - x^3 - x^2 - x - 1]
```

This function guesses the number w as the gcd of the $\#k(v)^*$ for unramified v above odd primes, then computes the roots in nf of the w-th cyclotomic polynomial: the algorithm is polynomial time with respect to the field degree and the bitsize of the multiplication table in nf (both of them polynomially bounded in terms of the size of the discriminant). Fields of degree up to 100 or so should require less than one minute.

The library syntax is `GEN rootsof1(GEN nf)`. Also available is `GEN rootsof1_kannan(GEN nf)`, that computes all algebraic integers of T_2 norm equal to the field degree (all roots of 1, by Kronecker's theorem). This is in general a little faster than the default when there *are* roots of 1 in the field (say twice faster), but can be much slower (say, *days* slower), since the algorithm is a priori exponential in the field degree.

3.6.111 nfsnf(nf, x): given a $\mathbf{Z}_K$-module x associated to the integral pseudo-matrix (A, I, J), returns an ideal list $d_1, \dots, d_n$ 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 $i \geq 2$.

See Section 3.6.4.1 for the definition of integral pseudo-matrix; briefly, it is input as a 3-component row vector $[A, I, J]$ where $I = [b_1, \dots, b_n]$ and $J = [a_1, \dots, a_n]$ are two ideal lists, and A is a square $n \times n$ matrix with columns $(A_1, \dots, A_n)$, seen as elements in K^n (with canonical basis $(e_1, \dots, e_n)$). This data defines the $\mathbf{Z}_K$ module x given by

$$(b_1 e_1 \oplus \cdots \oplus b_n e_n)/(a_1 A_1 \oplus \cdots \oplus a_n A_n) \ ,$$

The integrality condition is $a_{i,j} \in b_i a_j^{-1}$ for all i, j. If it is not satisfied, then the d_i will not be integral. Note that every finitely generated torsion module is isomorphic to a module of this form and even with $b_i = Z_K$ for all i.

The library syntax is `GEN nfsnf(GEN nf, GEN x)`.

3.6.112 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 `GEN nfsolvemodpr(GEN nf, GEN a, GEN b, GEN pr)`. This function is normally useless in library mode. Project your inputs to the residue field using `nfM_to_FqM`, then work there.

3.6.113 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 `GEN nfsubfields(GEN pol, long d)`.

3.6.114 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 `GEN polcompositum0(GEN P, GEN Q, long flag)`. Also available are `GEN compositum(GEN P, GEN Q)` (*flag* $= 0$) and `GEN compositum2(GEN P, GEN Q)` (*flag* $= 1$).

3.6.115 polgalois(x): Galois group of the non-constant polynomial $x \in \mathbf{Q}[X]$. In the present version 2.5.5, 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 behavior 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 `GEN polgalois(GEN x, long prec)`. To enable the new format in library mode, set the global variable `new_galois_format` to 1.

3.6.116 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 primitive elements defining these subfields, the second giving the corresponding minimal polynomials.

```
? M = polred(x^4 + 8, 2)
%1 =
[1 x - 1]

[1/2*x^2 x^2 + 2]

[1/4*x^3 x^4 + 2]

[x x^4 + 8]
? minpoly(Mod(M[2,1], x^4+8))
%2 = x^2 + 2
```

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 `GEN polred0(GEN x, long flag, GEN fa = NULL)`. Also available is `GEN polred(GEN x)` (*flag* = 0). The function `polred0` is provided for backward compatibility; instead of the above hardcoded numerical flags (which happen to be inconsistent), one should use `GEN Polred(GEN x, long flag, GEN fa)` where flag is an or-ed combination of `nf_PARTIALFACT` (partial factorization of the discriminant) and `nf_ORIG` (give also elements).

3.6.117 polredabs$(T, \{flag = 0\})$: returns a canonical defining polynomial P for the same number field defined by T, such that the sum of the squares of the modulus of the roots (i.e. the T_2-norm) is minimal. Different T defining isomorphic number fields will yield the same P. All T accepted by `nfinit` are also allowed here: non-monic polynomials, `nf`, `bnf`, `[T, 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; in that case, 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 `Mod(a, P)` is a root of the original T.

4: gives *all* polynomials of minimal T_2 norm; of the two polynomials $P(x)$ and $\pm P(-x)$, 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 the polynomial P is no longer canonical, and it may happen that it does not have minimal T_2 norm. You should always include this flag; without it, the routine will often spend infinite time trying to factor the discriminant of T. As long as the discriminant of the *field* $\mathbf{Q}[X]/(T)$ is easy to factor (has at most one large prime factor not in the `addprimes` table), the result is the same.

The library syntax is `GEN polredabs0(GEN T, long flag)`. Instead of the above hardcoded numerical flags, one should use an or-ed combination of

• `nf_PARTIALFACT`: partial factorization of the discriminant, possibly work in a non-maximal order. You should always include this.

• `nf_ORIG`: return $[P, a]$, where `Mod(a, P)` is a root of T.

• `nf_RAW`: return $[P, b]$, where `Mod(b, T)` is a root of P. The algebraic integer b is the raw result produced by the small vectors enumeration in the maximal order; P was computed as the characteristic polynomial of `Mod(b, T)`. `Mod(a, P)` as in `nf_ORIG` is obtained with `modreverse`.

• `nf_ADDZK`: if r is the result produced with some of the above flags (of the form P or $[P, c]$), return `[r,zk]`, where `zk` is a $\mathbf{Z}$-basis for the maximal order of $\mathbf{Q}[X]/(P)$.

• `nf_ALL`: return a vector of results of the above form, for all polynomials of minimal T_2-norm.

3.6.118 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. Useless function: try `polred(,1)` or `polredabs(,16)`.

The library syntax is `GEN ordred(GEN x)`.

3.6.119 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 `GEN tschirnhaus(GEN x)`.

3.6.120 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 `GEN rnfalgtobasis(GEN rnf, GEN x)`.

3.6.121 rnfbasis(bnf, M): let K the field represented by bnf, as output by `bnfinit`. M is a projective $\mathbf{Z}_K$-module of rank n ($M \otimes K$ is an n-dimensional K-vector space), given by a pseudo-basis of size n. The routine returns either a true $\mathbf{Z}_K$-basis of M (of size n) if it exists, or an $n+1$-element generating set of M if not.

It is allowed to use an irreducible polynomial P in $K[X]$ instead of M, in which case, M is defined as the ring of integers of $K[X]/(P)$, viewed as a $\mathbf{Z}_K$-module.

The library syntax is `GEN rnfbasis(GEN bnf, GEN M)`.

3.6.122 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 `GEN rnfbasistoalg(GEN rnf, GEN x)`.

3.6.123 rnfcharpoly$(nf, T, a, \{var = x\})$: 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).

```
? nf = nfinit(y^2+1);
? rnfcharpoly(nf, x^2+y*x+1, x+y)
%2 = x^2 + Mod(-y, y^2 + 1)*x + 1
```

The library syntax is `GEN rnfcharpoly(GEN nf, GEN T, GEN a, long var = -1)`, where `var` is a variable number.

3.6.124 rnfconductor(bnf, pol): 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.

The library syntax is `GEN rnfconductor(GEN bnf, GEN pol, long )`.

3.6.125 rnfdedekind$(nf, pol, \{pr\}, \{flag = 0\})$: given a number field K coded by nf and a monic polynomial $P \in \mathbf{Z}_K[X]$, irreducible over K and thus defining a relative extension L of K, applies Dedekind's criterion to the order $\mathbf{Z}_K[X]/(P)$, at the prime ideal pr. It is possible to set pr to a vector of prime ideals (test maximality at all primes in the vector), or to omit altogether, in which case maximality at *all* primes is tested; in this situation *flag* is automatically set to 1.

The default historic behavior (*flag* is 0 or omitted and pr is a single prime ideal) is not so useful since `rnfpseudobasis` gives more information and is generally not that much slower. It returns a 3-component vector $[max, basis, v]$:

• *basis* is a pseudo-basis of an enlarged order O produced by Dedekind's criterion, containing the original order $\mathbf{Z}_K[X]/(P)$ with index a power of pr. Possibly equal to the original order.

• *max* is a flag equal to 1 if the enlarged order O could be proven to be pr-maximal and to 0 otherwise; it may still be maximal in the latter case if pr is ramified in L,

• v is the valuation at pr of the order discriminant.

If *flag* is non-zero, on the other hand, we just return 1 if the order $\mathbf{Z}_K[X]/(P)$ is pr-maximal (resp. maximal at all relevant primes, as described above), and 0 if not. This is much faster than the default, since the enlarged order is not computed.

```
? nf = nfinit(y^2-3); P = x^3 - 2*y;
? pr3 = idealprimedec(nf,3)[1];
? rnfdedekind(nf, P, pr3)
%2 = [1, [[1, 0, 0; 0, 1, 0; 0, 0, 1], [1, 1, 1]], 8]
? rnfdedekind(nf, P, pr3, 1)
%3 = 1
```

In this example, `pr3` is the ramified ideal above 3, and the order generated by the cube roots of y is already `pr3`-maximal. The order-discriminant has valuation 8. On the other hand, the order is not maximal at the prime above 2:

```
? pr2 = idealprimedec(nf,2)[1];
? rnfdedekind(nf, P, pr2, 1)
%5 = 0
? rnfdedekind(nf, P, pr2)
%6 = [0, [[2, 0, 0; 0, 1, 0; 0, 0, 1], [[1, 0; 0, 1], [1, 0; 0, 1],
     [1, 1/2; 0, 1/2]]], 2]
```

The enlarged order is not proven to be `pr2`-maximal yet. In fact, it is; it is in fact the maximal order:

```
? B = rnfpseudobasis(nf, P)
%7 = [[1, 0, 0; 0, 1, 0; 0, 0, 1], [1, 1, [1, 1/2; 0, 1/2]],
     [162, 0; 0, 162], -1]
? idealval(nf,B[3], pr2)
%4 = 2
```

It is possible to use this routine with non-monic $P = \sum_{i\leq n} a_i X^i \in \mathbf{Z}_K[X]$ if $flag = 1$; in this case, we test maximality of Dedekind's order generated by

$$1, a_n\alpha, a_n\alpha^2 + a_{n-1}\alpha, \ldots, a_n\alpha^{n-1} + a_{n-1}\alpha^{n-2} + \cdots + a_1\alpha.$$

The routine will fail if P is 0 on the projective line over the residue field $\mathbf{Z}_K/\mathtt{pr}$ (FIXME).

The library syntax is `GEN rnfdedekind(GEN nf, GEN pol, GEN pr = NULL, long flag)`.

3.6.126 rnfdet(nf, M): given a pseudo-matrix M over the maximal order of nf, computes its determinant.

The library syntax is `GEN rnfdet(GEN nf, GEN M)`.

3.6.127 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 2.5.3.

The library syntax is `GEN rnfdiscf(GEN nf, GEN pol)`.

3.6.128 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 `GEN rnfelementabstorel(GEN rnf, GEN x)`.

3.6.129 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 `GEN rnfelementdown(GEN rnf, GEN x)`.

3.6.130 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 `GEN rnfelementreltoabs(GEN rnf, GEN x)`.

3.6.131 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 `GEN rnfelementup(GEN rnf, GEN x)`.

3.6.132 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 behavior, 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 2.5.3). 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 `GEN rnfequation0(GEN nf, GEN pol, long flag)`. Also available are `GEN rnfequation(GEN nf, GEN pol)` (*flag* $= 0$) and `GEN rnfequation2(GEN nf, GEN pol)` (*flag* $= 1$).

3.6.133 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 `GEN rnfhnfbasis(GEN bnf, GEN x)`.

3.6.134 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 `GEN rnfidealabstorel(GEN rnf, GEN x)`.

3.6.135 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 3.6.134), returns the ideal of K below x, i.e. the intersection of x with K.

The library syntax is `GEN rnfidealdown(GEN rnf, GEN x)`.

3.6.136 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 `GEN rnfidealhermite(GEN rnf, GEN x)`.

3.6.137 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 `GEN rnfidealmul(GEN rnf, GEN x, GEN y)`.

3.6.138 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 `GEN rnfidealnormabs(GEN rnf, GEN x)`.

3.6.139 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 `GEN rnfidealnormrel(GEN rnf, GEN x)`.

3.6.140 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 `GEN rnfidealreltoabs(GEN rnf, GEN x)`.

3.6.141 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 `GEN rnfidealtwoelement(GEN rnf, GEN x)`.

3.6.142 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( matalgtobasis(nf, y) ) );
```

The library syntax is `GEN rnfidealup(GEN rnf, GEN x)`.

3.6.143 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 2.5.3) 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 `GEN rnfinit(GEN nf, GEN pol)`.

3.6.144 rnfisabelian(nf, T): T being a relative polynomial with coefficients in *nf*, return 1 if it defines an abelian extension, and 0 otherwise.

```
? K = nfinit(y^2 + 23);
? rnfisabelian(K, x^3 - 3*x - y)
%2 = 1
```

The library syntax is `long rnfisabelian(GEN nf, GEN T)`.

3.6.145 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 `long rnfisfree(GEN bnf, GEN x)`.

3.6.146 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 `GEN rnfisnorm(GEN T, GEN a, long flag)`.

3.6.147 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 `GEN rnfisnorminit(GEN pol, GEN polrel, long flag)`.

3.6.148 rnfkummer$(bnr, \{subgp\}, \{d = 0\})$: *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 *subgp* is omitted). If d is positive, outputs the list of all relative equations of degree d contained in the ray class field defined by *bnr*, with the *same* conductor as $(bnr, subgp)$.

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 `GEN rnfkummer(GEN bnr, GEN subgp = NULL, long d, long prec)`.

3.6.149 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 `GEN rnflllgram(GEN nf, GEN pol, GEN order, long prec)`.

3.6.150 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.bnf` 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.gen`. 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 `GEN rnfnormgroup(GEN bnr, GEN pol)`.

3.6.151 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 2.5.5, this is slower and less efficient than `rnfpolredabs`.

The library syntax is `GEN rnfpolred(GEN nf, GEN pol, long prec)`.

3.6.152 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 3.6.117.

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 `GEN rnfpolredabs(GEN nf, GEN pol, long flag)`.

3.6.153 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 `GEN rnfpseudobasis(GEN nf, GEN pol)`.

