User's Guide

to

PARI / GP

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

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

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

Primary ftp site:
ftp://pari.math.u-bordeaux.fr/pub/pari/

last updated 11 December 2003
for version 2.2.7

Table of Contents

Chapter 1: Overview of the PARI system . . . 5
1.1 Introduction . . . 5
1.2 The PARI types . . . 6
1.3 Operations and functions . . . 9
Chapter 2: Specific Use of the GP Calculator . . . 13
2.1 Defaults and output formats . . . 14
2.2 Simple metacommands . . . 21
2.3 Input formats for the PARI types . . . 25
2.4 GP operators . . . 28
2.5 The general GP input line . . . 31
2.6 The GP/PARI programming language . . . 32
2.7 Errors and error recovery . . . 42
2.8 Interfacing GP with other languages . . . 46
2.9 The preferences file . . . 47
2.10 Using GP under GNU Emacs . . . 49
2.11 Using GP with readline . . . 50
Chapter 3: Functions and Operations Available in PARI and GP . . . 53
3.1 Standard monadic or dyadic operators . . . 54
3.2 Conversions and similar elementary functions or commands . . . 58
3.3 Transcendental functions . . . 68
3.4 Arithmetic functions . . . 76
3.5 Functions related to elliptic curves . . . 89
3.6 Functions related to general number fields . . . 96
3.7 Polynomials and power series . . . 135
3.8 Vectors, matrices, linear algebra and sets . . . 141
3.9 Sums, products, integrals and similar functions . . . 153
3.10 Plotting functions . . . 156
3.11 Programming under GP . . . 162
Chapter 4: Programming PARI in Library Mode . . . 171
4.1 Introduction: initializations, universal objects . . . 171
4.2 Important technical notes . . . 172
4.3 Creation of PARI objects, assignments, conversions . . . 175
4.4 Garbage collection . . . 179
4.5 Implementation of the PARI types . . . 186
4.6 PARI variables . . . 190
4.7 Input and output . . . 192
4.8 A complete program . . . 197
4.9 Adding functions to PARI . . . 201
Chapter 5: Technical Reference Guide for Low-Level Functions . . . 207
5.1 Level 0 kernel (operations on unsigned longs) . . . 207
5.2 Level 1 kernel (operations on longs, integers and reals) . . . 208
5.3 Level 2 kernel (operations on general PARI objects) . . . 215
Appendix A: Installation Guide for the UNIX Versions . . . 221
Appendix B: A Sample program and Makefile . . . 229
Appendix C: Summary of Available Constants . . . 231
Index . . . 233

Chapter 1:
Overview of the PARI system

1.1 Introduction.

The PARI system is a package which is capable of doing formal computations on recursive types at high speed; it is primarily aimed at number theorists, but can be used by anybody whose primary need is speed.

Although quite an amount of symbolic manipulation is possible in PARI, this system does very badly compared to much more sophisticated systems like Axiom, Macsyma, Maple, Mathematica or Reduce on such manipulations (e.g. multivariate polynomials, formal integration, etc...). On the other hand, the three main advantages of the system are its speed (which can be between 5 and 100 times better on many computations), the possibility of using directly data types which are familiar to mathematicians, and its extensive algebraic number theory module which has no equivalent in the above-mentioned systems.

It is possible to use PARI in three different ways:

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

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

3) the compiler GP2C can translate GP code to C, and load them into the GP interpreter. A typical script compiled by GP2C will typically run 3 to 10 times faster. The C code can also be edited and optimized by hand.

The use of GP is explained in chapters 2 and 3, and the programming in library mode is explained in chapters 3, 4 and 5. In the present Chapter 1, we give an overview of the system.

Important note: A tutorial for GP is provided in the standard distribution (`tutorial.dvi`) and you should read this first (at least the beginning of it, you can skip the specialized topics you're not interested in). You can then start over and read the more boring stuff which lies ahead. But you should do that eventually, at the very least the various Chapter headings. 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 then refer to the complete function description in Chapter 3.

How to get the latest version?

This package can be obtained by anonymous ftp from quite a number of sites (ask your favourite Web search engine for the site nearest to you). But, if you want the very latest version (including development versions), you should use the anonymous ftp address

`ftp://pari.math.u-bordeaux.fr/pub/pari`

where you will find all the different ports and possibly some binaries. A lot of version information, mailing list archives, and various tips can be found on PARI's home page:

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

Implementation notes: (You can skip this section and switch to Section 1.2 if you're not interested in hardware technicalities. You won't miss anything that would be mentioned here.)

The PARI package contains essentially three variants. See Appendix A for how to set up one of these on your system.

In a first version, some time-critical parts of the multiprecision kernel are written in assembler, a few hundred lines at most. At present there exists three versions for the Sparc architecture: one for Sparc version 7 (e.g. Sparcstation 1, 1+, IPC, IPX or 2), one for Sparc version 8 with SuperSparc processors (e.g. SparcStation 10 and 20) and one for Sparc version 8 with MicroSparc I or II processors (e.g. SparcClassic or SparcStation 4 and 5). No specific version is written for the UltraSparc since it can use the MicroSparc II version. In addition, versions exist for the DEC Alpha 64-bit processors, and finally the whole ix86 family (Intel, AMD, Cyrix) starting at the 386 (up to the Xbox gaming console!). Finally, there are unmaintained, much less optimized and not really usable anymore, micro-kernels for the HP-PA architecture (only for HPUX 9, does not work with HPUX 10), and for the PowerPC architecture (only for the 601, does not compile anymore).

A second version uses the GNU MP library to implement its multiprecision kernel, and should run on any platform where GMP could be installed. The GMP kernel is about as fast as the native one for small operands, and much faster for huge ones (say, about twice faster around 10000 decimal digits).

A third version is written entirely in ANSI C, with a C++-compatible syntax, and should be portable without much trouble to any 32 or 64-bit computer having no real memory constraints. It is about 2 times slower than versions including an assembly kernel. This version has been tested for example on MIPS based DECstations 3100 and 5000 and SGI computers, on various Macs (based on 680x0 chips or more recent PowerPCs, like the G3 or G4), new Intel chips like ARM (handheld devices) or ia64 (Itanium). In addition, PARI has been ported to a considerable number of smaller and larger machines, for example the VAX, 68000-based machines like the Atari, Mac Classic or Amiga 500, 68020 machines such as the Amiga 2500 or 3000, or mainframes like the IBM/S390.

An historical version of the PARI/GP kernel, written in 1985, was specific to 680x0 based computers, and was entirely writen in MC68020 assembly language (around 6000 lines). It ran on SUN-3/xx, Sony News, NeXT cubes and on 680x0 based Macs. Since 1997, this version was unmaintained; it has been removed from the PARI distribution in version 2.2.5. To run PARI with a 68k assembler micro-kernel, one now uses the GMP version.

1.2 The PARI types.

The crucial word in PARI is recursiveness: most of the types it knows about are recursive. For example, the basic type **Complex** exists (actually called `t_COMPLEX`). 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 $\mathbf{Z}/n\mathbf{Z}$ ($(\mathbf{Z}/n\mathbf{Z})[i]$ when this makes sense), or p-adic numbers when $p \equiv 3 \bmod 4$ ($\mathbf{Q}_p[i]$).

This feature must of course not be used too rashly: for example you are in principle allowed to create objects which are "complex numbers of complex numbers", but don't expect PARI to make sensible use of such objects: you will mainly get nonsense.

On the other hand, one thing which *is* allowed is to have components of different, but compatible, types. For example, taking again complex numbers, the real part could be of type integer, and the imaginary part of type rational.

By compatible, we mean types which can be freely mixed in operations like + or ×. For example if the real part is of type real, the imaginary part cannot be of type integermod (integers modulo a given number n).

Let us now describe the types. As explained above, they are built recursively from basic types which are as follows. We use the letter T to designate any type; the symbolic names 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`	$\mathbf{Z}/n\mathbf{Z}$	Integermods (integers modulo n)
type `t_FRAC`	$\mathbf{Q}$	Rational numbers (in irreducible form)
type `t_FRACN`	$\mathbf{Q}$	Rational numbers (not necessarily in irreducible form)
type `t_COMPLEX`	$T[i]$	Complex numbers
type `t_PADIC`	$\mathbf{Q}_p$	p-adic numbers
type `t_QUAD`	$\mathbf{Q}[w]$	Quadratic Numbers (where $[\mathbf{Z}[w] : \mathbf{Z}] = 2$)
type `t_POLMOD`	$T[X]/P(X)T[X]$	Polmods (polynomials modulo P)
type `t_POL`	$T[X]$	Polynomials
type `t_SER`	$T((X))$	Power series (finite Laurent series)
type `t_RFRAC`	$T(X)$	Rational functions (in irreducible form)
type `t_RFRACN`	$T(X)$	Rational functions (not necessarily in irreducible form)
type `t_VEC`	T^n	Row (i.e. horizontal) vectors
type `t_COL`	T^n	Column (i.e. vertical) vectors
type `t_MAT`	$\mathcal{M}_{m,n}(T)$	Matrices
type `t_LIST`	T^n	Lists
type `t_STR`		Character strings

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

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

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

1.2.1 Integers and reals: they are of arbitrary and varying length (each number carrying in its internal representation its own length or precision) with the following mild restrictions (given for 32-bit machines, the restrictions for 64-bit machines being so weak as to be considered inexistent): integers must be in absolute value less than $2^{268435454}$ (i.e. roughly 80807123 digits). The precision of real numbers is also at most 80807123 significant decimal digits, and the binary exponent must be in absolute value less than $2^{23} = 8388608$.

Note that PARI has been optimized so that it works as fast as possible on numbers with at most a few thousand decimal digits. In particular, not too much effort has been put into fancy multiplication techniques (only the Karatsuba multiplication is implemented). Hence, although it is possible to use PARI to do computations with 10^7 decimal digits, much better programs can be written for such huge numbers.

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

1.2.2 Integermods, rational numbers (irreducible or not), p-adic numbers, polmods, and rational functions: these are recursive, but in a restricted way.

For integermods 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 $\varprojlim \mathbf{Z}/p^k\mathbf{Z}$, and $\mathbf{Q}_p$ as its field of fractions. Like real numbers, the codewords contain an exponent (giving essentially the p-adic valuation of the number) and also the information on the precision of the number (which is in fact redundant with p^k, but is included for the sake of efficiency).

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

Complex and quadratic numbers are partially recursive: the two components a and b can be of type integer, real, rational, integermod or p-adic, 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.

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

Note that in the present version 2.2.7 of PARI, there is a bug in the handling of power series of power series (i.e. power series in several variables). However power series of polynomials (which are power series in several variables of a special type) are OK. The reason for this bug is known, but it is difficult to correct because the mathematical problem itself contains some amount of imprecision.

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

1.2.6 Notes:

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

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

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

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

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

1.3 Operations and functions.

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

More specifically, if you do an operation (not a transcendental one) between exact objects, you will get an exact object. For example, dividing 1 by 3 does not give $0.33333\cdots$ as you might expect, but simply the rational number (1/3). If you really want the result in type real, evaluate 1./3 or add 0. to (1/3).

The result of operations between imprecise objects will be as precise as possible. Consider for example one of the most difficult cases, that is 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 (i.e. their exponents), and also on the size of $x + y$. PARI works out automatically the right precision for the result, even when it is working in calculator mode GP where there is a default precision.

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

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

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

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

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

An ambiguity would arise with square matrices. PARI always considers that you want to do componentwise function evaluation, hence to get for example the exponential of a square matrix you would need to use a function with a different name, `matexp` for instance. In the present version 2.2.7, this is not yet implemented. See however the program in Appendix C, which is a first attempt for this particular function.

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

1.3.2 Standard operations.

Of course, the four standard operators `+`, `-`, `*`, `/` exist. It should once more be emphasized that division is, as far as possible, an exact operation: 4 divided by 3 gives `(4/3)`. In addition to this, operations on integers or polynomials, like `\` (Euclidean division), `%` (Euclidean remainder) exist (and for integers, `\/` computes the quotient such that the remainder has smallest possible absolute value). There is also the exponentiation operator `^`, when the exponent is of type integer. Otherwise, it is considered as a transcendental function. Finally, the logical operators `!` (`not` prefix operator), `&&` (`and` operator), `||` (`or` operator) exist, giving as results `1` (true) or `0` (false). Note that `&` and `|` are also accepted as synonyms respectively for `&&` and `||`. However, there is no bitwise `and` or `or`.

1.3.3 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 integermod or a polmod.

1.3.4 Transcendental functions.

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

1.3.5 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. In the present version 2.2.7, a primitive, but useful version of Lenstra's Elliptic Curve Method (ECM) has been implemented.

There is now a very large package which enables the number theorist to work with ease in 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 (including explicit class field theory), and also many functions dealing with elliptic curves over **Q** or over local fields.

1.3.6 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 Romberg integration (open or closed, on a finite or infinite interval), real root finding (when the root is bracketed), polynomial interpolation, infinite series evaluation, and plotting are included. See the last sections of Chapter 3 for details.

Chapter 2:
Specific Use of the GP Calculator

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

For starting the calculator, the general command line syntax is:

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

where items within brackets are optional*. The [*files*] argument is a list of files written in the GP scripting language, which will be loaded on startup. The ones starting with a minus sign are *flags*, setting some internal parameters of GP, or *defaults*. See Section 2.1 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`).

UNIX: Some features were developed on UNIX platforms, and depend heavily on the operating system in use. It is *possible* that some of these will be ported to other operating systems (BeOS, MacOS, DOS, OS/2, Windows, etc.) in future versions (most of them should be relatively easy tasks). As for now, most of them were not. So, whenever a specific feature of the UNIX version is discussed in a paragraph, a UNIX sign sticks out in the left margin, like here. Just skip these if you're stranded on a different operating system: the core GP functions (i.e. at least everything which is even faintly mathematical in nature) will still be available to you. It should also be possible (and then definitely advisable) to install Linux or FreeBSD on your machine.

Note (added in version 2.0.12): Most UNIX goodies are now available for DOS, OS/2 and Windows, thanks to the `EMX/RSX` runtime package (`install` excluded under DOS, since DLLs are not supported by the OS). For Windows 95 and higher, you can also use the `Cygwin` compatibility library to run GP almost as if running a genuine Unix system. Note that a native **Linux** binary will be faster than one using any of these compatibility packages (see the `MACHINES` benchmark file, included in the distribution).

EMACS: If you have GNU Emacs, you can work in a special Emacs shell (see Section 2.10), which is started by typing `M-x gp` (where as usual `M` is the `Meta` key) if you accept the default stack, prime and buffer sizes, or `C-u M-x gp` which will ask you for the name of the gp executable, the stack size, the prime limit and the buffer size. Specific features of this Emacs shell will be indicated by an EMACS sign.

If a preferences file (or `gprc`, to be discussed in Section 2.9) can be found, GP will then read it and execute the commands it contains. This provides an easy way to customize GP without having to delve into the code to hardwire it to your likings. The *files* argument is processed right after the `gprc`.

* On the Macintosh, even after clicking on the gp icon, once in the MPW Shell, you still need to type explicitly a command of the above form.

A copyright message then appears which includes the version number, and a lot of useful technical information. The present manual is written for version 2.2.7, and has undergone major changes since the 1.39.xx versions.

After the copyright, the computer works for a few seconds (it is in fact computing and storing a table of primes), writes the top-level help information, some initial defaults, and then waits after printing its prompt (by default: `?`).

Note that at any point the user can type `Ctrl-C` (that is press simultaneously the `Control` and `C` keys): the current computation will be interrupted and control given back to the user at the GP prompt.

The top-level help information tells you that (as in many systems) to get help, you should type a `?`. When you do this and hit return, a menu appears, describing the eleven main categories of available functions and what to do to get more detailed help. If you now type `?`n with $1 \leq n \leq 11$, you will 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.

UNIX: If extended help (see Section 2.2.1) 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 compiled with the right options (see Appendix A), a line editor will be available to correct the command line, get automatic completions, and so on. See Section 2.11.1 for a short summary of available commands. This might not be available for all architectures.

Whether extended on-line help and line editing are available or not is indicated in the GP banner, between the version number and the copyright message.

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

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

As a general rule, under GP, commands starting with `\` or with some other symbols like `?` or `#`, are not computing commands, but are metacommands which allow the user 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 2.2).

2.1 Defaults and output formats.

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 2.9), but some of them are useful interactively (set timer on, increase precision, etc.).

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

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

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

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

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

(somewhat technical) Note: As we will see in Section 2.6.6, the second argument to `default` will be subject to string context expansion, which means you can use run-time values. In other words, something like

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

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

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

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

```
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 the C and bash shells — `~` by itself stands for your home directory, and `~user` is expanded to `user`'s home directory. This is applied to all filenames.

2.1.1 buffersize (default `30k`): GP input is buffered, which means only so many bytes of data can be read at a time before a command is executed. This used to be a very important variable, to allow for very large input files to be read into GP, for example large matrices, without it complaining about "unused characters". Currently, `buffersize` is automatically adjusted to the size of the data that are to be read. It will never go down by itself though. Thus this option may come in handy to decrease the buffer size after some unusually large `read`, when you don't need to keep gigantic buffers around anymore.

UNIX: **2.1.2 colors** (default `""`): this default is only usable if GP is running within certain color-capable terminals. For instance `rxvt`, `color_xterm` and modern versions of `xterm` under X Windows, or standard Linux/DOS text consoles. It causes GP to use a small palette of colors for its output. With xterms, the colormap used corresponds to the resources `Xterm*color`n where n ranges from 0 to 15 (see the file `misc/color.dft` for an example). Legal values for this default are strings "a_1,...,a_k" where $k \leq 7$ and each a_i is either

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

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

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

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

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

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

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

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

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

The hacker's way to check if this is the case would be to look in the `readline.h` include file (wherever your readline include files are) for the string `RL_PROMPT_START_IGNORE`. If it's there, you are safe.

A more sensible way is to make some experiments, and get a more recent `readline` if yours doesn't work the way you would like it to. See the file `misc/gprc.dft` for some examples.

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

`compatible` = 0: no backward compatibility. In this mode, a very handy function, to be described in Section 3.11.2.27, is `whatnow`, which tells you what has become of your favourite functions, which GP suddenly can't seem to remember.

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

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

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

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

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

2.1.4 datapath (default: the location of installed precomputed data): the name of directory containing the optional data files. For now, only the `galdata` package, needed by polgalois in degrees 8 to 11.

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

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

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

Important Note: As it noticeably slows down the performance (and triggers bugs in some versions of a popular compiler), the first functionality (memory usage) is disabled if you're not running a version compiled for debugging (see Appendix A).

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

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

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

UNIX: **2.1.10 help** (default: the location of the `gphelp` script): the name of the external help program which will be used from within GP when extended help is invoked, usually through a `??` or `???` request (see Section 2.2.1), or `M-H` under readline (see Section 2.11.1).

2.1.11 histsize (default `5000`): GP keeps a history of the last `histsize` results computed so far, which you can recover using the `%` notation (see Section 2.2.4). 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 don't need anymore.

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

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

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

1: plain logfile

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

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

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

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

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

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

The third format is the *prettyprint format*, or beautified format. In the present version 2.2.7, this is not beautiful at all.

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

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

2.1.17 parisize (default, 1M bytes on the Mac, 4M otherwise): GP, and in fact any program using the PARI library, needs a stack in which to do its computations. `parisize` is the stack size, in bytes. It is strongly recommended you increase this default (using the `-s` command-line switch, or a `gprc`) if you can afford it. Don't increase it beyond the actual amount of RAM installed on your computer or GP will spend most of its time paging.

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

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

UNIX: **2.1.19 prettyprinter** (default `"tex2mail -TeX -noindent -ragged -by_par"`) the name of an external prettyprinter to use when `output` is 3 (*alternate prettyprinter*). **This is experimental** but the default `tex2mail` looks already much nicer than the built-in "beautified format" (`output` = 2).

2.1.20 primelimit (default `200k` on the Mac, and `500k` otherwise): GP precomputes a list of all primes less than `primelimit` at initialization time. These are used by many arithmetical functions. If you don't plan to invoke any of them, you can just set this to 1.

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

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

If you use `readline`, escape sequences in your prompt will result in display bugs. If you have a relatively recent `readline` (see the comment at the end of Section 2.1.2), you can brace them with special sequences (`\[` and `\]`), and you will be safe. If these just result in extra spaces in your prompt, then you'll have to get a more recent `readline`. See the file `misc/gprc.dft` for an example.

EMACS: **Caution**: Emacs needs to know about the prompt pattern to separate your input from previous GP results, without ambiguity. It's not a trivial problem to adapt automatically this regular expression to an arbitrary prompt (which can be self-modifying!). Thus, in this version 2.2.7, Emacs relies on the prompt being the default one. So, do not tamper with the `prompt` variable *unless* you modify it simultaneously in your `.emacs` file (see `emacs/pariemacs.txt` and `misc/gprc.dft` for examples).

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

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

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

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

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

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

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

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

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

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

1: use `\frac` instead of `\over` for fractions.

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

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

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

2.1.32 Note on output formats.

A zero real number is printed in `e` format as $0.Exx$ where xx is the (usually negative) *decimal* exponent of the number (cf. Section 1.2.6.3). This allows the user to check the accuracy of the zero in question (this could also be done using `\x`, but that would be more technical).

When the integer part of a real number x is not known exactly because the exponent of x is greater than the internal precision, the real number is printed in `e` format (note that in versions before 1.38.93, this was instead printed with a $*$ at the end).

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

2.2 Simple metacommands.

Simple metacommands are meant as shortcuts and should not be used in GP scripts (see Section 3.11). Beware that these, as all of GP input, are *case sensitive*. For example, `\Q` is not identical to `\q`. In the following list, braces are used to denote optional arguments, with their default values when applicable, e.g. $\{n = 0\}$ means that if n is not there, it is assumed to be 0. Whitespace (or spaces) between the metacommand and its arguments and within arguments is optional. (This can cause problems only with `\w`, when you insist on having a filename whose first character is a digit, and with `\r` or `\w`, if the filename itself contains a space. In such cases, just use the underlying `read` or `write` function; see Section 3.11.2.28).

2.2.1 `?` {*command*}: GP on-line help interface. As already mentioned, 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!). Much better help can be obtained through the extended help system (see below).

UNIX: If the line before the copyright message indicates that extended help is available (this means `perl` is installed on your system, GP was told about it at compile time, and the whole PARI distribution was correctly installed), you can add more `?` signs for extended functionalities:

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

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

In case of conflicts between function and default names (e.g `log`, `simplify`), the function has higher priority. Use `?? default /`*defaultname* to get the default help.

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

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

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

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

message. Also, you can ask for extended help on section number n in Chapter 3, just by typing `??` n (where `?`n would yield merely a list of functions). Finally, a few key concepts in GP are documented in this way: metacommands (e.g `?? "??"`), defaults (e.g `?? psfile`) and type names

(e.g `t_INT` or `integer`), as well as various miscellaneous keywords such as `edit` (short summary of line editor commands), `operator`, `member`, `"user defined"`, `nf`, `ell`, ...

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

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

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

2.2.3 `\\`: one-line comment. The rest of the line is ignored by GP.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.19 `\r` {*filename*}: reads into GP all the commands contained in the named file as if they had been typed from the keyboard, one line after the other. Can be used in combination with the `\w` command (see below). Related but not equivalent to the function `read` (see Section 3.11.2.21); in particular, if the file contains more than one line of input, there will be one history entry for each of them, whereas `read` would only record the last one. If *filename* is omitted, re-read the previously used input file (fails if no file has ever been successfully read in the current session). If a GP `binary file` (see Section 3.11.2.30) is read using this command, it is silently loaded, without cluttering the history.

UNIX: 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.

2.2.20 `\s`: prints the state of the PARI stack and heap. This is used primarily as a debugging device for PARI, and is not intended for the casual user.

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

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

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

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

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

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

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

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

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

2.3 Input formats for the PARI types.

Before describing more sophisticated functions in the next section, let us 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).

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

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

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

2.3.3 Integermods (type `t_INTMOD`): to enter $n \bmod m$, type `Mod(n,m)`, *not* `n%m` (see Section 3.2.3).

2.3.4 Rational numbers (types `t_FRAC` and `t_FRACN`): under GP, all fractions are automatically reduced to lowest terms, so it is in principle impossible to work with reducible fractions (of type `t_FRACN`), although of course in library mode this is easy. To enter n/m just type it as written. As explained in Section 3.1.4, division will *not* be performed, only reduction to lowest terms.

If you really want a reducible fraction under GP, you must use the `type` function (see Section 3.11.2.26), by typing `type(x,FRACN)`. Be warned however that this function must be used with extreme care.

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

2.3.6 p-adic numbers (type `t_PADIC`): to enter a p-adic number, simply write a rational or integer expression and add to it `O(p^k)`, where p and k are integers. This last expression indicates three things to GP: first that it is dealing with a `t_PADIC` type (the fact that p is an integer, and not a polynomial, which would be used to enter a series, see Section 2.3.10), secondly the "prime" p (note that it is not checked whether p is indeed prime; you can work on 10-adics if you want, but beware of disasters as soon as you do something non-trivial like taking a square root), and finally the number of significant p-adic digits k. Note that `O(25)` is not the same as `O(5^2)`; you probably 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)`.

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

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

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

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

2.3.8 Polmods (type `t_POLMOD`): exactly as for integermods, to enter $x \bmod y$ (where x and y are polynomials), type `Mod(x,y)`, not `x%y` (see Section 3.2.3). 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.

Important remark. Since the variables occurring in a polmod are not free variables, it is essential in order to avoid inconsistencies that polmods use the same variable in internal operations (i.e. between polmods) and variables of lower priority (which have been introduced later in the GP session) for external operations (typically between a polynomial and a polmod). For example, PARI will not recognize that `Mod(y, y^2 + 1)` is the same as `Mod(x, x^2 + 1)`. Hopefully, this problem will pass away when type "element of a number field" is eventually introduced. * See Section 2.6.2 for a definition of "priority" and a discussion of (PARI's idea of) multivariate polynomial arithmetic.

On the other hand, `Mod(x, x^2 + 1) + Mod(x, x^2 + 1)` (which gives `Mod(2*x, x^2 + 1)`) and `x + Mod(y, y^2 + 1)` (which gives a result mathematically equivalent to `x` $+ i$ with $i^2 = -1$) are completely correct, while `y + Mod(x, x^2 + 1)` gives `Mod(x + y, x^2 + 1)`, which may not be what you want (`y` is treated here as a numerical parameter, not as a polynomial variable).

* On the other hand, one can argue that there is no reason to consider these quantities equal. E.g., one can be the opposite of another. Compare with numerous discussions on whether "the algebraic closure of $\mathbf{Q}$ is canonically defined", or one needs to consider a groupoid of algebraic closures.

Note (added in version 2.0.16) As long as the main variables are the same, it is allowed to mix `t_POL` and `t_POLMOD`s. The result will be the expected `t_POLMOD`. For instance `x + Mod(x, x^2 + 1)` is equal to `Mod(2*x, x^2 + 1)`. This wasn't the case prior to version 2.0.16: it returned a polynomial in `x` equivalent to $x + i$, which was in fact an invalid object (you couldn't `lift` it).

2.3.9 Polynomials (type `t_POL`): type the polynomial in a natural way, not forgetting to put a "$*$" between a coefficient and a formal variable (this $*$ does not appear in beautified output). Any variable name can be used except for the reserved names `I` (used exclusively for the square root of -1), `Pi` ($3.14\ldots$), `Euler` (Euler's constant), and all the function names: predefined functions, as described in Chapter 3 (use `\c` to get the complete list of them) and user-defined functions, which you ought to know about (use `\u` if you are subject to memory lapses). The total number of different variable names is limited to 16384 and 65536 on 32-bit and 64-bit machines respectively, which should be enough. If you ever need hundreds of variables, you should probably be using vectors instead. See Section 2.6.2 for a discussion of multivariate polynomial rings.

2.3.10 Power series (type `t_SER`): type a rational function or polynomial expression and add to it `O(`*expr* `^k)`, where *expr* is an expression which has non-zero valuation (it can be a polynomial, power series, or a rational function; the most common case being simply a variable name). This indicates to GP that it is dealing with a power series, and the desired precision is k times the valuation of *expr* with respect to the main variable of *expr* (to check the ordering of the variables, or to modify it, use the function `reorder`; see Section 3.11.2.22).

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

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

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

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

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

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

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

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

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

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

2.4 GP operators.

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

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

is equivalent to

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

GP knows quite a lot of different operators, some of them unary (having only one argument), some binary, plus special selection operators. Unary operators are defined for either prefix (preceding their single argument: $op\ x$) or postfix (following the argument: $x\ op$) position, never both (some are syntactically correct in both positions, but with different meanings). Binary operators all use the syntax $x\ op\ y$. Most of them are well known, some are borrowed from C syntax, and a few are specific to GP. Beware that some GP operators may differ slightly from their C counterparts.

For instance, GP's postfix `++` returns the *new* value, like the prefix `++` of C, and the binary shifts `<<`, `>>` have a priority which is different from (higher than) that of their C counterparts. When in doubt, just surround everything by parentheses (besides, your code will probably be more legible).

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

• Priority 10

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

• Priority 9

op`=`, where *op* is any simple binary operator (i.e. a binary operator with no side effects, i.e. one of those defined below) which is not a boolean operator (comparison or logical). `x` *op*`=` y assigns (`x` *op* y) to `x`, and returns the new value of `x`, *not* a reference to the variable `x`. (Thus an assignment cannot occur on the left hand side of another assignment.)

• Priority 8

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

• Priority 7

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

• Priority 6

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

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

`^`: powering.

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

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

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

`.`*member* (unary, postfix): x`.`*member* extracts *member* from structure x (see Section 2.6.5).

• Priority 5

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

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

• Priority 4

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

• Priority 3

`*`: multiplication.

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

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

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

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

• Priority 2

`+`, `-`: addition/subtraction.

• Priority 1

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

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

`==`: test for (exact) equality.

• Priority 0

`&`, `&&`: logical *and*.

`|`, `||`: logical (inclusive) *or*. Any sequence of logical *or* and *and* operations is evaluated from left to right, and aborted as soon as the final truth value is known. Thus, for instance, `(x && 1/x)` or `(type(p) == "t_INT" && isprime(p))` will never produce an error since the second argument need not (and will not) be processed when the first is already zero (false).

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

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

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

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

2.5 The general GP input line.

2.5.1 Generalities. User interaction with a GP session proceeds as follows: a sequence of characters is typed by the user at the GP prompt. This can be either a `\` command, a function definition, an expression, or a sequence of expressions (i.e. a program). In the latter two cases, after the last expression has been computed its result is put into an internal ("history") array, and printed. The successive elements of this 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.

If you want to suppress the printing of the result, for example because it is a long unimportant intermediate result, end the expression with a `;` sign. This same sign is used as an instruction separator when several instructions are written on the same line (note that for the pleasure of BASIC addicts, the `:` sign can also be used, but we will try to stick to C-style conventions in this manual). The final expression computed, even if not printed, will still be assigned to the history array, so you may have to pay close attention when you intend to refer back to it by number since this number does not appear explicitly. Of course, if you just want to use it on the next line, use `%` as usual.

Any legal expression can be typed in, and is evaluated using the conventions about operator priorities and left to right associativity (see the previous section), using the available operator symbols, function names (including user-defined functions and member functions see Section 2.6.4), and special variables. Please note that, from version 1.900 on, there *is* a distinction between lowercase and uppercase. Also, note that, outside of constant strings, blanks are completely ignored in the input to GP.

The special variable names known to GP are `Euler` (Euler's constant $\gamma = 0.577\ldots$), `I` (the square root of -1), `Pi` ($3.14\ldots$) — which could be thought of as functions with no arguments, and which may therefore be invoked without parentheses —, and `O` which obeys the following syntax:

`O(`*expr*`^k)`

When *expr* is an integer or a rational number, this creates an *expr*-adic number (zero in fact) of precision `k`. When *expr* is a polynomial, a power series or a rational function whose main variable is X, say, this creates a power series (also zero) of precision $v * \mathtt{k}$ where v is the X-adic valuation of *expr* (see 2.3.6 and 2.3.9).

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

The first one is simply to use the backslash character `\` at the end of the line that you are typing, just before hitting `<Return>`. This tells GP that what you will write on the next line is the physical continuation of what you have just written. In other words, it makes GP forget your newline character. For example if you use this while defining a function, and if you ask for the definition of the function using `?name`, you will see that your backslash has disappeared and that everything is on the same line. You can type a `\` anywhere. It will be interpreted as above only if (apart from ignored whitespace characters) it is immediately followed by a newline. For example, you can type

```
? 3 + \
4
```

instead of typing `3 + 4`.

The second one is a slight variation on the first, and is mostly useful when defining a user function (see Section 2.6.4): since an equal sign can never end a valid expression, GP will disregard a newline immediately following an `=`.

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

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

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

would silently produce garbage, since what GP will really see is `a=bb=c` which will assign the value of `c` to both `bb` and `a` (if this really is what you intended, you're a hopeless case).

2.6 The GP/PARI programming language.

The GP calculator uses a purely interpreted language. 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 * (see Section 3.11 for a complete list), and the main loop does not really execute, but rather evaluates (sequences of) expressions. Of course, it is by no means a true LISP.

2.6.1 Variables and symbolic expressions.

In GP you can use up to 16383 variable names (up to 65535 on 64-bit machines). These names can be any standard identifier names, i.e. they must start with a letter and contain only valid keyword characters: `_` or alphanumeric characters ([`_A-Za-z0-9`]). To avoid confusion with other symbols, you must not use other non-alphanumeric symbols like `$`, or '`.`'. In addition to the function names which you must not use (see the list with `\c`), there are exactly three special variable names which you are not allowed to use: `Pi` and `Euler`, which represent well known constants, and `I` $= \sqrt{-1}$.

Note that GP names are case sensitive since version 1.900. This means for instance that the symbol `i` is perfectly safe to use, and will not be mistaken for $\sqrt{-1}$, and that `o` is not synonymous anymore to `O`. If you grew addicted to the previous behaviour, you can have it back by setting the default `compatible` to 3.

Now the main thing to understand is that PARI/GP is *not* a symbolic manipulation package, although it shares some of the functionalities. One of the main consequences of this fact is that all expressions are evaluated as soon as they are written, they never stay in a purely abstract form**. As an important example, consider what happens when you use a variable name *before* assigning a value into it. This is perfectly acceptable to GP, which considers this variable in fact as a polynomial of degree 1, with coefficients 1 in degree 1, 0 in degree 0, whose variable is the variable name you used.

If later you assign a value to that variable, the objects which you have created before will still be considered as polynomials. If you want to obtain their value, use the function `eval` (see Section 3.7.3).

Finally, note that if the variable x contains a vector or list, you can assign a result to $x[m]$ (i.e. write something like $x[k] = expr$). If x is a matrix, you can assign a result to $x[m, n]$, but *not* to $x[m]$. If you want to assign an expression to the m-th column of a matrix x, use $x[, m] = expr$ instead. Similarly, use $x[m,] = expr$ to assign an expression to the m-th row of x. This process is recursive, so if x is a matrix of matrices of ..., an expression such as $x[1, 1][, 3][4] = 1$ would be perfectly valid (assuming of course that all matrices along the way have the correct dimensions).

Note: We'll see in Section 2.6.4 that it is possible to restrict the use of a given variable by declaring it to be `global` or `local`. This can be useful to enforce clean programming style, but is in no way mandatory.

---

* 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, GP only expands this one.

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

(Technical) Note: Each variable has a stack of values, implemented as a linked list. When a new scope is entered (during a function call which uses it as a parameter, or if the variable is used as a loop index, see Section 2.6.4 and Section 3.11), the value of the actual parameter is pushed on the stack. If the parameter is not supplied, a special 0 value called **gnil** is pushed on the stack (this value is not printed if it is returned as the result of a GP expression sequence). Upon exit, the stack decreases. You can `kill` a variable, decreasing the stack yourself. However, the stack has a bottom: the value of a variable is the monomial of degree 1 in this variable, as is natural for a mathematician.

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

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

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

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

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

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

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

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

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

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

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

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

In the last exemple, 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 = 1
? 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 dependance on implicit orderings. This will hopefully be corrected in future versions.

2.6.3 Expressions and expression sequences.

An expression is formed by combining the GP operators, functions (including user-defined functions, see below) and control statements. It may be preceded by an assignment statement '=' into a variable. It always has a value, which can be any PARI object.

Several expressions can be combined on a single line by separating them with semicolons (';') and also with colons (':') for those who are used to BASIC. Such an expression sequence will be called simply a *seq*. A *seq* also has a value, which is the value of the last non-empty expression in the sequence. Under GP, the value of the *seq*, and only this last value, is always put on the stack (i.e. it will become the next object $\%n$). 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* (or of course of an expression if there is only one) is printed if the line does not end with a semicolon (';').

2.6.4 User defined functions.

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

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

which looks better written on consecutive lines:

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

  ...
}
```

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

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

`list of formal variables` is the list of variables corresponding to those which you will actually use when calling your function. The number of actual parameters supplied when calling the function has to be less than the number of formal variables.

Uninitialized formal variables will be given a default value. An equal (`=`) sign following a variable name in the function definition, followed by any expression, gives the variable a default value. The said expression gets evaluated the moment the function is called, hence may involve the function parameters. A variable for which you supply no default value will be initialized to zero.

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

Example: For instance

```
foo(x=1, y=2, z=3) = print(x ":" y ":" z)
```

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

```
? foo(6,7)
6:7:3
? foo(,5)
1:5:3
? foo
1:2:3
```

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

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

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

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

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

The iterative constructs which use a variable name (`for`*xxx*, `prod`*xxx*, `sum`*xxx*, `vector`, `matrix`, `plot`, etc.) also consider the given variable to be local to the construct. A value is pushed on entry and pulled on exit. So, it is not necessary for a function using such a construct to declare the variable as a dummy formal parameter.

In particular, since loop variables are not visible outside their loops, the variable `n` need not be declared in the protoype of our `zet` function above.

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

would be a perfectly sensible (and in fact better) definition. Since local/global scope is a very tricky point, here's one more example. What's wrong with the following definition?

```
? first_prime_div(x) =
{
  local(p);
  forprime(p=2, x, if (x%p == 0, break));
```

```
  p
}
? first_prime_div(10)
%1 = 0
```

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

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

Again, it is strongly recommended to declare all other local variables that are used inside a function: if a function accesses a variable which is not one of its formal parameters, the value used will be the one which happens to be on top of the stack at the time of the call. This could be a "global" value, or a local value belonging to any function higher in the call chain. So, be warned.

Recursive functions can easily be written as long as one pays proper attention to variable scope. Here's a last 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) =
{
  local(v);

  if (type(P) != "t_POL",
    for (i=0, nbvar-1, P = [P]);
    return (P)
  );
  v = vector(poldegree(P)+1, i, polcoeff(P,i-1));
  vector(length(v), i, coeffs(v[i], 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]`. What would happen if the declaration `local(v)` had been omitted ?

The operating system will automatically limit the recursion depth:

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

There's no way to increase the recursion limit (which may be different on your machine) from within, since it would simply crash the GP process. To increase it before launching GP, you can use `ulimit` or `limit`, depending on your shell, to raise the process available stack space (increase `stacksize`).

Function which take functions as parameters ? This is easy in GP using the following trick (neat example due to Bill Daly):

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

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

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

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

Also, the statement `global(x, y, z, t)` (see Section 3.11.2.11) declares the corresponding variables to be global. It is then forbidden to use them as formal parameters or loop indexes as described above, since the parameter would "shadow" the variable.

Implementation note. For the curious reader, here is how these stacks are handled: a hashing function is computed from the identifier, and used as an index in `hashtable`, a table of pointers. Each of these pointers begins a linked list of structures (type `entree`). The linked list is searched linearly for the identifier (each list will typically have less than 7 components or so). When the correct `entree` is found, it points to the top of the stack of values for that identifier if it is a variable, to the function itself if it is a predefined function, and to a copy of the text of the function if it is a user-defined function. When an error occurs, all of this maze (rather a tree, in fact) is searched and (hopefully) restored to the state preceding the last call of the main evaluator.

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

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

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

2.6.5 Member functions.

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

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

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

```
}
```

You can redefine one of your own member functions simply by typing a new definition for it. On the other hand, as a safety measure, you can't redefine the built-in member functions, so typing the above text would in fact produce an error (you'd have to call it e.g. `x.j2` in order for GP to accept it).

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

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

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

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

2.6.6 Strings and Keywords.

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

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

• Since there are cases where expansion is not really 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).

Here is a useful example, used to create 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]
```

Note that the argument of `Str` is evaluated in string context, and really consists of 5 pieces. (Exercise: why are the empty strings necessary?). This part could also have been written as `concat(concat(Str(s), i), j)` (but *not* as `concat(Str(s), concat(i,j))`!). More simply, we could have written `concat([Str(s), i,j])`, or even `concat([s,i,j])`, silently assuming that `s` will indeed be a string.

In version 2.2.5, the prototype of `Str` was extended to allow more than one argument, which are concatenated into a single string. So, finally, `Str(s, i, j)` should now be preferred to the above solution, since it is less cryptic and more efficient.

A last example: the function `hist` returns all history entries from `%`a to `%`b neatly packed into a single vector

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

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`
`system`
all the `print`*xxx* functions
all the `write`*xxx* functions

The arguments of the following functions are processed as keywords:

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

2.7 Errors and error recovery.

2.7.1 Errors.

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

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

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

possibly enlarged to a full arrow given enough trailing context

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

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

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

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

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

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

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

```
? 1/0
  ***   division by zero in gdiv, gdivgs or ginv
```

The first half of the sentence is crystal clear, and the second one only gives more context as to where exactly the problem occured in the library. Unfortunately library errors do not give context, so it can be hard to track down exactly where in your program the error occured.

2.7.2 Error recovery

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

A slightly different situation, still related to error recovery, is when you you actually foresee that some error may occur, are unable to prevent it, but quite capable of recovering from it, given the chance. Examples include lazy factorization (cf. `addprimes`), where you knowingly use a pseudo prime N as if it were prime; you may then encounter an "impossible" situation, but this would

usually exhibit a factor of N, enabling you to refine the factorization and go on. Or you might run an expensive computation at low precision to guess the size of the output, hence the right precision to use. You can then encounter errors like "precision loss in truncation", e.g when trying to convert `1E1000`, known to 28 digits of accuracy, to an integer; or "division by 0", e.g inverting `0E1000` when all accuracy has been lost, and no significant digit remains. It would be enough to restart part of the computation at a slightly higher precision.

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

2.7.3 Break loop

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

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

In current version 2.2.7, an error is non-fatal if and only if it was initiated by a `C-c` typed by the user.

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

A break loop looks like this:

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

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

```
break> v
```

```
0
```

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

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

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

2.7.4 Error handlers

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

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

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

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

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

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

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

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

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

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

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

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

`accurer`: accuracy problem

`gdiver`: division by 0

`invmoder`: impossible inverse modulo

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

`typeer`: wrong type

`errpile`: the PARI stack overflows

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

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

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

2.8 Interfacing GP with other languages.

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

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

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

We are aware of three PARI-related public domain libraries. We *neither endorse nor support* any of them. You might want to give them a try if you are familiar with the languages they are based on. First, there are `PariPerl`*, written by Ilya Zakharevich (`ilya@math.ohio-state.edu`), and `PariPython`**, by Stéfane Fermigier (`fermigie@math.jussieu.fr`). Finaly, Michael Stoll

* see `http://nswt.tuwien.ac.at:8000/htdocs/internet/unix/perl/math-pari.html`

** see `http://www.math.jussieu.fr/~fermigie/PariPython/readme.html`

(Michael_Stoll@math.uni-bonn.de) has integrated PARI into CLISP, which is a Common Lisp implementation by Bruno Haible, Marcus Daniels and others. These provide interfaces to GP functions for use in perl, python or Lisp programs. To our knowledge, only the python and perl interfaces have been upgraded to version 2.0 of PARI, the CLISP one being still based on version 1.39.*xx*.

2.9 The preferences file.

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

2.9.1 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. No other places are examined.

• If the operating system supports environment variables (essentially, anything but MacOS), 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.

2.9.2 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).

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

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

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

`EMACS`: `true` if GP is running in an Emacs or TeXmacs shell (see Section 2.10).

`READL`: `true` if GP is compiled with `readline` support (see Section 2.11.1).

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

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

• *default* `=` *value*, where *default* is one of the available defaults (see Section 2.1), 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";
}
```

2.10 Using GP under GNU Emacs.

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

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

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

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

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

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

Nevertheless, if extended help is available on your system (see Section 2.2.1), you should use it instead of the above, since it's nicer (it ran through TEX) and understands many more keywords.

Finally you can use command completion in the following way. After the prompt, type the first few letters of the command, then `<TAB>` where `<TAB>` is the TAB key. If there exists a unique command starting with the letters you have typed, the command name will be completed. If not,

either the list of commands starting with the letters you typed will be displayed in a separate window (which you can then kill by typing as usual `C-x 1` or by typing in more letters), or "no match found" will be displayed in the Emacs command line. If your GP was linked with the readline library, read the section on completion in the section below (the paragraph on online help is not relevant).

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

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

2.11 Using GP with readline.

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

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

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

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

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

```
$if Pari-GP
  set show-all-if-ambiguous
```

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

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

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

Note: In recent 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 don't want to fall back to the `Esc` combination, put the following two lines in your `.inputrc`:

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

2.11.2 Command completion and online help

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

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

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

• an ending comma if we're completing a default,

• a pair of parentheses if we're 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'll probably find it very convenient. For instance, you can obtain `default(seriesprecision,10)`, just by hitting `def<TAB>se<TAB>10`, which saves 18 keystrokes (out of 27).

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

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

Chapter 3:
Functions and Operations Available in PARI and GP

The functions and operators available in PARI and in the GP/PARI calculator are numerous and everexpanding. Here is a description of the ones available in version 2.2.7. It should be noted that many of these functions accept quite different types as arguments, but others are more restricted. The list of acceptable types will be given for each function or class of functions. Except when stated otherwise, it is understood that a function or operation which should make natural sense is legal. In this chapter, we will describe the functions according to a rough classification. The general entry looks something like:

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

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

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

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

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

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

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

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

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

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

Mnemonics for flags. Numeric flags as mentionned above are obscure, error-prone, and quite rigid: should the authors want to adopt a new flag numbering scheme (for instance when noticing flags with the same meaning but different numeric values), 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 (taken from description of `log`($x, \{flag = 0\}$)):

If *flag* is equal to 1 = `AGM`, use an agm formula...

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

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

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

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

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

foo($x, \&e$)

it means that, besides the normal return value, the function may assign a value to e as a side effect. When passing the argument, the & sign has to be typed in explicitly. As of version 2.2.7, this `pointer` argument is optional for all documented functions, hence the & will always appear between brackets as in `issquare`($x, \{\&e\}$).

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

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

3.1 Standard monadic or dyadic operators.

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

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

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

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

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

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

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

3.1.4 `/`: The expression x `/` y is the quotient of x and y. In addition to the impossibilities for multiplication, note that if the divisor is a matrix, it must be an invertible square matrix, and in that case the result is $x*y^{-1}$. Furthermore note that the result is as exact as possible: in particular, division of two integers always gives a rational number (which may be an integer if the quotient is exact) and *not* the Euclidean quotient (see x `\` y for that), and similarly the quotient of two polynomials is a rational function in general. To obtain the approximate real value of the quotient of two integers, add `0.` to the result; to obtain the approximate p-adic value of the quotient of two integers, add `O(p^k)` to the result; finally, to obtain the Taylor series expansion of the quotient of two polynomials, add `O(X^k)` to the result or use the `taylor` function (see Section 3.7.32).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.9 ^: The expression x`^`n is powering. If the exponent is an integer, then exact operations are performed using binary (left-shift) powering techniques. In particular, in this case x cannot be a vector or matrix unless it is a square matrix (and moreover 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, 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 */
```

The library syntax is **gpow**$(x, n, prec)$ for $x\hat{}n$.

3.1.10 shift$(x, n, \{flag = 0\})$ or x `<<` n $(= x$ `>>` $(-n))$: shifts x componentwise left by n bits if $n \geq 0$ and right by $|n|$ bits if $n < 0$. A left shift by n corresponds to multiplication by 2^n. A right shift of an integer x by $|n|$ corresponds to a Euclidean division of x by $2^{|n|}$ with a remainder of the same sign as x, hence is not the same (when $x < 0$) as $x\backslash 2^{|n|}$. If *flag* is non-zero, this behaviour is modified and right shift of a negative x *is* the same as $x\backslash 2^{|n|}$ (which is consistent with 2-complement semantic of negative numbers).

The library syntax is **gshift3**$(x, n, flag)$ where n is a `long`. Also available is **gshift**(x, n) for the case $flag = 0$.

3.1.11 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 `shift` Euclidean division takes place when $n < 0$ hence if x is an integer the result is still an integer.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1.16 vecmax(x): if x is a vector or a matrix, returns the maximum of the elements of x, otherwise returns a copy of x. Returns $-\infty$ in the form of $-(2^{31} - 1)$ (or $-(2^{63} - 1)$ for 64-bit machines) if x is empty.

The library syntax is **vecmax**(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. Returns $+\infty$ in the form of $2^{31} - 1$ (or $2^{63} - 1$ for 64-bit machines) if x is empty.

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

3.2 Conversions and similar elementary functions or commands.

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

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

This is useless in library mode.

3.2.2 Mat$(x = [\,])$: transforms the object x into a matrix. If x is not a vector or a matrix, this creates a 1×1 matrix. If x is a row (resp. column) vector, this creates a 1-row (resp. 1-column) matrix. If x is already a matrix, a copy of x is created.

This function can be useful in connection with the function `concat` (see there).

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

3.2.3 Mod$(x, y, \{flag = 0\})$: creates the PARI object $(x \bmod y)$, i.e. an integermod 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.

If *flag* is equal to 1, the modulus of the created result is put on the heap and not on the stack, and hence becomes a permanent copy which cannot be erased later by garbage collecting (see Section 4.4). Functions will operate faster on such objects and memory consumption will be lower. On the other hand, care should be taken to avoid creating too many such objects.

Under GP, the same effect can be obtained by assigning the object to a GP variable (the value of which is a permanent object for the duration of the relevant library function call, and is treated as such). This value is subject to garbage collection, since it will be deleted when the value changes. This is preferable and the above flag is only retained for compatibility reasons (it can still be useful in library mode).

The library syntax is **Mod0**$(x, y, flag)$. Also available are

- for *flag* $= 1$: **gmodulo**(x, y).
- for *flag* $= 0$: **gmodulcp**(x, y).

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

Warning: this is *not* a substitution function. It is intended to be quick and dirty. So if you try `Pol(a,y)` on the polynomial `a = x+y`, you will get `y+y`, which is not a valid PARI object.

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

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

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

3.2.6 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.

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

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

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

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

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

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

3.2.9 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 2.6.6.

```
? 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.

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

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

3.2.11 Strexpand($\{x\}*$): converts its argument list into a single character string (type `t_STR`, the empty string if x is omitted). Then performe environment expansion, see Section 2.1. 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 2.6.6.

3.2.12 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 2.6.6.

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

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

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

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

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

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

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

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

Negative numbers behave as if modulo a huge power of 2.

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

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

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

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

Negative numbers behave as if modulo a huge power of 2.

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

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

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

Negative numbers behave as if modulo a huge power of 2.

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

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

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

Negative numbers behave as if modulo a huge power of 2.

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

3.2.20 bittest$(x, n, \{c = 1\})$: extracts $|c|$ bits starting from number n from the right in the development of $|x|$ (i.e. the coefficient of 2^n in the binary expansion of x), returning the bits as an integer bitmap. That is, if $x = \sum x_i 2^i$ with the x_i in $\{0, 1\}$, this routine returns the integer

$$\sum_{0 \le i < |c|} x_{n+i} 2^i$$

Bits at negative offsets are 0. A negative value of c means that negative values of x are treated in the spirit of 2-complement arithmetic (i.e modulo a big power of 2). To extract several bits (or groups of bits if $|c| > 1$) separately at once as a vector, pass a vector for n.

The library syntax is **gbittest3**(x, n, c). Also available are **gbittest**(x, n) (default case $c = 1$) and for simple cases **bittest**(x, n), where n and the result are `long`s.

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

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

Negative numbers behave as if modulo a huge power of 2.

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

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

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

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

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

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

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

3.2.25 components of a PARI object:

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

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

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

The two other methods are more natural but more restricted. The function **polcoeff**(x, n) gives the coefficient of degree n of the polynomial or power series x, with respect to the main variable of x (to check variable ordering, or to change it, use the function `reorder`, see Section 3.11.2.22). In particular if n is less than the valuation of x or in the case of a polynomial, greater than the degree, the result is zero (contrary to `compo` which would send an error message). If x is a power series and n is greater than the largest significant degree, then an error message is issued.

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

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

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

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

write. If x is a matrix, `x[m,n]` represents the coefficient of row `m` and column `n` of the matrix, `x[m,]` represents the m^{th} *row* of x, and `x[,n]` represents the n^{th} *column* of x.

Finally note that in library mode, the macros **coeff** and **mael** are available to deal with the non-recursivity of the `GEN` type from the compiler's point of view. See the discussion on typecasts in Chapter 4.

3.2.26 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, integermods, fractions or p-adics. The only forbidden type is polmod (see `conjvec` for this).

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

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

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

3.2.28 denominator(x): lowest denominator of x. The meaning of this is clear when x is a rational number or function. When x is an integer or a polynomial, the result is equal to 1. When x is a vector or a matrix, the lowest common denominator of the components of x is computed. All other types are forbidden.

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 2.6.2.

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

3.2.29 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 euclidian quotient of the numerator by the denominator.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.36 numerator(x): numerator of x. When x is a rational number or function, the meaning is clear. When x is an integer or a polynomial, the result is x itself. When x is a vector or a matrix, then `numerator(x)` is defined to be `denominator(x)*x`. All other types are forbidden.

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 2.6.2.

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.41 random($\{N = 2^{31}\}$): gives a random integer between 0 and $N - 1$. N can be arbitrary large. This is an internal PARI function and does not depend on the system's random number generator. Note that the resulting integer is obtained by means of linear congruences and will not be well distributed in arithmetic progressions.

The library syntax is **genrand**(N).

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

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

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

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

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

whereas

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

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

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

3.2.44 simplify(x): this function tries to simplify the object x as much as it can. The simplifications do not concern rational functions (which PARI automatically tries to simplify), but type changes. Specifically, a complex or quadratic number whose imaginary part is exactly equal to 0 (i.e. not a real zero) is converted to its real part, and a polynomial of degree zero is converted to its constant term. For all types, this of course occurs recursively. This function is useful in any case, but in particular before the use of arithmetic functions which expect integer arguments, and not for example a complex number of 0 imaginary part and integer real part (which is however printed as an integer).

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

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

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

3.2.46 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 `length(Str(x))`, which is much slower.

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

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

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

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

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

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

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

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

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

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

3.3 Transcendental functions.

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

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

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

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

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

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

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

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

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

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

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

• If the argument is a vector or a matrix, the result is the componentwise evaluation of the function. In particular, transcendental functions on square matrices, which are not implemented in the present version 2.2.7 (see Appendix B however), will have a slightly 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*ln(x))`. It can be applied to p-adic numbers as well as to the more usual types.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.16 bernvec(x): creates a vector containing, as rational numbers, the Bernoulli numbers B_0, B_2,..., B_{2x}. This routine is obsolete. Use `bernfrac` instead each time you need a Bernoulli number in exact form.

Note: this routine is implemented using repeated independant calls to `bernfrac`, which is faster than the standard recursion in exact arithmetic. It is only kept for backward compatibility: it is not faster than individual calls to `bernfrac`, its output uses a lot of memory space, and coping with the index shift is awkward.

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

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

The library syntax is **hbessel1**$(nu, x, prec)$.

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

The library syntax is **hbessel2**$(nu, x, prec)$.

3.3.19 besseli(nu, x): I-Bessel function of index nu and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu + 1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

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

3.3.20 besselj(nu, x): J-Bessel function of index nu and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu + 1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

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

3.3.21 besseljh(n, x): J-Bessel function of half integral index. More precisely, `besseljh`(n, x) computes $J_{n+1/2}(x)$ where n must be of type integer, and x is any element of **C**. In the present version 2.2.7, this function is not very accurate when x is small.

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

3.3.22 besselk$(nu, x, \{flag = 0\})$: K-Bessel function of index nu (which can be complex) and argument x. Only real and positive arguments x are allowed in the present version 2.2.7. If *flag* is equal to 1, uses another implementation of this function which is often faster.

The library syntax is **kbessel**$(nu, x, prec)$ and **kbessel2**$(nu, x, prec)$ respectively.

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

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

3.3.24 cos(x): cosine of x.

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

3.3.25 cosh(x): hyperbolic cosine of x.

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

3.3.26 cotan(x): cotangent of x.

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

3.3.27 dilog(x): principal branch of the dilogarithm of x, i.e. analytic continuation of the power series $\log_2(x) = \sum_{n\geq 1} x^n/n^2$.

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

3.3.28 eint1$(x, \{n\})$: exponential integral $\int_x^\infty \frac{e^{-t}}{t}\, dt$ $(x \in \mathbf{R})$

If n is present, outputs the n-dimensional vector $[\texttt{eint1}(x), \ldots, \texttt{eint1}(nx)]$ $(x \geq 0)$. This is faster than repeatedly calling `eint1(`i `* x)`.

The library syntax is **veceint1**$(x, n, prec)$. Also available is **eint1**$(x, prec)$.

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

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

3.3.30 eta$(x, \{flag = 0\})$: Dedekind's η function, without the $q^{1/24}$. This means the following: if x is a complex number with positive imaginary part, the result is $\prod_{n=1}^\infty (1 - q^n)$, where $q = e^{2i\pi x}$. If x is a power series (or can be converted to a power series) with positive valuation, the result is $\prod_{n=1}^\infty (1 - x^n)$.

If $flag = 1$ and x can be converted to a complex number (i.e. is not a power series), computes the true η function, including the leading $q^{1/24}$.

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

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

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

3.3.32 gammah(x): gamma function evaluated at the argument $x + 1/2$. When x is an integer, this is much faster than using `gamma`$(x + 1/2)$.

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

3.3.33 gamma(x): gamma function of x. In the present version 2.2.7 the p-adic gamma function is not implemented.

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

3.3.34 hyperu(a, b, x): U-confluent hypergeometric function with parameters a and b. The parameters a and b can be complex but the present implementation requires x to be positive.

The library syntax is **hyperu**$(a, b, x, prec)$.

3.3.35 incgam(s, x, y): incomplete gamma function.

x must be positive and s real. 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 tremendously speed up the computation.

The library syntax is **incgam**$(s, x, prec)$ and **incgam0**$(s, x, y, prec)$, respectively (an omitted y is coded as `NULL`). There exist also the functions **incgam1** and **incgam2** which are used for internal purposes.

3.3.36 incgamc(s, x): complementary incomplete gamma function.

The arguments s and x must be positive. The result returned is $\int_0^x e^{-t}t^{s-1}\,dt$, when x is not too large.

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

3.3.37 log$(x, \{flag = 0\})$: principal branch of the natural logarithm of x, i.e. such that $\mathrm{Im}(\ln(x)) \in]-\pi, \pi]$. The result is complex (with imaginary part equal to π) if $x \in \mathbf{R}$ and $x < 0$.

p-adic arguments are also accepted for x, with the convention that $\ln(p) = 0$. Hence in particular $\exp(\ln(x))/x$ will not in general be equal to 1 but to a $(p-1)$-th root of unity (or ± 1 if $p = 2$) times a power of p.

If *flag* is equal to $1 =$ `AGM`, use an agm formula suggested by Mestre, when x is real, otherwise identical to `log`.

The library syntax is **glog**$(x, prec)$ or **glogagm**$(x, prec)$.

3.3.38 lngamma(x): principal branch of the logarithm of the gamma function of x. Can have much larger arguments than `gamma` itself. In the present version 2.2.7, the p-adic `lngamma` function is not implemented.

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

3.3.39 polylog$(m, x, flag = 0)$: one of the different polylogarithms, depending on *flag*:

If $flag = 0$ or is omitted: m^{th} polylogarithm of x, i.e. analytic continuation of the power series $\mathrm{Li}_m(x) = \sum_{n\geq 1} x^n/n^m$. The program uses the power series when $|x|^2 \leq 1/2$, and the power series expansion in $\log(x)$ otherwise. It is valid in a large domain (at least $|x| < 230$), but should not be used too far away from the unit circle since it is then better to use the functional equation linking the value at x to the value at $1/x$, which takes a trivial form for the variant below. Power series, polynomial, rational and vector/matrix arguments are allowed.

For the variants to follow we need a notation: let $\Re_m$ denotes $\Re$ or $\Im$ depending whether m is odd or even.

If $flag = 1$: modified m^{th} polylogarithm of x, called $\tilde{D}_m(x)$ in Zagier, 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: modified m^{th} polylogarithm of x, called $D_m(x)$ in Zagier, defined for $|x| \le 1$ 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: another modified m^{th} polylogarithm of x, called $P_m(x)$ in Zagier, defined for $|x| \le 1$ by

$$\Re_m\left(\sum_{k=0}^{m-1}\frac{2^kB_k}{k!}(\log|x|)^k\mathrm{Li}_{m-k}(x) - \frac{2^{m-1}B_m}{m!}(\log|x|)^m\right).$$

These three functions satisfy the functional equation $f_m(1/x) = (-1)^{m-1}f_m(x)$.

The library syntax is **polylog0**($m, x, \mathit{flag}, \mathit{prec}$).

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

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

3.3.41 sin(x): sine of x.

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

3.3.42 sinh(x): hyperbolic sine of x.

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

3.3.43 sqr(x): square of x. This operation is not completely straightforward, i.e. identical to $x * x$, since it can usually be computed more efficiently (roughly one-half of the elementary multiplications can be saved). Also, squaring a 2-adic number increases its precision. For example,

```
? (1 + O(2^4))^2
%1 = 1 + O(2^5)
? (1 + O(2^4)) * (1 + O(2^4))
%2 = 1 + O(2^4)
```

Note that this function is also called whenever one multiplies two objects which are known to be *identical*, e.g. they are the value of the same variable, or we are computing a power.

```
? x = (1 + O(2^4)); x * x
%3 = 1 + O(2^5)
? (1 + O(2^4))^4
%4 = 1 + O(2^6)
```

(note the difference between `%2` and `%3` above).

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

3.3.44 sqrt(x): principal branch of the square root of x, i.e. such that $\mathrm{Arg}(\mathrm{sqrt}(x)) \in]-\pi/2, \pi/2]$, or in other words such that $\Re(\mathrm{sqrt}(x)) > 0$ or $\Re(\mathrm{sqrt}(x)) = 0$ and $\Im(\mathrm{sqrt}(x)) \geq 0$. If $x \in \mathbf{R}$ and $x < 0$, then the result is complex with positive imaginary part.