3.6.154 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 `GEN rnfsteinitz(GEN nf, GEN x)`.

3.6.155 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.

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 `GEN subgrouplist0(GEN bnr, GEN bound = NULL, long flag)`.

3.6.156 zetak$(nfz, x, \{flag = 0\})$: *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 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 `GEN gzetakall(GEN nfz, GEN x, long flag, long prec)`. See also `GEN glambdak(GEN znf, GEN x, long prec)` or `GEN gzetak(GEN znf, GEN x, long prec)`.

3.6.157 zetakinit(bnf): computes a number of initialization data concerning the number field associated to `bnf` so as to be able to compute the Dedekind zeta and lambda functions, respectively `zetak`(x) and `zetak`($x, 1$), at the current real precision. If you do not need the `bnfinit` data somewhere else, you may call it with an irreducible polynomial instead of a bnf: it will call `bnfinit` itself.

The result is a 9-component vector v whose components are very technical and cannot really be used except through the `zetak` function.

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 `GEN initzeta(GEN bnf, long prec)`.

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($p\hat{}e$): 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 `GEN ggrando()`. `GEN zeropadic(GEN p, long e)` for a p-adic and `GEN zeroser(long v, long e)` for a power series zero in variable v.

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 `GEN deriv(GEN x, long v = -1)`, where `v` is a variable number.

3.7.3 diffop$(x, v, d, \{n = 1\})$: Let v be a vector of variables, and d a vector of the same length, return the image of x by the n-power (1 if n is not given) of the differential operator D that assumes the value `d[i]` on the variable `v[i]`. The value of D on a scalar type is zero, and D applies componentwise to a vector or matrix. When applied to a `t_POLMOD`, if no value is provided for the variable of the modulus, such value is derived using the implicit function theorem.

Some examples: This function can be used to differentiate formal expressions: If $E = \exp(X^2)$ then we have $E' = 2 * X * E$. We can derivate $X * exp(X^2)$ as follow:

```
? diffop(E*X,[X,E],[1,2*X*E])
%1 = (2*X^2 + 1)*E
```

Let `Sin` and `Cos` be two function such that $\texttt{Sin}^2 + \texttt{Cos}^2 = 1$ and $\texttt{Cos}' = -\texttt{Sin}$. We can differentiate `Sin`/`Cos` as follow, PARI inferring the value of $\texttt{Sin}'$ from the equation:

```
? diffop(Mod('Sin/'Cos,'Sin^2+'Cos^2-1),['Cos],[-'Sin])
%1 = Mod(1/Cos^2, Sin^2 + (Cos^2 - 1))
```

Compute the Bell polynomials (both complete and partial) via the Faa di Bruno formula:

```
Bell(k,n=-1)=
{
  my(var(i)=eval(Str("X",i)));
  my(x,v,dv);
  v=vector(k,i,if(i==1,'E,var(i-1)));
  dv=vector(k,i,if(i==1,'X*var(1)*'E,var(i)));
  x=diffop('E,v,dv,k)/'E;
  if(n<0,subst(x,'X,1),polcoeff(x,n,'X))
}
```

The library syntax is `GEN diffop0(GEN x, GEN v, GEN d, long n)`.

For $n = 1$, the function `GEN diffop(GEN x, GEN v, GEN d)` is also available.

3.7.4 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")
  *** eval: Warning: unused characters: a.
% 1 = 1
```

The library syntax is **geval**(`GEN x`).

3.7.5 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 much less efficient.

The library syntax is `GEN factorpadic0(GEN pol, GEN p, long r, long flag)`.

`GEN factorpadic(GEN f,GEN p, long r)` corresponds to the default $flag = 0$.

3.7.6 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 `GEN integ(GEN x, long v = -1)`, where `v` is a variable number.

3.7.7 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 `GEN padicappr(GEN pol, GEN a)`.

3.7.8 padicfields($p, N, \{flag = 0\}$): returns a vector of polynomials generating all the extensions of degree N of the field $\mathbf{Q}_p$ of p-adic rational numbers; N is allowed to be a 2-component vector $[n, d]$, in which case we return the extensions of degree n and discriminant p^d.

The list is minimal in the sense that two different polynomials generate non-isomorphic extensions; in particular, the number of polynomials is the number of classes of isomorphic extensions. If P is a polynomial in this list, α is any root of P and $K = \mathbf{Q}_p(\alpha)$, then α is the sum of a uniformizer and a (lift of a) generator of the residue field of K; in particular, the powers of α generate the ring of p-adic integers of K.

If $flag = 1$, replace each polynomial P by a vector $[P, e, f, d, c]$ where e is the ramification index, f the residual degree, d the valuation of the discriminant, and c the number of conjugate

fields. If *flag* = 2, only return the *number* of extensions in a fixed algebraic closure (Krasner's formula), which is much faster.

The library syntax is `GEN padicfields0(GEN p, GEN N, long flag)`. Also available is `GEN padicfields(GEN p, long n, long d, long flag)`, which computes extensions of $\mathbf{Q}_p$ of degree n and discriminant p^d.

3.7.9 polchebyshev$(n, \{flag = 1\}, \{a =' x\})$: returns the n^{th} Chebyshev polynomial of the first kind T_n (*flag* = 1) or the second kind U_n (*flag* = 2), evaluated at a (`'x` by default). Both series of polynomials satisfy the 3-term relation

$$P_{n+1} = 2xP_n - P_{n-1},$$

and are determined by the initial conditions $U_0 = T_0 = 1$, $T_1 = x$, $U_1 = 2x$. In fact $T_n' = nU_{n-1}$ and, for all complex numbers z, we have $T_n(\cos z) = \cos(nz)$ and $U_{n-1}(\cos z) = \sin(nz)/\sin z$. If $n \geq 0$, then these polynomials have degree n. For $n < 0$, T_n is equal to T_{-n} and U_n is equal to $-U_{-2-n}$. In particular, $U_{-1} = 0$.

The library syntax is `GEN polchebyshev_eval(long n, long flag, GEN a = NULL)`. Also available are `GEN polchebyshev1(long n, long v)` and `GEN polchebyshev2(long n, long v)` for T_n and U_n respectively.

3.7.10 polcoeff$(x, n, \{v\})$: coefficient of degree n of the polynomial x, with respect to the main variable if v is omitted, with respect to v otherwise. If n is greater than the degree, the result is zero.

Naturally applies to scalars (polynomial of degree 0), as well as to rational functions whose denominator is a monomial. It also applies to power series: if n is less than the valuation, the result is zero. If it 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 `component(x,n)`.

The library syntax is `GEN polcoeff0(GEN x, long n, long v = -1)`, where `v` is a variable number.

3.7.11 polcyclo$(n, \{a =' x\})$: n-th cyclotomic polynomial, evaluated at a (`'x` by default). The integer n must be positive.

Algorithm used: reduce to the case where n is squarefree; to compute the cyclotomic polynomial, use $\Phi_{np}(x) = \Phi_n(x^p)/\Phi(x)$; to compute it evaluated, use $\Phi_n(x) = \prod_{d|n}(x^d - 1)^{\mu(n/d)}$. In the evaluated case, the algorithm can deal with all rational values a; otherwise it assumes that $a^d - 1$ is invertible for all $d \mid n$. If this is not the case, use `subst(polcyclo(n),x,a)`.

The library syntax is `GEN polcyclo_eval(long n, GEN a = NULL)`. The variant `GEN polcyclo(long n, long v)` returns the n-th cyclotomic polynomial in variable v.

3.7.12 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 `long poldegree(GEN x, long v = -1)`, where `v` is a variable number.

3.7.13 poldisc$(pol, \{v\})$: discriminant of the polynomial *pol* in the main variable if v is omitted, in v otherwise. The algorithm used is the subresultant algorithm.

The library syntax is `GEN poldisc0(GEN pol, long v = -1)`, where `v` is a variable number.

3.7.14 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 `GEN reduceddiscsmith(GEN f)`.

3.7.15 polhensellift(A, B, p, e): given a prime p, an integral polynomial A whose leading coefficient is a p-unit, a vector B of integral polynomials that are monic and pairwise relatively prime modulo p, and whose product is congruent to $A/\mathrm{lc}(A)$ modulo p, lift the elements of B to polynomials whose product is congruent to A modulo p^e.

More generally, if T is an integral polynomial irreducible mod p, and B is a factorization of A over the finite field $\mathbf{F}_p[t]/(T)$, you can lift it to $\mathbf{Z}_p[t]/(T, p^e)$ by replacing the p argument with $[p, T]$:

```
? { T = t^3 - 2; p = 7; A = x^2 + t + 1;
    B = [x + (3*t^2 + t + 1), x + (4*t^2 + 6*t + 6)];
    r = polhensellift(A, B, [p, T], 6) }
%1 = [x + (20191*t^2 + 50604*t + 75783), x + (97458*t^2 + 67045*t + 41866)]
? lift(lift( r[1] * r[2] * Mod(Mod(1,p^6),T) ))
%2 = x^2 + (t + 1)
```

The library syntax is `GEN polhensellift(GEN A, GEN B, GEN p, long e)`.

3.7.16 polhermite$(n, \{a =' x\})$: n^{th} Hermite polynomial H_n evaluated at a (`'x` by default), i.e.

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}.$$

The library syntax is `GEN polhermite_eval(long n, GEN a = NULL)`. The variant `GEN polhermite(long n, long v)` returns the n-th Hermite polynomial in variable v.

3.7.17 polinterpolate$(X, \{Y\}, \{x\}, \{\&e\})$: given the data vectors X and Y of the same length n (X containing the x-coordinates, and Y the corresponding y-coordinates), this function finds the interpolating polynomial passing through these points and evaluates it at x. If Y is omitted, return the polynomial interpolating the $(i, X[i])$. If present, e will contain an error estimate on the returned value.

The library syntax is `GEN polint(GEN X, GEN Y = NULL, GEN x = NULL, GEN *e = NULL)`.

3.7.18 polisirreducible(pol): *pol* being a polynomial (univariate in the present version 2.5.5), 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 `GEN gisirreducible(GEN pol)`.

3.7.19 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 `GEN pollead(GEN x, long v = -1)`, where `v` is a variable number.

3.7.20 pollegendre$(n, \{a =' x\})$: n^{th} Legendre polynomial evaluated at a (`'x` by default).

The library syntax is `GEN pollegendre_eval(long n, GEN a = NULL)`. To obtain the n-th Legendre polynomial in variable v, use `GEN pollegendre(long n, long v)`.

3.7.21 polrecip(pol): reciprocal polynomial of *pol*, i.e. the coefficients are in reverse order. *pol* must be a polynomial.

The library syntax is `GEN polrecip(GEN pol)`.

3.7.22 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 you also need the u and v such that $x * u + y * v = res(x, y)$, use the `bezoutres` function.

If *flag* $= 0$ (default), uses the the algorithm best suited to the inputs, either the subresultant algorithm (Lazard/Ducos variant, generic case), a modular algorithm (inputs in $\mathbf{Q}[X]$) or Sylvester's matrix (inexact inputs).

If *flag* $= 1$, uses the determinant of Sylvester's matrix instead; this should always be slower than the default.

The library syntax is `GEN polresultant0(GEN x, GEN y, long v = -1, long flag)`, where `v` is a variable number.

3.7.23 polroots$(x, \{flag = 0\})$: 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, nor to give accurate results, but is rather fast when it does. If you get the messages "too many iterations in roots" or "INTERNAL ERROR: incorrect result in roots", use the default algorithm.

The library syntax is `GEN roots0(GEN x, long flag, long prec)`. Also available is `GEN roots(GEN x, long prec)`, as well as `GEN cleanroots(GEN x, long prec)` which ensures that real roots of real polynomials are returned as `t_REAL` (instead of `t_COMPLEX`s with 0 imaginary part).

3.7.24 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.

```
? polrootsmod(x^2-1,2)
%1 = [Mod(1, 2)]~
? polrootsmod(x^2-1,4)
%2 = [Mod(1, 4), Mod(3, 4)]~
```

If p is very small, you may set $flag = 1$, which uses a naive search.

The library syntax is `GEN rootmod0(GEN pol, GEN p, long flag)`.

3.7.25 polrootspadic(x, 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 `GEN rootpadic(GEN x, GEN p, long r)`.

3.7.26 polsturm$(pol, \{a\}, \{b\})$: number of real roots of the real squarefree 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 `long sturmpart(GEN pol, GEN a = NULL, GEN b = NULL)`. Also available is `long sturm(GEN pol)` (total number of real roots).

3.7.27 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, possibly empty. To get a vector in all cases, use `concat([], polsubcyclo(n,d))`

The function `galoissubcyclo` allows to specify more closely which sub-Abelian extension should be computed.

The library syntax is `GEN polsubcyclo(long n, long d, long v = -1)`, where `v` is a variable number.

3.7.28 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 `GEN sylvestermatrix(GEN x, GEN y)`.

3.7.29 polsym(x, n): creates the column vector of the symmetric powers of the roots of the polynomial x up to power n, using Newton's formula.

The library syntax is `GEN polsym(GEN x, long n)`.

3.7.30 poltchebi$(n, \{v = x\})$: creates the n^{th} Chebyshev polynomial T_n of the first kind in variable v. This function is retained for backward compatibility only. Use `polchebyshev`.

The library syntax is `GEN polchebyshev1(long n, long v = -1)`, where `v` is a variable number.

3.7.31 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 `GEN polzag(long n, long m)`.

3.7.32 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 `GEN convol(GEN x, GEN y)`.

3.7.33 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 `GEN laplace(GEN x)`.

3.7.34 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 `GEN recip(GEN x)`.

3.7.35 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]))
  ***   at top-level: subst(1,x,Mat([0,1])
  ***                 ^--------------------
  *** subst: 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. Finally, if x is a vector, matrix or list, the substitution is applied to each individual entry.

Use the function `substvec` to replace several variables at once, or the function `substpol` to replace a polynomial expression.

The library syntax is `GEN gsubst(GEN x, long y, GEN z)`, where `y` is a variable number.

3.7.36 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 behavior as `subst` above apply.

The difference with `subst` is that y is allowed to be any polynomial here. The substitution is done moding out all components of x (recursively) by $y - t$, where t is a new free variable of lowest priority. Then substituting t by z in the resulting expression. 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 `GEN gsubstpol(GEN x, GEN y, GEN z)`. Further, `GEN gdeflate(GEN T, long v, long d)` attempts to write $T(x)$ in the form $t(x^d)$, where $x =$`pol_x`(v), and returns `NULL` if the substitution fails (for instance in the example `%2` above).

3.7.37 substvec(x, v, w): v being a vector of monomials of degree 1 (variables), w a vector of expressions of the same length, replace in the expression x all occurrences 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 `GEN gsubstvec(GEN x, GEN v, GEN w)`.

3.7.38 taylor$(x, t, \{d = seriesprecision\})$: Taylor expansion around 0 of x with respect to the simple variable t. x can be of any reasonable type, for example a rational function. Contrary to `Ser`, which takes the valuation into account, this function adds $O(t^d)$ to all components of x.