Integermod a prime and p-adics are allowed as arguments. In that case, the square root (if it exists) which is returned is the one whose first p-adic digit (or its unique p-adic digit in the case of integermods) is in the interval $[0, p/2]$. When the argument is an integermod a non-prime (or a non-prime-adic), the result is undefined.

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

3.3.45 sqrtn$(x, n, \{\&z\})$: principal branch of the nth root of x, i.e. such that $\mathrm{Arg}(\mathrm{sqrt}(x)) \in]-\pi/n, \pi/n]$.

Integermod 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.

The following script computes all roots in all possible cases:

```
sqrtnall(x,n)=
{
  local(V,r,z,r2);
  r = sqrtn(x,n, &z);
  if (!z, error("Impossible case in sqrtn"));
  if (type(x) == "t_INTMOD" || type(x)=="t_PADIC" ,
    r2 = r*z; n = 1;
    while (r2!=r, r2*=z;n++));
  V = vector(n); V[1] = r;
  for(i=2, n, V[i] = V[i-1]*z);
  V
}
addhelp(sqrtnall,"sqrtnall(x,n):compute the vector of nth-roots of x");
```

The library syntax is **gsqrtn**$(x, n, \&z, prec)$.

3.3.46 tan(x): tangent of x.

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

3.3.47 tanh(x): hyperbolic tangent of x.

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

3.3.48 teichmuller(x): Teichmüller character of the p-adic number x.

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

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

The library syntax is **theta**$(q, z, prec)$.

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

The library syntax is **thetanullk**$(q, k, prec)$, where k is a `long`.

3.3.51 weber$(x, \{flag = 0\})$: one of Weber's three f functions. If $flag = 0$, returns

$$f(x) = \exp(-i\pi/24) \cdot \eta((x+1)/2) \,/\, \eta(x) \quad \text{such that} \quad j = (f^{24} - 16)^3/f^{24}\,,$$

where j is the elliptic j-invariant (see the function `ellj`). If $flag = 1$, returns

$$f_1(x) = \eta(x/2) \,/\, \eta(x) \quad \text{such that} \quad j = (f_1^{24} + 16)^3/f_1^{24}\,.$$

Finally, if $flag = 2$, returns

$$f_2(x) = \sqrt{2}\eta(2x) \,/\, \eta(x) \quad \text{such that} \quad j = (f_2^{24} + 16)^3/f_2^{24}.$$

Note the identities $f^8 = f_1^8 + f_2^8$ and $f f_1 f_2 = \sqrt{2}$.

The library syntax is **weber0**$(x, flag, prec)$, or **wf**$(x, prec)$, **wf1**$(x, prec)$ or **wf2**$(x, prec)$.

3.3.52 zeta(s): 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.

The library syntax is **gzeta**$(s, prec)$.

3.4 Arithmetic functions.

These functions are by definition functions whose natural domain of definition is either $\mathbf{Z}$ (or $\mathbf{Z}_{>0}$), or sometimes polynomials over a base ring. Functions which concern polynomials exclusively will be explained in the next section. The way these functions are used is completely different from transcendental functions: in general only the types integer and polynomial are accepted as arguments. If a vector or matrix type is given, the function will be applied on each coefficient independently.

In the present version 2.2.7, all arithmetic functions in the narrow sense of the word — Euler's totient function, the Moebius function, the sums over divisors or powers of divisors etc.— call, after trial division by small primes, the same versatile factoring machinery described under `factorint`. It includes Shanks SQUFOF, Pollard Rho, ECM and MPQS stages, and has an early exit option for the functions **moebius** and (the integer function underlying) **issquarefree**. Note that it relies on a (fairly strong) probabilistic primality test, see `ispseudoprime`.

3.4.1 addprimes$(\{x = []\})$: adds the "primes" contained in the vector x (or the single integer x) to the table computed upon GP initialization (by `pari_init` in library mode), and returns a row vector entries contain all such user primes. Whenever `factor` or `smallfact` is subsequently called, first the primes in the table computed by `pari_init` will be checked, and then the additional primes in this table. If x is empty or omitted, just returns the current list of extra primes.

The entries in x are not checked for primality. They need only be positive integers not divisible by any of the pre-computed primes. It's in fact a nice trick to add composite numbers, which for example the function `factor`$(x, 0)$ was not able to factor. In case the message "impossible inverse modulo ⟨*some INTMOD*⟩" shows up afterwards, you have just stumbled over a non-trivial factor. Note that the arithmetic functions in the narrow sense, like **eulerphi**, do *not* use this extra table.

To remove primes from the list use `removeprimes`.

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

3.4.2 bestappr$(x, A, \{B\})$: if B is omitted, finds the best rational approximation to $x \in \mathbf{R}$ (or $\mathbf{R}[X]$, or $\mathbf{R}^n$, ...) with denominator at most equal to A using continued fractions.

If B is present, x is assumed to be of type `t_INTMOD` modulo M (or a recursive combination of those), and the routine returns the unique fraction a/b in coprime integers $a \le A$ and $b \le B$ which is congruent to x modulo M. If $M \le 2AB$, uniqueness is not guaranteed and the function fails with an error message. If rational reconstruction is not possible (no such a/b exists for at least one component of x), returns -1.

The library syntax is **bestappr0**(x, A, B). Also available is **bestappr**(x, A) corresponding to an omitted B.

3.4.3 bezout(x, y): finds u and v minimal in a natural sense such that $x * u + y * v = \gcd(x, y)$. The arguments must be both integers or both polynomials, and the result is a row vector with three components u, v, and $\gcd(x, y)$.

The library syntax is **vecbezout**(x, y) to get the vector, or **gbezout**$(x, y, \&u, \&v)$ which gives as result the address of the created gcd, and puts the addresses of the corresponding created objects into u and v.

3.4.4 bezoutres(x, y): as `bezout`, with the resultant of x and y replacing the gcd.

The library syntax is **vecbezoutres**(x, y) to get the vector, or **subresext**$(x, y, \&u, \&v)$ which gives as result the address of the created gcd, and puts the addresses of the corresponding created objects into u and v.

3.4.5 bigomega(x): number of prime divisors of $|x|$ counted with multiplicity. x must be an integer.

The library syntax is **bigomega**(x), the result is a `long`.

3.4.6 binomial(x, y): binomial coefficient $\binom{x}{y}$. Here y must be an integer, but x can be any PARI object.

The library syntax is **binome**(x, y), where y must be a `long`.

3.4.7 chinese$(x, \{y\})$: if x and y are both integermods 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 **chinois**(x, y).

3.4.8 content(x): computes the gcd of all the coefficients of x, when this gcd makes sense. If x is a scalar, this simply returns the absolute value of x if x is rationnal (`t_INT`, `t_FRAC` or `t_FRACN`), and x otherwise. If x is a polynomial (and by extension a power series), it gives the usual content of x. If x is a rational function, it gives the ratio of the contents of the numerator and the denominator. Finally, if x is a vector or a matrix, it gives the gcd of the contents of all entries.

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

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

The number of partial quotients n is limited to $nmax$. If x is a real number, the expansion stops at the last significant partial quotient if $nmax$ is omitted. x can also be a rational function or a power series.

If a vector b is supplied, the numerators will be equal to the coefficients of b (instead of all equal to 1 as above). The length of the result is then equal to the length of b, unless a partial remainder is encountered which is equal to zero, in which case the expansion stops. In the case of real numbers, the stopping criterion is thus different from the one mentioned above since, if b is too long, some partial quotients may not be significant.

If b is an integer, the command is understood as `contfrac(`$x, nmax$`)`.

The library syntax is **contfrac0**$(x, b, nmax)$. Also available are **gboundcf**$(x, nmax)$, **gcf**(x), or **gcf2**(b, x), where $nmax$ is a C integer.

3.4.10 contfracpnqn(x): when x is a vector or a one-row matrix, x is considered as the list of partial quotients $[a_0, a_1, \ldots, a_n]$ of a rational number, and the result is the 2 by 2 matrix $[p_n, p_{n-1}; q_n, q_{n-1}]$ in the standard notation of continued fractions, so $p_n/q_n = a_0 + 1/(a_1 + \ldots + 1/a_n)\ldots)$. If x is a matrix with two rows $[b_0, b_1, \ldots, b_n]$ and $[a_0, a_1, \ldots, a_n]$, this is then considered as a generalized continued fraction and we have similarly $p_n/q_n = 1/b_0(a_0 + b_1/(a_1 + \ldots + b_n/a_n)\ldots)$. Note that in this case one usually has $b_0 = 1$.

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

3.4.11 core$(n, \{flag = 0\})$: if n is a non-zero integer written as $n = df^2$ with d squarefree, returns d. If *flag* is non-zero, returns the two-element row vector $[d, f]$.

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

3.4.12 coredisc$(n, \{flag\})$: if n is a non-zero integer written as $n = df^2$ with d fundamental discriminant (including 1), returns d. If *flag* is non-zero, returns the two-element row vector $[d, f]$. Note that if n is not congruent to 0 or 1 modulo 4, f will be a half integer and not an integer.

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

3.4.13 dirdiv(x, y): x and y being vectors of perhaps different lengths but with $y[1] \neq 0$ considered as Dirichlet series, computes the quotient of x by y, again as a vector.

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

3.4.14 direuler$(p = a, b, expr, \{c\})$: computes the Dirichlet series to b terms of 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 Dirichlet series (or more precisely the local factor) $expr(p^{-s})$. If c is present, output only the first c coefficients in the series.

The library syntax is **direuler**(`entree *ep, GEN a, GEN b, char *expr`)

3.4.15 dirmul(x, y): x and y being vectors of perhaps different lengths considered as Dirichlet series, computes the product of x by y, again as a vector.

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

3.4.16 divisors(x): creates a row vector whose components are the positive divisors of the integer x in increasing order. The factorization of x (as output by `factor`) can be used instead.

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

3.4.17 eulerphi(x): Euler's ϕ (totient) function of $|x|$, in other words $|(\mathbf{Z}/x\mathbf{Z})^*|$. x must be of type integer.

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

3.4.18 factor$(x, \{lim = -1\})$: general factorization function. If x is of type integer, rational, polynomial or rational function, the result is a two-column matrix, the first column being the irreducibles dividing x (prime numbers or polynomials), and the second the exponents. If x is a vector or a matrix, the factoring is done componentwise (hence the result is a vector or matrix of two-column matrices). By definition, 0 is factored as 0^1.

If x is of type integer or rational, the factors are *pseudoprimes* (see `ispseudoprime`), and in general not rigorously proven primes. In fact, any factor which is $\leq 10^{13}$ is a genuine prime number. Use `isprime` to prove primality of other factors, as in

```
fa = factor(2^2^7 +1)
isprime( fa[,1] )
```

An argument *lim* can be added, meaning that we look only for factors up to *lim*, or to `primelimit`, whichever is lowest (except when $lim = 0$ where the effect is identical to setting $lim =$ `primelimit`). In this case, the remaining part may actually be a proven composite! See `factorint` for more information about the algorithms used.

The polynomials or rational functions to be factored must have scalar coefficients. In particular PARI does *not* know how to factor multivariate polynomials. See `factormod` and `factorff` for the algorithms used over finite fields, `factornf` for the algorithms over number fields. Over $\mathbf{Q}$, van Hoeij's method is used, which is able to cope with hundreds of modular factors.

Note that PARI tries to guess in a sensible way over which ring you want to factor. Note also that factorization of polynomials is done up to multiplication by a constant. In particular, the factors of rational polynomials will have integer coefficients, and the content of a polynomial or rational function is discarded and not included in the factorization. If needed, you can always ask for the content explicitly:

```
? factor(t^2 + 5/2*t + 1)
%1 =
```

```
[2*t + 1 1]

[t + 2 1]

? content(t^2 + 5/2*t + 1)
%2 = 1/2
```

See also `factornf` and `nffactor`.

The library syntax is **factor0**(x, lim), where lim is a C integer. Also available are **factor**(x) (= **factor0**$(x, -1)$), **smallfact**(x) (= **factor0**$(x, 0)$).

3.4.19 factorback$(f, \{e\}, \{nf\})$: gives back the factored object corresponding to a factorization. If the last argument is of number field type (e.g. created by `nfinit` or `bnfinit`), assume we are dealing with an ideal factorization in the number field. The resulting ideal product is given in HNF form.

If e is present, e and f must be vectors of the same length (e being integral), and the corresponding factorization is the product of the $f[i]^{e[i]}$.

If not, and f is vector, it is understood as in the preceding case with e a vector of 1 (the product of the $f[i]$ is returned). Finally, f can be a regular factorization, as produced with any `factor` command. A few examples:

```
? factorback([2,2; 3,1])
%1 = 12
? factorback([2,2], [3,1])
%2 = 12
? factorback([5,2,3])
%3 = 30
? factorback([2,2], [3,1], nfinit(x^3+2))
%4 =
[16 0 0]

[0 16 0]

[0 0 16]
? nf = nfinit(x^2+1); fa = idealfactor(nf, 10)
%5 =
[[2, [1, 1]~, 2, 1, [1, 1]~] 2]

[[5, [-2, 1]~, 1, 1, [2, 1]~] 1]

[[5, [2, 1]~, 1, 1, [-2, 1]~] 1]
? factorback(fa)
  ***   forbidden multiplication t_VEC * t_VEC.
? factorback(fa, nf)
%6 =
[10 0]

[0 10]
```

In the fourth example, 2 and 3 are interpreted as principal ideals in a cubic field. In the fifth one, `factorback(fa)` is meaningless since we forgot to indicate the number field, and the entries in the first column of `fa` can't be multiplied.

The library syntax is **factorback0**(f, e, nf), where an omitted nf or e is entered as `NULL`. Also available is `factorback`(f, nf) (case $e =$ `NULL`) where an omitted nf is entered as `NULL`.

3.4.20 factorcantor(x, p): factors the polynomial x modulo the prime p, using distinct degree plus Cantor-Zassenhaus. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two-column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. If you want only the *degrees* of the irreducible polynomials (for example for computing an L-function), use `factormod`$(x, p, 1)$. Note that the `factormod` algorithm is usually faster than `factorcantor`.

The library syntax is **factcantor**(x, p).

3.4.21 factorff(x, p, a): factors the polynomial x in the field $\mathbf{F}_q$ defined by the irreducible polynomial a over $\mathbf{F}_p$. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two-column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. It is recommended to use for the variable of a (which will be used as variable of a polmod) a name distinct from the other variables used, so that a `lift()` of the result will be legible. If all the coefficients of x are in $\mathbf{F}_p$, a much faster algorithm is applied, using the computation of isomorphisms between finite fields.

The library syntax is **factmod9**(x, p, a).

3.4.22 factorial(x) or $x!$: factorial of x. The expression $x!$ gives a result which is an integer, while `factorial`(x) gives a real number.

The library syntax is **mpfact**(x) for $x!$ and **mpfactr**$(x, prec)$ for `factorial`(x). x must be a `long` integer and not a PARI integer.

3.4.23 factorint$(n, \{flag = 0\})$: factors the integer n into a product of pseudoprimes (see `ispseudoprime`), using a combination of the Shanks SQUFOF and Pollard Rho method (with modifications due to Brent), Lenstra's ECM (with modifications by Montgomery), and MPQS (the latter adapted from the LiDIA code with the kind permission of the LiDIA maintainers), as well as a search for pure powers with exponents≤ 10. The output is a two-column matrix as for `factor`. Use `isprime` on the result if you want to guarantee primality.

This gives direct access to the integer factoring engine called by most arithmetical functions. *flag* is optional; its binary digits mean 1: avoid MPQS, 2: skip first stage ECM (we may still fall back to it later), 4: avoid Rho and SQUFOF, 8: don't run final ECM (as a result, a huge composite may be declared to be prime). Note that a (strong) probabilistic primality test is used; thus composites might (very rarely) not be detected.

The machinery underlying this function is still in a somewhat experimental state, but should be much faster on average than pure ECM as used by all PARI versions up to 2.0.8, at the expense of heavier memory use. You are invited to play with the flag settings and watch the internals at work by using GP's `debuglevel` default parameter (level 3 shows just the outline, 4 turns on time keeping, 5 and above show an increasing amount of internal details). If you see anything funny happening, please let us know.

The library syntax is **factorint**$(n, flag)$.

3.4.24 factormod$(x, p, \{flag = 0\})$: factors the polynomial x modulo the prime integer p, using Berlekamp. The coefficients of x must be operation-compatible with $\mathbf{Z}/p\mathbf{Z}$. The result is a two-column matrix, the first column being the irreducible polynomials dividing x, and the second the exponents. If *flag* is non-zero, outputs only the *degrees* of the irreducible polynomials (for example, for computing an L-function). A different algorithm for computing the mod p factorization is `factorcantor` which is sometimes faster.

The library syntax is **factormod**$(x, p, flag)$. Also available are **factmod**(x, p) (which is equivalent to **factormod**$(x, p, 0)$) and **simplefactmod**(x, p) (= **factormod**$(x, p, 1)$).

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

The library syntax is **fibo**(x). x must be a `long`.

3.4.26 ffinit$(p, n, \{v = x\})$: computes a monic polynomial of degree n which is irreducible over $\mathbf{F}_p$. For instance if `P = ffinit(3,2,y)`, you can represent elements in $\mathbf{F}_{3^2}$ as polmods modulo `P`. Starting with version 2.2.3 this function use a fast variant of Adleman–Lenstra algorithm, and is much faster than in earlier versions.

The library syntax is **ffinit**(p, n, v), where v is a variable number.

3.4.27 gcd$(x, \{y\}, \{flag = 0\})$: creates the greatest common divisor of x and y. x and y can be of quite general types, for instance both rational numbers. Vector/matrix types are also accepted, in which case the GCD is taken recursively on each component. Note that for these types, `gcd` is not commutative. *flag* is obsolete and should not be used.

If y is omitted and x is a vector, return the gcd of all components of x. The algorithm used is a naive Euclid except for the following inputs:

• integers: use modified right-shift binary ("plus-minus" variant).

• univariate polynomials with coeffients in the same number field (in particular rational): use modular gcd algorithm.

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

The library syntax is **ggcd**(x, y). For general polynomial inputs, **srgcd**(x, y) is also available. For univariate *rational* polynomials, one also has **modulargcd**(x, y).

3.4.28 hilbert$(x, y, \{p\})$: Hilbert symbol of x and y modulo p. If x and y are of type integer or fraction, an explicit third parameter p must be supplied, $p = 0$ meaning the place at infinity. Otherwise, p needs not be given, and x and y can be of compatible types integer, fraction, real, integermod a prime (result is undefined if the modulus is not prime), or p-adic.

The library syntax is **hil**(x, y, p).

3.4.29 isfundamental(x): true (1) if x is equal to 1 or to the discriminant of a quadratic field, false (0) otherwise.

The library syntax is **gisfundamental**(x), but the simpler function **isfundamental**(x) which returns a `long` should be used if x is known to be of type integer.

3.4.30 isprime$(x, \{flag = 0\})$: true (1) if x is a (proven) prime number, false (0) otherwise. This can be very slow when x is indeed prime and has more than 1000 digits, say. Use `ispseudoprime` to quickly check for pseudo primality.

If *flag* $= 0$, use a combination of Baillie-PSW pseudo primality test (see `ispseudoprime`), Selfridge "$p-1$" test if $x-1$ is smooth enough, and Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL) for general x.

If *flag* $= 1$, use Selfridge-Pocklington-Lehmer "$p-1$" test and output a primality certificate as follows: return 0 if x is composite, 1 if x is small enough that passing Baillie-PSW test guarantees its primality (currently $x < 10^{13}$), 2 if x is a large prime whose primality could only sensibly be proven (given the algorithms implemented in PARI) using the APRCL test. Otherwise (x is large and $x-1$ is smooth) output a three column matrix as a primality certificate. The first column contains the prime factors p of $x-1$, the second the corresponding elements a_p as in Proposition 8.3.1 in GTM 138, and the third the output of isprime(p,1). The algorithm fails if one of the pseudo-prime factors is not prime, which is exceedingly unlikely (and well worth a bug report).

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

The library syntax is **gisprime**$(x, flag)$, but the simpler function **isprime**(x) which returns a `long` should be used if x is known to be of type integer.

3.4.31 ispseudoprime$(x, \{flag\})$: true (1) if x is a strong pseudo prime (see below), false (0) otherwise. If this function returns false, x is not prime; if, on the other hand it returns true, it is only highly likely that x is a prime number. Use `isprime` (which is of course much slower) to prove that x is indeed prime.

If *flag* $= 0$, checks whether x is a Baillie-Pomerance-Selfridge-Wagstaff pseudo prime (strong Rabin-Miller pseudo prime for base 2, followed by strong Lucas test for the sequence $(P, -1)$, P smallest positive integer such that $P^2 - 4$ is not a square mod x).

There are no known composite numbers passing this test (in particular, all composites $\leq 10^{13}$ are correctly detected), although it is expected that infinitely many such numbers exist.

If *flag* > 0, checks whether x is a strong Miller-Rabin pseudo prime for *flag* randomly chosen bases (with end-matching to catch square roots of -1).

The library syntax is **gispseudoprime**$(x, flag)$, but the simpler function **ispseudoprime**(x) which returns a `long` should be used if x is known to be of type integer.

3.4.32 issquare$(x, \{\&n\})$: true (1) if x is square, false (0) if not. x can be of any type. If n is given and an exact square root had to be computed in the checking process, puts that square root in n. This is in particular the case when x is an integer or a polynomial. This is *not* the case for intmods (use quadratic reciprocity) or series (only check the leading coefficient).

The library syntax is **gcarrecomplet**$(x, \&n)$. Also available is **gcarreparfait**(x).

3.4.33 issquarefree(x): true (1) if x is squarefree, false (0) if not. Here x can be an integer or a polynomial.

The library syntax is **gissquarefree**(x), but the simpler function **issquarefree**(x) which returns a `long` should be used if x is known to be of type integer. This **issquarefree** is just the square of the Moebius function, and is computed as a multiplicative arithmetic function much like the latter.

3.4.34 kronecker(x, y): Kronecker symbol $(x|y)$, where x and y must be of type integer. By definition, this is the extension of Legendre symbol to $\mathbf{Z} \times \mathbf{Z}$ by total multiplicativity in both arguments with the following special rules for $y = 0, -1$ or 2:

- $(x|0) = 1$ if $|x| = 1$ and 0 otherwise.
- $(x|-1) = 1$ if $x \geq 0$ and -1 otherwise.
- $(x|2) = 0$ if x is even and 1 if $x = 1, -1 \bmod 8$ and -1 if $x = 3, -3 \bmod 8$.

The library syntax is **kronecker**(x, y), the result (0 or ± 1) is a `long`.

3.4.35 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, return the lcm of all components of x.

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

3.4.36 moebius(x): Moebius μ-function of $|x|$. x must be of type integer.

The library syntax is **mu**(x), the result (0 or ± 1) is a `long`.

3.4.37 nextprime(x): finds the smallest pseudoprime (see `ispseudoprime`) greater than or equal to x. x can be of any real type. Note that if x is a pseudoprime, this function returns x and not the smallest pseudoprime strictly larger than x. To rigorously prove that the result is prime, use `isprime`.

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

3.4.38 numdiv(x): number of divisors of $|x|$. x must be of type integer.

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

3.4.39 numbpart(n): gives the number of unrestricted partitions of n, usually called $p(n)$ in the litterature; in other words the number of nonnegative integer solutions to $a + 2b + 3c + \cdots = n$. n must be of type integer and $1 \leq n < 10^{15}$. The algorithm uses the Hardy-Ramanujan-Rademacher formula.

The library syntax is **numbpart**(n).

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

The library syntax is **omega**(x), the result is a `long`.

3.4.41 precprime(x): finds the largest pseudoprime (see `ispseudoprime`) less than or equal to x. x can be of any real type. Returns 0 if $x \leq 1$. Note that if x is a prime, this function returns x and not the largest prime strictly smaller than x. To rigorously prove that the result is prime, use `isprime`.

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

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

The library syntax is **prime**(x). x must be a `long`.

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

The library syntax is **primes**(x). x must be a `long`.

3.4.44 qfbclassno$(D, \{flag = 0\})$: ordinary class number of the quadratic order of discriminant D. In the present version 2.2.7, a $O(D^{1/2})$ algorithm is used for $D > 0$ (using Euler product and the functional equation) so D should not be too large, say $D < 10^8$, for the time to be reasonable. On the other hand, for $D < 0$ one can reasonably compute `qfbclassno(`D`)` for $|D| < 10^{25}$, since the routine uses Shanks's method which is in $O(|D|^{1/4})$. For larger values of $|D|$, see `quadclassunit`.

If *flag* $= 1$, compute the class number using Euler products and the functional equation. However, it is in $O(|D|^{1/2})$.

Important warning. For $D < 0$, this function may give incorrect results when the class group has a low exponent (has many cyclic factors), because implementing Shanks's method in full generality slows it down immensely. It is therefore strongly recommended to double-check results using either the version with *flag* $= 1$, the function `qfbhclassno`$(-D)$ or the function `quadclassunit`.

Warning. contrary to what its name implies, this routine does not compute the number of classes of binary primitive forms of discriminant D, which is equal to the *narrow* class number. The two notions are the same when $D < 0$ or the fundamental unit ε has negative norm; when $D > 0$ and $N\varepsilon > 0$, the number of classes of forms is twice the ordinary class number. This is a problem which we cannot fix for backward compatibility reasons. Use the following routine if you are only interested in the number of classes of forms:

```
QFBclassno(D) =
  qfbclassno(D) * if (D < 0 || norm(quadunit(D)) < 0, 1, 2)
```

Here are a few examples:

```
? qfbclassno(400000028)
time = 3,140 ms.
%1 = 1
? quadclassunit(400000028).no
time = 20 ms. \\ much faster
%2 = 1
? qfbclassno(-400000028)
time = 0 ms.
%3 = 7253 \\ correct, and fast enough
? quadclassunit(-400000028).no
time = 0 ms.
%4 = 7253
```

The library syntax is **qfbclassno0**$(D, flag)$. Also available: **classno**(D) $(=$ **qfbclassno**$(D))$, **classno2**(D) $(=$ **qfbclassno**$(D, 1))$, and finally there exists the function **hclassno**(D) which computes the class number of an imaginary quadratic field by counting reduced forms, an $O(|D|)$ algorithm. See also `qfbhclassno`.

3.4.45 qfbcompraw(x, y) composition of the binary quadratic forms x and y, without reduction of the result. This is useful e.g. to compute a generating element of an ideal.

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

3.4.46 qfbhclassno(x): Hurwitz class number of x, where x is non-negative and congruent to 0 or 3 modulo 4. See also `qfbclassno`.

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

3.4.47 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 library syntax is **nucomp**(x, y, l). The auxiliary function **nudupl**(x, l) should be used instead for speed when $x = y$.

3.4.48 qfbnupow(x, n): n-th power of the primitive positive definite binary quadratic form x using Shanks's NUCOMP and NUDUPL algorithms (see `qfbnucomp`).

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

3.4.49 qfbpowraw(x, n): n-th power of the binary quadratic form x, computed without doing any reduction (i.e. using `qfbcompraw`). Here n must be non-negative and $n < 2^{31}$.

The library syntax is **powraw**(x, n) where n must be a `long` integer.

3.4.50 qfbprimeform(x, p): prime binary quadratic form of discriminant x whose first coefficient is the prime number p. By abuse of notation, $p = 1$ is a valid special case which returns the unit form. Returns an error if x is not a quadratic residue mod p. In the case where $x > 0$, the "distance" component of the form is set equal to zero according to the current precision.

The library syntax is **primeform**$(x, p, prec)$, where the third variable *prec* is a `long`, but is only taken into account when $x > 0$.

3.4.51 qfbred$(x, \{flag = 0\}, \{D\}, \{isqrtD\}, \{sqrtD\})$: reduces the binary quadratic form x (updating Shanks's distance function if x is indefinite). The binary digits of *flag* are toggles meaning

1: perform a single reduction step

2: don't update Shanks's distance

D, *isqrtD*, *sqrtD*, if present, supply the values of the discriminant, $\lfloor\sqrt{D}\rfloor$, and $\sqrt{D}$ respectively (no checking is done of these facts). If $D < 0$ these values are useless, and all references to Shanks's distance are irrelevant.

The library syntax is **qfbred0**$(x, flag, D, isqrtD, sqrtD)$. Use `NULL` to omit any of D, *isqrtD*, *sqrtD*.

Also available are

redimag(x) (= **qfbred**(x) where x is definite),

and for indefinite forms:

redreal(x) (= **qfbred**(x)),

rhoreal(x) (= **qfbred**$(x, 1)$),

redrealnod(x, sq) (= **qfbred**$(x, 2, , isqrtD)$),

rhorealnod(x, sq) (= **qfbred**$(x, 3, , isqrtD)$).

3.4.52 qfbsolve(Q, p): Solve the equation $Q(x, y) = p$ over the integers, where Q is a imaginary 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 functions return only one solution and not all the solutions.

The library syntax is **qfbsolve**(Q, n).

3.4.53 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.

If *flag* is non-zero *and* $D > 0$, computes the narrow class group and regulator, instead of the ordinary (or wide) ones. In the current version 2.2.7, this doesn't 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 and c_2 are positive real numbers which control the execution time and the stack size. To get maximum speed, set $c_2 = c$. To get a rigorous result (under GRH) you must take $c_2 = 6$. Reasonable values for c are between 0.1 and 2.

The result of this function is a vector v with 3 components if $D < 0$, and 4 otherwise. The correspond respectively to

- $v[1]$: the class number
- $v[2]$: a vector giving the structure of the class group as a product of cyclic groups;
- $v[3]$: a vector giving generators of those cyclic groups (as binary quadratic forms).
- $v[4]$: (omitted if $D < 0$) the regulator, computed to an accuracy which is the maximum of an internal accuracy determined by the program and the current default (note that once the regulator is known to a small accuracy it is trivial to compute it to very high accuracy, see the tutorial).

The library syntax is **quadclassunit0**$(D, flag, tech)$. Also available are **buchimag**(D, c_1, c_2) and **buchreal**$(D, flag, c_1, c_2)$.

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

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

3.4.55 quadhilbert$(D, \{flag = 0\})$: relative equation defining the Hilbert class field of the quadratic field of discriminant D. If *flag* is non-zero and $D < 0$, outputs $[form, root(form)]$ (to be used for constructing subfields). If *flag* is non-zero and $D > 0$, try hard to get the best modulus. Uses complex multiplication in the imaginary case and Stark units in the real case.

The library syntax is **quadhilbert**$(D, flag, prec)$.

3.4.56 quadgen(D): creates the quadratic number $\omega = (a + \sqrt{D})/2$ where $a = 0$ if $x \equiv 0 \bmod 4$, $a = 1$ if $D \equiv 1 \bmod 4$, so that $(1, \omega)$ is an integral basis for the quadratic order of discriminant D. D must be an integer congruent to 0 or 1 modulo 4, which is not a square.

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

3.4.57 quadpoly$(D, \{v = x\})$: creates the "canonical" quadratic polynomial (in the variable v) corresponding to the discriminant D, i.e. the minimal polynomial of `quadgen`(D). D must be an integer congruent to 0 or 1 modulo 4, which is not a square.

The library syntax is **quadpoly0**(x, v).

3.4.58 quadray$(D, f, \{flag = 0\})$: relative equation for the ray class field of conductor f for the quadratic field of discriminant D (which can also be a `bnf`), using analytic methods.

For $D < 0$, uses the σ function. *flag* has the following meaning: if it's an odd integer, outputs instead the vector of $[ideal, corresponding\ root]$. It can also be a two-component vector $[\lambda, flag]$, where *flag* is as above and λ is the technical element of `bnf` necessary for Schertz's method. In that case, returns 0 if λ is not suitable.