```
? taylor(x/(1+y), y, 5)
%1 = (y^4 - y^3 + y^2 - y + 1)*x + O(y^5)
? Ser(x/(1+y), y, 5)
 ***   at top-level: Ser(x/(1+y),y,5)
 ***                 ^----------------
 *** Ser: main variable must have higher priority in gtoser.
```

The library syntax is `GEN tayl(GEN x, long t, long precdl)`, where `t` is a variable number.

3.7.39 thue$(tnf, a, \{sol\})$: returns all solutions of the equation $P(x, y) = a$ in integers x and y, where *tnf* was created with `thueinit`(P). If present, *sol* must contain the solutions of $\mathrm{Norm}(x) = a$ modulo units of positive norm in the number field defined by P (as computed by `bnfisintnorm`). If there are infinitely many solutions, an error will be issued.

If *tnf* was computed without assuming GRH (flag 1 in `thueinit`), then the result is unconditional. Otherwise, it depends in principle of the truth of the GRH, but may still be unconditionally correct in some favourable cases. The result is conditional on the GRH if $a \neq \pm 1$ and, P has a single irreducible rational factor, whose associated tentative class number h and regulator R (as computed assuming the GRH) satisfy

- $h > 1$,
- $R/0.2 > 1.5$.

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 $(x, y) = (1, 1)$, and the result is unconditional. On the other hand:

```
? P = x^3-2*x^2+3*x-17; tnf = thueinit(P);
? thue(tnf, -15)
%2 = [[1, 1]]  \\ a priori conditional on the GRH.
? K = bnfinit(P); K.no
%3 = 3
? K.ref
%4 = 2.868218513926287367470603447549875583 4
```

This time the result is conditional. All results computed using this particular *tnf* are likewise conditional, *except* for a right-hand side of ± 1. The above result is in fact correct, so we did not just disprove the GRH:

```
? tnf = thueinit(x^3-2*x^2+3*x-17, 1 /*unconditional*/);
? thue(tnf, -15)
%4 = [[1, 1]]
```

Note that reducible or non-monic polynomials are allowed:

```
? tnf = thueinit((2*x+1)^5 * (4*x^3-2*x^2+3*x-17), 1);
? thue(tnf, 128)
%2 = [[-1, 0], [1, 0]]
```

Reducible polynomials are in fact much easier to handle.

The library syntax is `GEN thue(GEN tnf, GEN a, GEN sol = NULL)`.

3.7.40 thueinit$(P, \{flag = 0\})$: initializes the *tnf* corresponding to P, a univariate polynomial with integer coefficients. The result is meant to be used in conjunction with `thue` to solve Thue equations $P(X/Y)Y^{\deg P} = a$, where a is an integer.

If *flag* is non-zero, certify results unconditionally. Otherwise, assume GRH, this being much faster of course. In the latter case, the result may still be unconditionally correct, see `thue`. For instance in most cases where P is reducible (not a pure power of an irreducible), *or* conditional computed class groups are trivial *or* the right hand side is ± 1, then results are always unconditional.

The library syntax is `GEN thueinit(GEN P, long flag, long 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. We thus assume that 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 that the base ring is $\mathbf{Z}$.

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 non-zero value of *flag* may improve on the default behavior if the input number is known to a *huge* accuracy, and you suspect the last bits are incorrect (this truncates the number, throwing away the least significant bits), but default values are usually sufficient:

```
\\\\\\\\\ LLL
? \p200
? algdep(2^(1/6)+3^(1/5), 30);      \\ wrong in 0.8s
? algdep(2^(1/6)+3^(1/5), 30, 100); \\ wrong in 0.4s
? algdep(2^(1/6)+3^(1/5), 30, 170); \\ right in 0.8s
? algdep(2^(1/6)+3^(1/5), 30, 200); \\ wrong in 1.0s
? \p250
? algdep(2^(1/6)+3^(1/5), 30);      \\ right in 1.0s
? algdep(2^(1/6)+3^(1/5), 30, 200); \\ right in 1.0s
? \p500
? algdep(2^(1/6)+3^(1/5), 30);      \\ right in 2.9s
? \p1000
? algdep(2^(1/6)+3^(1/5), 30);      \\ right in 10.6s
\\\\\\\\\ PSLQ
? \p200
? algdep(2^(1/6)+3^(1/5), 30, -3);  \\ failure in 15s
```

```
? \p250
? algdep(2^(1/6)+3^(1/5), 30, -3);  \\ right in 20s
? \p500
? algdep(2^(1/6)+3^(1/5), 30, -3);  \\ right in 52s
? \p1000
? algdep(2^(1/6)+3^(1/5), 30, -3);  \\ right in 164s
```

The changes in `defaultprecision` only affect the quality of the initial approximation to $2^{1/6}+3^{1/5}$, `algdep` itself uses exact operations (the size of its operands depend on the accuracy of the input of course: more accurate input means slower operations).

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 15 times slower at that accuracy).

The above example is the test case studied in a 2000 paper by Borwein and Lisonek: Applications of integer relation algorithms, *Discrete Math.*, **217**, p. 65–82. The paper concludes 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 200 as much time as the current version.

The library syntax is `GEN algdep0(GEN x, long k, long flag)`. Also available is `GEN algdep(GEN x, long k)` (*flag* $= 0$).

3.8.2 charpoly$(A, \{v = x\}, \{flag = 3\})$: 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. Let n be the dimension of A.

If *flag* $= 0$, same method (Le Verrier's) as for computing the adjoint matrix, i.e. using the traces of the powers of A. Assumes that $n!$ is invertible; uses $O(n^4)$ scalar operations.

If *flag* $= 1$, uses Lagrange interpolation which is usually the slowest method. Assumes that $n!$ is invertible; uses $O(n^4)$ scalar operations.

If *flag* $= 2$, uses the Hessenberg form. Assumes that the base ring is a field. Uses $O(n^3)$ scalar operations, but suffers from coefficient explosion unless the base field is finite or $\mathbf{R}$.

If *flag* $= 3$, uses Berkowitz's division free algorithm, valid over any ring (commutative, with unit). Uses $O(n^4)$ scalar operations.

If *flag* $= 4$, x must be integral. Uses a modular algorithm.

In practice one should use the default (Berkowitz) unless the base ring is $\mathbf{Z}$ (use *flag* $= 4$) or a field where coefficient explosion does not occur, e.g. a finite field or the reals (use *flag* $= 2$).

The library syntax is `GEN charpoly0(GEN A, long v = -1, long flag)`, where `v` is a variable number. Also available are `GEN caract(GEN A, long v)` (*flag* $= 1$), `GEN carhess(GEN A, long v)` (*flag* $= 2$), `GEN carberkowitz(GEN A, long v)` (*flag* $= 3$) and `GEN caradj(GEN A, long v, GEN *pt)`. In this last case, if *pt* is not `NULL`, `*pt` receives 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 `GEN concat(GEN x, GEN y = NULL)`. `GEN concat1(GEN x)` is a shortcut for `concat(x,NULL)`.

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. To get meaningful results in the latter case, the parameter *flag* should be smaller than 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. Faster than the alternatives if it converges, especially when the accuracy is much larger than what is really necessary; usually diverges, though.

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. Don't use it!

The library syntax is `GEN lindep0(GEN x, long flag)`. Also available are `GEN lindep(GEN x)` (*flag* $= 0$) `GEN lindep2(GEN x, long bit)` (*flag* ≥ 0, bypasses the check for p-adic inputs) and `GEN deplin(GEN x)` (*flag* $= -2$).

3.8.5 listcreate(): creates an empty list. This routine used to have a mandatory argument, which is now ignored (for backward compatibility). In fact, this function has become redundant and obsolete; it will disappear in future versions of PARI: just use `List()`

3.8.6 listinsert(L, x, n): inserts the object x at position n in L (which must be of type `t_LIST`). This has complexity $O(\#L - n + 1)$: all the remaining elements of *list* (from position $n+1$ onwards) are shifted to the right.

The library syntax is `GEN listinsert(GEN L, GEN x, long n)`.

3.8.7 listkill(L): obsolete, retained for backward compatibility. Just use `L = List()` instead of `listkill(L)`. In most cases, you won't even need that, e.g. local variables are automatically cleared when a user function returns.

The library syntax is `void listkill(GEN L)`.

3.8.8 listpop$(list, \{n\})$: removes the n-th element of the list *list* (which must be of type `t_LIST`). If n is omitted, or greater than the list current length, removes the last element. This runs in time $O(\#L - n + 1)$.

The library syntax is `void listpop(GEN list, long n)`.

3.8.9 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 length, appends x. You may put an element into an occupied cell (not changing the list length), but it is easier to use the standard `list[n] = x` construct. This runs in time $O(\#L)$ in the worst case (when the list must be reallocated), but in time $O(1)$ on average: any number of successive `listput`s run in time $O(\#L)$, where $\#L$ denotes the list *final* length.

The library syntax is `GEN listput(GEN list, GEN x, long n)`.

3.8.10 listsort$(L, \{flag = 0\})$: sorts the `t_LIST` *list* in place. If *flag* is non-zero, suppresses all repeated coefficients. This is faster than the `vecsort` command since no copy has to be made. No value returned.

The library syntax is `void listsort(GEN L, long flag)`.

3.8.11 matadjoint$(x, \{flag = 0\})$: 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 of dimension n. If *flag* is 0 or omitted, use a fast algorithm which assumes that $n!$ is invertible. If *flag* is 1, use a slower division-free algorithm.

```
? a = [1,2,3;3,4,5;6,7,8] * Mod(1,2);
? matadjoint(a)
 ***   at top-level: matadjoint([1,2,3;3,
 ***                 ^--------------------
 *** matadjoint: impossible inverse modulo: Mod(0, 2).
? matadjoint(a, 1)  \\ use safe algorithm
%2 =
[Mod(1, 2) Mod(1, 2) Mod(0, 2)]

[Mod(0, 2) Mod(0, 2) Mod(0, 2)]

[Mod(1, 2) Mod(1, 2) Mod(0, 2)]
```

Both algorithms use $O(n^4)$ operations in the base ring.

The library syntax is `GEN matadjoint0(GEN x, long flag)`. Also available are `GEN adj(GEN x)` (*flag*=0) and `GEN adjsafe(GEN x)` (*flag*=1).

3.8.12 matcompanion(x): the left companion matrix to the polynomial x.

The library syntax is `GEN matcompanion(GEN x)`.

3.8.13 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 `GEN det0(GEN x, long flag)`. Also available are `GEN det(GEN x)` ($flag = 0$) and `GEN det2(GEN x)` ($flag = 1$).

3.8.14 matdetint(x): x being an $m \times n$ matrix with integer coefficients, this function computes a non-zero *multiple* of the determinant of the lattice generated by the columns of x if it has maximal rank m, and returns zero otherwise, using the Gauss-Bareiss algorithm. When x is square, the exact determinant is obtained.

This function is useful in conjunction with `mathnfmod`, which needs to know such a multiple. If the rank is maximal and the matrix non-square, you can obtain the exact determinant using

```
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 `GEN detint(GEN x)`.

3.8.15 matdiagonal(x): x being a vector, creates the diagonal matrix whose diagonal entries are those of x.

The library syntax is `GEN diagonal(GEN x)`.

3.8.16 mateigen(x): gives the eigenvectors of x as columns of a matrix.

The library syntax is `GEN eigen(GEN x, long prec)`.

3.8.17 matfrobenius($M, \{flag\}, \{v = x\}$): returns the Frobenius form of the square matrix `M`. If *flag* $= 1$, returns only the elementary divisors as a vector 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 `GEN matfrobenius(GEN M, long flag, long v = -1)`, where `v` is a variable number.

3.8.18 mathess(x): returns a matrix similar to the square matrix x, which is in upper Hessenberg form (zero entries below the first subdiagonal).

The library syntax is `GEN hess(GEN x)`.

3.8.19 mathilbert(n): 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 `GEN mathilbert(long n)`.

3.8.20 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 is in general fastest but may require too much memory as the dimension gets large (bigger than 100, say), in which case you may try `mathnfmod(x, matdetint(x))` when x has maximal rank, and `mathnf(x, 4)` otherwise.

If *flag* $= 1$, outputs a two-component row vector $[H, U]$, where H is the Hermite normal form of x defined as above, and U is the unimodular transformation matrix such that $xU = [0|H]$. When the kernel is large, U has in general huge coefficients. In the worst case, the running time is exponential with respect to the dimension, but the routine behaves well in small dimension (less than 50 or 100, say).

If *flag* $= 3$, uses Batut's algorithm and output $[H, U, P]$, such that H and U are as before and P is a permutation of the rows such that P applied to xU gives H. This is in general slower than *flag* $= 1$ but the matrix U is smaller; it may still be large.

If *flag* $= 4$, as in case 1 above, but uses a 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 in general slow, although provably polynomial-time.

The library syntax is `GEN mathnf0(GEN x, long flag)`. Also available are `GEN hnf(GEN x)` (*flag* $= 0$) and `GEN hnfall(GEN 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 quite technical and the calling interface may change, they are not documented yet. Look at the code in `basemath/alglin1.c`.

3.8.21 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. Even when d is known, this is in general slower than `mathnf` but uses much less memory.

The library syntax is `GEN hnfmod(GEN x, GEN d)`.

3.8.22 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 `GEN hnfmodid(GEN x, GEN d)`.

3.8.23 matid(n): creates the $n \times n$ identity matrix.

The library syntax is `GEN matid(long n)`.

3.8.24 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` (much slower: keep the default flag!).

The library syntax is `GEN matimage0(GEN x, long flag)`. Also available is `GEN image(GEN x)` ($flag = 0$).