For $D > 0$, uses Stark's conjecture. If *flag* is non-zero, try hard to get the best modulus. The function may fail with the following message

```
"Cannot find a suitable modulus in FindModulus"
```

See `bnrstark` for more details about the real case.

The library syntax is **quadray**$(D, f, flag)$.

3.4.59 quadregulator(x): regulator of the quadratic field of positive discriminant x. Returns an error if x is not a discriminant (fundamental or not) or if x is a square. See also `quadclassunit` if x is large.

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

3.4.60 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 3.4.56).

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

3.4.61 removeprimes$(\{x = []\})$: removes the primes listed in x from the prime number table. In particular `removeprimes(addprimes)` empties the extra prime table. x can also be a single integer. List the current extra primes if x is omitted.

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

3.4.62 sigma$(x, \{k = 1\})$: sum of the k^{th} powers of the positive divisors of $|x|$. x must be of type integer.

The library syntax is **sumdiv**(x) (= **sigma**(x)) or **gsumdivk**(x, k) (= **sigma**(x, k)), where k is a C long integer.

3.4.63 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 **racine**(x).

3.4.64 znlog(x, g): g must be a primitive root mod a prime p, and the result is the discrete log of x in the multiplicative group $(\mathbf{Z}/p\mathbf{Z})^*$. This function uses a simple-minded combination of Pohlig-Hellman algorithm and Shanks baby-step/giant-step which requires $O(\sqrt{q})$ storage, where q is the largest prime factor of $p-1$. Hence it cannot be used when the largest prime divisor of $p-1$ is greater than about 10^{13}.

The library syntax is **znlog**(x, g).

3.4.65 znorder(x): 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 library syntax is **order**(x).

3.4.66 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 $k \geq 0$).

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

3.4.67 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 **znstar**(n).

3.5 Functions related to elliptic curves.

We have implemented a number of functions which are useful for number theorists working on elliptic curves. We always use Tate's notations. The functions assume that the curve is given by a general Weierstrass model

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6,$$

where a priori the a_i can be of any scalar type. This curve can be considered as a five-component vector `E=[a1,a2,a3,a4,a6]`. Points on `E` are represented as two-component vectors `[x,y]`, except for the point at infinity, i.e. the identity element of the group law, represented by the one-component vector `[0]`.

It is useful to have at one's disposal more information. This is given by the function `ellinit` (see there), which usually gives a 19 component vector (which we will call a long vector in this section). If a specific flag is added, a vector with only 13 component will be output (which we will call a medium vector). A medium vector just gives the first 13 components of the long vector corresponding to the same curve, but is of course faster to compute. The following member functions are available to deal with the output of `ellinit`:

`a1`–`a6`, `b2`–`b8`, `c4`–`c6` : coefficients of the elliptic curve.
`area` : volume of the complex lattice defining E.
`disc` : discriminant of the curve.
`j` : j-invariant of the curve.
`omega` : $[\omega_1, \omega_2]$, periods forming a basis of the complex lattice defining E (ω_1 is the real period, and ω_2/ω_1 belongs to Poincaré's half-plane).
`eta` : quasi-periods $[\eta_1, \eta_2]$, such that $\eta_1\omega_2 - \eta_2\omega_1 = i\pi$.

`roots` : roots of the associated Weierstrass equation.
`tate` : $[u^2, u, v]$ in the notation of Tate.
`w` : Mestre's w (this is technical).

Their use is best described by an example: assume that E was output by `ellinit`, then typing E`.disc` will retrieve the curve's discriminant. 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 $\mathbf{Q}_p$.

Some functions, in particular those relative to height computations (see `ellheight`) require also that the curve be in minimal Weierstrass form. This is achieved by the function `ellminimalmodel`.

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. For instance, in functions which do not use the extra information given by long vectors, the curve can be given either as a five-component vector, or by one of the longer vectors computed by `ellinit`.

3.5.1 elladd$(E, z1, z2)$: sum of the points $z1$ and $z2$ on the elliptic curve corresponding to the vector E.

The library syntax is **addell**$(E, z1, z2)$.

3.5.2 ellak(E, n): computes the coefficient a_n of the L-function of the elliptic curve E, i.e. in principle coefficients of a newform of weight 2 assuming Taniyama-Weil conjecture (which is now known to hold in full generality thanks to the work of Breuil, Conrad, Diamond, Taylor and Wiles). E must be a medium or long vector of the type given 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` first before using `ellak`.

The library syntax is **akell**(E, n).

3.5.3 ellan(E, n): computes the vector of the first n a_k corresponding to the elliptic curve E. All comments in `ellak` description remain valid.

The library syntax is **anell**(E, n), where n is a C integer.

3.5.4 ellap$(E, p, \{flag = 0\})$: computes the a_p corresponding to the elliptic curve E and the prime number p. These are defined by the equation $\#E(\mathbf{F}_p) = p + 1 - a_p$, where $\#E(\mathbf{F}_p)$ stands for the number of points of the curve E over the finite field $\mathbf{F}_p$. When *flag* is 0, this uses the baby-step giant-step method and a trick due to Mestre. This runs in time $O(p^{1/4})$ and requires $O(p^{1/4})$ storage, hence becomes unreasonable when p has about 30 digits.

If *flag* is 1, computes the a_p as a sum of Legendre symbols. This is slower than the previous method as soon as p is greater than 100, say.

No checking is done that p is indeed prime. E must be a medium or long vector of the type given by `ellinit`, defined over $\mathbf{Q}$, $\mathbf{F}_p$ or $\mathbf{Q}_p$. E must be given by a Weierstrass equation minimal at p.

The library syntax is **ellap0**$(E, p, flag)$. Also available are **apell**(E, p), corresponding to $flag = 0$, and **apell2**(E, p) $(flag = 1)$.

3.5.5 ellbil$(E, z1, z2)$: if $z1$ and $z2$ are points on the elliptic curve E, this function computes the value of the canonical bilinear form on $z1$, $z2$:

$$\texttt{ellheight}(E, z1+z2) - \texttt{ellheight}(E, z1) - \texttt{ellheight}(E, z2)$$

where `+` denotes of course addition on E. In addition, $z1$ or $z2$ (but not both) can be vectors or matrices. Note that this is equal to twice some normalizations. E is assumed to be integral, given by a minimal model.

The library syntax is **bilhell**$(E, z1, z2, prec)$.

3.5.6 ellchangecurve(E, v): changes the data for the elliptic curve E by changing the coordinates using the vector `v=[u,r,s,t]`, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$. The vector E must be a medium or long vector of the type given by `ellinit`.

The library syntax is **coordch**(E, v).

3.5.7 ellchangepoint(x, v): changes the coordinates of the point or vector of points x using the vector `v=[u,r,s,t]`, i.e. if x' and y' are the new coordinates, then $x = u^2x' + r$, $y = u^3y' + su^2x' + t$ (see also `ellchangecurve`).

The library syntax is **pointch**(x, v).

3.5.8 elleisnum$(E, k, \{flag = 0\})$: E being an elliptic curve as output by `ellinit` (or, alternatively, given by a 2-component vector $[\omega_1, \omega_2]$ representing its periods), and k being an even positive integer, computes the numerical value of the Eisenstein series of weight k at E, namely

$$(2i\pi/\omega_2)^k \Big(1 + 2/\zeta(1-k) \sum_{n \geq 0} n^{k-1} q^n/(1-q^n)\Big),$$

where $q = e(\omega_1/\omega_2)$.

When *flag* is non-zero and $k = 4$ or 6, returns the elliptic invariants g_2 or g_3, such that

$$y^2 = 4x^3 - g_2x - g_3$$

is a Weierstrass equation for E.

The library syntax is **elleisnum**$(E, k, flag)$.

3.5.9 elleta(om): returns the two-component row vector $[\eta_1, \eta_2]$ of quasi-periods associated to $\texttt{om} = [\omega_1, \omega_2]$

The library syntax is **elleta**$(om, prec)$

3.5.10 ellglobalred(E): calculates the arithmetic conductor, the global minimal model of E and the global Tamagawa number c. Here E is an elliptic curve given by a medium or long vector of the type given 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 **globalreduction**(E).

3.5.11 ellheight$(E, z, \{flag = 0\})$: global N"'eron-Tate height of the point z on the elliptic curve E. The vector E must be a long vector of the type given by `ellinit`, with $flag = 1$. 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, which is much slower. E is assumed to be integral, given by a minimal model.

The library syntax is **ellheight0**$(E, z, flag, prec)$. The Archimedean contribution alone is given by the library function **hell**$(E, z, prec)$. Also available are **ghell**$(E, z, prec)$ ($flag = 0$) and **ghell2**$(E, z, prec)$ ($flag = 1$).

3.5.12 ellheightmatrix(E, x): x being a vector of points, this function outputs the Gram matrix of x with respect to the Néron-Tate height, in other words, the (i, j) component of the matrix is equal to `ellbil(`E`,x[`i`],x[`j`])`. The rank of this matrix, at least in some approximate sense, gives the rank of the set of points, and if x is a basis of the Mordell-Weil group of E, its determinant is equal to the regulator of E. Note that this matrix should be divided by 2 to be in accordance with certain normalizations. E is assumed to be integral, given by a minimal model.

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

3.5.13 ellinit$(E, \{flag = 0\})$: computes some fixed data concerning the elliptic curve given by the five-component vector E, which will be essential for most further computations on the curve. The result is a 19-component vector E (called a long vector in this section), shortened to 13 components (medium vector) if $flag = 1$. Both contain the following information in the first 13 components:

$$a_1, a_2, a_3, a_4, a_6, b_2, b_4, b_6, b_8, c_4, c_6, \Delta, j.$$

In particular, the discriminant is $E[12]$ (or E`.disc`), and the j-invariant is $E[13]$ (or E`.j`).

The other six components are only present if *flag* is 0 (or omitted!). Their content depends on whether the curve is defined over $\mathbf{R}$ or not:

• When E is defined over $\mathbf{R}$, $E[14]$ (E`.roots`) is a vector whose three components contain the roots of the right hand side of the associated Weierstrass equation.

$$(y + a_1x/2 + a_3/2)^2 = g(x)$$

If the roots are all real, then they are ordered by decreasing value. If only one is real, it is the first component.

$E[15]$ (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 $E[16]$ (E`.omega[2]`) is a complex period. In other words, $\omega_1 = E[15]$ and $\omega_2 = E[16]$ form a basis of the complex lattice defining E (E`.omega`), with $\tau = \frac{\omega_2}{\omega_1}$ having positive imaginary part.

$E[17]$ and $E[18]$ are the corresponding values η_1 and η_2 such that $\eta_1\omega_2 - \eta_2\omega_1 = i\pi$, and both can be retrieved by typing E`.eta` (as a row vector whose components are the η_i).

Finally, $E[19]$ (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 $E[14]$ (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[15] is equal to the square of the u-value, in the notation of Tate.

E[16] is the u-value itself, if it belongs to $\mathbf{Q}_p$, otherwise zero.

E[17] is the value of Tate's q for the curve E.

E`.tate` will yield the three-component vector $[u^2, u, q]$.

E[18] (E`.w`) is the value of Mestre's w (this is technical), and E[19] is arbitrarily set equal to zero.

For all other base fields or rings, the last six components are arbitrarily set equal to zero. See also the description of member functions related to elliptic curves at the beginning of this section.

The library syntax is **ellinit0**(E, *flag*, *prec*). Also available are **initell**(E, *prec*) (*flag* = 0) and **smallinitell**(E, *prec*) (*flag* = 1).

3.5.14 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.

The library syntax is **oncurve**(E, z), and the result is a `long`.

3.5.15 ellj(x): elliptic j-invariant. x must be a complex number with positive imaginary part, or convertible into a power series or a p-adic number with positive valuation.

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

3.5.16 elllocalred(E, p): calculates the Kodaira type of the local fiber of the elliptic curve E at the prime p. E must be given by a medium or long vector of the type given 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 **localreduction**(E, p).

3.5.17 elllseries($E, s, \{A = 1\}$): E being a medium or long vector given by `ellinit`, this computes the value of the L-series of E at s. In the present version 2.2.7, s must be a real number. It is assumed that E is a minimal model over $\mathbf{Z}$. 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 **lseriesell**(E, s, A, *prec*) where *prec* is a `long` and an omitted A is coded as `NULL`.

3.5.18 ellminimalmodel$(E, \{\&v\})$: return the standard minimal integral model of the rational elliptic curve E. If present, sets v to the corresponding change of variables, which is a vector $[u, r, s, t]$ with rational components. The return value is identical to that of `ellchangecurve(E, v)`.

The resulting model has integral coefficients, is everywhere minimal, a_1 is 0 or 1, a_2 is 0, 1 or -1 and a_3 is 0 or 1. Such a model is unique, and the vector v is unique if we specify that u is positive, which we do.

The library syntax is **ellminimalmodel**$(E, \&v)$, where an omitted v is coded as `NULL`.

3.5.19 ellorder(E, z): gives the order of the point z on the elliptic curve E if it is a torsion point, zero otherwise. In the present version 2.2.7, this is implemented only for elliptic curves defined over $\mathbf{Q}$.

The library syntax is **orderell**(E, z).

3.5.20 ellordinate(E, x): gives a 0, 1 or 2-component vector containing the y-coordinates of the points of the curve E having x as x-coordinate.

The library syntax is **ordell**(E, x).

3.5.21 ellpointtoz(E, z): if E is an elliptic curve with coefficients in $\mathbf{R}$, this computes a complex number t (modulo the lattice defining E) corresponding to the point z, i.e. such that, in the standard Weierstrass model, $\wp(t) = z[1], \wp'(t) = z[2]$. In other words, this is the inverse function of `ellztopoint`.

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 a long vector output by `ellinit`.

The library syntax is **zell**$(E, z, prec)$.

3.5.22 ellpow(E, z, n): computes n times the point z for the group law on the elliptic curve E. Here, n can be in $\mathbf{Z}$, or n can be a complex quadratic integer if the curve E has complex multiplication by n (if not, an error message is issued).

The library syntax is **powell**(E, z, n).

3.5.23 ellrootno$(E, \{p = 1\})$: E being a medium or long vector given by `ellinit`, this computes the local (if $p \neq 1$) or global (if $p = 1$) root number of the L-series of the elliptic curve E. Note that the global root number is the sign of the functional equation and conjecturally is the parity of the rank of the Mordell-Weil group. The equation for E must have coefficients in $\mathbf{Q}$ but need *not* be minimal.

The library syntax is **ellrootno**(E, p) and the result (equal to ± 1) is a `long`.

3.5.24 ellsigma$(E, z, \{flag = 0\})$: value of the Weierstrass σ function of the lattice associated to E as given by `ellinit` (alternatively, E can be given as a lattice $[\omega_1, \omega_2]$).

If $flag = 1$, computes an (arbitrary) determination of $\log(\sigma(z))$.

If $flag = 2, 3$, same using the product expansion instead of theta series. The library syntax is **ellsigma**$(E, z, flag)$

3.5.25 ellsub$(E, z1, z2)$: difference of the points $z1$ and $z2$ on the elliptic curve corresponding to the vector E.

The library syntax is **subell**$(E, z1, z2)$.

3.5.26 elltaniyama(E): computes the modular parametrization of the elliptic curve E, where E is given in the (long or medium) format output by `ellinit`, in the form of a two-component vector $[u, v]$ of power series, given to the current default series precision. This vector is characterized by the following two properties. First the point $(x, y) = (u, v)$ satisfies the equation of the elliptic curve. Second, the differential $du/(2v+a_1u+a_3)$ is equal to $f(z)dz$, a differential form on $H/\Gamma_0(N)$ where N is the conductor of the curve. The variable used in the power series for u and v is x, which is implicitly understood to be equal to $\exp(2i\pi z)$. It is assumed that the curve is a *strong* Weil curve, and 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 **taniyama**(E), and the precision of the result is determined by the global variable `precdl`.

3.5.27 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 a long vector 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 medium vector.

The library syntax is **elltors0**$(E, flag)$.

3.5.28 ellwp$(E, \{z = x\}, \{flag = 0\})$:

Computes the value at z of the Weierstrass $\wp$ function attached to the elliptic curve E as given by `ellinit` (alternatively, E can be given as a lattice $[\omega_1, \omega_2]$).

If z is omitted or is a simple variable, computes the *power series* expansion in z (starting $z^{-2} + O(z^2)$). The number of terms to an *even* power in the expansion is the default serieslength in GP, and the second argument (C long integer) in library mode.

Optional *flag* is (for now) only taken into account when z is numeric, and means 0: compute only $\wp(z)$, 1: compute $[\wp(z), \wp'(z)]$.

The library syntax is **ellwp0**$(E, z, flag, prec, precdl)$. Also available is **weipell**$(E, precdl)$ for the power series (in $x =$ `polx`[0]).

3.5.29 ellzeta(E, z): value of the Weierstrass ζ function of the lattice associated to E as given by `ellinit` (alternatively, E can be given as a lattice $[\omega_1, \omega_2]$).

The library syntax is **ellzeta**(E, z).

3.5.30 ellztopoint(E, z): E being a long vector, computes the coordinates $[x, y]$ on the curve E corresponding to the complex number z. Hence this is the inverse function of `ellpointtoz`. In other words, if the curve is put in Weierstrass form, $[x, y]$ represents the Weierstrass $"wp$-function and its derivative. If z is in the lattice defining E over $\mathbf{C}$, the result is the point at infinity [0].

The library syntax is **pointell**$(E, z, prec)$.

3.6 Functions related to general number fields.

In this section can be found functions which are used almost exclusively for working in general number fields. Other less specific functions can be found in the next section on polynomials. Functions related to quadratic number fields can be found in the section Section 3.4 (Arithmetic functions).

We shall use the following conventions:

• *nf* denotes a number field, i.e. a 9-component vector in the format output by `nfinit`. This contains the basic arithmetic data associated to the number field: signature, maximal order, discriminant, etc.

• *bnf* denotes a big number field, i.e. a 10-component vector in the format output by `bnfinit`. This contains *nf* and the deeper invariants of the field: units, class groups, as well as a lot of technical data necessary for some complex functions like `bnfisprincipal`.

• *bnr* denotes a big "ray number field", i.e. some data structure output by `bnrinit`, even more complicated than *bnf*, corresponding to the ray class group structure of the field, for some modulus.

• *rnf* denotes a relative number field (see below).

• *ideal* can mean any of the following:

– a $\mathbf{Z}$-basis, in Hermite normal form (HNF) or not. In this case x is a square matrix.

– an *idele*, i.e. a 2-component vector, the first being an ideal given as a $\mathbf{Z}$–basis, the second being a $r_1 + r_2$-component row vector giving the complex logarithmic Archimedean information.

– a $\mathbf{Z}_K$-generating system for an ideal.

– a *column* vector x expressing an element of the number field on the integral basis, in which case the ideal is treated as being the principal idele (or ideal) generated by x.

– a prime ideal, i.e. a 5-component vector in the format output by `idealprimedec`.

– a polmod x, i.e. an algebraic integer, in which case the ideal is treated as being the principal idele generated by x.

– an integer or a rational number, also treated as a principal idele.

• a *character* on the Abelian group $\bigoplus(\mathbf{Z}/N_i\mathbf{Z})g_i$ is given by a row vector $\chi = [a_1, \ldots, a_n]$ such that $\chi(\prod g_i^{n_i}) = exp(2i\pi \sum a_i n_i/N_i)$.

Warnings:

1) An element in *nf* can be expressed either as a polmod or as a *column* vector of components on the integral basis *nf*`.zk`. A row vector will not be recognized.

2) When giving an ideal by a $\mathbf{Z}_K$ generating system to a function expecting an ideal, it must be ensured that the function understands that it is a $\mathbf{Z}_K$-generating system and not a $\mathbf{Z}$-generating system. When the number of generators is strictly less than the degree of the field, there is no ambiguity and the program assumes that one is giving a $\mathbf{Z}_K$-generating set. When the number of generators is greater than or equal to the degree of the field, however, the program assumes on the contrary that you are giving a $\mathbf{Z}$-generating set. If this is not the case, you *must* change it into a $\mathbf{Z}$-generating set, using `idealhnf(`*nf*`,`*x*`)`.

Concerning relative extensions, some additional definitions are necessary. When defining a relative extension, the base field *nf* must be defined by a variable having a lower priority (see Section 2.6.2) than the variable defining the extension. For example, under GP you can use the variable name y (or t) to define the base field, and the variable name x to define the relative extension.

• A *relative matrix* is a matrix whose entries are elements of a (fixed) number field *nf*, always expressed as column vectors on the integral basis *nf*`.zk`. Hence it is a matrix of vectors.

• An *ideal list* is a row vector of (fractional) ideals of the number field *nf*.

• A *pseudo-matrix* is a pair (A, I) where A is a relative matrix and I an ideal list whose length is the same as the number of columns of A. This pair is represented by a 2-component row vector.

• The *module* generated by a pseudo-matrix (A, I) is the sum $\sum_i \mathbf{a}_j A_j$ where the $\mathbf{a}_j$ are the ideals of I and A_j is the j-th column of A.

• A pseudo-matrix (A, I) is a *pseudo-basis* of the module it generates if A is a square matrix with non-zero determinant and all the ideals of I are non-zero. We say that it is in Hermite Normal Form (HNF) if it is upper triangular and all the elements of the diagonal are equal to 1.

• The *determinant* of a pseudo-basis (A, I) is the ideal equal to the product of the determinant of A by all the ideals of I. The determinant of a pseudo-matrix is the determinant of any pseudo-basis of the module it generates.

Now a last set of definitions concerning the way big ray number fields (or *bnr*) are input, using class field theory. These are defined by a triple $a1$, $a2$, $a3$, where the defining set $[a1, a2, a3]$ can have any of the following forms: [*bnr*], [*bnr*, *subgroup*], [*bnf*, *module*], [*bnf*, *module*, *subgroup*], where:

• *bnf* is as output by `bnfclassunit` or `bnfinit`, where units are mandatory unless the ideal is trivial; *bnr* by `bnrclass` (with *flag* > 0) or `bnrinit`. This is the ground field.

• *module* is either an ideal in any form (see above) or a two-component row vector containing an ideal and an r_1-component row vector of flags indicating which real Archimedean embeddings to take in the module.

• *subgroup* is the HNF matrix of a subgroup of the ray class group of the ground field for the modulus *module*. 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 then the subfield of the ray class field of the ground field for the given modulus, associated to the given subgroup.

All the functions which are specific to relative extensions, number fields, big number fields, big number rays, share the prefix `rnf`, `nf`, `bnf`, `bnr` respectively. They are meant to 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). The program will make the effort of converting to what it needs. On the other hand, if the program requires a big number field, the program will *not* launch `bnfinit` for you, which is a costly operation. Instead, it will give you a specific error message.

The data types corresponding to the structures described above are rather complicated. Thus, as we already have seen it with elliptic curves, GP provides you with some "member functions" to retrieve the data you need from these structures (once they have been initialized of course). The relevant types of number fields are indicated between parentheses:

`bnf` (*bnr*, *bnf*) : big number field.
`clgp` (*bnr*, *bnf*) : classgroup. This one admits the following three subclasses:
  `cyc` : cyclic decomposition (SNF).
  `gen` : generators.
  `no` : number of elements.
`diff` (*bnr*, *bnf*, *nf*) : the different ideal.
`codiff` (*bnr*, *bnf*, *nf*) : the codifferent (inverse of the different in the ideal group).
`disc` (*bnr*, *bnf*, *nf*) : discriminant.
`fu` (*bnr*, *bnf*, *nf*) : fundamental units.
`futu` (*bnr*, *bnf*) : $[u, w]$, u is a vector of fundamental units, w generates the torsion.
`nf` (*bnr*, *bnf*, *nf*) : number field.
`reg` (*bnr*, *bnf*) : regulator.
`roots` (*bnr*, *bnf*, *nf*) : roots of the polynomial generating the field.
`sign` (*bnr*, *bnf*, *nf*) : $[r_1, r_2]$ the signature of the field. This means that the field has r_1 real embeddings, $2r_2$ complex ones.
`t2` (*bnr*, *bnf*, *nf*) : the T2 matrix (see `nfinit`).
`tu` (*bnr*, *bnf*) : a generator for the torsion units.
`tufu` (*bnr*, *bnf*) : as `futu`, but outputs $[w, u]$.
`zk` (*bnr*, *bnf*, *nf*) : integral basis, i.e. a $\mathbf{Z}$-basis of the maximal order.
`zkst` (*bnr*) : structure of $(\mathbf{Z}_K/m)^*$ (can be extracted also from an *idealstar*).

For instance, assume that $bnf = \texttt{bnfinit}(pol)$, for some polynomial. Then *bnf*`.clgp` retrieves the class group, and *bnf*`.clgp.no` the class number. If we had set $bnf = \texttt{nfinit}(pol)$, 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`.

Some of the functions starting with `bnf` are implementations of the sub-exponential algorithms for finding class and unit groups under GRH, due to Hafner-McCurley, Buchmann and Cohen-Diaz-Olivier. The general call to the functions concerning class groups of general number fields (i.e. excluding `quadclassunit`) involves a polynomial P and a technical vector

$$tech = [c, c2, nrpid],$$

where the parameters are to be understood as follows:

P is the defining polynomial for the number field, which must be in $\mathbf{Z}[X]$, irreducible and, preferably, monic. In fact, if you supply a non-monic polynomial at this point, GP will issue a warning, then *transform your polynomial* so that it becomes monic. Instead of the normal result, say `res`, you then get a vector `[res,Mod(a,Q)]`, where `Mod(a,Q)=Mod(X,P)` gives the change of variables.

The numbers c and $c2$ are positive real numbers which control the execution time and the stack size. To get maximum speed, set $c2 = c$. To get a rigorous result (under GRH) you must take $c2 = 12$ (or $c2 = 6$ in the quadratic case, but then you should use the much faster function `quadclassunit`). Reasonable values for c are between 0.1 and 2. (The defaults are $c = c2 = 0.3$.)

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).

Remarks.

Apart from the polynomial P, you don't need to supply any of the technical parameters (under the library you still need to send at least an empty vector, `cgetg(1,t_VEC)`). 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 $nrel$, you must give some values as well for c and $c2$, and provide a vector $[c, c2, nrel]$.

Note also that you can use an *nf* instead of P, which avoids recomputing the integral basis and analogous quantities.

3.6.1 bnfcertify(bnf): *bnf* being a big number field as output by `bnfinit` or `bnfclassunit`, checks whether the result is correct, i.e. whether it is possible to remove the assumption of the Generalized Riemann Hypothesis. If it is correct, the answer is 1. If not, the program may output some error message, but more probably will loop indefinitely. In *no* occasion can the program give a wrong answer (barring bugs of course): if the program answers 1, the answer is certified.

The library syntax is **certifybuchall**(bnf), and the result is a C long.

3.6.2 bnfclassunit($P, \{flag = 0\}, \{tech = []\}$): Buchmann's sub-exponential algorithm for computing the class group, the regulator and a system of fundamental units of the general algebraic number field K defined by the irreducible polynomial P with integer coefficients.

The result of this function is a vector v with many components (it is *not* a *bnf*, you need `bnfinit` for that), which for ease of presentation is in fact output as a one column matrix. First we describe the default behaviour ($flag = 0$):

$v[1]$ is equal to the polynomial P. Note that for optimum performance, P should have gone through `polred` or `nfinit`$(x, 2)$.

$v[2]$ is the 2-component vector $[r1, r2]$, where $r1$ and $r2$ are as usual the number of real and half the number of complex embeddings of the number field K.

$v[3]$ is the 2-component vector containing the field discriminant and the index.

$v[4]$ is an integral basis in Hermite normal form.

$v[5]$ (v`.clgp`) is a 3-component vector containing the class number (v`.clgp.no`), the structure of the class group as a product of cyclic groups of order n_i (v`.clgp.cyc`), and the corresponding generators of the class group of respective orders n_i (v`.clgp.gen`).

$v[6]$ (v`.reg`) is the regulator computed to an accuracy which is the maximum of an internally determined accuracy and of the default.

$v[7]$ is deprecated, maintained for backward compatibility and always equal to 1.

$v[8]$ (v`.tu`) a vector with 2 components, the first being the number w of roots of unity in K and the second a primitive w-th root of unity expressed as a polynomial.

$v[9]$ (v`.fu`) is a system of fundamental units also expressed as polynomials.

If *flag* $= 1$, and the precision happens to be insufficient for obtaining the fundamental units exactly, the internal precision is doubled and the computation redone, until the exact results are obtained. The user should be warned that this can take a very long time when the coefficients of the fundamental units on the integral basis are very large, for example in the case of large real quadratic fields. In that case, there are alternate methods for representing algebraic numbers which are not implemented in PARI.

If *flag* $= 2$, the fundamental units and roots of unity are not computed. Hence the result has only 7 components, the first seven ones.

tech is a technical vector (empty by default) containing c, $c2$, *nrel*, *borne*, *nbpid*, *minsfb*, in this order (see the beginning of the section or the keyword `bnf`). You can supply any number of these *provided you give an actual value to each of them* (the "empty arg" trick won't work here). Careful use of these parameters may speed up your computations considerably.

The library syntax is **bnfclassunit0**$(P, \mathit{flag}, \mathit{tech}, \mathit{prec})$.

3.6.3 bnfclgp$(P, \{\mathit{tech} = [\,]\})$: as `bnfclassunit`, but only outputs $v[5]$, i.e. the class group.

The library syntax is **bnfclassgrouponly**$(P, \mathit{tech}, \mathit{prec})$, where *tech* is as described under `bnfclassunit`.

3.6.4 bnfdecodemodule(nf, m): if m is a module as output in the first component of an extension given by `bnrdisclist`, outputs the true module.

The library syntax is **decodemodule**(nf, m).

3.6.5 bnfinit$(P, \{\mathit{flag} = 0\}, \{\mathit{tech} = [\,]\})$: essentially identical to `bnfclassunit` except that the output contains a lot of technical data, and should not be printed out explicitly in general. The result of `bnfinit` is used in programs such as `bnfisprincipal`, `bnfisunit` or `bnfnarrow`. The result is a 10-component vector *bnf*.

• The first 6 and last 2 components are technical and in principle are not used by the casual user. However, for the sake of completeness, their description is as follows. We use the notations explained in the book by H. Cohen, *A Course in Computational Algebraic Number Theory*, Graduate Texts in Maths **138**, Springer-Verlag, 1993, Section 6.5, and subsection 6.5.5 in particular.

bnf$[1]$ contains the matrix W, i.e. the matrix in Hermite normal form giving relations for the class group on prime ideal generators $(\wp_i)_{1 \le i \le r}$.

bnf$[2]$ contains the matrix B, i.e. the matrix containing the expressions of the prime ideal factorbase in terms of the $\wp_i$. It is an $r \times c$ matrix.

bnf$[3]$ contains the complex logarithmic embeddings of the system of fundamental units which has been found. It is an $(r_1 + r_2) \times (r_1 + r_2 - 1)$ matrix.

bnf[4] contains the matrix M''_C of Archimedean components of the relations of the matrix $(W|B)$.

bnf[5] contains the prime factor base, i.e. the list of prime ideals used in finding the relations.

bnf[6] used to contain a permutation of the prime factor base, but has been obsoleted. It contains a dummy 0.

bnf[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] $= [U_r, g_a, GD_a]$.

Finally, *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 less technical components are as follows:

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 last 6 components of `bnfclassunit[,1]`, i.e. the classgroup *bnf*`.clgp`, the regulator *bnf*`.reg`, the general "check" number which should be close to 1, the number of roots of unity and a generator *bnf*`.tu`, the fundamental units *bnf*`.fu`, and finally the check on their computation. If the precision becomes insufficient, GP outputs a warning (`fundamental units too large, not given`) and does not strive to compute the units by default (*flag* = 0).

When *flag* = 1, GP insists on finding the fundamental units exactly, the internal precision being doubled and the computation redone, until the exact results are obtained. The user should be warned that this can take a very long time when the coefficients of the fundamental units on the integral basis are very large.

When *flag* = 2, on the contrary, it is initially agreed that GP will not compute units. Note that the resulting *bnf* will not be suitable for `bnrinit`, and that this flag provides negligible time savings. In short, do not use it without a very good reason.

When *flag* = 3, computes a very small version of `bnfinit`, a "small big number field" (or *sbnf* for short) which contains enough information to recover the full *bnf* vector very rapidly, but which is much smaller and hence easy to store and print. It is supposed to be used in conjunction with `bnfmake`. The output is a 12 component vector v, as follows. Let *bnf* be the result of a full `bnfinit`, complete with units. Then $v[1]$ is the polynomial P, $v[2]$ is the number of real embeddings r_1, $v[3]$ is the field discriminant, $v[4]$ is the integral basis, $v[5]$ is the list of roots as in the sixth component of `nfinit`, $v[6]$ is the matrix MD of `nfinit` giving a $\mathbf{Z}$-basis of the different, $v[7]$ is the matrix `W` = *bnf*[1], $v[8]$ is the matrix `matalpha` = *bnf*[2], $v[9]$ is the prime ideal factor base *bnf*[5] coded in a compact way, and ordered according to the permutation *bnf*[6], $v[10]$ is the 2-component vector giving the number of roots of unity and a generator, expressed on the integral basis, $v[11]$ is the list of fundamental units, expressed on the integral basis, $v[12]$ is a vector containing the algebraic numbers alpha corresponding to the columns of the matrix `matalpha`, expressed on the integral basis.

Note that all the components are exact (integral or rational), except for the roots in $v[5]$. In practice, this is the only component which a user is allowed to modify, by recomputing the roots to a higher accuracy if desired. Note also that the member functions will *not* work on *sbnf*, you have to use `bnfmake` explicitly first.

The library syntax is **bnfinit0**($P, flag, tech, prec$).

3.6.6 bnfisintnorm(bnf, x): computes a complete system of solutions (modulo units of positive norm) of the absolute norm equation $\mathrm{Norm}(a) = x$, where a is an integer in *bnf*. If *bnf* has not been certified, the correctness of the result depends on the validity of GRH.

See also `bnfisnorm`.

The library syntax is **bnfisintnorm**(bnf, x).

3.6.7 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 $12\log(disc(Bnf))^2$, where *Bnf* is the Galois closure of *bnf*.

See also `bnfisintnorm`.

The library syntax is **bnfisnorm**($bnf, x, flag, prec$), where *flag* and *prec* are `long`s.

3.6.8 bnfissunit(bnf, sfu, x): *bnf* being output by `bnfinit`, *sfu* by `bnfsunit`, gives the column vector of exponents of x on the fundamental S-units and the roots of unity. If x is not a unit, outputs an empty vector.

The library syntax is **bnfissunit**(bnf, sfu, x).

3.6.9 bnfisprincipal($bnf, x, \{flag = 1\}$): *bnf* being the number field data output by `bnfinit`, and x being either a $\mathbf{Z}$-basis of an ideal in the number field (not necessarily in HNF) or a prime ideal in the format output by the function `idealprimedec`, this function tests whether the ideal is principal or not. The result is more complete than a simple true/false answer: it gives a row vector $[v_1, v_2]$, where

v_1 is the vector of components c_i of the class of the ideal x in the class group, expressed on the generators g_i given by `bnfinit` (specifically *bnf*`.gen`). The c_i are chosen so that $0 \le c_i < n_i$ where n_i is the order of g_i (the vector of n_i being *bnf*`.cyc`).

v_2 gives on the integral basis the components of α such that $x = \alpha \prod_i g_i^{c_i}$. In particular, x is principal if and only if v_1 is equal to the zero vector. In the latter case, $x = \alpha\mathbf{Z}_K$ where α is given by v_2. Note that if α is too large to be given, a warning message will be printed and v_2 will be set equal to the empty vector.

If $flag = 0$, outputs only v_1, which is much easier to compute.

If $flag = 2$, does as if *flag* were 0, but doubles the precision until a result is obtained.

If $flag = 3$, as in the default behaviour ($flag = 1$), but doubles the precision until a result is obtained.

The user is warned that these two last setting may induce *very* lengthy computations.

The library syntax is **isprincipalall**(*bnf*, *x*, *flag*).

3.6.10 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 (found by `bnfclassunit` or `bnfinit`), 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 $i \leq r$ and is an integer modulo the order of ζ for $i = r+1$.

The library syntax is **isunit**(*bnf*, *x*).

3.6.11 bnfmake(*sbnf*): *sbnf* being a "small *bnf*" as output by `bnfinit`$(x,3)$, computes the complete `bnfinit` information. The result is *not* identical to what `bnfinit` would yield, but is functionally identical. The execution time is very small compared to a complete `bnfinit`. Note that if the default precision in GP (or *prec* in library mode) is greater than the precision of the roots *sbnf*[5], these are recomputed so as to get a result with greater accuracy.

Note that the member functions are *not* available for *sbnf*, you have to use `bnfmake` explicitly first.

The library syntax is **makebigbnf**(*sbnf*, *prec*), where *prec* is a C long integer.

3.6.12 bnfnarrow(*bnf*): *bnf* being a big number field 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 `bnrclass`.

The library syntax is **buchnarrow**(*bnf*).

3.6.13 bnfsignunit(*bnf*): *bnf* being a big number field output by `bnfinit`, this computes an $r_1 \times (r_1 + r_2 - 1)$ matrix having ± 1 components, giving the signs of the real embeddings of the fundamental units. The following functions compute generators for the totally positive units:

```
/* exponents of totally positive units generators on bnf.tufu */
tpuexpo(bnf)=
{ local(S,d,K);

  S = bnfsignunit(bnf); d = matsize(S);
  S = matrix(d[1],d[2], i,j, if (S[i,j] < 0, 1,0));
  S = concat(S, vectorv(d[1],i,1));   \\ sign(-1)
  K = lift(matker(S * Mod(1,2)));
  if (K, mathnfmodid(K, 2), 2*matid(d[1]))
}

/* totally positive units */
tpu(bnf)=
{ local(vu, ex = tpuexpo(bnf));

  vu = nfbasistoalg(bnf, bnf.tufu);
  vector(length(ex)-1, i, factorback([vu, ex[,i+1]]))  \\ ex[,1] is 1
```

`}`

The library syntax is **signunits**(*bnf*).

3.6.14 bnfreg(*bnf*): *bnf* being a big number field output by `bnfinit`, computes its regulator.

The library syntax is **regulator**(*bnf*, *tech*, *prec*), where *tech* is as in `bnfclassunit`.

3.6.15 bnfsunit(*bnf*, *S*): computes the fundamental S-units of the number field *bnf* (output by `bnfinit`), where S is a list of prime ideals (output by `idealprimedec`). The output is a vector v with 6 components.

$v[1]$ gives a minimal system of (integral) generators of the S-unit group modulo the unit group.

$v[2]$ contains technical data needed by `bnfissunit`.

$v[3]$ is an empty vector (used to give the logarithmic embeddings of the generators in $v[1]$ in version 2.0.16).

$v[4]$ is the S-regulator (this is the product of the regulator, the determinant of $v[2]$ and the natural logarithms of the norms of the ideals in S).

$v[5]$ gives the S-class group structure, in the usual format (a row vector whose three components give in order the S-class number, the cyclic components and the generators).

$v[6]$ is a copy of S.

The library syntax is **bnfsunit**(*bnf*, *S*, *prec*).

3.6.16 bnfunit(*bnf*): *bnf* being a big number field as output by `bnfinit`, outputs the vector of fundamental units of the number field.

This function is mostly useless, since it will only succeed if *bnf* contains the units, in which case `bnf.fu` is recommanded instead, or *bnf* was produced with `bnfinit(,,2)`, which is itself deprecated.

The library syntax is **buchfu**(*bnf*).

3.6.17 bnrL1(*bnr*, {*subgroup*}, {*flag* = 0}): *bnr* being the number field data which is output by `bnrinit(,,1)` and *subgroup* being a square matrix defining a congruence subgroup of the ray class group corresponding to *bnr* (the trivial congruence subgroup if omitted), returns for each character χ of the ray class group which is trivial on this subgroup, the value at $s = 1$ (or $s = 0$) of the abelian L-function associated to χ. For the value at $s = 0$, the function returns in fact for each character χ a vector $[r_\chi, c_\chi]$ where r_χ is the order of $L(s, \chi)$ at $s = 0$ and c_χ the first non-zero term in the expansion of $L(s, \chi)$ at $s = 0$; in other words

$$L(s, \chi) = c_\chi \cdot s^{r_\chi} + O(s^{r_\chi+1})$$

near 0. *flag* is optional, default value is 0; its binary digits mean 1: compute at $s = 1$ if set to 1 or $s = 0$ if set to 0, 2: compute the primitive L-functions associated to χ if set to 0 or the L-function with Euler factors at prime ideals dividing the modulus of *bnr* removed if set to 1 (this is the so-called $L_S(s, \chi)$ function where S is the set of infinite places of the number field together with the finite prime ideals dividing the modulus of *bnr*, see the example below), 3: returns also the character.

Example:

```
bnf = bnfinit(x^2 - 229);
bnr = bnrinit(bnf,1,1);
bnrL1(bnr)
```

returns the order and the first non-zero term of the abelian L-functions $L(s,\chi)$ at $s=0$ where χ runs through the characters of the class group of $\mathbf{Q}(\sqrt{229})$. Then

```
bnr2 = bnrinit(bnf,2,1);
bnrL1(bnr2,,2)
```

returns the order and the first non-zero terms of the abelian L-functions $L_S(s,\chi)$ at $s=0$ where χ runs through the characters of the class group of $\mathbf{Q}(\sqrt{229})$ and S is the set of infinite places of $\mathbf{Q}(\sqrt{229})$ together with the finite prime 2 (note that the ray class group modulo 2 is in fact the class group, so `bnrL1(bnr2,0)` returns exactly the same answer as `bnrL1(bnr,0)`!).

The library syntax is **bnrL1**(*bnr*, *subgroup*, *flag*, *prec*), where an omitted *subgroup* is coded as `NULL`.

3.6.18 bnrclass(*bnf*, *ideal*, {*flag* = 0}): *bnf* being a big number field as output by `bnfinit` (the units are mandatory unless the ideal is trivial), and *ideal* being either an ideal in any form or a two-component row vector containing an ideal and an r_1-component row vector of flags indicating which real Archimedean embeddings to take in the module, computes the ray class group of the number field for the module *ideal*, as a 3-component vector as all other finite Abelian groups (cardinality, vector of cyclic components, corresponding generators).

If *flag* = 2, the output is different. It is a 6-component vector w. $w[1]$ is *bnf*. $w[2]$ is the result of applying `idealstar`(*bnf*, I, 2). $w[3]$, $w[4]$ and $w[6]$ are technical components used only by the function `bnrisprincipal`. $w[5]$ is the structure of the ray class group as would have been output with *flag* = 0.

If *flag* = 1, as above, except that the generators of the ray class group are not computed, which saves time.

The library syntax is **bnrclass0**(*bnf*, *ideal*, *flag*).

3.6.19 bnrclassno(*bnf*, I): *bnf* being a big number field as output by `bnfinit` (units are mandatory unless the ideal is trivial), and I being either an ideal in any form or a two-component row vector containing an ideal and an r_1-component row vector of flags indicating which real Archimedean embeddings to take in the modulus, computes the ray class number of the number field for the modulus I. This is faster than `bnrclass` and should be used if only the ray class number is desired.

The library syntax is **rayclassno**(*bnf*, I).

3.6.20 bnrclassnolist(*bnf*, *list*): *bnf* being a big number field as output by `bnfinit` (units are mandatory unless the ideal is trivial), and *list* being a list of modules as output by `ideallist` of `ideallistarch`, outputs the list of the class numbers of the corresponding ray class groups.

The library syntax is **rayclassnolist**(*bnf*, *list*).

3.6.21 bnrconductor$(a_1, \{a_2\}, \{a_3\}, \{flag = 0\})$: conductor f of the subfield of a ray class field as defined by $[a_1, a_2, a_3]$ (see `bnr` at the beginning of this section).

If $flag = 0$, returns f.

If $flag = 1$, returns $[f, Cl_f, H]$, where Cl_f is the ray class group modulo f, as output by `bnrclass`$(, f, 1)$, and H is the subgroup of Cl_f defining the extension. If a_1 is a `bnr`, the algorithm requires that it has been computed by `bnrinit`$(, , 1)$, i.e. with generators.

If $flag = 2$, returns $[f, bnr(f), H]$, as above except Cl_f is replaced by a `bnr` structure, as output by `bnrinit`$(, f, 1)$.

The library syntax is **conductor**$(bnr, subgroup, flag)$, where an omitted subgroup (trivial subgroup, i.e. ray class field) is input as `NULL`, and *flag* is a C long.

3.6.22 bnrconductorofchar(bnr, chi): *bnr* being a big ray number field as output by `bnrclass`, and *chi* being a row vector representing a character as expressed on the generators of the ray class group, gives the conductor of this character as a modulus.

The library syntax is **bnrconductorofchar**(bnr, chi).

3.6.23 bnrdisc$(a1, \{a2\}, \{a3\}, \{flag = 0\})$: $a1$, $a2$, $a3$ defining a big ray number field L over a ground field K (see `bnr` at the beginning of this section for the meaning of $a1$, $a2$, $a3$), outputs a 3-component row vector $[N, R_1, D]$, where N is the (absolute) degree of L, R_1 the number of real places of L, and D the discriminant of $L/\mathbf{Q}$, including sign (if $flag = 0$).

If $flag = 1$, as above but outputs relative data. N is now the degree of L/K, R_1 is the number of real places of K unramified in L (so that the number of real places of L is equal to R_1 times the relative degree N), and D is the relative discriminant ideal of L/K.

If $flag = 2$, does as in case 0, except that if the modulus is not the exact conductor corresponding to the L, no data is computed and the result is 0 (`gzero`).

If $flag = 3$, as case 2, outputting relative data.

The library syntax is **bnrdisc0**$(a1, a2, a3, flag)$.

3.6.24 bnrdisclist$(bnf, bound, \{arch\}, \{flag = 0\})$: *bnf* being a big number field as output by `bnfinit` (the units are mandatory), computes a list of discriminants of Abelian extensions of the number field by increasing modulus norm up to bound *bound*, where the ramified Archimedean places are given by *arch* (unramified at infinity if *arch* is void or omitted). If *flag* is non-zero, give *arch* all the possible values. (See `bnr` at the beginning of this section for the meaning of $a1$, $a2$, $a3$.)

The alternative syntax `bnrdisclist`$(bnf, list)$ is supported, where *list* is as output by `ideallist` or `ideallistarch` (with units).

The output format is as follows. The output v is a row vector of row vectors, allowing the bound to be greater than 2^{16} for 32-bit machines, and $v[i][j]$ is understood to be in fact $V[2^{15}(i-1)+j]$ of a unique big vector V (note that 2^{15} is hardwired and can be increased in the source code only on 64-bit machines and higher).

Such a component $V[k]$ is itself a vector W (maybe of length 0) whose components correspond to each possible ideal of norm k. Each component $W[i]$ corresponds to an Abelian extension L of *bnf* whose modulus is an ideal of norm k and no Archimedean components (hence the extension is

unramified at infinity). The extension $W[i]$ is represented by a 4-component row 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 the absolute discriminant. Each prime ideal $pr = [p, \alpha, e, f, \beta]$ in the prime factorization m is coded as $p \cdot n^2 + (f-1) \cdot n + (j-1)$, 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`.

The library syntax is **bnrdisclist0**($a1, a2, a3$, *bound*, *arch*, *flag*).

3.6.25 bnrinit(*bnf*, *ideal*, {*flag* = 0}): *bnf* is as output by `bnfinit`, *ideal* is a valid ideal (or a module), initializes data linked to the ray class group structure corresponding to this module. This is the same as `bnrclass`(*bnf*, *ideal*, *flag* + 1).

The library syntax is **bnrinit0**(*bnf*, *ideal*, *flag*).

3.6.26 bnrisconductor($a1$, {$a2$}, {$a3$}): $a1$, $a2$, $a3$ represent an extension of the base field, given by class field theory for some modulus encoded in the parameters. Outputs 1 if this modulus is the conductor, and 0 otherwise. This is slightly faster than `bnrconductor`.

The library syntax is **bnrisconductor**($a1, a2, a3$) and the result is a `long`.

3.6.27 bnrisprincipal(*bnr*, x, {*flag* = 1}): *bnr* being the number field data which is output by `bnrinit`(, , 1) and x being an ideal in any form, outputs the components of x on the ray class group generators in a way similar to `bnfisprincipal`. That is a 2-component vector v where $v[1]$ is the vector of components of x on the ray class group generators, $v[2]$ gives on the integral basis an element α such that $x = \alpha \prod_i g_i^{x_i}$.

If *flag* = 0, outputs only v_1. In that case, *bnr* need not contain the ray class group generators, i.e. it may be created with `bnrinit`(, , 0)

The library syntax is **isprincipalrayall**(*bnr*, x, *flag*).

3.6.28 bnrrootnumber(*bnr*, *chi*, {*flag* = 0}): if $\chi =$ *chi* is a (not necessarily primitive) character over *bnr*, let $L(s, \chi) = \sum_{id} \chi(id) N(id)^{-s}$ be the associated Artin L-function. Returns the so-called Artin root number, i.e. the complex number $W(\chi)$ of modulus 1 such that

$$\Lambda(1-s, \chi) = W(\chi)\Lambda(s, \overline{\chi})$$

where $\Lambda(s, \chi) = A(\chi)^{s/2}\gamma_\chi(s)L(s, \chi)$ is the enlarged L-function associated to L.

The generators of the ray class group are needed, and you can set *flag* = 1 if the character is known to be primitive. Example:

```
bnf = bnfinit(x^2 - 145);
bnr = bnrinit(bnf,7,1);
bnrrootnumber(bnr, [5])
```

returns the root number of the character χ of $Cl_7(\mathbf{Q}(\sqrt{145}))$ such that $\chi(g) = \zeta^5$, where g is the generator of the ray-class field and $\zeta = e^{2i\pi/N}$ where N is the order of g ($N = 12$ as `bnr.cyc` readily tells us).

The library syntax is **bnrrootnumber**(*bnf*, *chi*, *flag*)

3.6.29 bnrstark$(bnr, \{subgroup\}, \{flag = 0\})$: *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 using Stark units (omit *subgroup* $= 0$ 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.
flag

is optional and may be set to 0 (default) to obtain a reduced relative polynomial, 1 to be satisfied with any relative polynomial, 2 to obtain an absolute polynomial and 3 to obtain the irreducible relative polynomial of the Stark unit. Example:

```
bnf = bnfinit(y^2 - 3);
bnr = bnrinit(bnf, 5, 1);
bnrstark(bnr, 0)
```

returns the ray class field of $\mathbf{Q}(\sqrt{3})$ modulo 5.

Remark. The result of the computation depends on the choice of a modulus verifying special conditions. By default the function will try few moduli, choosing the one giving the smallest result. In some cases where the result is however very large, you can tell the function to try more moduli by adding 4 to the value of flag. Whether this flag is set or not, the function may fail in some extreme cases, returning the error message

`"Cannot find a suitable modulus in FindModule"`.

In this case, the corresponding congruence group is a product of cyclic groups and, for the time being, the class field has to be obtained by splitting this group into its cyclic components.

The library syntax is **bnrstark**$(bnr, subgroup, flag)$, where an omitted *subgroup* is coded by `NULL`.

3.6.30 dirzetak(nf, b): gives as a vector the first b coefficients of the Dedekind zeta function of the number field *nf* considered as a Dirichlet series.

The library syntax is **dirzetak**(nf, b).

3.6.31 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 2.6.2). 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. For example

```
? factornf(x^2 + Mod(y, y^2+1), y^2+1);
? factornf(x^2 + 1, y^2+1); \\ these two are OK
? factornf(x^2 + Mod(z,z^2+1), y^2+1)
  ***   incorrect type in gmulsg
```

The library syntax is **polfnf**(x, t).

3.6.32 galoisexport($gal, \{flag = 0\}$): *gal* being be a Galois field as output by `galoisinit`, export the underlying permutation group as a string suitable for (no flags or $flag = 0$) GAP or ($flag = 1$) Magma.

The following example compute the index of the underlying abstract group in the GAP library:

```
? G=galoisinit(x^6+108);
? s=galoisexport(G)
%2 = "Group((1, 2, 3)(4, 5, 6), (1, 4)(2, 6)(3, 5))"
? extern("echo \"IdGroup("s");\"| gap -q")
%3 = [6, 1]
? galoisidentify(G)
%4 = [6, 1]
```

The library syntax is **galoisexport**($gal, flag$).

3.6.33 galoisfixedfield($gal, perm, \{flag = 0\}, \{v = y\}$)): *gal* being be a Galois field as output by `galoisinit` and *perm* an element of *gal.group* or a vector of such elements, computes the fixed field of *gal* by the automorphism defined by the permutations *perm* of the roots *gal.roots*. P is guaranteed to be squarefree modulo *gal.p*.

If no flags or $flag = 0$, output format is the same as for `nfsubfield`, returning $[P, x]$ such that P is a polynomial defining the fixed field, and x is a root of P expressed as a polmod in *gal.pol*.

If $flag = 1$ return only the polynomial P.

If $flag = 2$ return $[P, x, F]$ where P and x are as above and F is the factorization of *gal.pol* over the field defined by P, where variable v (y by default) stands for a root of P. The priority of v must be less than the priority of the variable of *gal.pol* (see Section 2.6.2). Example:

```
G = galoisinit(x^4+1);
galoisfixedfield(G,G.group[2],2)
  [x^2 + 2, Mod(x^3 + x, x^4 + 1), [x^2 - y*x - 1, x^2 + y*x - 1]]
```

computes the factorization $x^4 + 1 = (x^2 - \sqrt{-2}x - 1)(x^2 + \sqrt{-2}x - 1)$

The library syntax is **galoisfixedfield**($gal, perm, flag$,v), where v is a variable number, an omitted v being coded by -1.

3.6.34 galoisidentify(gal): *gal* being be a Galois field as output by `galoisinit`, output the isomorphism class of the underlying abstract group as a two-components vector $[o, i]$, where o is the group order, and i is the group index in the GAP4 Small Group library, by Hans Ulrich Besche, Bettina Eick and Eamonn O'Brien.

The current implementation is limited to degree less or equal to 127. Some larger "easy" orders are also supported.

The output is similar to the output of the function `IdGroup` in GAP4. Note that GAP4 `IdGroup` handles all groups of order less than 2000 except 1024, so you can use `galoisexport` and GAP4 to identify large Galois groups.

The library syntax is **galoisidentify**(gal).

3.6.35 galoisinit$(pol, \{den\})$: computes the Galois group and all necessary information for computing the fixed fields of the Galois extension $K/\mathbf{Q}$ where K is the number field defined by *pol* (monic irreducible polynomial in $\mathbf{Z}[X]$ or a number field as output by `nfinit`). The extension $K/\mathbf{Q}$ must be Galois with Galois group "weakly" super-solvable (see `nfgaloisconj`)

The output is an 8-component vector *gal*.

gal[1] contains the polynomial *pol* (*gal*`.pol`).

gal[2] is a three–components vector $[p, e, q]$ where p is a prime number (*gal*`.p`) such that *pol* totally split modulo p , e is an integer and $q = p^e$ (*gal*`.mod`) is the modulus of the roots in *gal*`.roots`.

gal[3] is a vector L containing the p-adic roots of *pol* as integers implicitly modulo *gal*`.mod`. (*gal*`.roots`).

gal[4] is the inverse of the Van der Monde matrix of the p-adic roots of *pol*, multiplied by *gal*[5].

gal[5] is a multiple of the least common denominator of the automorphisms expressed as polynomial in a root of *pol*.

gal[6] is the Galois group G expressed as a vector of permutations of L (*gal*`.group`).

gal[7] is a generating subset $S = [s_1, \ldots, s_g]$ of G expressed as a vector of permutations of L (*gal*`.gen`).

gal[8] contains the relative orders $[o_1, \ldots, o_g]$ of the generators of S (*gal*`.orders`).

Let H be the maximal normal supersolvable subgroup of G, we have the following properties:

- if $G/H \simeq A_4$ then $[o_1, \ldots, o_g]$ ends by $[2, 2, 3]$.
- if $G/H \simeq S_4$ then $[o_1, \ldots, o_g]$ ends by $[2, 2, 3, 2]$.
- else G is super-solvable.
- for $1 \leq i \leq g$ the subgroup of G generated by $[s_1, \ldots, s_g]$ is normal, with the exception of $i = g - 2$ in the second case and of $i = g - 3$ in the third.
- the relative order o_i of s_i is its order in the quotient group $G/\langle s_1, \ldots, s_{i-1}\rangle$, 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 **galoisinit**(gal, den).

3.6.36 galoisisabelian$(gal, fl = 0)$: *gal* being as output by `galoisinit`, return 0 if *gal* is not an abelian group, and the HNF matrix of *gal* over `gal.gen` if $fl = 0$, 1 if $fl = 1$.

The library syntax is **galoisisabelian**(gal, fl) where *fl* is a C long integer.

3.6.37 galoispermtopol$(gal, perm)$: *gal* being a Galois field as output by `galoisinit` and *perm* a element of *gal.group*, return the polynomial defining the Galois automorphism, as output by `nfgaloisconj`, associated with the permutation *perm* of the roots *gal.roots*. *perm* can also be a vector or matrix, in this case, `galoispermtopol` is applied to all components recursively.

Note that

```
G = galoisinit(pol);
galoispermtopol(G, G[6])~
```

is equivalent to `nfgaloisconj(pol)`, if degree of *pol* is greater or equal to 2.

The library syntax is **galoispermtopol**$(gal, perm)$.

3.6.38 galoissubcyclo$(N, H, \{fl = 0\}, \{v\})$: computes the subextension of $\mathbf{Q}(\zeta_n)$ fixed by the subgroup $H \subset (\mathbf{Z}/n\mathbf{Z})^*$. By the Kronecker-Weber theorem, all abelian number fields can be generated in this way (uniquely if n is taken to be minimal).

The pair (n, H) is deduced from the parameters (N, H) as follows

• N an integer: then $n = N$; H is a generator, i.e. an integer or an integer modulo n; or a vector of generators.

• N the output of `znstar(`n`)`. H as in the first case above, or a matrix, taken to be a HNF left divisor of the SNF for $(\mathbf{Z}/n\mathbf{Z})^*$ (of type N`.cyc`), giving the generators of H in terms of N`.gen`.

• N the output of `bnrinit(bnfinit(y),` m`, 1)` where m is a module. H as in the first case, or a matrix taken to be a HNF left divisor of the SNF for the ray class group modulo m (of type N`.cyc`), giving the generators of H in terms of N`.gen`.

In this last case, beware that H is understood relatively to N; in particular, if the infinite place does not divide the module, e.g if m is an integer, then it is not a subgroup of $(\mathbf{Z}/n\mathbf{Z})^*$, but of its quotient by $\{\pm 1\}$.

If $fl = 0$, compute a polynomial (in the variable v) defining the the subfield of $\mathbf{Q}(\zeta_n)$ fixed by the subgroup H of $(\mathbf{Z}/n\mathbf{Z})^*$.

If $fl = 1$, compute only the conductor of the abelian extension, as a module.

If $fl = 2$, output $[pol, N]$, where *pol* is the polynomial as output when $fl = 0$ and N the conductor as output when $fl = 1$.

The following function can be used to compute all subfields of $\mathbf{Q}(\zeta_n)$ (of exact degree `d`, if `d` is set):

```
subcyclo(n, d = -1)=
{
  local(bnr,L,IndexBound);
  IndexBound = if (d < 0, n, [d]);
  bnr = bnrinit(bnfinit(y), [n,[1]], 1);
  L = subgrouplist(bnr, IndexBound, 1);
  vector(#L,i, galoissubcyclo(bnr,L[i]));
}
```

Setting `L = subgrouplist(bnr, [d])` would produce subfields of exact conductor $n\infty$.

The library syntax is **galoissubcyclo**(N, H, fl, v) where *fl* is a C long integer, and v a variable number.

3.6.39 galoissubfields$(G, \{fl = 0\}, \{v\})$: Output all the subfields of the Galois group G, as a vector. This works by applying `galoisfixedfield` to all subgroups. The meaning of the flag *fl* is the same as for `galoisfixedfield`.

The library syntax is **galoissubfields**(G, fl, v), where *fl* is a long and v a variable number.

3.6.40 galoissubgroups(G): Output all the subgroups of the Galois group `G`. 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.

The library syntax is **galoissubgroups**(G).

3.6.41 idealadd(nf, x, y): sum of the two ideals x and y in the number field nf. When x and y are given by **Z**-bases, this does not depend on nf and can be used to compute the sum of any two **Z**-modules. The result is given in HNF.

The library syntax is **idealadd**(nf, x, y).

3.6.42 idealaddtoone$(nf, x, \{y\})$: x and y being two co-prime integral ideals (given in any form), this gives a two-component row vector $[a, b]$ such that $a \in x$, $b \in y$ and $a + b = 1$.

The alternative syntax `idealaddtoone`(nf, v), is supported, where v is a k-component vector of ideals (given in any form) which sum to $\mathbf{Z}_K$. This outputs a k-component vector e such that $e[i] \in x[i]$ for $1 \le i \le k$ and $\sum_{1 \le i \le k} e[i] = 1$.

The library syntax is **idealaddtoone0**(nf, x, y), where an omitted y is coded as `NULL`.

3.6.43 idealappr$(nf, x, \{flag = 0\})$: if x is a fractional ideal (given in any form), gives an element α in nf such that for all prime ideals $\wp$ such that the valuation of x at $\wp$ is non-zero, we have $v_\wp(\alpha) = v_\wp(x)$, and. $v_\wp(\alpha) \ge 0$ for all other $\wp$.

If *flag* is non-zero, x must be given as a prime ideal factorization, as output by `idealfactor`, but possibly with zero or negative exponents. This yields an element α such that for all prime ideals $\wp$ occurring in x, $v_\wp(\alpha)$ is equal to the exponent of $\wp$ in x, and for all other prime ideals, $v_\wp(\alpha) \ge 0$. 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 **idealappr0**$(nf, x, flag)$.

3.6.44 idealchinese(nf, x, y): x being a prime ideal factorization (i.e. a 2 by 2 matrix whose first column contain prime ideals, and the second column integral exponents), y a vector of elements in nf indexed by the ideals in x, computes an element b such that

$v_\wp(b - y_\wp) \ge v_\wp(x)$ for all prime ideals in x and $v_\wp(b) \ge 0$ for all other $\wp$.

The library syntax is **idealchinese**(nf, x, y).

3.6.45 idealcoprime(nf, x, y): given two integral ideals x and y in the number field nf, finds a β in the field, expressed on the integral basis $nf[7]$, such that $\beta \cdot x$ is an integral ideal coprime to y.

The library syntax is **idealcoprime**(nf, x, y).

3.6.46 idealdiv$(nf, x, y, \{flag = 0\})$: quotient $x \cdot y^{-1}$ of the two ideals x and y in the number field nf. The result is given in HNF.

If *flag* is non-zero, the quotient $x \cdot y^{-1}$ is assumed to be an integral ideal. This can be much faster when the norm of the quotient is small even though the norms of x and y are large.