3.8.25 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 `GEN imagecompl(GEN x)`.

3.8.26 matindexrank(x): x being a matrix of rank r, returns a vector with two `t_VECSMALL` components 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 `GEN indexrank(GEN x)`.

3.8.27 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 $\mathbf{Q}$-vector space equal to the intersection of the spaces spanned by the columns of x and y respectively. The faster function `idealintersect` can be used to intersect fractional ideals (projective $\mathbf{Z}_K$ modules of rank 1); the slower but much more general function `nfhnf` can be used to intersect general $\mathbf{Z}_K$-modules.

The library syntax is `GEN intersect(GEN x, GEN y)`.

3.8.28 matinverseimage(x, y): given a matrix x and a column vector or matrix y, returns a preimage z of y by x if one exists (i.e such that $xz = y$), an empty vector or matrix otherwise. The complete inverse image is $z + \mathrm{Ker}x$, where a basis of the kernel of x may be obtained by `matker`.

```
? M = [1,2;2,4];
? matinverseimage(M, [1,2]~)
%2 = [1, 0]~
? matinverseimage(M, [3,4]~)
%3 = []~     \\ no solution
? matinverseimage(M, [1,3,6;2,6,12])
%4 =
[1 3 6]

[0 0 0]
? matinverseimage(M, [1,2;3,4])
%5 = [;]     \\ no solution
? K = matker(M)
%6 =
[-2]

[1]
```

The library syntax is `GEN inverseimage(GEN x, GEN y)`.

3.8.29 matisdiagonal(x): returns true (1) if x is a diagonal matrix, false (0) if not.

The library syntax is `GEN isdiagonal(GEN x)`.

3.8.30 matker$(x, \{flag = 0\})$: gives a basis for the kernel of the matrix x as columns of a matrix. The matrix can have entries of any type, provided they are compatible with the generic arithmetic operations ($+$, $\times$ and $/$).

If x is known to have integral entries, set $flag = 1$.

The library syntax is `GEN matker0(GEN x, long flag)`. Also available are `GEN ker(GEN x)` ($flag = 0$), `GEN keri(GEN x)` ($flag = 1$).

3.8.31 matkerint$(x, \{flag = 0\})$: gives an LLL-reduced $\mathbf{Z}$-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 an integer LLL algorithm.

If $flag = 1$, uses `matrixqz`$(x, -2)$. Many orders of magnitude slower than the default: never use this.

The library syntax is `GEN matkerint0(GEN x, long flag)`. See also `GEN kerint(GEN x)` ($flag = 0$), which is a trivial wrapper around

```
ZM_lll(ZM_lll(x, 0.99, LLL_KER), 0.99, LLL_INPLACE);
```

Remove the outermost `ZM_lll` if LLL-reduction is not desired (saves time).

3.8.32 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 `GEN matmuldiagonal(GEN x, GEN d)`.

3.8.33 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 `GEN matmultodiagonal(GEN x, GEN y)`.

3.8.34 matpascal$(n, \{q\})$: 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 `GEN matqpascal(long n, GEN q = NULL)`. Also available is `GEN matpascal(GEN x)`.

3.8.35 matrank(x): rank of the matrix x.

The library syntax is `long rank(GEN x)`.

3.8.36 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.

3.8.37 matrixqz$(A, \{p = 0\})$: A being an $m \times n$ matrix in $M_{m,n}(\mathbf{Q})$, let $\mathrm{Im}_{\mathbf{Q}} A$ (resp. $\mathrm{Im}_{\mathbf{Z}} A$) the $\mathbf{Q}$-vector space (resp. the $\mathbf{Z}$-module) spanned by the columns of A. This function has varying behavior depending on the sign of p:

If $p \geq 0$, A is assumed to have maximal rank $n \leq m$. The function returns a matrix $B \in M_{m,n}(\mathbf{Z})$, with $\mathrm{Im}_{\mathbf{Q}} B = \mathrm{Im}_{\mathbf{Q}} A$, such that the GCD of all its $n \times n$ minors is coprime to p; in particular, if $p = 0$ (default), this GCD is 1.

```
? minors(x) = vector(#x[,1], i, matdet( vecextract(x, Str("^",i), "..") ));
? A = [3,1/7; 5,3/7; 7,5/7]; minors(A)
%1 = [4/7, 8/7, 4/7]   \\ determinants of all 2x2 minors
? B = matrixqz(A)
%2 =
[3 1]

[5 2]

[7 3]
? minors(%)
%3 = [1, 2, 1]   \\ B integral with coprime minors
```

If $p = -1$, returns the HNF basis of the lattice $\mathbf{Z}^n \cap \mathrm{Im}_{\mathbf{Z}} A$.

If $p = -2$, returns the HNF basis of the lattice $\mathbf{Z}^n \cap \mathrm{Im}_{\mathbf{Q}} A$.

```
? matrixqz(A,-1)
%4 =
[8 5]

[4 3]

[0 1]

? matrixqz(A,-2)
%5 =
```

```
[2 -1]

[1 0]

[0 1]
```

The library syntax is `GEN matrixqz0(GEN A, GEN p = NULL)`.

3.8.38 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 `GEN matsize(GEN x)`.

3.8.39 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, normalized so that $d_n \mid d_{n-1} \mid \ldots \mid d_1$.

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. If X is not a square matrix, then D will be a square diagonal matrix padded with zeros on the left or the top.

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 'vector of elementary divisors' or $[U, V, D]$ as would normally be output with the cleanup flag unset.

The library syntax is `GEN matsnf0(GEN X, long flag)`.

3.8.40 matsolve(M, B): M being an invertible matrix and B a column vector, finds the solution X of $MX = B$, using Gaussian elimination. This has the same effect as, but is a bit faster, than $M^{-1} * B$.

The library syntax is `GEN gauss(GEN M, GEN B)`.

3.8.41 matsolvemod$(M, D, B, \{flag = 0\})$: M being any integral matrix, D a column vector of non-negative integer moduli, and B an integral column vector, gives a small integer solution to the system of congruences $\sum_i m_{i,j} x_j \equiv b_i \pmod{d_i}$ if one exists, otherwise returns zero. Shorthand notation: B (resp. D) can be given as a single integer, in which case all the b_i (resp. d_i) above are taken to be equal to B (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]~
? matsolvemod(M, [3,0]~, [1,2]~) \\ x + 2y = 1 (mod 3), 3x + 4y = 2 (in Z)
%4 = [6, -4]~
```

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 `GEN matsolvemod0(GEN M, GEN D, GEN B, long flag)`. Also available are `GEN gaussmodulo(GEN M, GEN D, GEN B)` ($flag = 0$) and `GEN gaussmodulo2(GEN M, GEN D, GEN B)` ($flag = 1$).

3.8.42 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 `GEN suppl(GEN x)`.

3.8.43 mattranspose(x): transpose of x (also x~). This has an effect only on vectors and matrices.

The library syntax is `GEN gtrans(GEN x)`.

3.8.44 minpoly($A, \{v = x\}$): 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 `GEN minpoly(GEN A, long v = -1)`, where `v` is a variable number.

3.8.45 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 off-diagonal entries on each line represent the bilinear forms. More precisely, if (a_{ij}) denotes the output, one has

$$q(x) = \sum_i a_{ii}(x_i + \sum_{j \neq i} a_{ij} x_j)^2$$

```
? qfgaussred([0,1;1,0])
%1 =
[1/2 1]

[-1 -1/2]
```

This means that $2xy = (1/2)(x + y)^2 - (1/2)(x - y)^2$.

The library syntax is `GEN qfgaussred(GEN q)`. `GEN qfgaussred_positive(GEN q)` assumes that q is positive definite and is a little faster; returns `NULL` if a vector with negative norm occurs (non positive matrix or too many rounding errors).

3.8.46 qfjacobi(x): x being a real symmetric matrix, this gives a vector having two components: the first one is the vector of (real) eigenvalues of x, sorted in increasing order, 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 `GEN jacobi(GEN x, long prec)`.

3.8.47 qflll$(x, \{flag = 0\})$: LLL algorithm applied to the *columns* of the matrix x. The columns of x may be linearly dependent. 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. Note that if x is not of maximal rank T will not be square. The LLL parameters are $(0.51, 0.99)$, meaning that the Gram-Schmidt coefficients for the final basis satisfy $\mu_{i,j} \leq |0.51|$, and the Lovász's constant is 0.99.

If *flag* $= 0$ (default), assume that x has either exact (integral or rational) or real floating point entries. The matrix is rescaled, converted to integers and the behavior is then as in *flag* $= 1$.

If *flag* $= 1$, assume that x is integral. Computations involving Gram-Schmidt vectors are approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr). Adapted from Nguyen and Stehlé's algorithm and Stehlé's code (`fplll-1.3`).

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 faster than *flag* $= 1$, esp. when one row is huge compared to the other rows (knapsack-style), and should quickly produce relatively short vectors. The resulting basis is *not* LLL-reduced in general. If LLL reduction is eventually desired, avoid this partial reduction: applying LLL to the partially reduced matrix is significantly *slower* than starting from a knapsack-type lattice.

If *flag* $= 4$, as *flag* $= 1$, returning a vector $[K, T]$ of matrices: the columns of K represent a basis of the integer kernel of x (not LLL-reduced in general) and T is the transformation matrix 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 `GEN qflll0(GEN x, long flag)`. Also available are `GEN lll(GEN x)` (*flag* $= 0$), `GEN lllint(GEN x)` (*flag* $= 1$), and `GEN lllkerim(GEN x)` (*flag* $= 4$).

3.8.48 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 quadratic form (not necessarily definite: x needs not have maximal rank). 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. See `qflll` for further details about the LLL implementation.

If *flag* $= 0$ (default), assume that G has either exact (integral or rational) or real floating point entries. The matrix is rescaled, converted to integers and the behavior is then as in *flag* $= 1$.

If *flag* $= 1$, assume that G is integral. Computations involving Gram-Schmidt vectors are approximate, with precision varying as needed (Lehmer's trick, as generalized by Schnorr). Adapted from Nguyen and Stehlé's algorithm and Stehlé's code (`fplll-1.3`).

flag $= 4$: G has integer entries, gives the kernel and reduced image of x.

flag $= 5$: same as 4, but G may have polynomial coefficients.

The library syntax is `GEN qflllgram0(GEN G, long flag)`. Also available are `GEN lllgram(GEN G)` (*flag* $= 0$), `GEN lllgramint(GEN G)` (*flag* $= 1$), and `GEN lllgramkerim(GEN G)` (*flag* $= 4$).

3.8.49 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, storing at most m vectors (no limit if m is omitted). The function searches for the minimal non-zero vectors if b is omitted. The behavior is undefined if x is not positive definite (a "precision too low" error is most likely, although more precise error messages are possible). The precise behavior 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, unless m was omitted). The vectors are returned in no particular order.

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 these two cases, x must 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 raises 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!

```
? x = matid(2);
? qfminim(x)  \\ 4 minimal vectors of norm 1: ±[0,1], ±[1,0]
%2 = [4, 1, [0, 1; 1, 0]]
? { x =
[4, 2, 0, 0, 0,-2, 0, 0, 0, 0, 0, 0, 1,-1, 0, 0, 0, 1, 0,-1, 0, 0, 0,-2;
 2, 4,-2,-2, 0,-2, 0, 0, 0, 0, 0, 0, 0,-1, 0, 0, 0, 0, 0,-1, 0, 1,-1,-1;
 0,-2, 4, 0,-2, 0, 0, 0, 0, 0, 0, 0,-1, 1, 0, 0, 1, 0, 0, 1,-1,-1, 0, 0;
 0,-2, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0, 1,-1, 0, 0, 0, 1,-1, 0, 1,-1, 1, 0;
 0, 0,-2, 0, 4, 0, 0, 0, 1,-1, 0, 0, 1, 0, 0, 0,-2, 0, 0,-1, 1, 1, 0, 0;
-2, -2,0, 0, 0, 4,-2, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,-1, 1, 1;
 0, 0, 0, 0, 0,-2, 4,-2, 0, 0, 0, 0, 0, 1, 0, 0, 0,-1, 0, 0, 0, 1,-1, 0;
 0, 0, 0, 0, 0, 0,-2, 4, 0, 0, 0, 0,-1, 0, 0, 0, 0, 0,-1,-1,-1, 0, 1, 0;
 0, 0, 0, 0, 1,-1, 0, 0, 4, 0,-2, 0, 1, 1, 0,-1, 0, 1, 0, 0, 0, 0, 0, 0;
 0, 0, 0, 0,-1, 0, 0, 0, 0, 4, 0, 0, 1, 1,-1, 1, 0, 0, 0, 1, 0, 0, 1, 0;
 0, 0, 0, 0, 0, 0, 0, 0,-2, 0, 4,-2, 0,-1, 0, 0, 0,-1, 0,-1, 0, 0, 0, 0;
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,-2, 4,-1, 1, 0, 0,-1, 1, 0, 1, 1, 1,-1, 0;
 1, 0,-1, 1, 1, 0, 0,-1, 1, 1, 0,-1, 4, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1,-1;
-1,-1, 1,-1, 0, 0, 1, 0, 1, 1,-1, 1, 0, 4, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1;
 0, 0, 0, 0, 0, 0, 0, 0, 0,-1, 0, 0, 0, 1, 4, 0, 0, 0, 1, 0, 0, 0, 0, 0;
 0, 0, 0, 0, 0, 0, 0, 0,-1, 1, 0, 0, 1, 1, 0, 4, 0, 0, 0, 0, 1, 1, 0, 0;
 0, 0, 1, 0,-2, 0, 0, 0, 0, 0, 0,-1, 0, 0, 0, 0, 4, 1, 1, 1, 0, 0, 1, 1;
 1, 0, 0, 1, 0, 0,-1, 0, 1, 0,-1, 1, 1, 0, 0, 0, 1, 4, 0, 1, 1, 0, 1, 0;
 0, 0, 0,-1, 0, 1, 0,-1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 4, 0, 1, 1, 0, 1;
-1, -1,1, 0,-1, 1, 0,-1, 0, 1,-1, 1, 0, 1, 0, 0, 1, 1, 0, 4, 0, 0, 1, 1;
 0, 0,-1, 1, 1, 0, 0,-1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 4, 1, 0, 1;
```