The library syntax is **idealdiv0**$(nf, x, y, flag)$. Also available are **idealdiv**(nf, x, y) ($flag = 0$) and **idealdivexact**(nf, x, y) ($flag = 1$).

3.6.47 idealfactor(nf, x): factors into prime ideal powers the ideal x in the number field nf. The output format is similar to the `factor` function, and the prime ideals are represented in the form output by the `idealprimedec` function, i.e. as 5-element vectors.

The library syntax is **idealfactor**(nf, x).

3.6.48 idealhnf$(nf, a, \{b\})$: gives the Hermite normal form matrix of the ideal a. The ideal can be given in any form whatsoever (typically by an algebraic number if it is principal, by a $\mathbf{Z}_K$-system of generators, as a prime ideal as given by `idealprimedec`, or by a $\mathbf{Z}$-basis).

If b is not omitted, assume the ideal given was $a\mathbf{Z}_K + b\mathbf{Z}_K$, where a and b are elements of K given either as vectors on the integral basis $nf[7]$ or as algebraic numbers.

The library syntax is **idealhnf0**(nf, a, b) where an omitted b is coded as `NULL`. Also available is **idealhermite**(nf, a) (b omitted).

3.6.49 idealintersect(nf, x, y): intersection of the two ideals x and y in the number field nf. When x and y are given by $\mathbf{Z}$-bases, this does not depend on nf and can be used to compute the intersection of any two $\mathbf{Z}$-modules. The result is given in HNF.

The library syntax is **idealintersect**(nf, x, y).

3.6.50 idealinv(nf, x): inverse of the ideal x in the number field nf. The result is the Hermite normal form of the inverse of the ideal, together with the opposite of the Archimedean information if it is given.

The library syntax is **idealinv**(nf, x).

3.6.51 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. This information can be either the HNF of the ideal or the `idealstar` with possibly some additional information.

If *flag* is present, its binary digits are toggles meaning

1: give also the generators in the `idealstar`.

2: output $[L, U]$, where L is as before and U is a vector of `zinternallog`s of the units.

4: give only the ideals and not the `idealstar` or the `ideallog` of the units.

The library syntax is **ideallist0**$(nf, bound, flag)$, where *bound* must be a C long integer. Also available is **ideallist**$(nf, bound)$, corresponding to the case $flag = 0$.

3.6.52 ideallistarch$(nf, list, \{arch = []\}, \{flag = 0\})$: vector of vectors of all `idealstarinit` (see `idealstar`) of all modules in *list*, with Archimedean part *arch* added (void if omitted). *list* is a vector of big ideals, as output by `ideallist`$(\ldots, flag)$ for instance. *flag* is optional; its binary digits are toggles meaning: 1: give generators as well, 2: list format is $[L, U]$ (see `ideallist`).

The library syntax is **ideallistarch0**$(nf, list, arch, flag)$, where an omitted *arch* is coded as `NULL`.

3.6.53 ideallog(nf, x, bid): *nf* being a number field, *bid* being a "big ideal" as output by `idealstar` and x being a non-necessarily integral element of *nf* which must have valuation equal to 0 at all prime ideals dividing $I = bid[1]$, computes the "discrete logarithm" of x on the generators given in $bid[2]$. In other words, if g_i are these generators, of orders d_i respectively, the result is a column vector of integers (x_i) such that $0 \le x_i < d_i$ and

$$x \equiv \prod_i g_i^{x_i} \pmod{{}^*I} \ .$$

Note that when I is a module, this implies also sign conditions on the embeddings.

The library syntax is **zideallog**(nf, x, bid).

3.6.54 idealmin$(nf, x, \{vdir\})$: computes a minimum of the ideal x in the direction *vdir* in the number field *nf*.

The library syntax is **minideal**$(nf, x, vdir, prec)$, where an omitted *vdir* is coded as `NULL`.

3.6.55 idealmul$(nf, x, y, \{flag = 0\})$: ideal multiplication of the ideals x and y in the number field *nf*. The result is a generating set for the ideal product with at most n elements, and is in Hermite normal form if either x or y is in HNF or is a prime ideal as output by `idealprimedec`, and this is given together with the sum of the Archimedean information in x and y if both are given.

If *flag* is non-zero, reduce the result using `idealred`.

The library syntax is **idealmul**(nf, x, y) (*flag* $= 0$) or **idealmulred**$(nf, x, y, prec)$ (*flag* $\neq 0$), where as usual, *prec* is a C long integer representing the precision.

3.6.56 idealnorm(nf, x): computes the norm of the ideal x in the number field *nf*.

The library syntax is **idealnorm**(nf, x).

3.6.57 idealpow$(nf, x, k, \{flag = 0\})$: computes the k-th power of the ideal x in the number field *nf*. k can be positive, negative or zero. The result is NOT reduced, it is really the k-th ideal power, and is given in HNF.

If *flag* is non-zero, reduce the result using `idealred`. Note however that this is NOT the same as as `idealpow`(nf, x, k) followed by reduction, since the reduction is performed throughout the powering process.

The library syntax corresponding to *flag* $= 0$ is **idealpow**(nf, x, k). If k is a `long`, you can use **idealpows**(nf, x, k). Corresponding to *flag* $= 1$ is **idealpowred**$(nf, vp, k, prec)$, where *prec* is a `long`.

3.6.58 idealprimedec(nf, p): computes the prime ideal decomposition of the prime number p in the number field nf. p must be a (positive) prime number. Note that the fact that p is prime is not checked, so if a non-prime number p is given it may lead to unpredictable results.

The result is a vector of 5-component vectors, each representing one of the prime ideals above p in the number field nf. The representation $vp = [p, a, e, f, b]$ of a prime ideal means the following. The prime ideal is equal to $p\mathbf{Z}_K + \alpha\mathbf{Z}_K$ where $\mathbf{Z}_K$ is the ring of integers of the field and $\alpha = \sum_i a_i\omega_i$ where the ω_i form the integral basis nf`.zk`, e is the ramification index, f is the residual index, and b is an n-component column vector representing a $\beta \in \mathbf{Z}_K$ such that $vp^{-1} = \mathbf{Z}_K + \beta/p\mathbf{Z}_K$ which will be useful for computing valuations, but which the user can ignore. The number α is guaranteed to have a valuation equal to 1 at the prime ideal (this is automatic if $e > 1$).

The library syntax is **primedec**(nf, p).

3.6.59 idealprincipal(nf, x): creates the principal ideal generated by the algebraic number x (which must be of type integer, rational or polmod) in the number field nf. The result is a one-column matrix.

The library syntax is **principalideal**(nf, x).

3.6.60 idealred$(nf, I, \{vdir = 0\})$: LLL reduction of the ideal I in the number field nf, along the direction $vdir$. If $vdir$ is present, it must be an $r1 + r2$-component vector ($r1$ and $r2$ number of real and complex places of nf as usual).

This function finds a "small" a in I (it is an LLL pseudo-minimum along direction $vdir$). The result is the Hermite normal form of the LLL-reduced ideal rI/a, where r is a rational number such that the resulting ideal is integral and primitive. This is often, but not always, a reduced ideal in the sense of Buchmann. If I is an idele, the logarithmic embeddings of a are subtracted to the Archimedean part.

More often than not, a principal ideal will yield the identity matrix. This is a quick and dirty way to check if ideals are principal without computing a full `bnf` structure, but it's not a necessary condition; hence, a non-trivial result doesn't prove the ideal is non-trivial in the class group.

Note that this is *not* the same as the LLL reduction of the lattice I since ideal operations are involved.

The library syntax is **idealllred**$(nf, x, vdir, prec)$, where an omitted $vdir$ is coded as `NULL`.

3.6.61 idealstar$(nf, I, \{flag = 1\})$: nf being a number field, and I either and 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, outputs necessary data for computing in the group $(\mathbf{Z}_K/I)^*$.

If $flag = 2$, the result is a 5-component vector w. $w[1]$ is the ideal or module I itself. $w[2]$ is the structure of the group. The other components are difficult to describe and are used only in conjunction with the function `ideallog`.

If $flag = 1$ (default), as $flag = 2$, but do not compute explicit generators for the cyclic components, which saves time.

If $flag = 0$, computes the structure of $(\mathbf{Z}_K/I)^*$ as a 3-component vector v. $v[1]$ is the order, $v[2]$ is the vector of SNF cyclic components and $v[3]$ the corresponding generators. When the row vector is explicitly included, the non-zero elements of this vector are considered as real embeddings of nf

in the order given by `polroots`, i.e. in $nf[6]$ (nf`.roots`), and then I is a module with components at infinity.

To solve discrete logarithms (using `ideallog`), you have to choose $flag = 2$.

The library syntax is **idealstar0**$(nf, I, flag)$.

3.6.62 idealtwoelt$(nf, x, \{a\})$: computes a two-element representation of the ideal x in the number field nf, using a straightforward (exponential time) search. x can be an ideal in any form, (including perhaps an Archimedean part, which is ignored) and the result is a row vector $[a, \alpha]$ with two components such that $x = a\mathbf{Z}_K + \alpha\mathbf{Z}_K$ and $a \in \mathbf{Z}$, where a is the one passed as argument if any. If x is given by at least two generators, a is chosen to be the positive generator of $x \cap \mathbf{Z}$.

Note that when an explicit a is given, we use an asymptotically faster method, however in practice it is usually slower.

The library syntax is **ideal_two_elt0**(nf, x, a), where an omitted a is entered as `NULL`.

3.6.63 idealval(nf, x, vp): gives the valuation of the ideal x at the prime ideal vp in the number field nf, where vp must be a 5-component vector as given by `idealprimedec`.

The library syntax is **idealval**(nf, x, vp), and the result is a `long` integer.

3.6.64 ideleprincipal(nf, x): creates the principal idele generated by the algebraic number x (which must be of type integer, rational or polmod) in the number field nf. The result is a two-component vector, the first being a one-column matrix representing the corresponding principal ideal, and the second being the vector with $r_1 + r_2$ components giving the complex logarithmic embedding of x.

The library syntax is **principalidele**(nf, x).

3.6.65 matalgtobasis(nf, x): nf being a number field in `nfinit` format, and x a matrix whose coefficients are expressed as polmods in nf, transforms this matrix into a matrix whose coefficients are expressed on the integral basis of nf. This is the same as applying `nfalgtobasis` to each entry, but it would be dangerous to use the same name.

The library syntax is **matalgtobasis**(nf, x).

3.6.66 matbasistoalg(nf, x): nf being a number field in `nfinit` format, and x a matrix whose coefficients are expressed as column vectors on the integral basis of nf, transforms this matrix into a matrix whose coefficients are algebraic numbers expressed as polmods. This is the same as applying `nfbasistoalg` to each entry, but it would be dangerous to use the same name.

The library syntax is **matbasistoalg**(nf, x).

3.6.67 modreverse(a): a being a polmod $A(X)$ modulo $T(X)$, finds the "reverse polmod" $B(X)$ modulo $Q(X)$, where Q is the minimal polynomial of a, which must be equal to the degree of T, and such that if θ is a root of T then $\theta = B(\alpha)$ for a certain root α of Q.

This is very useful when one changes the generating element in algebraic extensions.

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

3.6.68 newtonpoly(x, p): gives the vector of the slopes of the Newton polygon of the polynomial x with respect to the prime number p. The n components of the vector are in decreasing order, where n is equal to the degree of x. Vertical slopes occur iff the constant coefficient of x is zero and are denoted by `VERYBIGINT`, the biggest single precision integer representable on the machine ($2^{31} - 1$ (resp. $2^{63} - 1$) on 32-bit (resp. 64-bit) machines), see Section 3.2.48.

The library syntax is **newtonpoly**(x, p).

3.6.69 nfalgtobasis(nf, x): this is the inverse function of `nfbasistoalg`. Given an object x whose entries are expressed as algebraic numbers in the number field nf, transforms it so that the entries are expressed as a column vector on the integral basis nf`.zk`.

The library syntax is **algtobasis**(nf, x).

3.6.70 nfbasis$(x, \{flag = 0\}, \{fa\})$: integral basis of the number field defined by the irreducible, preferably monic, polynomial x, using a modified version of the round 4 algorithm by default, due to David Ford, Sebastian Pauli and Xavier Roblot. The binary digits of *flag* have the following meaning:

1: assume that no square of a prime greater than the default `primelimit` divides the discriminant of x, i.e. that the index of x has only small prime divisors.

2: use round 2 algorithm. For small degrees and coefficient size, this is sometimes a little faster. (This program is the translation into C of a program written by David Ford in Algeb.)

Thus for instance, if $flag = 3$, this uses the round 2 algorithm and outputs an order which will be maximal at all the small primes.

If fa is present, we assume (without checking!) that it is the two-column matrix of the factorization of the discriminant of the polynomial x. Note that it does *not* have to be a complete factorization. This is especially useful if only a local integral basis for some small set of places is desired: only factors with exponents greater or equal to 2 will be considered.

The library syntax is **nfbasis0**$(x, flag, fa)$. An extended version is **nfbasis**$(x, \&d, flag, fa)$, where d will receive the discriminant of the number field (*not* of the polynomial x), and an omitted fa should be input as `NULL`. Also available are **base**$(x, \&d)$ ($flag = 0$), **base2**$(x, \&d)$ ($flag = 2$) and **factoredbase**$(x, fa, \&d)$.

3.6.71 nfbasistoalg(nf, x): this is the inverse function of `nfalgtobasis`. Given an object x whose entries are expressed on the integral basis nf`.zk`, transforms it into an object whose entries are algebraic numbers (i.e. polmods).

The library syntax is **basistoalg**(nf, x).

3.6.72 nfdetint(nf, x): given a pseudo-matrix x, computes a non-zero ideal contained in (i.e. multiple of) the determinant of x. This is particularly useful in conjunction with `nfhnfmod`.

The library syntax is **nfdetint**(nf, x).

3.6.73 nfdisc$(x, \{flag = 0\}, \{fa\})$: field discriminant of the number field defined by the integral, preferably monic, irreducible polynomial x. *flag* and fa are exactly as in `nfbasis`. That is, fa provides the matrix of a partial factorization of the discriminant of x, and binary digits of *flag* are as follows:

1: assume that no square of a prime greater than `primelimit` divides the discriminant.

2: use the round 2 algorithm, instead of the default round 4. This should be slower except maybe for polynomials of small degree and coefficients.

The library syntax is **nfdiscf0**$(x, flag, fa)$ where, to omit fa, you should input `NULL`. You can also use **discf**(x) $(flag = 0)$.

3.6.74 nfeltdiv(nf, x, y): given two elements x and y in nf, computes their quotient x/y in the number field nf.

The library syntax is **element_div**(nf, x, y).

3.6.75 nfeltdiveuc(nf, x, y): given two elements x and y in nf, computes an algebraic integer q in the number field nf such that the components of $x - qy$ are reasonably small. In fact, this is functionally identical to `round(nfeltdiv(`nf`,x,y))`.

The library syntax is **nfdiveuc**(nf, x, y).

3.6.76 nfeltdivmodpr(nf, x, y, pr): given two elements x and y in nf and pr a prime ideal in `modpr` format (see `nfmodprinit`), computes their quotient x/y modulo the prime ideal pr.

The library syntax is **element_divmodpr**(nf, x, y, pr).

3.6.77 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 **nfdivres**(nf, x, y).

3.6.78 nfeltmod(nf, x, y): given two elements x and y in nf, computes an element r of nf of the form $r = x - qy$ with q and algebraic integer, and such that r is small. This is functionally identical to

$$\texttt{x} - \texttt{nfeltmul}(nf, \texttt{round}(\texttt{nfeltdiv}(nf, \texttt{x}, \texttt{y})), \texttt{y}).$$

The library syntax is **nfmod**(nf, x, y).

3.6.79 nfeltmul(nf, x, y): given two elements x and y in nf, computes their product $x * y$ in the number field nf.

The library syntax is **element_mul**(nf, x, y).

3.6.80 nfeltmulmodpr(nf, x, y, pr): given two elements x and y in nf and pr a prime ideal in `modpr` format (see `nfmodprinit`), computes their product $x * y$ modulo the prime ideal pr.

The library syntax is **element_mulmodpr**(nf, x, y, pr).

3.6.81 nfeltpow(nf, x, k): given an element x in nf, and a positive or negative integer k, computes x^k in the number field nf.

The library syntax is **element_pow**(nf, x, k).

3.6.82 nfeltpowmodpr(nf, x, k, pr): given an element x in nf, an integer k and a prime ideal pr in `modpr` format (see `nfmodprinit`), computes x^k modulo the prime ideal pr.

The library syntax is **element_powmodpr**(nf, x, k, pr).

3.6.83 nfeltreduce$(nf, x, ideal)$: given an ideal in Hermite normal form and an element x of the number field nf, finds an element r in nf such that $x - r$ belongs to the ideal and r is small.

The library syntax is **element_reduce**$(nf, x, ideal)$.

3.6.84 nfeltreducemodpr(nf, x, pr): given an element x of the number field nf and a prime ideal pr in `modpr` format compute a canonical representative for the class of x modulo pr.

The library syntax is **nfreducemodpr**(nf, x, pr).

3.6.85 nfeltval(nf, x, pr): given an element x in nf and a prime ideal pr in the format output by `idealprimedec`, computes their the valuation at pr of the element x. The same result could be obtained using `idealval(`*nf*`,x,`*pr*`)` (since x would then be converted to a principal ideal), but it would be less efficient.

The library syntax is **element_val**(nf, x, pr), and the result is a `long`.

3.6.86 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 2.6.2). However if the polynomial defining the number field occurs explicitly in the coefficients of x (as modulus of a `t_POLMOD`), its main variable must be *the same* as the main variable of x. For example,

```
? nf = nfinit(y^2 + 1);
? nffactor(nf, x^2 + y); \\ OK
? nffactor(nf, x^2 + Mod(y, y^2+1)); \\  OK
? nffactor(nf, x^2 + Mod(z, z^2+1)); \\  WRONG
```

The library syntax is **nffactor**(nf, x).

3.6.87 nffactormod(nf, x, pr): factorization of the univariate polynomial x modulo the prime ideal pr in the number field nf. x can have coefficients in the number field (scalar, polmod, polynomial, column vector) or modulo the prime ideal (integermod modulo the rational prime under pr, polmod or polynomial with integermod coefficients, column vector of integermod). 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 2.6.2). However if the coefficients of the number field occur explicitly (as polmods) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t (see `nffactor`).

The library syntax is **nffactormod**(nf, x, pr).

3.6.88 nfgaloisapply(nf, aut, x): nf being a number field as output by `nfinit`, and aut being a Galois automorphism of nf expressed either as a polynomial or a polmod (such automorphisms being found using for example one of the variants of `nfgaloisconj`), computes the action of the automorphism aut on the object x in the number field. x can be an element (scalar, polmod, polynomial or column vector) of the number field, an ideal (either given by $\mathbf{Z}_K$-generators or by a $\mathbf{Z}$-basis), a prime ideal (given as a 5-element row vector) or an idele (given as a 2-element row vector). Because of possible confusion with elements and ideals, other vector or matrix arguments are forbidden.

The library syntax is **galoisapply**(nf, aut, x).

3.6.89 nfgaloisconj$(nf, \{flag = 0\}, \{d\})$: nf being a number field as output by `nfinit`, computes the conjugates of a root r of the non-constant polynomial $x = nf[1]$ expressed as polynomials in r. This can be used even if the number field nf is not Galois since some conjugates may lie in the field.

nf can simply be a polynomial if $flag \neq 1$.

If no flags or $flag = 0$, if nf is a number field use a combination of flag 4 and 1 and the result is always complete, else use a combination of flag 4 and 2 and the result is subject to the restriction of $flag = 2$, but a warning is issued when it is not proven complete.

If $flag = 1$, use `nfroots` (require a number field).

If $flag = 2$, use complex approximations to the roots and an integral LLL. The result is not guaranteed to be complete: some conjugates may be missing (no warning issued), especially so if the corresponding polynomial has a huge index. In that case, increasing the default precision may help.

If $flag = 4$, use Allombert's algorithm and permutation testing. If the field is Galois with "weakly" super solvable Galois group, return the complete list of automorphisms, else only the identity element. If present, d is assumed to be a multiple of the least common denominator of the conjugates expressed as polynomial in a root of pol.

A group G is "weakly" super solvable (WKSS) if it contains a super solvable normal subgroup H such that $G = H$, or $G/H \simeq A_4$, or $G/H \simeq S_4$. Abelian and nilpotent groups are WKSS. In practice, almost all groups of small order are WKSS, the exceptions having order 36(1 exception), 48(2), 56(1), 60(1), 72(5), 75(1), 80(1), 96(10) and ≥ 108.

Hence $flag = 4$ permits to quickly check whether a polynomial of order strictly less than 36 is Galois or not. This method is much faster than `nfroots` and can be applied to polynomials of degree larger than 50.

This routine can only compute $\mathbf{Q}$-automorphisms, but it may be used to get K-automorphism for any base field K as follows:

```
rnfgaloisconj(nfK, R) = \\ K-automorphisms of L = K[X] / (R)
{ local(polabs, N, H);
  R *= Mod(1, nfK.pol);             \\ convert coeffs to polmod elts of K
  polabs = rnfequation(nfK, R);
  N = nfgaloisconj(polabs) % R;     \\ Q-automorphisms of L
  H = [];
  for(i=1, #N,                      \\ select the ones that fix K
    if (subst(R, variable(R), Mod(N[i],R)) == 0,
```

```
      H = concat(H,N[i])
    )
  ); H
}
K  = nfinit(y^2 + 7);
polL = x^4 - y*x^3 - 3*x^2 + y*x + 1;
rnfgaloisconj(K, polL)            \\ K-automorphisms of L
```

The library syntax is **galoisconj0**$(nf, flag, d, prec)$. Also available are **galoisconj**(nf) for $flag = 0$, **galoisconj2**$(nf, n, prec)$ for $flag = 2$ where n is a bound on the number of conjugates, and **galoisconj4**(nf, d) corresponding to $flag = 4$.

3.6.90 nfhilbert$(nf, a, b, \{pr\})$: if pr is omitted, compute the global Hilbert symbol (a, b) in nf, that is 1 if $x^2 - ay^2 - bz^2$ has a non trivial solution (x, y, z) in nf, and -1 otherwise. Otherwise compute the local symbol modulo the prime ideal pr (as output by `idealprimedec`).

The library syntax is **nfhilbert**(nf, a, b, pr), where an omitted pr is coded as `NULL`.

3.6.91 nfhnf(nf, x): given a pseudo-matrix (A, I), finds a pseudo-basis in Hermite normal form of the module it generates.

The library syntax is **nfhermite**(nf, x).

3.6.92 nfhnfmod$(nf, x, detx)$: given a pseudo-matrix (A, I) and an ideal $detx$ which is contained in (read integral multiple of) the determinant of (A, I), finds a pseudo-basis in Hermite normal form of the module generated by (A, I). This avoids coefficient explosion. $detx$ can be computed using the function `nfdetint`.

The library syntax is **nfhermitemod**$(nf, x, detx)$.

3.6.93 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`), 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 $nf[1]$, i.e. $[\mathbf{Z}_K : \mathbf{Z}[\theta]]$, where θ is any root of $nf[1]$.

$nf[5]$ is a vector containing 7 matrices M, G, $T2$, T, MD, TI, MDI useful for certain computations in the number field K.

• M is the $(r1+r2)\times n$ matrix whose columns represent the numerical values of the conjugates of the elements of the integral basis.

• G is such that $T2 = {}^tGG$, where $T2$ is the quadratic form $T_2(x) = \sum |\sigma(x)|^2$, σ running over the embeddings of K into $\mathbf{C}$.

• The $T2$ component is deprecated and currently unused.

• T is the $n \times n$ matrix whose coefficients are $\mathrm{Tr}(\omega_i\omega_j)$ where the ω_i are the elements of the integral basis. Note also that $\det(T)$ is equal to the discriminant of the field K.

• The columns of MD (*nf*`.diff`) express a $\mathbf{Z}$-basis of the different of K on the integral basis.

• TI is equal to $d(K)T^{-1}$, which has integral coefficients. Note that, understood as as ideal, the matrix T^{-1} generates the codifferent ideal.

• Finally, MDI is a two-element representation (for faster ideal product) of $d(K)$ times the codifferent ideal (*nf*`.disc`$*$*nf*`.codiff`, which is an integral ideal). MDI is only used in `idealinv`.

nf[6] is the vector containing the $r1+r2$ roots (*nf*`.roots`) of *nf*[1] corresponding to the $r1+r2$ embeddings of the number field into $\mathbf{C}$ (the first $r1$ components are real, the next $r2$ have positive imaginary part).

nf[7] is an integral basis for $\mathbf{Z}_K$ (*nf*`.zk`) expressed on the powers of θ. Its first element is guaranteed to be 1. This basis is LLL-reduced with respect to T_2 (strictly speaking, it is a permutation of such a basis, due to the condition that the first element be 1).

nf[8] is the $n \times n$ integral matrix expressing the power basis in terms of the integral basis, and finally

nf[9] is the $n \times n^2$ matrix giving the multiplication table of the integral basis.

If a non monic polynomial is input, `nfinit` will transform it into a monic one, then reduce it (see *flag* = 3). It is allowed, though not very useful given the existence of `nfnewprec`, to input a `nf` or a `bnf` instead of a polynomial.

The special input format $[x, B]$ is also accepted where x is a polynomial as above and B is the integer basis, as would be computed by `nfbasis`. This can be useful if the integer basis is known in advance.

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 $[nf, \mathtt{Mod}(a, P)]$, where *nf* is as before and $\mathtt{Mod}(a, P) = \mathtt{Mod}(x, pol)$ gives the change of variables. This is implicit when *pol* is not monic: first a linear change of variables is performed, to get a monic polynomial, then a `polred` reduction.

If *flag* = 4, as 2 but uses a partial `polred`.

If *flag* = 5, as 3 using a partial `polred`.

The library syntax is **nfinit0**$(x, flag, prec)$.

3.6.94 nfisideal(nf, x): returns 1 if x is an ideal in the number field *nf*, 0 otherwise.

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

3.6.95 nfisincl(x, y): tests whether the number field K defined by the polynomial x is conjugate to a subfield of the field L defined by y (where x and y must be in $\mathbf{Q}[X]$). If they are not, the output is the number 0. If they are, the output is a vector of polynomials, each polynomial a representing an embedding of K into L, i.e. being such that $y \mid x \circ a$.

If y is a number field (nf), a much faster algorithm is used (factoring x over y using `nffactor`). Before version 2.0.14, this wasn't guaranteed to return all the embeddings, hence was triggered by a special flag. This is no more the case.

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

3.6.96 nfisisom(x, y): as `nfisincl`, but tests for isomorphism. If either x or y is a number field, a much faster algorithm will be used.

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

3.6.97 nfnewprec(nf): transforms the number field nf into the corresponding data using current (usually larger) precision. This function works as expected if nf is in fact a bnf (update bnf to current precision) but may be quite slow (many generators of principal ideals have to be computed).

The library syntax is **nfnewprec**$(nf, prec)$.

3.6.98 nfkermodpr(nf, a, pr): kernel of the matrix a in $\mathbf{Z}_K/pr$, where pr is in **modpr** format (see `nfmodprinit`).

The library syntax is **nfkermodpr**(nf, a, pr).

3.6.99 nfmodprinit(nf, pr): transforms the prime ideal pr into `modpr` format necessary for all operations modulo pr in the number field nf. Returns a two-component vector $[P, a]$, where P is the Hermite normal form of pr, and a is an integral element congruent to 1 modulo pr, and congruent to 0 modulo p/pr^e. Here $p = \mathbf{Z} \cap pr$ and e is the absolute ramification index.

The library syntax is **nfmodprinit**(nf, pr).

3.6.100 nfsubfields$(pol, \{d = 0\})$: finds all subfields of degree d of the number field defined by the (monic, integral) polynomial pol (all subfields if d is null or omitted). The result is a vector of subfields, each being given by $[g, h]$, where g is an absolute equation and h expresses one of the roots of g in terms of the root x of the polynomial defining nf. This routine uses J. Klüners's algorithm in the general case, and B. Allombert's `galoissubfields` when nf is Galois (with weakly supersolvable Galois group).

The library syntax is **subfields**(nf, d).

3.6.101 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 2.6.2). However if the coefficients of the number field occur explicitly (as polmods) as coefficients of x, the variable of these polmods *must* be the same as the main variable of t (see `nffactor`).

The library syntax is **nfroots**(nf, x).

3.6.102 nfrootsof1(nf): computes the number of roots of unity w and a primitive w-th root of unity (expressed on the integral basis) belonging to the number field nf. The result is a two-component vector $[w, z]$ where z is a column vector expressing a primitive w-th root of unity on the integral basis nf`.zk`.

The library syntax is **rootsof1**(nf).

3.6.103 nfsnf(nf, x): given a torsion module x as a 3-component row vector $[A, I, J]$ where A is a square invertible $n \times n$ matrix, I and J are two ideal lists, outputs an ideal list $d_1, \dots, d_n$ 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 $i \geq 2$. The link between x and $[A, I, J]$ is as follows: if e_i is the canonical basis of K^n, $I = [b_1, \dots, b_n]$ and $J = [a_1, \dots, a_n]$, then x is isomorphic to

$$(b_1 e_1 \oplus \cdots \oplus b_n e_n)/(a_1 A_1 \oplus \cdots \oplus a_n A_n) \ ,$$

where the A_j are the columns of the matrix A. Note that every finitely generated torsion module can be given in this way, and even with $b_i = Z_K$ for all i.

The library syntax is **nfsmith**(nf, x).

3.6.104 nfsolvemodpr(nf, a, b, pr): solution of $a \cdot x = b$ in $\mathbf{Z}_K/pr$, where a is a matrix and b a column vector, and where pr is in **modpr** format (see `nfmodprinit`).

The library syntax is **nfsolvemodpr**(nf, a, b, pr).

3.6.105 polcompositum($P, Q, \{flag = 0\}$): P and Q being squarefree polynomials in $\mathbf{Z}[X]$ in the same variable, outputs the simple factors of the étale $\mathbf{Q}$-algebra $A = \mathbf{Q}(X, Y)/(P(X), Q(Y))$. The factors are given by a list of polynomials R in $\mathbf{Z}[X]$, associated to the number field $\mathbf{Q}(X)/(R)$, and sorted by increasing degree (with respect to lexicographic ordering for factors of equal degrees). Returns an error if one of the polynomials is not squarefree.

Note that it is more efficient to reduce to the case where P and Q are irreducible first. The routine will not perform this for you, since it may be expensive, and the inputs are irreducible in most applications anyway. Assuming P is irreducible (of smaller degree than Q for efficiency), it is in general *much* faster to proceed as follows

```
nf = nfinit(P); L = nffactor(nf, Q)[,1];
vector(#L, i, rnfequation(nf, L[i]))
```

to obtain the same result. If you are only interested in the degrees of the simple factors, the `rnfequation` instruction can be replaced by a trivial `poldegree(P) * poldegree(L[i])`.

If $flag = 1$, outputs a vector of 4-component vectors $[R, a, b, k]$, where R ranges through the list of all possible compositums as above, and a (resp. b) expresses the root of P (resp. Q) as an element of $\mathbf{Q}(X)/(R)$. Finally, k is a small integer such that $b - ka = X$ modulo R.

A compositum is quite often defined by a complicated polynomial, which it is advisable to reduce before further work. Here is a simple example involving the field $\mathbf{Q}(\zeta_5, 5^{1/5})$:

```
? z = polcompositum(x^5 - 5, polcyclo(5), 1)[1];
? pol = z[1]                    \\ pol defines the compositum
%2 = x^20 + 5*x^19 + 15*x^18 + 35*x^17 + 70*x^16 + 141*x^15 + 260*x^14 \
  + 355*x^13 + 95*x^12 - 1460*x^11 - 3279*x^10 - 3660*x^9 - 2005*x^8    \
  + 705*x^7 + 9210*x^6 + 13506*x^5 + 7145*x^4 - 2740*x^3 + 1040*x^2     \
```

```
  - 320*x + 256
? a = z[2]; a^5 - 5          \\ a is a fifth root of 5
%3 = 0
? z = polredabs(pol, 1);     \\ look for a simpler polynomial
? pol = z[1]
%5 = x^20 + 25*x^10 + 5
? a = subst(a.pol, x, z[2])  \\ a in the new coordinates
%6 = Mod(-5/22*x^19 + 1/22*x^14 - 123/22*x^9 + 9/11*x^4, x^20 + 25*x^10 + 5)
```

The library syntax is **polcompositum0**$(P, Q, flag)$.

3.6.106 polgalois(x): Galois group of the non-constant polynomial $x \in \mathbf{Q}[X]$. In the present version 2.2.7, 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 3-component vector $[n, s, k]$ 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).

k is more arbitrary and the choice made up to version 2.2.3 of PARI is rather unfortunate: for $n > 7$, k is the numbering of the group among all transitive subgroups of S_n, as given in "The transitive groups of degree up to eleven", G. Butler and J. McKay, Communications in Algebra, vol. 11, 1983, pp. 863–911 (group k is denoted T_k there). And for $n \leq 7$, it was ad hoc, so as to ensure that a given triple would design a unique group. Specifically, for polynomials of degree ≤ 7, the groups are coded as follows, using standard notations

In degree 1: $S_1 = [1, 1, 1]$.

In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1]$, $S_3 = [6, -1, 1]$.

In degree 4: $C_4 = [4, -1, 1]$, $V_4 = [4, 1, 1]$, $D_4 = [8, -1, 1]$, $A_4 = [12, 1, 1]$, $S_4 = [24, -1, 1]$.

In degree 5: $C_5 = [5, 1, 1]$, $D_5 = [10, 1, 1]$, $M_{20} = [20, -1, 1]$, $A_5 = [60, 1, 1]$, $S_5 = [120, -1, 1]$.

In degree 6: $C_6 = [6, -1, 1]$, $S_3 = [6, -1, 2]$, $D_6 = [12, -1, 1]$, $A_4 = [12, 1, 1]$, $G_{18} = [18, -1, 1]$, $S_4^- = [24, -1, 1]$, $A_4 \times C_2 = [24, -1, 2]$, $S_4^+ = [24, 1, 1]$, $G_{36}^- = [36, -1, 1]$, $G_{36}^+ = [36, 1, 1]$, $S_4 \times C_2 = [48, -1, 1]$, $A_5 = PSL_2(5) = [60, 1, 1]$, $G_{72} = [72, -1, 1]$, $S_5 = PGL_2(5) = [120, -1, 1]$, $A_6 = [360, 1, 1]$, $S_6 = [720, -1, 1]$.

In degree 7: $C_7 = [7, 1, 1]$, $D_7 = [14, -1, 1]$, $M_{21} = [21, 1, 1]$, $M_{42} = [42, -1, 1]$, $PSL_2(7) = PSL_3(2) = [168, 1, 1]$, $A_7 = [2520, 1, 1]$, $S_7 = [5040, -1, 1]$.

This is deprecated and obsolete, but for reasons of backward compatibility, we cannot change this behaviour yet. So you can use the default `new_galois_format` to switch to a consistent naming scheme, namely k is always the standard numbering of the group among all transitive subgroups of S_n. If this default is in effect, the above groups will be coded as:

In degree 1: $S_1 = [1, 1, 1]$.

In degree 2: $S_2 = [2, -1, 1]$.

In degree 3: $A_3 = C_3 = [3, 1, 1]$, $S_3 = [6, -1, 2]$.

In degree 4: $C_4 = [4, -1, 1]$, $V_4 = [4, 1, 2]$, $D_4 = [8, -1, 3]$, $A_4 = [12, 1, 4]$, $S_4 = [24, -1, 5]$.

In degree 5: $C_5 = [5, 1, 1]$, $D_5 = [10, 1, 2]$, $M_{20} = [20, -1, 3]$, $A_5 = [60, 1, 4]$, $S_5 = [120, -1, 5]$.

In degree 6: $C_6 = [6, -1, 1]$, $S_3 = [6, -1, 2]$, $D_6 = [12, -1, 3]$, $A_4 = [12, 1, 4]$, $G_{18} = [18, -1, 5]$, $A_4 \times C_2 = [24, -1, 6]$, $S_4^+ = [24, 1, 7]$, $S_4^- = [24, -1, 8]$, $G_{36}^- = [36, -1, 9]$, $G_{36}^+ = [36, 1, 10]$, $S_4 \times C_2 = [48, -1, 11]$, $A_5 = PSL_2(5) = [60, 1, 12]$, $G_{72} = [72, -1, 13]$, $S_5 = PGL_2(5) = [120, -1, 14]$, $A_6 = [360, 1, 15]$, $S_6 = [720, -1, 16]$.

In degree 7: $C_7 = [7, 1, 1]$, $D_7 = [14, -1, 2]$, $M_{21} = [21, 1, 3]$, $M_{42} = [42, -1, 4]$, $PSL_2(7) = PSL_3(2) = [168, 1, 5]$, $A_7 = [2520, 1, 6]$, $S_7 = [5040, -1, 7]$.

Warning: The method used is that of resolvent polynomials and is sensitive to the current precision. The precision is updated internally but, in very rare cases, a wrong result may be returned if the initial precision was not sufficient.

The library syntax is **galois**$(x, prec)$. To enable the new format in library mode, set the global variable `new_galois_format` to 1.

3.6.107 polred$(x, \{flag = 0\}, \{fa\})$: finds polynomials with reasonably small coefficients defining subfields of the number field defined by x. One of the polynomials always defines $\mathbf{Q}$ (hence is equal to $x - 1$), and another always defines the same number field as x if x is irreducible. All x accepted by `nfinit` are also allowed here (e.g. non-monic polynomials, `nf`, `bnf`, `[x,Z_K_basis]`).

The following binary digits of *flag* are significant:

1: possibly use a suborder of the maximal order. The primes dividing the index of the order chosen are larger than `primelimit` or divide integers stored in the `addprimes` table.

2: gives also elements. The result is a two-column matrix, the first column giving the elements defining these subfields, the second giving the corresponding minimal polynomials.

If fa is given, it is assumed that it is the two-column matrix of the factorization of the discriminant of the polynomial x.

The library syntax is **polred0**$(x, flag, fa)$, where an omitted fa is coded by `NULL`. Also available are **polred**(x) and **factoredpolred**(x, fa), both corresponding to $flag = 0$.

3.6.108 polredabs$(x, \{flag = 0\})$: finds one of the polynomial defining the same number field as the one defined by x, and such that the sum of the squares of the modulus of the roots (i.e. the T_2-norm) is minimal. All x accepted by `nfinit` are also allowed here (e.g. non-monic polynomials, `nf`, `bnf`, `[x,Z_K_basis]`).

Warning: this routine uses an exponential-time algorithm to enumerate all potential generators, and may be exceedingly slow when the number field has many subfields, hence a lot of elements of small T_2-norm. E.g. do not try it on the compositum of many quadratic fields, use `polred` instead.

The binary digits of *flag* mean

1: outputs a two-component row vector $[P, a]$, where P is the default output and a is an element expressed on a root of the polynomial P, whose minimal polynomial is equal to x.

4: gives *all* polynomials of minimal T_2 norm (of the two polynomials $P(x)$ and $P(-x)$, only one is given).

16: possibly use a suborder of the maximal order. The primes dividing the index of the order chosen are larger than `primelimit` or divide integers stored in the `addprimes` table. In that case it may happen that the output polynomial does not have minimal T_2 norm.

The library syntax is **polredabs0**$(x, flag)$.

3.6.109 polredord(x): finds polynomials with reasonably small coefficients and of the same degree as that of x defining suborders of the order defined by x. One of the polynomials always defines $\mathbf{Q}$ (hence is equal to $(x-1)^n$, where n is the degree), and another always defines the same order as x if x is irreducible.

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

3.6.110 poltschirnhaus(x): applies a random Tschirnhausen transformation to the polynomial x, which is assumed to be non-constant and separable, so as to obtain a new equation for the étale algebra defined by x. This is for instance useful when computing resolvents, hence is used by the `polgalois` function.

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

3.6.111 rnfalgtobasis(rnf, x): expresses x on the relative integral basis. Here, *rnf* is a relative number field extension L/K as output by `rnfinit`, and x an element of L in absolute form, i.e. expressed as a polynomial or polmod with polmod coefficients, *not* on the relative integral basis.

The library syntax is **rnfalgtobasis**(rnf, x).

3.6.112 rnfbasis(bnf, L): gives either a true *bnf*-basis of $\mathbf{Z}_L$ if it exists, or an $n+1$-element generating set of L if not, where n is the rank of L over *bnf*. Here, *bnf* is as output by `bnfinit`; L is either a polynomial with coefficients in *bnf* defining a relative extension of *bnf*, or a pseudo-basis of such an extension, as ouput by `rnfpseudobasis`.

The library syntax is **rnfbasis**(bnf, x).

3.6.113 rnfbasistoalg(rnf, x): computes the representation of x as a polmod with polmods coefficients. Here, *rnf* is a relative number field extension L/K as output by `rnfinit`, and x an element of L expressed on the relative integral basis.

The library syntax is **rnfbasistoalg**(rnf, x).

3.6.114 rnfcharpoly$(nf, T, a, \{v = x\})$: characteristic polynomial of a over *nf*, where a belongs to the algebra defined by T over *nf*, i.e. $nf[X]/(T)$. Returns a polynomial in variable v (x by default).

The library syntax is **rnfcharpoly**(nf, T, a, v), where v is a variable number.

3.6.115 rnfconductor$(bnf, pol, \{flag = 0\})$: given *bnf* as output by `bnfinit`, and *pol* a relative polynomial defining an Abelian extension, computes the class field theory conductor of this Abelian extension. The result is a 3-component vector $[conductor, rayclgp, subgroup]$, where *conductor* is the conductor of the extension given as a 2-component row vector $[f_0, f_\infty]$, *rayclgp* is the full ray class group corresponding to the conductor given as a 3-component vector [h,cyc,gen] as usual for a group, and *subgroup* is a matrix in HNF defining the subgroup of the ray class group on the given generators gen. If *flag* is non-zero, check that *pol* indeed defines an Abelian extension, return 0 if it does not.

The library syntax is **rnfconductor**$(rnf, pol, flag)$.

3.6.116 rnfdedekind(nf, pol, pr): given a number field *nf* as output by `nfinit` and a polynomial *pol* with coefficients in *nf* defining a relative extension L of *nf*, evaluates the relative Dedekind criterion over the order defined by a root of *pol* for the prime ideal *pr* and outputs a 3-component vector as the result. The first component is a flag equal to 1 if the enlarged order could be proven to be *pr*-maximal and to 0 otherwise (it may be maximal in the latter case if *pr* is ramified in L), the second component is a pseudo-basis of the enlarged order and the third component is the valuation at *pr* of the order discriminant.

The library syntax is **rnfdedekind**(nf, pol, pr).

3.6.117 rnfdet(nf, M): given a pseudo-matrix M over the maximal order of *nf*, computes its determinant.

The library syntax is **rnfdet**(nf, M).

3.6.118 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 2.6.2.

The library syntax is **rnfdiscf**(bnf, pol).

3.6.119 rnfeltabstorel(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being an element of L expressed as a polynomial modulo the absolute equation *rnf*`.pol`, computes x as an element of the relative extension L/K as a polmod with polmod coefficients.

The library syntax is **rnfelementabstorel**(rnf, x).

3.6.120 rnfeltdown(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being an element of L expressed as a polynomial or polmod with polmod coefficients, computes x as an element of K as a polmod, assuming x is in K (otherwise an error will occur). If x is given on the relative integral basis, apply `rnfbasistoalg` first, otherwise PARI will believe you are dealing with a vector.

The library syntax is **rnfelementdown**(rnf, x).

3.6.121 rnfeltreltoabs(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being an element of L expressed as a polynomial or polmod with polmod coefficients, computes x as an element of the absolute extension $L/\mathbf{Q}$ as a polynomial modulo the absolute equation *rnf*`.pol`. If x is given on the relative integral basis, apply `rnfbasistoalg` first, otherwise PARI will believe you are dealing with a vector.

The library syntax is **rnfelementreltoabs**(rnf, x).

3.6.122 rnfeltup(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being an element of K expressed as a polynomial or polmod, computes x as an element of the absolute extension $L/\mathbf{Q}$ as a polynomial modulo the absolute equation *rnf*`.pol`. If x is given on the integral basis of K, apply `nfbasistoalg` first, otherwise PARI will believe you are dealing with a vector.

The library syntax is **rnfelementup**(rnf, x).

3.6.123 rnfequation($nf, pol, \{flag = 0\}$): given a number field *nf* as output by `nfinit` (or simply a polynomial) and a polynomial *pol* with coefficients in *nf* defining a relative extension L of *nf*, computes the absolute equation of L over $\mathbf{Q}$.

If *flag* is non-zero, outputs a 3-component row vector $[z, a, k]$, where z is the absolute equation of L over $\mathbf{Q}$, as in the default behaviour, a expresses as an element of L a root α of the polynomial defining the base field *nf*, and k is a small integer such that $\theta = \beta + k\alpha$ where θ is a root of z and β a root of *pol*.

The main variable of *nf* *must* be of lower priority than that of *pol* (see Section 2.6.2). Note that for efficiency, this does not check whether the relative equation is irreducible over *nf*, but only if it is squarefree. If it is reducible but squarefree, the result will be the absolute equation of the étale algebra defined by *pol*. If *pol* is not squarefree, an error message will be issued.

The library syntax is **rnfequation0**($nf, pol, flag$).

3.6.124 rnfhnfbasis(bnf, x): given a big number field *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 **rnfhermitebasis**(nf, x).

3.6.125 rnfidealabstorel(rnf, x): let *rnf* be a relative number field extension L/K as output by `rnfinit`, and x an ideal of the absolute extension $L/\mathbf{Q}$ given by a $\mathbf{Z}$-basis of elements of L. Returns the relative pseudo-matrix in HNF giving the ideal x considered as an ideal of the relative extension L/K.

If x is an ideal in HNF form, associated to an *nf* structure, for instance as output by `idealhnf`($nf, \ldots$), use `rnfidealabstorel(rnf, nf.zk * x)` to convert it to a relative ideal.

The library syntax is **rnfidealabstorel**(rnf, x).

3.6.126 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 3.6.125), returns the ideal of K below x, i.e. the intersection of x with K.

The library syntax is **rnfidealdown**(rnf, x).

3.6.127 rnfidealhnf(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being a relative ideal (which can be, as in the absolute case, of many different types, including of course elements), computes the HNF pseudo-matrix associated to x, viewed as a $\mathbf{Z}_K$-module.

The library syntax is **rnfidealhermite**(rnf, x).

3.6.128 rnfidealmul(rnf, x, y): *rnf* being a relative number field extension L/K as output by `rnfinit` and x and y being ideals of the relative extension L/K given by pseudo-matrices, outputs the ideal product, again as a relative ideal.

The library syntax is **rnfidealmul**(rnf, x, y).

3.6.129 rnfidealnormabs(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being a relative ideal (which can be, as in the absolute case, of many different types, including of course elements), computes the norm of the ideal x considered as an ideal of the absolute extension $L/\mathbf{Q}$. This is identical to `idealnorm(rnfidealnormrel(`*rnf*`,x))`, but faster.

The library syntax is **rnfidealnormabs**(rnf, x).

3.6.130 rnfidealnormrel(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being a relative ideal (which can be, as in the absolute case, of many different types, including of course elements), computes the relative norm of x as a ideal of K in HNF.

The library syntax is **rnfidealnormrel**(rnf, x).

3.6.131 rnfidealreltoabs(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), 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) =
{
  local(z); z = nfalgtobasis(nf, y);
  z[1] = Mat(z[1]); mathnf( concat(z) );
}
```

The library syntax is **rnfidealreltoabs**(rnf, x).

3.6.132 rnfidealtwoelt(rnf, x): *rnf* being a relative number field extension L/K as output by `rnfinit` and x being an ideal of the relative extension L/K given by a pseudo-matrix, gives a vector of two generators of x over $\mathbf{Z}_L$ expressed as polmods with polmod coefficients.

The library syntax is **rnfidealtwoelement**(rnf, x).

3.6.133 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) =
{
  local(z); z = nfalgtobasis(nf, y);
  z[1] = Mat(z[1]); mathnf( concat(z) );
}
```

The library syntax is **rnfidealup**(rnf, x).

3.6.134 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 2.6.2) than that of *nf*, and the coefficients of *pol* must be in *nf*.

The result is a row vector, whose components are technical. In the following description, we let K be the base field defined by *nf*, m the degree of the base field, n the relative degree, L the large field (of relative degree n or absolute degree nm), r_1 and r_2 the number of real and complex places of K.

rnf[1] contains the relative polynomial *pol*.

rnf[2] is currently unused.

rnf[3] is a two-component row vector $[\mathfrak{d}(L/K), s]$ where $\mathfrak{d}(L/K)$ is the relative ideal discriminant of L/K and s is the discriminant of L/K viewed as an element of $K^*/(K^*)^2$, in other words it is the output of `rnfdisc`.

rnf[4] is the ideal index $\mathfrak{f}$, i.e. such that $d(pol)\mathbf{Z}_K = \mathfrak{f}^2\mathfrak{d}(L/K)$.

rnf[5] is currently unused.

rnf[6] is currently unused.

rnf[7] is a two-component row vector, where the first component is the relative integral pseudo basis expressed as polynomials (in the variable of *pol*) with polmod coefficients in *nf*, and the second component is the ideal list of the pseudobasis in HNF.

rnf[8] is the inverse matrix of the integral basis matrix, with coefficients polmods in *nf*.

rnf[9] is currently unused.

rnf[10] is *nf*.

rnf[11] is the output of `rnfequation(nf, pol, 1)`. Namely, a vector *vabs* with 3 entries describing the *absolute* extension $L/\mathbf{Q}$. *vabs*[1] is an absolute equation, more conveniently obtained as `rnf.pol`. *vabs*[2] expresses the generator α of the number field *nf* as a polynomial modulo the absolute equation *vabs*[1]. *vabs*[3] is a small integer k such that, if β is an abstract root of *pol* and α the generator of *nf*, the generator whose root is *vabs* will be $\beta + k\alpha$. Note that one must be very careful if $k \neq 0$ when dealing simultaneously with absolute and relative quantities since the generator chosen for the absolute extension is not the same as for the relative one. If this happens,

one can of course go on working, but we strongly advise to change the relative polynomial so that its root will be $\beta + k\alpha$. Typically, the GP instruction would be

```
pol = subst(pol, x, x - k*Mod(y,nf.pol))
```

rnf[12] is by default unused and set equal to 0. This field is used to store further information about the field as it becomes available (which is rarely needed, hence would be too expensive to compute during the initial `rnfinit` call).

The library syntax is **rnfinitalg**(*nf*, *pol*, *prec*).

3.6.135 rnfisfree(*bnf*, *x*): given a big number field *bnf* as output by `bnfinit`, and either a polynomial x with coefficients in *bnf* defining a relative extension L of *bnf*, or a pseudo-basis x of such an extension, returns true (1) if L/bnf is free, false (0) if not.

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

3.6.136 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 $a = Norm(x) * q$. The algorithm looks for a solution x which is an S-integer, with S a list of places of K containing at least the ramified primes, the generators of the class group of L, as well as those primes dividing a. If L/K is Galois, then this is enough; otherwise, *flag* is used to add more primes to S: all the places above the primes $p \leq flag$ (resp. $p|flag$) if $flag > 0$ (resp. $flag < 0$).

The answer is guaranteed (i.e. a is a norm iff $q = 1$) if the field is Galois, or, under GRH, if S contains all primes less than $12 \log^2 |\mathrm{disc}(M)|$, where M is the normal closure of L/K.

If `rnfisnorminit` has determined (or was told) that L/K is Galois, and $flag \neq 0$, a Warning is issued (so that you can set $flag = 1$ to check whether L/K is known to be Galois, according to T). Example:

```
bnf = bnfinit(y^3 + y^2 - 2*y - 1);
p = x^2 + Mod(y^2 + 2*y + 1, bnf.pol);
T = rnfisnorminit(bnf, p);
rnfisnorm(T, 17)
```

checks whether 17 is a norm in the Galois extension $\mathbf{Q}(\beta)/\mathbf{Q}(\alpha)$, where $\alpha^3 + \alpha^2 - 2\alpha - 1 = 0$ and $\beta^2 + \alpha^2 + 2\alpha + 1 = 0$ (it is).

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

3.6.137 rnfisnorminit(*pol*, *polrel*, $\{flag = 2\}$): let K be defined by a root of *pol*, and L/K the extension defined by the polynomial *polrel*. As usual, *pol* can in fact be an *nf*, or *bnf*, etc; if *pol* has degree 1 (the base field is $\mathbf{Q}$), polrel is also allowed to be an *nf*, etc. Computes technical data needed by `rnfisnorm` to solve norm equations $Nx = a$, for x in L, and a in K.

If $flag = 0$, do not care whether L/K is Galois or not.

If $flag = 1$, L/K is assumed to be Galois (unchecked), which speeds up `rnfisnorm`.

If $flag = 2$, let the routine determine whether L/K is Galois.

The library syntax is **rnfisnorminit**(*pol*, *polrel*, *flag*).

3.6.138 rnfkummer($bnr, \{subgroup\}, \{deg = 0\}$): *bnr* being as output by `bnrinit`, finds a relative equation for the class field corresponding to the module in *bnr* and the given congruence subgroup (the full ray class field if *subgroup* is omitted). If *deg* is positive, outputs the list of all relative equations of degree *deg* contained in the ray class field defined by *bnr*, with the *same* conductor as (*bnr*, *subgroup*).

Warning: this routine only works for subgroups of prime index. It uses Kummer theory, adjoining necessary roots of unity (it needs to compute a tough `bnfinit` here), and finds a generator via Hecke's characterization of ramification in Kummer extensions of prime degree. If your extension does not have prime degree, for the time being, you have to split it by hand as a tower / compositum of such extensions.

The library syntax is **rnfkummer**(*bnr*, *subgroup*, *deg*, *prec*), where *deg* is a `long` and an omitted *subgroup* is coded as `NULL`

3.6.139 rnflllgram($nf, pol, order$): given a polynomial *pol* with coefficients in *nf* defining a relative extension L and a suborder *order* of L (of maximal rank), as output by `rnfpseudobasis`(nf, pol) or similar, gives $[[neworder], U]$, where *neworder* is a reduced order and U is the unimodular transformation matrix.

The library syntax is **rnflllgram**(*nf*, *pol*, *order*, *prec*).

3.6.140 rnfnormgroup(bnr, pol): *bnr* being a big ray class field as output by `bnrinit` and *pol* a relative polynomial defining an Abelian extension, computes the norm group (alias Artin or Takagi group) corresponding to the Abelian extension of $bnf = bnr[1]$ defined by *pol*, where the module corresponding to *bnr* is assumed to be a multiple of the conductor (i.e. *pol* defines a subextension of bnr). The result is the HNF defining the norm group on the given generators of $bnr[5][3]$. Note that neither the fact that *pol* defines an Abelian extension nor the fact that the module is a multiple of the conductor is checked. The result is undefined if the assumption is not correct.

The library syntax is **rnfnormgroup**(*bnr*, *pol*).

3.6.141 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 2.2.7, this is slower and less efficient than `rnfpolredabs`.

The library syntax is **rnfpolred**(*nf*, *pol*, *prec*).

3.6.142 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 3.6.108.

Remark. In the present implementation, this is both faster and much more efficient than `rnfpolred`, the difference being more dramatic than in the absolute case. This is because the implementation of `rnfpolred` is based on (a partial implementation of) an incomplete reduction theory of lattices over number fields, the function `rnflllgram`, which deserves to be improved.

The library syntax is **rnfpolredabs**(*nf*, *pol*, *flag*, *prec*).

3.6.143 rnfpseudobasis(*nf*, *pol*): given a number field *nf* as output by `nfinit` and a polynomial *pol* with coefficients in *nf* defining a relative extension L of *nf*, computes a pseudo-basis (A, I) for the maximal order $\mathbf{Z}_L$ viewed as a $\mathbf{Z}_K$-module, and the relative discriminant of L. This is output as a four-element row vector $[A, I, D, d]$, where D is the relative ideal discriminant and d is the relative discriminant considered as an element of $nf^*/{nf^*}^2$.

The library syntax is **rnfpseudobasis**(*nf*, *pol*).

3.6.144 rnfsteinitz(*nf*, *x*): given a number field *nf* as output by `nfinit` and either a polynomial x with coefficients in *nf* defining a relative extension L of *nf*, or a pseudo-basis x of such an extension as output for example by `rnfpseudobasis`, computes another pseudo-basis (A, I) (not in HNF in general) such that all the ideals of I except perhaps the last one are equal to the ring of integers of *nf*, and outputs the four-component row vector $[A, I, D, d]$ as in `rnfpseudobasis`. The name of this function comes from the fact that the ideal class of the last ideal of I, which is well defined, is the Steinitz class of the $\mathbf{Z}_K$-module $\mathbf{Z}_L$ (its image in $SK_0(\mathbf{Z}_K)$).

The library syntax is **rnfsteinitz**(*nf*, *x*).

3.6.145 subgrouplist(*bnr*, {*bound*}, {*flag* = 0}): *bnr* being as output by `bnrinit` or a list of cyclic components of a finite Abelian group G, outputs the list of subgroups of G. Subgroups are given as HNF left divisors of the SNF matrix corresponding to G.

Warning: the present implementation cannot treat a group G where any cyclic factor has more than 2^{31}, resp. 2^{63} elements on a 32-bit, resp. 64-bit architecture. `forsubgroup` is a bit more general and can handle G if all p-Sylow subgroups of G satisfy the condition above.

If *flag* = 0 (default) and *bnr* is as output by `bnrinit`(,,1), gives only the subgroups whose modulus is the conductor. Otherwise, the modulus is not taken into account.

If *bound* is present, and is a positive integer, restrict the output to subgroups of index less than *bound*. If *bound* is a vector containing a single positive integer B, then only subgroups of index exactly equal to B are computed. For instance

```
? subgrouplist([6,2])
%1 = [[6, 0; 0, 2], [2, 0; 0, 2], [6, 3; 0, 1], [2, 1; 0, 1], [3, 0; 0, 2],
      [1, 0; 0, 2], [6, 0; 0, 1], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],3)    \\ index less than 3
%2 = [[2, 1; 0, 1], [1, 0; 0, 2], [2, 0; 0, 1], [3, 0; 0, 1], [1, 0; 0, 1]]
? subgrouplist([6,2],[3])  \\ index 3
%3 = [[3, 0; 0, 1]]
? bnr = bnrinit(bnfinit(x), [120,[1]], 1);
? L = subgrouplist(bnr, [8]);
```

In the last example, L corresponds to the 24 subfields of $\mathbf{Q}(\zeta_{120})$, of degree 8 and conductor 120∞ (by setting *flag*, we see there are a total of 43 subgroups of degree 8).

```
? vector(#L, i, galoissubcyclo(bnr, L[i]))
```

will produce their equations. (For a general base field, you would have to rely on `bnrstark`, or `rnfkummer`.)

The library syntax is **subgrouplist0**(*bnr*, *bound*, *flag*), where *flag* is a long integer, and an omitted *bound* is coded by `NULL`.

3.6.146 zetak($znf, x, \{flag = 0\}$): *znf* being a number field initialized by `zetakinit` (*not* by `nfinit`), computes the value of the Dedekind zeta function of the number field at the complex number x. If $flag = 1$ computes Dedekind Λ function instead (i.e. the product of the Dedekind zeta function by its gamma and exponential factors).

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 less significant digits than expected. Computing with 28 eight digits of relative accuracy, we have

```
? zeta(3)
%1 = 1.2020569031595942853997381615
? zeta(3-1e-20)
%2 = 1.2020569031595942854017194240
? zetak(zetakinit(x), 3-1e-20)
%3 = 1.2020569031595942854017205290  \\ the last 4 digits are wrong
? zetak(zetakinit(x), 3-1e-28)
%4 = 1.2020569140584772764962002780  \\ the last 20 digits are wrong
? zetak(zetakinit(x), 3-1e-40)
%5 = -1989629366932171.6339040216900 \\ junk
```

The library syntax is **glambdak**($znf, x, prec$) or **gzetak**($znf, x, prec$).

3.6.147 zetakinit(x): computes a number of initialization data concerning the number field defined by the polynomial x so as to be able to compute the Dedekind zeta and lambda functions (respectively `zetak`(x) and `zetak`($x, 1$)). This function calls in particular the `bnfinit` program. The result is a 9-component vector v whose components are very technical and cannot really be used by the user except through the `zetak` function. The only component which can be used if it has not been computed already is $v[1][4]$ which is the result of the `bnfinit` call.

This function is very inefficient and should be rewritten. It needs to computes millions of coefficients of the corresponding Dirichlet series if the precision is big. Unless the discriminant is small it will not be able to handle more than 9 digits of relative precision. For instance, `zetakinit(x^8 - 2)` needs 440MB of memory at default precision.

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

3.7 Polynomials and power series.

We group here all functions which are specific to polynomials or power series. Many other functions which can be applied on these objects are described in the other sections. Also, some of the functions described here can be applied to other types.

3.7.1 O(a^b): p-adic (if a is an integer greater or equal to 2) or power series zero (in all other cases), with precision given by b.

The library syntax is **ggrandocp**(a, b), where b is a `long`.

3.7.2 deriv($x, \{v\}$): derivative of x with respect to the main variable if v is omitted, and with respect to v otherwise. x can be any type except polmod. 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.

The library syntax is **deriv**(x, v), where v is a `long`, and an omitted v is coded as -1. When x is a `t_POL`, `derivpol`(x) is a shortcut for `deriv`($x, -1$).

3.7.3 eval(x): replaces in x the formal variables by the values that have been assigned to them after the creation of x. This is mainly useful in GP, and not in library mode. Do not confuse this with substitution (see `subst`). Applying this function to a character string yields the output from the corresponding GP command, as if directly input from the keyboard (see Section 2.6.6).

The library syntax is **geval**(x). The more basic functions **poleval**(q, x), **qfeval**(q, x), and **hqfeval**(q, x) evaluate q at x, where q is respectively assumed to be a polynomial, a quadratic form (a symmetric matrix), or an Hermitian form (an Hermitian complex matrix).

3.7.4 factorpadic($pol, p, r, \{flag = 0\}$): p-adic factorization of the polynomial pol to precision r, the result being a two-column matrix as in `factor`. The factors are normalized so that their leading coefficient is a power of p. r must be strictly larger than the p-adic valuation of the discriminant of pol for the result to make any sense. The method used is a modified version of the round 4 algorithm of Zassenhaus.

If $flag = 1$, use an algorithm due to Buchmann and Lenstra, which is usually less efficient.

The library syntax is **factorpadic4**(pol, p, r), where r is a `long` integer.

3.7.5 intformal($x, \{v\}$): formal integration of x with respect to the main variable if v is omitted, with respect to the variable v otherwise. Since PARI does not know about "abstract" logarithms (they are immediately evaluated, if only to a power series), logarithmic terms in the result will yield an error. x can be of any type. When x is a rational function, it is assumed that the base ring is an integral domain of characteristic zero.