```
 0, 1,-1,-1, 1,-1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 4, 0, 1;
 0,-1, 0, 1, 0, 1,-1, 1, 0, 1, 0,-1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 4, 1;
-2,-1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,-1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 4]; }
? qfminim(x,,0)  \\ the Leech lattice has 196560 minimal vectors of norm 4
time = 648 ms.
%4 = [196560, 4, [;]]
? qfminim(x,,0,2); \\ safe algorithm. Slower and unnecessary here.
time = 18,161 ms.
%5 = [196560, 4.000061035156250000, [;]]
```

In the last example, we store 0 vectors to limit memory use. All minimal vectors are nevertheless enumerated. Provided `parisize` is about 50MB, `qfminim(x)` succeeds in 2.5 seconds.

The library syntax is `GEN qfminim0(GEN x, GEN b = NULL, GEN m = NULL, long flag, long prec)`. Also available are `GEN minim(GEN x, GEN b = NULL, GEN m = NULL)` (*flag* $= 0$), `GEN minim2(GEN x, GEN b = NULL, GEN m = NULL)` (*flag* $= 1$).

3.8.50 qfperfection(G): G 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.

Since this requires computing the minimal vectors, the computations can become very lengthy as the dimension of x grows.

The library syntax is `GEN perf(GEN G)`.

3.8.51 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_VEC`

The library syntax is `GEN qfrep0(GEN q, GEN B, long flag)`.

3.8.52 qfsign(x): returns $[p, m]$ the signature of the quadratic form represented by the symmetric matrix x. Namely, p (resp. m) is the number of positive (resp. negative) eigenvalues of x.The result is computed using Gaussian reduction.

The library syntax is `GEN qfsign(GEN x)`.

3.8.53 setintersect(x, y): intersection of the two sets x and y (see `setisset`). The function also works if both x and y are vectors of strictly increasing entries, according to `<`); in that case we return a vector of strictly increasing entries, not a set. Otherwise, the result is undefined.

The library syntax is `GEN setintersect(GEN x, GEN y)`.

3.8.54 setisset(x): returns true (1) if x is a set, false (0) if not. In PARI, a set is a row vector whose entries are strictly increasing `t_STR`s. To convert any object into a set (this is most useful for vectors, of course), use the function `Set`.

```
? a = [3, 1, 1, 2];
? setisset(a)
%2 = 0
? Set(a)
%3 = ["1", "2", "3"]
```

The library syntax is `long setisset(GEN x)`.

3.8.55 setminus(x, y): difference of the two sets x and y (see `setisset`), i.e. set of elements of x which do not belong to y. The function also works if both x and y are vectors of strictly increasing entries, according to `<`); in that case we return a vector of strictly increasing entries, not a set. Otherwise, the result is undefined.

The library syntax is `GEN setminus(GEN x, GEN y)`.

3.8.56 setsearch($S, x, \{flag = 0\}$): searches if x belongs to the set S (see `setisset`). A set is a vector of `t_STR`, but this function works also if S is a arbitrary *sorted* vector or list (see `listsort`): if x is not a `t_STR`, we first replace it by `Str`(x) *unless* the first element of S is also not a `t_STR`.

If x belongs to the set and *flag* is zero or omitted, returns the index j such that $S[j] = x$, otherwise returns 0. If *flag* is non-zero returns the index j where x should be inserted, and 0 if it already belongs to S (this is meant to be used in conjunction with `listinsert`, see below).

```
? T = [2,3,5,7]; S = Set(T);
? setsearch(S, 2)      \\ search in a true set, t_INT 2 converted to string
%2 = 1
? setsearch(S, Str(2)) \\ search in a true set, no need for conversion
%3 = 1
? setsearch(T, 2)      \\ search in a sorted vector, no need for conversion
%4 = 1
? setsearch(T, Str(2)) \\ search in a sorted vector, t_STR "2" not found
%5 = 0
? setsearch(S, 4)      \\ not found
%6 = 0
? setsearch(S, 4, 1)   \\ should have been inserted at index 3
%7 = 3
```

The library syntax is `long setsearch(GEN S, GEN x, long flag)`.

3.8.57 setunion(x, y): union of the two sets x and y (see `setisset`). The function also works if both x and y are vectors of strictly increasing entries, according to `<`); in that case we return a vector of strictly increasing entries, not a set. Otherwise, the result is undefined.

The library syntax is `GEN setunion(GEN x, GEN y)`.

3.8.58 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 `GEN gtrace(GEN x)`.

3.8.59 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 `GEN extract0(GEN x, GEN y, GEN z = NULL)`.

3.8.60 vecsort$(x, \{cmp\}, \{flag = 0\})$: sorts the vector x in ascending order, using a mergesort method. x must be a list, vector or matrix (seen as a vector of its columns). Note that mergesort is stable, hence the initial ordering of "equal" entries (with respect to the sorting criterion) is not changed.

If `cmp` is omitted, we use the standard comparison function `<`, thereby restricting the possible types for the elements of x (integers, fractions or reals). If `cmp` is present, it is understood as a comparison function and we sort according to it. The following possibilities exist:

• an integer k: sort according to the value of the k-th subcomponents of the components of x.

• a vector: sort lexicographically according to the components listed in the vector. For example, if `cmp` $= [2, 1, 3]$, sort 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.

• a comparison function (`t_CLOSURE`), with two arguments x and y, and returning an integer which is < 0, > 0 or $= 0$ if $x < y$, $x > y$ or $x = y$ respectively. The `sign` function is very useful in this context:

```
? vecsort([3,0,2; 1,0,2],lex) \\ sort columns according to lex order
%1 =
[2 3]

[2 1]
? vecsort(v, (x,y)->sign(y-x))             \\ reverse sort
? vecsort(v, (x,y)->sign(abs(x)-abs(y)))   \\ sort by increasing absolute value
? cmp(x,y) = my(dx = poldisc(x), dy = poldisc(y)); sign(abs(dx) - abs(dy))
? vecsort([x^2+1, x^3-2, x^4+5*x+1], cmp)
```

The last example used the named `cmp` instead of an anonymous function, and sorts polynomials with respect to the absolute value of their discriminant. A more efficient approach would use precomputations to ensure a given discriminant is computed only once:

```
? DISC = vector(#v, i, abs(poldisc(v[i])));
? perm = vecsort(vector(#v,i,i), (x,y)->sign(DISC[x]-DISC[y]))
? vecextract(v, perm)
```

Similar ideas apply whenever we sort according to the values of a function which is expensive to compute.

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.

• 8: remove "duplicate" entries with respect to the sorting function (keep the first occurring entry). For example:

```
? vecsort([Pi,Mod(1,2),z], (x,y)->0, 8)    \\ make everything compare equal
%1 = [3.14159265358979323846264338327]
? vecsort([[2,3],[0,1],[0,3]], 2, 8)
%2 = [[0, 1], [2, 3]]
```

The library syntax is `GEN vecsort0(GEN x, GEN cmp = NULL, long flag)`.

3.8.61 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]
```

3.8.62 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.

3.8.63 vectorv$(n, \{X\}, \{expr = 0\})$: as `vector`, but returns a column vector (type `t_COL`).

3.9 Sums, products, integrals and similar functions.

Although the `gp` calculator is programmable, it is useful to have a number of preprogrammed 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. On the other hand, numerical routines code a function (to be integrated, summed, etc.) with two parameters named

```
GEN (*eval)(void*,GEN)
void *E;  \\ context: eval(E, x) must evaluate your function at x.
```

see the Libpari manual for details.

Numerical integration. Starting with version 2.2.9 the "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`, you must split the integral into a sum of subintegrals where the function has no singularities except at the endpoints. Polynomials in logarithms are not considered singular, and neglecting these logs, singularities are assumed to be algebraic (asymptotic to $C(x-a)^{-\alpha}$ for some $\alpha > -1$ 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 derivnum($X = a, expr$): numerical derivation of *expr* with respect to X at $X = a$.

```
? derivnum(x=0,sin(exp(x))) - cos(1)
%1 = -1.262177448 E-29
```

A clumsier approach, which would not work in library mode, is

```
? f(x) = sin(exp(x))
? f'(0) - cos(1)
%1 = -1.262177448 E-29
```

When a is a power series, compute `derivnum(t=a,f)` as $f'(a) = (f(a))'/a'$.

The library syntax is **derivnum**(`void *E, GEN (*eval)(void*,GEN), GEN a, long prec`). Also available is `GEN derivfun(void *E, GEN (*eval)(void *, GEN), GEN a, long prec)`, which also allows power series for a.

3.9.2 intcirc($X = a, R, expr, \{tab\}$): numerical integration of $(2i\pi)^{-1} expr$ with respect to X on the circle $|X - a| = R$. 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
%1 = -2.398082982 E-104 - 7.94487211 E-107*I
```

The library syntax is **intcirc**(`void *E, GEN (*eval)(void*,GEN), GEN a,GEN R,GEN tab, long prec`).

3.9.3 intfouriercos($X = a, b, z, expr, \{tab\}$): numerical integration of $expr(X)\cos(2\pi z X)$ from a to b, in other words Fourier cosine transform (from a to b) of the function represented by *expr*. Endpoints 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)(void*,GEN), GEN a, GEN b, GEN z, GEN tab, long prec`).

3.9.4 intfourierexp$(X = a, b, z, expr, \{tab\})$: numerical integration of $expr(X)\exp(-2i\pi zX)$ from a to b, in other words Fourier transform (from a to b) of the function represented by *expr*. Note the minus sign. Endpoints 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)(void*,GEN), GEN a, GEN b, GEN z, GEN tab, long prec`).

3.9.5 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*. Endpoints 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)(void*,GEN), GEN a, GEN b, GEN z, GEN tab, long prec`).

3.9.6 intfuncinit$(X = a, b, expr, \{flag = 0\}, \{m = 0\})$: initialize 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)(void*,GEN), 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.

3.9.7 intlaplaceinv$(X = sig, z, expr, \{tab\})$: numerical integration of $(2i\pi)^{-1}expr(X)e^{Xz}$ with respect to X on the line $\Re(X) = sig$. 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 abscissa of integration, and then the integrand 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 abscissa 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)(void*,GEN), GEN sig,GEN z, GEN tab, long prec`).

3.9.8 intmellininv$(X = sig, z, expr, \{tab\})$: numerical integration of $(2i\pi)^{-1}expr(X)z^{-X}$ with respect to X on the line $\Re(X) = sig$, 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 abscissa of integration, and then the 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)(void*,GEN), GEN sig, GEN z, GEN tab, long prec`).

3.9.9 intmellininvshort(sig, z, tab): numerical integration of $(2i\pi)^{-1}s(X)z^{-X}$ with respect to X on the line $\Re(X) = sig$. 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 `GEN intmellininvshort(GEN sig, GEN z, GEN tab, long prec)`.

3.9.10 intnum$(X = a, b, expr, \{tab\})$: numerical integration of *expr* on $]a, b[$ with respect to X. The integrand may have values belonging to a vector space over the real numbers; in particular, it can be complex-valued or vector-valued. But it is assumed that the function is regular on $]a, b[$. If the endpoints a and b are finite and the function is regular there, the situation is simple:

```
? intnum(x = 0,1, x^2)
%1 = 0.33333333333333333333333333333
? intnum(x = 0,Pi/2, [cos(x), sin(x)])
%2 = [1.0000000000000000000000000000, 1.0000000000000000000000000000]
```

An endpoint equal to $\pm\infty$ is coded as the single-component vector $[\pm 1]$. You are welcome to set, e.g `oo = [1]` or `INFINITY = [1]`, then using `+oo`, `-oo`, `-INFINITY`, etc. will have the expected behavior.

```
? oo = [1];  \\ for clarity
? intnum(x = 1,+oo, 1/x^2)
%2 = 1.0000000000000000000000000000
```

In basic usage, it is assumed that the function does not decrease exponentially fast at infinity:

```
? intnum(x=0,+oo, exp(-x))
  ***   at top-level: intnum(x=0,+oo,exp(-
  ***                 ^--------------------
  *** exp: exponent (expo) overflow
```

We shall see in a moment how to avoid the last problem, after describing the last argument *tab*, which is both optional and technical. The routine uses weights, which are mostly independent of the function being integrated, evaluated at many sampling points. If *tab* is

• a positive integer m, we use 2^m sampling points, hopefully increasing accuracy. But note that the running 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. If *tab* is omitted, the algorithm guesses a reasonable value for m depending on the current precision only, which should be sufficient for regular functions. That value may be obtained from `intnumstep`, and increased in case of difficulties.

• a set of integration tables as output by `intnuminit`, they are used directly. This is useful if several integrations of the same type are performed (on the same kind of interval and functions, for a given accuracy), in particular for multivariate integrals, since we then skip expensive precomputations.

Specifying the behavior at endpoints. This is done as follows. An endpoint a is either given as such (a scalar, real or complex, or $[\pm 1]$ for $\pm\infty$), or as a two component vector $[a, \alpha]$, to indicate the behavior of the integrand in a neighborhood of a.

If a is finite, the code $[a, \alpha]$ means the function has a singularity of the form $(x-a)^\alpha$, up to logarithms. (If $\alpha \geq 0$, we only assume the function is regular, which is the default assumption.) If a wrong singularity exponent is used, the result will lose a catastrophic number of decimals:

```
? intnum(x=0, 1, x^(-1/2))          \\ assume x^{-1/2} is regular at 0
%1 = 1.9999999999999999999990291881
? intnum(x=[0,-1/2], 1, x^(-1/2))   \\ no, it's not
%2 = 2.0000000000000000000000000000
? intnum(x=[0,-1/10], 1, x^(-1/2))
%3 = 1.9999999999999999999999946438 \\ using a wrong exponent is bad
```

If a is $\pm\infty$, which is coded as $[\pm 1]$, the situation is more complicated, and $[[\pm 1], \alpha]$ means:

• $\alpha = 0$ (or no α at all, i.e. simply $[\pm 1]$) assumes that the integrand 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 $\exp(-\alpha x)$. This includes oscillating but quickly decreasing functions such as $\exp(-x)\sin(x)$.

```
? oo = [1];
? intnum(x=0, +oo, exp(-2*x))
  ***   at top-level: intnum(x=0,+oo,exp(-
  ***                 ^--------------------
  *** exp: exponent (expo) overflow
? intnum(x=0, [+oo, 2], exp(-2*x))
%1 = 0.50000000000000000000000000000 \\ OK!
? intnum(x=0, [+oo, 4], exp(-2*x))
%2 = 0.49999999999999999999961990984 \\ wrong exponent => imprecise result
? intnum(x=0, [+oo, 20], exp(-2*x))
%2 = 0.49995249977390712838045102 27 \\ disaster
```

• $\alpha < -1$ 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 non-oscillating function tending slowly to 0 (e.g. like a negative power of x), then

• $\alpha = k * I$ assumes that the function behaves like $\cos(kx)g(x)$.

• $\alpha = -k * I$ 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 $\cos(kx + a)$.

Note. If $f(x) = \cos(kx)g(x)$ 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 $[[\pm 1], k*I]$, but a good rule of thumb is that if the oscillations are much weaker than the exponential decrease, choose $[[\pm 1], \alpha]$, otherwise choose $[[\pm 1], k*I]$, 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 integrand is almost always a product of an exponentially decreasing and an oscillating factor. 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.

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 67 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 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)(void*,GEN), GEN a,GEN b,GEN tab, long prec`), where an omitted *tab* is coded as `NULL`.

3.9.11 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 abscissa and weight corresponding to $t = 0$ ($\phi(0)$ and $\phi'(0)$). $tab[4]$ and $tab[5]$ contain the abscissas 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 abscissas 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 `GEN intnuminit(GEN a, GEN b, long m, long prec)`.

3.9.12 intnuminitgen$(t, a, b, ph, \{m = 0\}, \{flag = 0\})$: initialize tables for integrations from a to b using abscissas $ph(t)$ and weights $ph'(t)$. Note that there is no equal sign after the variable name t since t always goes from $-\infty$ to $+infty$, but it is $ph(t)$ which goes from a to b, and this is not checked. If *flag*= 1 or 2, multiply the reserved table length by 4^{flag}, to avoid corresponding error.

The library syntax is **intnuminitgen**(`void *E, GEN (*eval)(void*,GEN), GEN a, GEN b, long m, long flag, long prec`)

3.9.13 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)(void*,GEN), 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`.

3.9.14 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 `long intnumstep(long prec)`.

3.9.15 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 = 128 ms.
? prod(i=1, 100, 1 - X^i, 1 + O(X^101))
time = 8 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)
```

Of course, in this specific case, it is faster to use `eta`, which is computed using Euler's formula.

```
? prod(i=1, 1000, 1 - X^i, 1 + O(X^1001));
time = 589 ms.
? \ps1000
seriesprecision = 1000 significant terms
? eta(X) - %
time = 8ms.
%4 = O(X^1001)
```

The library syntax is **produit**(`GEN a, GEN b, char *expr, GEN x`).

3.9.16 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)(void*,GEN), GEN a,GEN b, long prec`).

3.9.17 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. In particular, non-convergent products result in infinite loops. 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)(void*,GEN), GEN a, long prec`) ($flag = 0$), or `prodinf1` with the same arguments ($flag = 1$).

3.9.18 solve$(X = a, b, expr)$: find a real root of expression *expr* between a and b, under the condition $expr(X = a) * expr(X = b) \leq 0$. (You will get an error message `roots must be bracketed in solve` if this does not hold.) 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)(void*,GEN),GEN a,GEN b,long prec`).

3.9.19 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, 10^4, 1/i); \\ rational number: denominator has 4345 digits.
time = 236 ms.
? sum(i=1, 5000, 1/i, 0.)
time = 8 ms.
%2 = 9.787606036044382264178477904
```

The library syntax is **somme**(`GEN a, GEN b, char *expr, GEN x`).

3.9.20 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 Cohen, Villegas and Zagier (*Experiment. Math.* **9** (2000), no. 1, 3–12).

If $flag = 1$, use a variant with slightly different polynomials. Sometimes faster.

The routine is heuristic and a rigorous proof assumes that the values of *expr* are the moments of a positive measure on $[0, 1]$. Divergent alternating series can sometimes be summed by this method, as well as series which are not exactly alternating (see for example Section 2.7). It should be used to try and guess the value of an infinite sum. (However, see the example at the end of Section 2.7.1.)

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)    \\ Had to hit ¡C-C¿
  ***   at top-level: 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 = -4.0369878149063348315649736135219061579435333859189783058 7 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)(void*,GEN),GEN a,long prec`). Also available is `sumalt2` with the same arguments (*flag* = 1).

3.9.21 sumdiv$(n, X, expr)$: sum of expression *expr* over the positive divisors of n. This function is a trivial wrapper essentially equivalent to

```
D = divisors(n);
for (i = 1, #D, X = D[i]; eval(expr))
```

(except that `X` is lexically scoped to the `sumdiv` loop). Arithmetic functions like `sigma` use the multiplicativity of the underlying expression to speed up the computation. Since there is no way to indicate that *expr* is multiplicative in n, specialized functions should always be preferred.

3.9.22 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)   \\ Had to hit ¡C-C¿
  ***   at top-level: 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)(void*,GEN), GEN a, long prec`).

3.9.23 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 ∞
? 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
%3 = -1.321115060 E320 + 0.E311*I \\ nonsense
? sumnum(n=1, [2,log(2)], 2^(-n), /*omitted*/, 1) - 1 \\ of real type
time = 5,860 ms.
%4 = -1.5... E-236 \\ slow and lost 70 decimals
? m = intnumstep()
%5 = 9
? sumnum(n=1,[2,log(2)], 2^(-n), m+1, 1) - 1
time = 11,770 ms.
%6 = -1.9... E-305 \\ now perfect, but slow.
```

The library syntax is **sumnum**(`void *E, GEN (*eval)(void*,GEN), GEN a,GEN sig,GEN tab,long flag, long prec`).

3.9.24 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.

Warning 2. 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); \\ abscissa 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)(void*,GEN), GEN a, GEN sig, GEN tab, long flag, long prec`).

3.9.25 sumnuminit$(sig, \{m = 0\}, \{sgn = 1\})$: initialize tables for numerical summation using `sumnum` (with $sgn = 1$) or `sumnumalt` (with $sgn = -1$), *sig* is the abscissa of integration coded as in `sumnum`, and m is as in `intnuminit`.

The library syntax is `GEN sumnuminit(GEN sig, long m, long sgn, long prec)`.

3.9.26 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, then we use `sumalt`. For regular functions, the function `sumnum` is in general much faster once the initializations have been made using `sumnuminit`.

The routine is heuristic and assumes that *expr* is more or less a decreasing function of X. In particular, the result will be completely wrong if *expr* is 0 too often. We do not check either that all terms have the same sign. As `sumalt`, this function should be used to try and guess the value of an infinite sum.

If $flag = 1$, use slightly different polynomials. Sometimes faster.

The library syntax is **sumpos**(`void *E, GEN (*eval)(void*,GEN),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. 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 using the `plotcolor` function) 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 `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, use `plotkill`. It is not possible to partially erase a window: erase it completely, initialize it again, 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 `plotscale`. 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.

Plotting functions are platform independent, but a number of graphical drivers are available for screen output: X11-windows (hence also for GUI's based on X11 such as Openwindows and Motif), and the Qt and FLTK graphical libraries. 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. This window can be closed, enlarged or reduced using the standard window manager functions. No zooming procedure is implemented though (yet).

* Among these, special thanks go to Klaus-Peter Nischke who suggested the recursive plotting and forking/resizing stuff the graphical window, and Ilya Zakharevich who rewrote the graphic code from scratch implementing many new primitives (splines, clipping). Nils Skoruppa and Bill Allombert wrote the `Qt` and `fltk` graphic drivers respectively.

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 identical with the screen output, but 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 committing the part you're interested in to the final picture.

3.10.8 plotcolor(w, c): set default color to c in rectwindow w. This is only implemented for the X-windows, fltk and Qt graphing engines. Possible values for c are given by the `graphcolormap` default, factory setting are

1=black, 2=blue, 3=violetred, 4=red, 5=green, 6=grey, 7=gainsborough.

but this can be considerably extended.

3.10.9 plotcopy$(sourcew, destw, dx, dy, \{flag = 0\})$: copy the contents of rectwindow *sourcew* to rectwindow *destw* with offset (dx,dy). If flag's bit 1 is set, dx and dy express fractions of the size of the current output device, otherwise dx and dy are in pixels. dx and dy are relative positions of northwest corners if other bits of flag vanish, otherwise of: 2: southwest, 4: southeast, 6: northeast corners

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($list, \{flag = 0\}$): 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. If $flag \neq 0$, x1, y1 etc. express fractions of the size of the current output device

3.10.12 ploth($X = a, b, expr, \{flags = 0\}, \{n = 0\}$): 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.: `ploth` may evaluate `expr` thousands of times; given the relatively low resolution of plotting devices, few significant digits of the result will be meaningful. Hence you should keep the current precision to a minimum (e.g. 9) before calling this function.

n specifies the number of reference point on the graph, where a value of 0 means we use the hardwired default values (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)], "Parametric")
ploth(X=0,2*Pi,[sin(X),cos(X)])
ploth(X=0,2*Pi,[X,X,sin(X),cos(X)], "Parametric")
```

draw successively a circle, two entwined sinusoidal curves and a circle cut by 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

```
ploth(X=-1,1, sin(1/X), "Recursive")
ploth(X=-1,1, sin(1/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), "Recursive")
```

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 used in conjunction with `plotscale` when graphing multiple functions on a rectwindow (as a `plotrecth` call):

```
s = plothsizes();
plotinit(0, s[2]-1, s[2]-1);
plotscale(0, -1,1, -1,1);
plotrecth(0, t=0,2*Pi, [cos(t),sin(t)], "Parametric|no_Rescale")
plotdraw([0, -1,1]);
```

This way we get a proper circle instead of the distorted ellipse produced by

```
ploth(t=0,2*Pi, [cos(t),sin(t)], "Parametric")
```

• 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.

• 4096 = `Complex`: is a parametric plot but where each member of `expr` is considered a complex number encoding the two coordinates of a point. For instance:

```
ploth(X=0,2*Pi,exp(I*X), "Complex")
ploth(X=0,2*Pi,[(1+I)*X,exp(I*X)], "Complex")
```

will draw respectively a circle and a circle cut by the line $y = x$.

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({*flag* = 0}): 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.

If *flag* = 0, sizes of ticks and characters are in pixels, otherwise are fractions of the screen size

3.10.15 plotinit$(w, \{x\}, \{y\}, \{\mathit{flag} = 0\})$: 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; omitting either x or y means we use the full size of the device in that direction. If $\mathit{flag} = 0$, 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$(w, X, Y, \{\mathit{flag} = 0\})$: 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`$(w, X = a, b, expr, flag, n)$. Returns a vector for the bounding box.

3.10.25 plotrecthraw$(w, data, \{flags = 0\})$: 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$(w, x, \{flags = 0\})$: draw on the rectwindow w the String x (see Section 2.9), 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$(list, \{flag = 0\})$: same as `plotdraw`, except that the output is a PostScript program appended to the `psfile`, and flag!=0 scales the plot from size of the current output device to the standard PostScript plotting size

3.10.32 psploth$(X = a, b, expr, \{flags = 0\}, \{n = 0\})$: same as `ploth`, except that the output is a PostScript program appended to the `psfile`.

3.10.33 psplothraw$(listx, listy, \{flag = 0\})$: same as `plothraw`, except that the output is a PostScript program appended to the `psfile`.

3.11 Programming in GP: 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 lexically scoped to the loop, leading to possibly unexpected behavior:

```
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 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); the integer n must be positive. If n is greater than the number of enclosing loops, all enclosing loops are exited.

3.11.2 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**.

3.11.3 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)=
{ my(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.
```

3.11.4 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.

The library syntax is **forell**(`void *data, long (*call)(void*,GEN), GEN a, GEN b`).

3.11.5 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 $\mathbf{R}$.

```
f(N) =
{
  forprime(p = 2, N,
    print(p);
    if (p == 3, p = 6);
  )
}
? f(12)
2
3
7
11
```

3.11.6 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
```

3.11.7 forsubgroup($H = G, \{bound\}, seq$): 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 `galois-subcyclo` and `galoisfixedfield` for applications to Galois theory.

The library syntax is **forsubgroup**(`void *data, long (*call)(void*,GEN), GEN G, GEN bound`).

3.11.8 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.

The type of X is the same as the type of v: `t_VEC` or `t_COL`.

3.11.9 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!).

3.11.10 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.

3.11.11 return$(\{x = 0\})$: returns from current subroutine, with result x. If x is omitted, return the `(void)` value (return no result, like `print`).

3.11.12 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`).

3.11.13 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.

3.12 Programming in GP: other specific functions.

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 2.9): 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.

3.12.1 Strprintf$(fmt, \{x\}*)$: returns a string built from the remaining arguments according to the format fmt. The format consists of ordinary characters (not %), printed unchanged, and conversions specifications. See `printf`.

3.12.2 addhelp(*sym*, *str*): changes the help message for the symbol `sym`. The string *str* is expanded on the spot and stored as the online help for `sym`. If `sym` 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. It is possible to attach a help text to an alias, but it will never be shown: aliases are expanded by the `?` help operator and we get the help of the functions the alias points to.

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!

The library syntax is `void addhelp(const char *sym, const char *str)`.

3.12.3 alarm($\{s = 0\}$): trigger an *alarmer* exception after s seconds, cancelling any previously set alarm. Stop a pending alarm if s = 0 or is omitted.

For example, the function `timefact(N,sec)` below will try to factor N and give up after *sec* seconds, returning a partial factorisation.

```
default(factor_add_primes,1);
default(primelimit,16777216);
timefact(N,sec)=
{
  trap(alarmer,factor(N,0),alarm(sec);my(F=factor(N));alarm(0);F);
}
```

3.12.4 alias(*newsym*, *sym*): defines the symbol *newsym* as an alias for the the symbol *sym*:

```
? alias("det", "matdet");
? det([1,2;3,4])
%1 = -2
```

You are not restricted to ordinary functions, as in the above example: to alias (from/to) member functions, prefix them with '`_.`'; to alias operators, use their internal name, obtained by writing `_` in lieu of the operators argument: for instance, `_!` and `!_` are the internal names of the factorial and the logical negation, respectively.

```
? alias("mod", "_.mod");
? alias("add", "_+_");
? alias("_.sin", "sin");
? mod(Mod(x,x^4+1))
%2 = x^4 + 1
? add(4,6)
%3 = 10
? Pi.sin
%4 = 0.E-37
```

Alias expansion is performed directly by the internal GP compiler. Note that since alias is performed at compilation-time, it does not require any run-time processing, however it only affects GP code compiled *after* the alias command is evaluated. A slower but more flexible alternative is to use variables. Compare

```
? fun = sin;
? g(a,b) = intnum(t=a,b,fun(t));
```

```
? g(0, Pi)
%3 = 2.0000000000000000000000000000000000000
? fun = cos;
? g(0, Pi)
%5 = 1.8830410776607851098 E-39
```

with

```
? alias(fun, sin);
? g(a,b) = intnum(t=a,b,fun(t));
? g(0,Pi)
%2 = 2.0000000000000000000000000000000000000
? alias(fun, cos);  \\ Oops. Does not affect *previous* definition!
? g(0,Pi)
%3 = 2.0000000000000000000000000000000000000
? g(a,b) = intnum(t=a,b,fun(t)); \\ Redefine, taking new alias into account
? g(0,Pi)
%5 = 1.8830410776607851098 E-39
```

A sample alias file `misc/gpalias` is provided with the standard distribution.

The library syntax is `void alias0(const char *newsym, const char *sym)`.

3.12.5 allocatemem($\{s = 0\}$): this very special operation 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. If $x = 0$, the size of the new stack is twice the size of the old one. The old stack is discarded.

Warning. This function should be typed at the `gp` prompt in interactive usage, or left by itself at the start of batch files. It cannot be used meaningfully in loop-like constructs, or as part of a larger expression sequence, e.g

```
allocatemem(); x = 1;    \\ This will not set x!
```

In fact, all loops are immediately exited, user functions terminated, and the rest of the sequence following `allocatemem()` is silently discarded, as well as all pending sequences of instructions. We just go on reading the next instruction sequence from the file we're in (or from the user). In particular, we have the following possibly unexpected behavior: in

```
read("file.gp"); x = 1
```

were `file.gp` contains an `allocatemem` statement, the `x = 1` is never executed, since all pending instructions in the current sequence are discarded.

The technical reason is that this routine moves the stack, so temporary objects created during the current expression evaluation are not correct anymore. (In particular byte-compiled expressions, which are allocated on the stack.) To avoid accessing obsolete pointers to the old stack, this routine ends by a `longjmp`.

3.12.6 apply(f, A): Apply the `t_CLOSURE f` to the entries of `A`. If `A` is a scalar, return `f(A)`. If `A` is a polynomial or power series, apply `f` on all coefficients. If `A` is a vector or list, return the elements $f(x)$ where x runs through `A`. If `A` is a matrix, return the matrix whose entries are the f(`A[i,j]`).

```
? apply(x->x^2, [1,2,3,4])
%1 = [1, 4, 9, 16]
? apply(x->x^2, [1,2;3,4])
%2 =
[1 4]

[9 16]
? apply(x->x^2, 4*x^2 + 3*x+ 2)
%3 = 16*x^2 + 9*x + 4
```

Note that many functions already act componentwise on vectors or matrices, but they almost never act on lists; in this case, `apply` is a good solution:

```
? L = List([Mod(1,3), Mod(2,4)]);
? lift(L)
  ***   at top-level: lift(L)
  ***                 ^-------
  *** lift: incorrect type in lift.
? apply(lift, L);
%2 = List([1, 2])
```

The library syntax is **genapply**(`void *E, GEN (*fun)(void*,GEN), GEN a`).

3.12.7 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 2.12 for an introduction to GP defaults, Section 3.13 for a list of available defaults, and Section 2.13 for some shortcut alternatives. Note that the shortcuts are meant for interactive use and usually display more information than `default`.

The library syntax is `GEN default0(const char *key = NULL, const char *val = NULL)`.

3.12.8 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 !")
```

3.12.9 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 output fed into `gp`, just as if read from a file.

3.12.10 externstr(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 output is returned as a vector of GP strings, one component per output line.

3.12.11 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 `GEN getheap()`.

3.12.12 getrand(): returns the current value of the seed used by the pseudo-random number generator `random`. Useful mainly for debugging purposes, to reproduce a specific chain of computations. The returned value is technical (reproduces an internal state array), and can only be used as an argument to `setrand`.

The library syntax is `GEN getrand()`.

3.12.13 getstack(): returns the current value of `top` − `avma`, i.e. the number of bytes used up to now on the stack. Useful mainly for debugging purposes.

The library syntax is `long getstack()`.

3.12.14 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 `long gettime()`.

3.12.15 global(*list of variables*): obsolete. Scheduled for deletion.

3.12.16 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 2.16) 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).

3.12.17 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 is 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. However, it will not 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.

3.12.18 kill(*sym*): restores the symbol `sym` to its "undefined" status, and deletes any help messages associated to `sym` using `addhelp`. Variable names remain known to the interpreter and keep their former priority: you cannot make a variable "less important" by killing it!

```
? z = y = 1; y
%1 = 1
? kill(y)
? y            \\ restored to ``undefined'' status
%2 = y
? variable()
%3 = [x, y, z] \\ but the variable name y is still known, with y > z !
```

For the same reason, killing a user function (which is an ordinary variable holding a `t_CLOSURE`) does not remove its name from the list of variable names.

If the symbol is associated to a variable — user functions being an important special case —, one may use the quote operator `a = 'a` to reset variables to their starting values. However, this will not delete a help message associated to `a`, and is also slightly slower than `kill(a)`.

```
? x = 1; addhelp(x, "foo"); x
%1 = 1
? x = 'x; x   \\ same as 'kill', except we don't delete help.
%2 = x
? ?x
foo
```

On the other hand, `kill` is the only way to remove aliases and installed functions.

```
? alias(fun, sin);
? kill(fun);
? install(addii, GG);
? kill(addii);
```

The library syntax is `void kill0(const char *sym)`.

3.12.19 print({*str*}∗): outputs its (string) arguments in raw format, ending with a newline.

3.12.20 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 !

3.12.21 printf($fmt, \{x\}*$): This function is based on the C library command of the same name. It prints its arguments according to the format *fmt*, which specifies how subsequent arguments are converted for output. The format is a character string composed of zero or more directives:

• ordinary characters (not `%`), printed unchanged,

• conversions specifications (`%` followed by some characters) which fetch one argument from the list and prints it according to the specification.

More precisely, a conversion specification consists in a `%`, one or more optional flags (among `#`, `0`, `-`, `+`, ‘ ’), an optional decimal digit string specifying a minimal field width, an optional precision in the form of a period (‘`.`’) followed by a decimal digit string, and the conversion specifier (among `d`,`i`, `o`, `u`, `x`,`X`, `p`, `e`,`E`, `f`, `g`,`G`, `s`).

The flag characters. The character `%` is followed by zero or more of the following flags:

• `#`: The value is converted to an “alternate form”. For `o` conversion (octal), a `0` is prefixed to the string. For `x` and `X` conversions (hexa), respectively `0x` and `0X` are prepended. For other conversions, the flag is ignored.

• `0`: The value should be zero padded. For `d`, `i`, `o`, `u`, `x`, `X` `e`, `E`, `f`, `F`, `g`, and `G` conversions, the value is padded on the left with zeros rather than blanks. (If the `0` and `-` flags both appear, the `0` flag is ignored.)

• `-`: The value is left adjusted on the field boundary. (The default is right justification.) The value is padded on the right with blanks, rather than on the left with blanks or zeros. A `-` overrides a `0` if both are given.

• ‘ ’ (a space): A blank is left before a positive number produced by a signed conversion.

• `+`: A sign (+ or -) is placed before a number produced by a signed conversion. A `+` overrides a space if both are used.

The field width. An optional decimal digit string (whose first digit is non-zero) specifying a *minimum* field width. If the value has fewer characters than the field width, it is padded with spaces on the left (or right, if the left-adjustment flag has been given). In no case does a small field width cause truncation of a field; if the value is wider than the field width, the field is expanded to contain the conversion result. Instead of a decimal digit string, one may write `*` to specify that the field width is given in the next argument.

The precision. An optional precision in the form of a period (‘`.`’) followed by a decimal digit string. This gives the number of digits to appear after the radix character for `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for `g` and `G` conversions, and the maximum number of characters to be printed from an `s` conversion. Instead of a decimal digit string, one may write `*` to specify that the field width is given in the next argument.

The length modifier. This is ignored under `gp`, but necessary for `libpari` programming. Description given here for completeness:

• `l`: argument is a `long` integer.

• `P`: argument is a `GEN`.

The conversion specifier. A character that specifies the type of conversion to be applied.

• `d`, `i`: A signed integer.

• `o`, `u`, `x`, `X`: An unsigned integer, converted to unsigned octal (`o`), decimal (`u`) or hexadecimal (`x` or `X`) notation. The letters `abcdef` are used for `x` conversions; the letters `ABCDEF` are used for `X` conversions.

• `e`, `E`: The (real) argument is converted in the style `[ -]d.ddd e[ -]dd`, where there is one digit before the decimal point, and the number of digits after it is equal to the precision; if the precision is missing, use the current `realprecision` for the total number of printed digits. If the precision is explicitly 0, no decimal-point character appears. An `E` conversion uses the letter `E` rather than `e` to introduce the exponent.

• `f`, `F`: The (real) argument is converted in the style `[ -]ddd.ddd`, where the number of digits after the decimal point is equal to the precision; if the precision is missing, use the current `realprecision` for the total number of printed digits. If the precision is explicitly 0, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

• `g`, `G`: The (real) argument is converted in style `e` or `f` (or `E` or `F` for `G` conversions) `[ -]ddd.ddd`, where the total number of digits printed is equal to the precision; if the precision is missing, use the current `realprecision`. If the precision is explicitly 0, it is treated as 1. Style `e` is used when the decimal exponent is < -4, to print `0.`, or when the integer part cannot be decided given the known significant digits, and the `f` format otherwise.

• `c`: The integer argument is converted to an unsigned char, and the resulting character is written.

• `s`: Convert to a character string. If a precision is given, no more than the specified number of characters are written.

• `p`: Print the address of the argument in hexadecimal (as if by `%#x`).

• `%`: A `%` is written. No argument is converted. The complete conversion specification is `%%`.

Examples:

```
? printf("floor: %d, field width 3: %3d, with sign: %+3d\n", Pi, 1, 2);
floor: 3, field width 3:   1, with sign:  +2

? printf("%.5g %.5g %.5g\n",123,123/456,123456789);
123.00 0.26974 1.2346 e8

? printf("%-2.5s:%2.5s:%2.5s\n", "P", "PARI", "PARIGP");
P :PARI:PARIG

\\ min field width and precision given by arguments
? x = 23; y=-1/x; printf("x=%+06.2f y=%+0*.*f\n", x, 6, 2, y);
x=+23.00 y=-00.04

\\ minimum fields width 5, pad left with zeroes
? for (i = 2, 5, printf("%05d\n", 10^i))
00100
01000
10000
100000  \\ don't truncate fields whose length is larger than the minimum width
? printf("%.2f  |%06.2f|", Pi,Pi)
```

```
3.14  |  3.14|
```

All numerical conversions apply recursively to the entries of vectors and matrices:

```
? printf("%4d", [1,2,3]);
[   1,   2,   3]
? printf("%5.2f", mathilbert(3));
[ 1.00  0.50  0.33]

[ 0.50  0.33  0.25]

[ 0.33  0.25  0.20]
```

Technical note. Our implementation of `printf` deviates from the C89 and C99 standards in a few places:

• whenever a precision is missing, the current `realprecision` is used to determine the number of printed digits (C89: use 6 decimals after the radix character).

• in conversion style `e`, we do not impose that the exponent has at least two digits; we never write a `+` sign in the exponent; 0 is printed in a special way, always as `0.E`*exp*.

• in conversion style `f`, we switch to style `e` if the exponent is greater or equal to the precision.

• in conversion `g` and `G`, we do not remove trailing zeros from the fractional part of the result; nor a trailing decimal point; 0 is printed in a special way, always as `0.E`*exp*.

3.12.22 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 2.12). You could for instance do:

```
default(logfile, "new.tex");
default(log, 1);
printtex(result);
```

3.12.23 quit({*status* = 0}): exits `gp` and return to the system with exit status `status`, a small integer. A non-zero exit status normally indicates abnormal termination. (Note: the system actually sees only `status` mod 256, see your man pages for `exit(3)` or `wait(2)`).

3.12.24 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 3.12.36), the file is loaded and the last object in the file is returned.

In case the file you read in contains an `allocatemem` statement (to be generally avoided), you should leave `read` instructions by themselves, and not part of larger instruction sequences.

3.12.25 readvec({*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 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.

The library syntax is `GEN gp_readvec_file(const char *filename)`. The underlying library function `GEN gp_readvec_stream(FILE *f)` is usually more flexible.

3.12.26 select(f, A): Given a vector, list or matrix `A` and a `t_CLOSURE f`, returns the elements x of `A` such that $f(x)$ is non-zero. In other words, `f` is seen as a selection function returning a boolean value.

```
? select(x->isprime(x), vector(50,i,i^2+1))
%1 = [2, 5, 17, 37, 101, 197, 257, 401, 577, 677, 1297, 1601]
? select(x->(x<100), %)
%2 = [2, 5, 17, 37]
```

returns the primes of the form $i^2 + 1$ for some $i \leq 50$, then the elements less than 100 in the preceding result. The following function lists the elements in $(\mathbf{Z}/N\mathbf{Z})^*$:

```
? invertibles(N) = select(x->gcd(x,N) == 1, vector(N,i,i))
```

Finally

```
? select(x->x, M)
```

selects the non-0 entries in `M`. If the latter is a `t_MAT`, we extract the matrix of non-0 columns. Note that *removing* entries instead of selecting them just involves replacing the selection function `f` with its negation:

```
? select(x->!isprime(x), vector(50,i,i^2+1))
```

The library syntax is **genselect**(`void *E, long (*fun)(void*,GEN), GEN a`).

3.12.27 setrand(n): reseeds the random number generator using the seed n. No value is returned. The seed is either a technical array output by `getrand`, or a small positive integer, used to generate deterministically a suitable state array. For instance, running a randomized computation starting by `setrand(1)` twice will generate the exact same output.

The library syntax is `void setrand(GEN n)`.

3.12.28 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.

3.12.29 trap($\{e\}, \{rec\}, seq$): tries to evaluate *seq*, trapping runtime 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. See Section 2.10.2 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
```

Note that *seq* is effectively evaluated up to the point that produced the error, and the recovery sequence is evaluated starting from that same context, it does not "undo" whatever happened in the other branch (restore the evaluation context):

```
? x = 1; trap (, /* recover: */ x, /* try: */ x = 0; 1/x)
%1 = 0
```

Note. The interface is currently not adequate for trapping individual exceptions. In the current version 2.5.5, 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`):

`alarmer`: alarm time-out

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

`errpile`: the PARI stack overflows

`gdiver`: division by 0

`impl`: not yet implemented

`invmoder`: impossible inverse modulo

`overflower`: all forms of arithmetic overflow, including length or exponent overflow (when a larger value is supplied than the implementation can handle).

`syntaxer`: syntax error

`talker`: miscellaneous error

`typeer`: wrong type

`user`: user error (from the `error` function)

The library syntax is `GEN trap0(const char *e = NULL, GEN rec = NULL, GEN seq = NULL)`.

3.12.30 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 `GEN type0(GEN x)`. The macro `typ` is usually simpler to use since it returns a `long` that can easily be matched with the symbols `t_*`. The name `type` was avoided since it is a reserved identifier for some compilers.

3.12.31 version(): returns the current version number as a `t_VEC` with three integer components (major version number, minor version number and patchlevel); if your sources were obtained through our version control system, this will be followed by a more precise version string, e.g. `git-`*commit hash*.

This function is present in all versions of PARI following releases 2.3.4 (stable) and 2.4.3 (testing).

Unless you are working with multiple development versions, you probably only care about the 3 first numeric components. In any case, the `lex` function offers a clever way to check against a particular version number, since it will compare each successive vector entry, numerically or as strings, and will not mind if the vectors it compares have different lengths :

```
if (lex(version(), [2,3,5]) >= 0,
  \\ code to be executed if we are running 2.3.5 or more recent.
,
  \\ compatibility code
);
```

On a number of different machines, `version()` could return either of

```
%1 = [2, 3, 4]    \\ released version, stable branch
%1 = [2, 4, 3]    \\ released version, testing branch
%1 = [2, 6, 0, "git-2cce227"] \\ development
```

In particular the first line of the gp introductory message can be essentially emulated by

```
v = version();
n = Str(v[1], ".", v[2], ".", v[3]);
s = if (#v > 3, v[4], "");
print("GP/PARI CALCULATOR Version ", n, " (", s, ")");
```

If you *are* working with many development versions of PARI/GP, the last component can be profitably included in the name of your logfile, for instance.

The library syntax is `GEN pari_version()`.

3.12.32 warning({*str*}$*$): outputs the message "user warning" and the argument list (each of them interpreted as a string). If colors are enabled, this warning will be in a different color, making it easy to distinguish.

```
warning(n, " is very large, this might take a while.")
```

3.12.33 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).

3.12.34 write(*filename*, {*str*}∗): writes (appends) to *filename* the remaining arguments, and appends a newline (same output as `print`).

3.12.35 write1(*filename*, {*str*}∗): writes (appends) to *filename* the remaining arguments without a trailing newline (same output as `print1`).

3.12.36 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 3.12.18).

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 2.5.3) 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).

The library syntax is `void gpwritebin(const char *filename, GEN x = NULL)`.

3.12.37 writetex(*filename*, {*str*}∗): as `write`, in TEX format.

3.13 GP defaults.

This section documents the GP defaults

3.13.1 TeXstyle: 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.

The default value is `0`.

3.13.2 breakloop: if true, enables the "break loop" debugging mode, see Section 2.10.3.

The default value is `1` if we are running an interactive `gp` session, and `0` otherwise.

3.13.3 colors: 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 "$a_1,\ldots,a_k$" 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 PariEmacs. Changing it will just fail silently (the alternative would be to display escape sequences as is, since Emacs will refuse to interpret them). You must customize color highlighting from the PariEmacs side, see its documentation.

The default value is "" (no colors).

3.13.4 compatible: 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 3.12.33, 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 behavior 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 behavior. 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`.

The default value is `0`.

3.13.5 datadir: the name of directory containing the optional data files. For now, this includes the `elldata`, `galdata`, `galpol`, `seadata` packages.

The default value is /usr/share/parigp(the location of installed precomputed data, can be specified via `Configure --datadir=`).

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

The default value is `0` (no debugging messages).

3.13.7 debugfiles: 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`).

The default value is `0` (no debugging messages).

3.13.8 debugmem: 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).

The default value is `0` (no debugging messages).

3.13.9 echo: this toggle is 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`).

The default value is `0` (no echo).

3.13.10 factor_add_primes: this toggle is either 1 (on) or 0 (off). If on, the integer factorization machinery calls `addprimes` on primes factor that were difficult to find (larger than 2^24), 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 lets you recover the prime factors already found. The downside is that a huge `addprimes` table unrelated to the current computations will slow down arithmetic functions relying on integer factorization; one should then empty the table using `removeprimes`.

The defaut value is `0`.

3.13.11 factor_proven: this toggle is either 1 (on) or 0 (off). By default, the factors output by the integer factorization machinery are only pseudo-primes, not proven primes. If this toggle is set, a primality proof is done for each factor and all results depending on integer factorization are fully proven. This flag does not affect partial factorization when it is explicitly requested. It also does not affect the private table managed by `addprimes`: its entries are included as is in factorizations, without being tested for primality.

The defaut value is `0`.

3.13.12 format: of the form x.n, where x (conversion style) is a letter in {`e`, `f`, `g`}, and n (precision) is an integer; this affects the way real numbers are printed:

• If the conversion style is `e`, real numbers are printed in scientific format, always with an explicit exponent, e.g. `3.3 E-5`.

• In style `f`, real numbers are generally printed in fixed floating point format without exponent, e.g. `0.000033`. A large real number, whose integer part is not well defined (not enough significant digits), is printed in style `e`. For instance `10.^100` known to ten significant digits is always printed in style `e`.

• In style `g`, non-zero real numbers are printed in `f` format, except when their decimal exponent is < -4, in which case they are printed in `e` format. Real zeroes (of arbitrary exponent) are printed in `e` format.

The precision 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). For more powerful formatting possibilities, see `printf` and `Strprintf`.

The default value is `"g.28"` and `"g.38"` on 32-bit and 64-bit machines, respectively.

3.13.13 graphcolormap: a vector of colors, to be used by hi-res graphing routines. Its length is arbitrary, but it must contain at least 3 entries: the first 3 colors are used for background, frame/ticks and axes respectively. All colors in the colormap may be freely used in `plotcolor` calls.

A color is either given as in the default by character strings or by an RGB code. For valid character strings, see the standard `rgb.txt` file in X11 distributions, where we restrict to lowercase letters and remove all whitespace from color names. An RGB code is a vector with 3 integer entries between 0 and 255. For instance `[250, 235, 215]` and `"antique white"` represent the same color. RGB codes are a little cryptic but often easier to generate.

The default value is [`"white"`, `"black"`, `"blue"`, `"violetred"`, `"red"`, `"green"`, `"grey"`, `"gainsboro"`].

3.13.14 graphcolors: entries in the `graphcolormap` that will be used to plot multi-curves. The successive curves are drawn in colors

`graphcolormap[graphcolors[1]]`, `graphcolormap[graphcolors[2]]`, ...

cycling when the `graphcolors` list is exhausted.

The default value is `[4,5]`.

3.13.15 help: 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 2.13.1), or `M-H` under readline (see Section 2.15.1).

The default value is the local of the `gphelp` script.

3.13.16 histfile: name of a file where `gp` will keep a history of all *input* commands (results are omitted). If this file exists when the value of `histfile` changes, it is read in and becomes part of the session history. Thus, setting this default in your gprc saves your readline history between sessions. Setting this default to the empty string `""` changes it to <`undefined`>

The default value is <`undefined`> (no history file).

3.13.17 histsize: `gp` keeps a history of the last `histsize` results computed so far, which you can recover using the `%` notation (see Section 2.13.4). 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.

The default value is `5000`.

3.13.18 lines: 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 3.12) are unaffected, so you can always type `print(%)` or `\a` to view the full result. If the actual screen width cannot be determined, a "line" is assumed to be 80 characters long.

The default value is `0`.

3.13.19 log: 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`).

The default value is `0`.

3.13.20 logfile: name of the log file to be used when the `log` toggle is on. Environment and time expansion are performed.

The default value is `"pari.log"`.

3.13.21 new_galois_format: this toggle is either 1 (on) or 0 (off). If on, 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 behavior and do not break unexpectedly.

The defaut value is `0`. This value will change to 1 (set) in the next major version.

3.13.22 output: there are three possible values: 0 (= *raw*), 1 (= *prettymatrix*), or 3 (= *external prettyprint*). This means that, independently of the default `format` for reals which we explained above, you can print results in three ways:

• *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.

• *prettymatrix format*: this is identical to raw format, except 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.

• *external prettyprint*: 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` and `\m` and `\B` respectively.

The default value is `1` (*prettymatrix*).

3.13.23 parisize: `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.

The default value is 4M, resp. 8M on a 32-bit, resp. 64-bit machine.

3.13.24 path: this is a list of directories, separated by colons ':' (semicolons ';' in the DOS world, since colons are preempted for drive names). When asked to read a file whose name is not given by an absolute path (does not start with `/`, `./` or `../`), `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.

The default value is `".:~:~/gp"` on UNIX systems, `".;C:\;C:\GP"` on DOS, OS/2 and Windows, and `"."` otherwise.

3.13.25 prettyprinter: 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).

The default value is `"tex2mail -TeX -noindent -ragged -by_par"`.

3.13.26 primelimit: `gp` precomputes a list of all primes less than `primelimit` at initialization time. These are used by many arithmetic functions, usually for trial division purposes. If you do not 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.

Since almost all arithmetic functions eventually require some table of prime numbers, PARI currently guarantees that the first 6547 primes, up to and including 65557, are precomputed, even if `primelimit` is 1.

The default value is `500k`.

3.13.27 prompt: 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 3.13.3), 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**: PariEmacs 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!). See PariEmacs's documentation.

The default value is `"?  "`.

3.13.28 prompt_cont: 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.

The defaut value is `""`.

3.13.29 psfile: 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.

The default value is `"pari.ps"`.

3.13.30 readline: 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.

The default value is `1`.

3.13.31 realprecision:

3.13.32 realprecision : 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)
```

The default value is `28`, resp. `38` on a 32-bit, resp .64-bit, machine.

3.13.33 recover: this toggle is either 1 (on) or 0 (off). If you change this to 0, any error becomes fatal and causes the gp interpreter to exit immediately. Can be useful in batch job scripts.

The default value is `1`.

3.13.34 secure: this toggle is 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.

The default value is `0`.

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

The default value is `16`.

3.13.36 simplify: this toggle is 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`).

The default value is `1`.

3.13.37 strictmatch: this toggle is 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 are unsure how many parentheses you have to close after complicated nested loops. Please do not use this; find a decent text-editor instead.

The default value is `1`.

3.13.38 timer: this toggle is 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 `##`).

The default value is `0`.

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 libraries and programs. 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. The download page on our website `http://pari.math.u-bordeaux.fr/download.html` contains pointers on how to get these.

• GNU `MP` library. This provides an alternative multiprecision kernel, which is faster than PARI's native one, but unfortunately binary incompatible, so the resulting PARI library SONAME is libpari-gmp.

• GNU `readline` library. This provides line editing under `gp`, an automatic context-dependent completion, and an editable history of commands.

• GNU `emacs` and the `PariEmacs` package. The `gp` calculator 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's GP-mode.

• GNU `gzip/gunzip/gzcat` package enables `gp` to read compressed data.

• `perl` provides extended online help (full text from the manual) about functions and concepts. The script handling this online help can be used under `gp` or independently.

2. Compiling the library and the gp calculator.

2.1. Basic configuration: 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`*osname*`-`*arch* where the object files and executables will be built (*build directory*). The *osname* and *arch* components depends on your architecture and operating system, thus you can build PARI/GP for several different machines from the same source tree (the builds are independent and can be done simultaneously).

```
./Configure --tune
```

fine tunes the library for the host used for compilation. This adjusts thresholds by running a large number of comparative tests and creates a file `tune.h` in the build directory, that will be used from now on, overriding the ones in `src/kernel/none/` and `src/kernel/gmp/`. It will take a while: about 30 minutes on a 2GHz machine. Expect a small performance boost, perhaps a 10% speed increase compared to default settings.

If you are using GMP, tune it first, then PARI. Make sure you tune PARI on the machine that will actually run your computations, and do not use a heavily loaded machine for tunings.

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*`).

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 build 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.

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).

2.4. Cross-compiling:

When cross-compiling, you can set the environment variable `RUNTEST` to a program that is able to run the target binaries (e.g. an emulator). It will be used for both the `Configure` tests and `make bench`.

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 about 30 of them, and default answers are provided. If you accept all default answers, `Configure` 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 `$prefix/include/pari`. Other system-dependent data go to `$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 `$share_prefix/man`, and other system-independent data under `$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:

`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 ppc64
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` stands for `auto-gmp`.

• The default kernel is `auto-auto`.

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. You can 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`

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 generally, 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 developers.

`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. Thorough, but slow. Some of the tests require extra packages (`elldata`, `galdata`, etc.) to be available. If you want to test such an extra package *before* `make install` (which would install it to its final location, where `gp` expects to find it), run

```
env GP_DATA_DIR=$PWD/data make test-all
```

from the PARI toplevel directory, otherwise the test will fail.

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.

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:

• `elldata`: This package contains the elliptic curves in John Cremona's database. It is needed by the functions `ellidentify`, `ellsearch`, `forell` 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).

• `seadata`: This package contains the database of modular polynomials extracted from the ECHIDNA databases and computed by David R. Kohel. It is needed by the functions `ellap` and `ellgroup` for primes larger than 10^{20}.

• `galpol`: This package contains the GALPOL database of polynomials defining Galois extensions of the rationals, accessed by `galoisgetpol`.

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 `pdftex` 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. 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 unsolicited 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 2.5.5}",
    year         = 2013,
    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