The library syntax is **integ**(x, v), where v is a `long` and an omitted v is coded as -1.

3.7.6 padicappr(pol, a): vector of p-adic roots of the polynomial pol congruent to the p-adic number a modulo p (or modulo 4 if $p = 2$), 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 $\mathbf{Q}_p$) or can be an element of a finite extension of $\mathbf{Q}_p$, in which case it is of type `t_POLMOD`, where at least one of the coefficients of the polmod is a p-adic number. 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 **apprgen9**(pol, a), but if a is known to be simply a p-adic number (type `t_PADIC`), the syntax **apprgen**(pol, a) can be used.

3.7.7 polcoeff$(x, s, \{v\})$: coefficient of degree s of the polynomial x, with respect to the main variable if v is omitted, with respect to v otherwise. Also applies to power series, scalars (polynomial of degree 0), and to rational functions provided the denominator is a monomial.

The library syntax is **polcoeff0**(x, s, v), where v is a `long` and an omitted v is coded as -1. Also available is **truecoeff**(x, v).

3.7.8 poldegree$(x, \{v\})$: degree of the polynomial x in the main variable if v is omitted, in the variable v otherwise. This is to be understood as follows:

The degree of 0 is $-$`VERYBIGINT` by convention (`VERYBIGINT` is $2^{31} - 1$ for 32-bit machines or $2^{63} - 1$ for 64-bit machines).

When x is a non-zero scalar, its degree is 0. When x is non-zero polynomial or rational function, it is the ordinary degree of x. Return an error otherwise.

The library syntax is **poldegree**(x, v), where v and the result are `long`s (and an omitted v is coded as -1). Also available is **degree**(x), which is equivalent to `poldegree(`x`,-1)`.

3.7.9 polcyclo$(n, \{v = x\})$: n-th cyclotomic polynomial, in variable v (x by default). The integer n must be positive.

The library syntax is **cyclo**(n, v), where n and v are `long` integers (v is a variable number, usually obtained through `varn`).

3.7.10 poldisc$(pol, \{v\})$: discriminant of the polynomial *pol* in the main variable is v is omitted, in v otherwise. The algorithm used is the subresultant algorithm.

The library syntax is **poldisc0**(x, v). Also available is **discsr**(x), equivalent to `poldisc0(x,-1)`.

3.7.11 poldiscreduced(f): reduced discriminant vector of the (integral, monic) polynomial f. This is the vector of elementary divisors of $\mathbf{Z}[\alpha]/f'(\alpha)\mathbf{Z}[\alpha]$, where α is a root of the polynomial f. The components of the result are all positive, and their product is equal to the absolute value of the discriminant of f.

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

3.7.12 polhensellift(x, y, p, e): given a prime p, an integral polynomial x whose leading coefficient is a p-unit, a vector y of integral polynomials that are pairwise relatively prime modulo p, and whose product is congruent to x modulo p, lift the elements of y to polynomials whose product is congruent to x modulo p^e.

The library syntax is **polhensellift**(x, y, p, e) where e must be a `long`.

3.7.13 polinterpolate$(xa, \{ya\}, \{v = x\}, \{\&e\})$: given the data vectors xa and ya of the same length n (xa containing the x-coordinates, and ya the corresponding y-coordinates), this function finds the interpolating polynomial passing through these points and evaluates it at v. If ya is omitted, return the polynomial interpolating the $(i, xa[i])$. If present, e will contain an error estimate on the returned value.

The library syntax is **polint**$(xa, ya, v, \&e)$, where e will contain an error estimate on the returned value.

3.7.14 polisirreducible(pol): *pol* being a polynomial (univariate in the present version 2.2.7), returns 1 if *pol* is non-constant and irreducible, 0 otherwise. Irreducibility is checked over the smallest base field over which *pol* seems to be defined.

The library syntax is **gisirreducible**(pol).

3.7.15 pollead($x, \{v\}$): leading coefficient of the polynomial or power series x. This is computed with respect to the main variable of x if v is omitted, with respect to the variable v otherwise.

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

3.7.16 pollegendre($n, \{v = x\}$): creates the n^{th} Legendre polynomial, in variable v.

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

3.7.17 polrecip(pol): reciprocal polynomial of *pol*, i.e. the coefficients are in reverse order. *pol* must be a polynomial.

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

3.7.18 polresultant($x, y, \{v\}, \{flag = 0\}$): resultant of the two polynomials x and y with exact entries, with respect to the main variables of x and y if v is omitted, with respect to the variable v otherwise. The algorithm used is the subresultant algorithm by default.

If $flag = 1$, uses the determinant of Sylvester's matrix instead (here x and y may have non-exact coefficients).

If $flag = 2$, uses Ducos's modified subresultant algorithm. It should be much faster than the default if the coefficient ring is complicated (e.g multivariate polynomials or huge coefficients), and slightly slower otherwise.

The library syntax is **polresultant0**($x, y, v, flag$), where v is a `long` and an omitted v is coded as -1. Also available are **subres**(x, y) ($flag = 0$) and **resultant2**(x, y) ($flag = 1$).

3.7.19 polroots($pol, \{flag = 0\}$): complex roots of the polynomial *pol*, given as a column vector where each root is repeated according to its multiplicity. The precision is given as for transcendental functions: under GP it is kept in the variable `realprecision` and is transparent to the user, but it must be explicitly given as a second argument in library mode.

The algorithm used is a modification of A. Schönhage's root-finding algorithm, due to and implemented by X. Gourdon. Barring bugs, it is guaranteed to converge and to give the roots to the required accuracy.

If $flag = 1$, use a variant of the Newton-Raphson method, which is *not* guaranteed to converge, but is rather fast. If you get the messages "too many iterations in roots" or "INTERNAL ERROR: incorrect result in roots", use the default algorithm. This used to be the default root-finding function in PARI until version 1.39.06.

The library syntax is **roots**($pol, prec$) or **rootsold**($pol, prec$).

3.7.20 polrootsmod$(pol, p, \{flag = 0\})$: row vector of roots modulo p of the polynomial *pol*. The particular non-prime value $p = 4$ is accepted, mainly for 2-adic computations. Multiple roots are *not* repeated.

If $p < 100$, you may try setting $flag = 1$, which uses a naive search. In this case, multiple roots *are* repeated with their order of multiplicity.

The library syntax is **rootmod**(pol, p) ($flag = 0$) or **rootmod2**(pol, p) ($flag = 1$).

3.7.21 polrootspadic(pol, p, r): row vector of p-adic roots of the polynomial *pol* with p-adic precision equal to r. Multiple roots are *not* repeated. p is assumed to be a prime, and *pol* to be non-zero modulo p.

The library syntax is **rootpadic**(pol, p, r), where r is a `long`.

3.7.22 polsturm$(pol, \{a\}, \{b\})$: number of real roots of the real polynomial *pol* in the interval $]a, b]$, using Sturm's algorithm. a (resp. b) is taken to be $-\infty$ (resp. $+\infty$) if omitted.

The library syntax is **sturmpart**(pol, a, b). Use `NULL` to omit an argument. **sturm**`(`*pol*`)` is equivalent to **sturmpart**`(`*pol*`,NULL,NULL)`. The result is a `long`.

3.7.23 polsubcyclo$(n, d, \{v = x\})$: gives polynomials (in variable v) defining the sub-Abelian extensions of degree d of the cyclotomic field $\mathbf{Q}(\zeta_n)$, where $d \mid \phi(n)$.

If there is exactly one such extension the output is a polynomial, else it is a vector of polynomials, eventually empty.

To be sure to get a vector, you can use `concat([],polsubcyclo(n,d))`

The function `galoissubcyclo` allows to specify more closely which sub-Abelian extension should be computed.

The library syntax is **polsubcyclo**(n, d, v), where n, d and v are `long` and v is a variable number. When $(\mathbf{Z}/n\mathbf{Z})^*$ is cyclic, you can use **subcyclo**(n, d, v), where n, d and v are `long` and v is a variable number.

3.7.24 polsylvestermatrix(x, y): forms the Sylvester matrix corresponding to the two polynomials x and y, where the coefficients of the polynomials are put in the columns of the matrix (which is the natural direction for solving equations afterwards). The use of this matrix can be essential when dealing with polynomials with inexact entries, since polynomial Euclidean division doesn't make much sense in this case.

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

3.7.25 polsym(x, n): creates the vector of the symmetric powers of the roots of the polynomial x up to power n, using Newton's formula.

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

3.7.26 poltchebi$(n, \{v = x\})$: creates the n^{th} Chebyshev polynomial, in variable v.

The library syntax is **tchebi**(n, v), where n and v are `long` integers (v is a variable number).

3.7.27 polzagier(n, m): creates Zagier's polynomial $P_n^{(m)}$ used in the functions `sumalt` and `sumpos` (with *flag* $= 1$). One must have $m \leq n$. The exact definition can be found in "Convergence acceleration of alternating series", Cohen et al., Experiment. Math., vol. 9, 2000, pp. 3–12.

The library syntax is **polzagreel**$(n, m, prec)$ if the result is only wanted as a polynomial with real coefficients to the precision *prec*, or **polzag**(n, m) if the result is wanted exactly, where n and m are `long`s.

3.7.28 serconvol(x, y): convolution (or Hadamard product) of the two power series x and y; in other words if $x = \sum a_k * X^k$ and $y = \sum b_k * X^k$ then `serconvol`$(x, y) = \sum a_k * b_k * X^k$.

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

3.7.29 serlaplace(x): x must be a power series with only non-negative exponents. If $x = \sum (a_k/k!) * X^k$ then the result is $\sum a_k * X^k$.

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

3.7.30 serreverse(x): reverse power series (i.e. x^{-1}, not $1/x$) of x. x must be a power series whose valuation is exactly equal to one.

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

3.7.31 subst(x, y, z): replace the simple variable y by the argument z in the "polynomial" expression x. Every type is allowed for x, but if it is not a genuine polynomial (or power series, or rational function), the substitution will be done as if the scalar components were polynomials of degree zero. In particular, beware that:

```
? subst(1, x, [1,2; 3,4])
%1 =
[1 0]

[0 1]

? subst(1, x, Mat([0,1]))
  ***   forbidden substitution by a non square matrix
```

If x is a power series, z must be either a polynomial, a power series, or a rational function.

The "variable" y is in fact allowed to be any polynomial, in which case, the substitution is done as per the following script:

```
subst_poly(pol, from, to) =
{ local(t = 'subst_poly_t, M = from - t);

  subst(lift(Mod(pol,M), variable(M)), t, to)
}
```

For instance

```
? subst(x^4 + x^2 + 1, x^2, y)
%1 = y^2 + y + 1
? subst(x^4 + x^2 + 1, x^3, y)
%2 = x^2 + y*x + 1
? subst(x^4 + x^2 + 1, (x+1)^2, y)
%3 = (-4*y - 6)*x + (y^2 + 3*y - 3)
```

The library syntax is **gsubst**(x, v, z), where v is the number of the variable y for regular usage. Also available is `gsubst0`(x, y, z) where y is a `GEN` polynomial.

3.7.32 taylor(x, y): Taylor expansion around 0 of x with respect to the simple variable y. x can be of any reasonable type, for example a rational function. The number of terms of the expansion is transparent to the user under GP, but must be given as a second argument in library mode.

The library syntax is **tayl**(x, y, n), where the `long` integer n is the desired number of terms in the expansion.

3.7.33 thue$(tnf, a, \{sol\})$: solves the equation $P(x, y) = a$ in integers x and y, where *tnf* was created with `thueinit`(P). *sol*, if present, contains the solutions of Norm$(x) = a$ modulo units of positive norm in the number field defined by P (as computed by `bnfisintnorm`). If *tnf* was computed without assuming GRH (*flag* $= 1$ in `thueinit`), the result is unconditional. For instance, here's how to solve the Thue equation $x^{13} - 5y^{13} = -4$:

```
? tnf = thueinit(x^13 - 5);
? thue(tnf, -4)
%1 = [[1, 1]]
```

Hence, assuming GRH, the only solution is $x = 1$, $y = 1$.

The library syntax is **thue**(tnf, a, sol), where an omitted *sol* is coded as `NULL`.

3.7.34 thueinit$(P, \{flag = 0\})$: initializes the *tnf* corresponding to P. It is meant to be used in conjunction with `thue` to solve Thue equations $P(x, y) = a$, where a is an integer. If *flag* is non-zero, certify the result unconditionnally, Otherwise, assume GRH, this being much faster of course.

The library syntax is **thueinit**$(P, flag, prec)$.

3.8 Vectors, matrices, linear algebra and sets.

Note that most linear algebra functions operating on subspaces defined by generating sets (such as `mathnf`, `qflll`, etc.) take matrices as arguments. As usual, the generating vectors are taken to be the *columns* of the given matrix.

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 (it's not even guaranteed to be irreducible!). One can check the closeness either by a polynomial evaluation or substitution, or by computing the roots of the polynomial given by algdep.

If x is p-adic, *flag* is meaningless and the algorithm LLL-reduces the "dual lattice" corresponding to the powers of x.

Otherwise, if *flag* is zero, the algorithm used is a variant of the LLL algorithm due to Hastad, Lagarias and Schnorr (STACS 1986). If the precision is too low, the routine may enter an infinite loop.

If *flag* is non-zero, use a standard LLL. *flag* then indicates a precision, which should be between 0.5 and 1.0 times the number of decimal digits to which x was computed.

The library syntax is **algdep0**$(x, k, flag, prec)$, where k and *flag* are `long`s. Also available is **algdep**$(x, k, prec)$ (*flag* $= 0$).

3.8.2 charpoly$(A, \{v = x\}, \{flag = 0\})$: characteristic polynomial of A with respect to the variable v, i.e. determinant of $v * I - A$ if A is a square matrix, determinant of the map "multiplication by A" if A is a scalar, in particular a polmod (e.g. `charpoly(I,x)=x^2+1`). Note that in the latter case, the minimal polynomial can be obtained as

```
minpoly(A)=
{
  local(y);
  y = charpoly(A);
  y / gcd(y,y')
}
```

The value of *flag* is only significant for matrices.

If *flag* $= 0$, the method used is essentially the same as for computing the adjoint matrix, i.e. computing the traces of the powers of A.

If *flag* $= 1$, uses Lagrange interpolation which is almost always slower.

If *flag* $= 2$, uses the Hessenberg form. This is faster than the default when the coefficients are integermod a prime or real numbers, but is usually slower in other base rings.

The library syntax is **charpoly0**$(A, v, flag)$, where v is the variable number. Also available are the functions **caract**(A, v) (*flag* $= 1$), **carhess**(A, v) (*flag* $= 2$), and **caradj**(A, v, pt) where, in this last case, pt is a `GEN*` which, if not equal to `NULL`, will receive the address of the adjoint matrix of A (see `matadjoint`), so both can be obtained at once.

3.8.3 concat$(x, \{y\})$: concatenation of x and y. If x or y is not a vector or matrix, it is considered as a one-dimensional vector. All types are allowed for x and y, but the sizes must be compatible. Note that matrices are concatenated horizontally, i.e. the number of rows stays the same. Using transpositions, it is easy to concatenate them vertically.

To concatenate vectors sideways (i.e. to obtain a two-row or two-column matrix), first transform the vector into a one-row or one-column matrix using the function `Mat`. 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, 2, 3, 4]~
? a[1] = Mat(a[1]); concat(a)
%3 =
[1 3]

[2 4]

? concat([1,2; 3,4], [5,6]~)
```

```
%4 =
[1 2 5]

[3 4 6]
? concat([%, [7,8]~, [1,2,3,4]])
%5 =
[1 2 5 7]

[3 4 6 8]

[1 2 3 4]
```

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

3.8.4 lindep$(x, \{flag = 0\})$:x being a vector with real or complex coefficients, finds a small integral linear combination among these coefficients.

If $flag = 0$, uses a variant of the LLL algorithm due to Hastad, Lagarias and Schnorr (STACS 1986).

If $flag > 0$, uses the LLL algorithm. *flag* is a parameter which should be between one half the number of decimal digits of precision and that number (see `algdep`).

If $flag < 0$, 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 library syntax is **lindep0**$(x, flag, prec)$. Also available is **lindep**$(x, prec)$ ($flag = 0$).

3.8.5 listcreate(n): creates an empty list of maximal length n.

This function is useless in library mode.

3.8.6 listinsert$(list, x, n)$: inserts the object x at position n in *list* (which must be of type `t_LIST`). All the remaining elements of *list* (from position $n + 1$ onwards) are shifted to the right. This and `listput` are the only commands which enable you to increase a list's effective length (as long as it remains under the maximal length specified at the time of the `listcreate`).

This function is useless in library mode.

3.8.7 listkill$(list)$: kill *list*. This deletes all elements from *list* and sets its effective length to 0. The maximal length is not affected.

This function is useless in library mode.

3.8.8 listput$(list, x, \{n\})$: sets the n-th element of the list *list* (which must be of type `t_LIST`) equal to x. If n is omitted, or greater than the list current effective length, just appends x. This and `listinsert` are the only commands which enable you to increase a list's effective length (as long as it remains under the maximal length specified at the time of the `listcreate`).

If you want to put an element into an occupied cell, i.e. if you don't want to change the effective length, you can consider the list as a vector and use the usual `list[n] = x` construct.

This function is useless in library mode.

3.8.9 listsort($list, \{flag = 0\}$): sorts *list* (which must be of type `t_LIST`) in place. If *flag* is non-zero, suppresses all repeated coefficients. This is much faster than the `vecsort` command since no copy has to be made.

This function is useless in library mode.

3.8.10 matadjoint(x): adjoint matrix of x, i.e. the matrix y of cofactors of x, satisfying $x * y = \det(x) * \mathrm{Id}$. x must be a (non-necessarily invertible) square matrix.

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

3.8.11 matcompanion(x): the left companion matrix to the polynomial x.

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

3.8.12 matdet($x, \{flag = 0\}$): determinant of x. x must be a square matrix.

If $flag = 0$, uses Gauss-Bareiss.

If $flag = 1$, uses classical Gaussian elimination, which is better when the entries of the matrix are reals or integers for example, but usually much worse for more complicated entries like multivariate polynomials.

The library syntax is **det**(x) ($flag = 0$) and **det2**(x) ($flag = 1$).

3.8.13 matdetint(x): x being an $m \times n$ matrix with integer coefficients, this function computes a *multiple* of the determinant of the lattice generated by the columns of x if it is of rank m, and returns zero otherwise. This function can be useful in conjunction with the function `mathnfmod` which needs to know such a multiple. To obtain the exact determinant (assuming the rank is maximal), you can compute `matdet(mathnfmod(x, matdetint(x)))`.

Note that as soon as one of the dimensions gets large (m or n is larger than 20, say), it will often be much faster to use `mathnf(x, 1)` or `mathnf(x, 4)` directly.

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

3.8.14 matdiagonal(x): x being a vector, creates the diagonal matrix whose diagonal entries are those of x.

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

3.8.15 mateigen(x): gives the eigenvectors of x as columns of a matrix.

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

3.8.16 mathess(x): Hessenberg form of the square matrix x.

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

3.8.17 mathilbert(x): x being a `long`, creates the Hilbert matrix of order x, i.e. the matrix whose coefficient (i,j) is $1/(i + j - 1)$.

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

3.8.18 mathnf$(x, \{flag = 0\})$: if x is a (not necessarily square) matrix, finds the *upper triangular* Hermite normal form of x. If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x.

If $flag = 0$, uses the naive algorithm. This should never be used if the dimension is at all large (larger than 10, say). It is recommanded to use either `mathnfmod(x, matdetint(x))` (when x has maximal rank) or `mathnf(x, 1)`. Note that the latter is in general faster than `mathnfmod`, and also provides a base change matrix.

If $flag = 1$, uses Batut's algorithm, which is much faster than the default. Outputs a two-component row vector $[H, U]$, where H is the *upper triangular* Hermite normal form of x defined as above, and U is the unimodular transformation matrix such that $xU = [0|H]$. U has in general huge coefficients, in particular when the kernel is large.

If $flag = 3$, uses Batut's algorithm, but outputs $[H, U, P]$, such that H and U are as before and P is a permutation of the rows such that P applied to xU gives H. The matrix U is smaller than with $flag = 1$, but may still be large.

If $flag = 4$, as in case 1 above, but uses a heuristic variant of LLL reduction along the way. The matrix U is in general close to optimal (in terms of smallest L_2 norm), but the reduction is slower than in case 1.

The library syntax is **mathnf0**$(x, flag)$. Also available are **hnf**(x) ($flag = 0$) and **hnfall**(x) ($flag = 1$). To reduce *huge* (say 400×400 and more) relation matrices (sparse with small entries), you can use the pair `hnfspec` / `hnfadd`. Since this is rather technical and the calling interface may change, they are not documented yet. Look at the code in `basemath/alglin1.c`.

3.8.19 mathnfmod(x, d): if x is a (not necessarily square) matrix of maximal rank with integer entries, and d is a multiple of the (non-zero) determinant of the lattice spanned by the columns of x, finds the *upper triangular* Hermite normal form of x.

If the rank of x is equal to its number of rows, the result is a square matrix. In general, the columns of the result form a basis of the lattice spanned by the columns of x. This is much faster than `mathnf` when d is known.

The library syntax is **hnfmod**(x, d).

3.8.20 mathnfmodid(x, d): outputs the (upper triangular) Hermite normal form of x concatenated with d times the identity matrix.

The library syntax is **hnfmodid**(x, d).

3.8.21 matid(n): creates the $n \times n$ identity matrix.

The library syntax is **idmat**(n) where n is a `long`.

Related functions are **gscalmat**(x, n), which creates x times the identity matrix (x being a `GEN` and n a `long`), and **gscalsmat**(x, n) which is the same when x is a `long`.

3.8.22 matimage$(x, \{flag = 0\})$: gives a basis for the image of the matrix x as columns of a matrix. A priori the matrix can have entries of any type. If $flag = 0$, use standard Gauss pivot. If $flag = 1$, use `matsupplement`.

The library syntax is **matimage0**$(x, flag)$. Also available is **image**(x) ($flag = 0$).

3.8.23 matimagecompl(x): gives the vector of the column indices which are not extracted by the function `matimage`. Hence the number of components of `matimagecompl(x)` plus the number of columns of `matimage(x)` is equal to the number of columns of the matrix x.

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

3.8.24 matindexrank(x): x being a matrix of rank r, gives two vectors y and z of length r giving a list of rows and columns respectively (starting from 1) such that the extracted matrix obtained from these two vectors using `vecextract`(x, y, z) is invertible.

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

3.8.25 matintersect(x, y): x and y being two matrices with the same number of rows each of whose columns are independent, finds a basis of the **Q**-vector space equal to the intersection of the spaces spanned by the columns of x and y respectively. See also the function `idealintersect`, which does the same for free **Z**-modules.

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

3.8.26 matinverseimage(x, y): gives a column vector belonging to the inverse image of the column vector y by the matrix x if one exists, the empty vector otherwise. To get the complete inverse image, it suffices to add to the result any element of the kernel of x obtained for example by `matker`.

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

3.8.27 matisdiagonal(x): returns true (1) if x is a diagonal matrix, false (0) if not.

The library syntax is **isdiagonal**(x), and this returns a `long` integer.

3.8.28 matker$(x, \{flag = 0\})$: gives a basis for the kernel of the matrix x as columns of a matrix. A priori the matrix can have entries of any type.

If x is known to have integral entries, set $flag = 1$.

Note: The library function `FpM_ker`(x, p), where x has integer entries *reduced mod* p and p is prime, is equivalent to, but orders of magnitude faster than, `matker(x*Mod(1,p))` and needs much less stack space. To use it under GP, type `install(FpM_ker, GG)` first.

The library syntax is **matker0**$(x, flag)$. Also available are **ker**(x) ($flag = 0$), **keri**(x) ($flag = 1$) and `FpM_ker`(x, p).

3.8.29 matkerint$(x, \{flag = 0\})$: gives an LLL-reduced **Z**-basis for the lattice equal to the kernel of the matrix x as columns of the matrix x with integer entries (rational entries are not permitted).

If $flag = 0$, uses a modified integer LLL algorithm.

If $flag = 1$, uses `matrixqz`$(x, -2)$. If LLL reduction of the final result is not desired, you can save time using `matrixqz(matker(x),-2)` instead.

The library syntax is **matkerint0**$(x, flag)$. Also available is **kerint**(x) ($flag = 0$).

3.8.30 matmuldiagonal(x, d): product of the matrix x by the diagonal matrix whose diagonal entries are those of the vector d. Equivalent to, but much faster than $x * $ `matdiagonal`(d).

The library syntax is **matmuldiagonal**(x, d).

3.8.31 matmultodiagonal(x, y): product of the matrices x and y assuming that the result is a diagonal matrix. Much faster than $x * y$ in that case. The result is undefined if $x * y$ is not diagonal.

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

3.8.32 matpascal$(x, \{q\})$: creates as a matrix the lower triangular Pascal triangle of order $x + 1$ (i.e. with binomial coefficients up to x). If q is given, compute the q-Pascal triangle (i.e. using q-binomial coefficients).

The library syntax is **matqpascal**(x, q), where x is a `long` and q = `NULL` is used to omit q. Also available is **matpascal**(x).

3.8.33 matrank(x): rank of the matrix x.

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

3.8.34 matrix$(m, n, \{X\}, \{Y\}, \{expr = 0\})$: creation of the $m \times n$ matrix whose coefficients are given by the expression *expr*. There are two formal parameters in *expr*, the first one (X) corresponding to the rows, the second (Y) to the columns, and X goes from 1 to m, Y goes from 1 to n. If one of the last 3 parameters is omitted, fill the matrix with zeroes.

The library syntax is **matrice**(`GEN nlig,GEN ncol,entree *e1,entree *e2,char *expr`).

3.8.35 matrixqz(x, p): x being an $m \times n$ matrix with $m \geq n$ with rational or integer entries, this function has varying behaviour depending on the sign of p:

If $p \geq 0$, x is assumed to be of maximal rank. This function returns a matrix having only integral entries, having the same image as x, such that the GCD of all its $n \times n$ subdeterminants is equal to 1 when p is equal to 0, or not divisible by p otherwise. Here p must be a prime number (when it is non-zero). However, if the function is used when p has no small prime factors, it will either work or give the message "impossible inverse modulo" and a non-trivial divisor of p.

If $p = -1$, this function returns a matrix whose columns form a basis of the lattice equal to $\mathbf{Z}^n$ intersected with the lattice generated by the columns of x.

If $p = -2$, returns a matrix whose columns form a basis of the lattice equal to $\mathbf{Z}^n$ intersected with the $\mathbf{Q}$-vector space generated by the columns of x.

The library syntax is **matrixqz0**(x, p).

3.8.36 matsize(x): x being a vector or matrix, returns a row vector with two components, the first being the number of rows (1 for a row vector), the second the number of columns (1 for a column vector).

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

3.8.37 matsnf$(X, \{flag = 0\})$: if X is a (singular or non-singular) matrix outputs the vector of elementary divisors of X (i.e. the diagonal of the Smith normal form of X).

The binary digits of *flag* mean:

1 (complete output): if set, outputs $[U, V, D]$, where U and V are two unimodular matrices such that UXV is the diagonal matrix D. Otherwise output only the diagonal of D.

2 (generic input): if set, allows polynomial entries, in which case the input matrix must be square. Otherwise, assume that X has integer coefficients with arbitrary shape.

4 (cleanup): if set, cleans up the output. This means that elementary divisors equal to 1 will be deleted, i.e. outputs a shortened vector D' instead of D. If complete output was required, returns $[U', V', D']$ so that $U'XV' = D'$ holds. If this flag is set, X is allowed to be of the form D or $[U, V, D]$ as would normally be output with the cleanup flag unset.

The library syntax is **matsnf0**$(X, flag)$. Also available is **smith**(X) ($flag = 0$).

3.8.38 matsolve(x, y): x being an invertible matrix and y a column vector, finds the solution u of $x * u = y$, using Gaussian elimination. This has the same effect as, but is a bit faster, than $x^{-1} * y$.

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

3.8.39 matsolvemod$(m, d, y, \{flag = 0\})$: m being any integral matrix, d a vector of positive integer moduli, and y an integral column vector, gives a small integer solution to the system of congruences $\sum_i m_{i,j} x_j \equiv y_i \pmod{d_i}$ if one exists, otherwise returns zero. Shorthand notation: y (resp. d) can be given as a single integer, in which case all the y_i (resp. d_i) above are taken to be equal to y (resp. d).

```
? m = [1,2;3,4];
? matsolvemod(m, [3,4], [1,2]~)
%2 = [-2, 0]~
? matsolvemod(m, 3, 1) \\ m X = [1,1]~ over F_3
%3 = [-1, 1]~
```

If $flag = 1$, all solutions are returned in the form of a two-component row vector $[x, u]$, where x is a small integer solution to the system of congruences and u is a matrix whose columns give a basis of the homogeneous system (so that all solutions can be obtained by adding x to any linear combination of columns of u). If no solution exists, returns zero.

The library syntax is **matsolvemod0**$(m, d, y, flag)$. Also available are **gaussmodulo**(m, d, y) ($flag = 0$) and **gaussmodulo2**(m, d, y) ($flag = 1$).

3.8.40 matsupplement(x): assuming that the columns of the matrix x are linearly independent (if they are not, an error message is issued), finds a square invertible matrix whose first columns are the columns of x, i.e. supplement the columns of x to a basis of the whole space.

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

3.8.41 mattranspose(x) or x~: transpose of x. This has an effect only on vectors and matrices.

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

3.8.42 qfgaussred(q): decomposition into squares of the quadratic form represented by the symmetric matrix q. The result is a matrix whose diagonal entries are the coefficients of the squares, and the non-diagonal entries represent the bilinear forms. More precisely, if (a_{ij}) denotes the output, one has

$$q(x) = \sum_i a_{ii}(x_i + \sum_{j>i} a_{ij}x_j)^2$$

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

3.8.43 qfjacobi(x): x being a real symmetric matrix, this gives a vector having two components: the first one is the vector of eigenvalues of x, the second is the corresponding orthogonal matrix of eigenvectors of x. The method used is Jacobi's method for symmetric matrices.

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

3.8.44 qflll$(x, \{flag = 0\})$: LLL algorithm applied to the *columns* of the (not necessarily square) matrix x. The columns of x must however be linearly independent, unless specified otherwise below. The result is a transformation matrix T such that $x \cdot T$ is an LLL-reduced basis of the lattice generated by the column vectors of x.

If $flag = 0$ (default), the computations are done with real numbers (i.e. not with rational numbers), using Householder matrices for orthogonalization (as presently programmed: slow but stable).

If $flag = 1$, it is assumed that the corresponding Gram matrix is integral. The computation is done entirely with integers and the algorithm is both accurate and quite fast. In this case, x needs not be of maximal rank, but if it is not, T will not be square.

If $flag = 2$, similar to case 1, except x should be an integer matrix whose columns are linearly independent. The lattice generated by the columns of x is first partially reduced before applying the LLL algorithm. [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 can be significantly faster than $flag = 1$ when one row is huge compared to the other rows.

If $flag = 4$, x is assumed to have integral entries, but needs not be of maximal rank. The result is a two-component vector of matrices: the columns of the first matrix represent a basis of the integer kernel of x (not necessarily LLL-reduced) and the second matrix is the transformation matrix T such that $x \cdot T$ is an LLL-reduced $\mathbf{Z}$-basis of the image of the matrix x.

If $flag = 5$, case as case 4, but x may have polynomial coefficients.

If $flag = 8$, same as case 0, where x may have polynomial coefficients.

The library syntax is **qflll0**$(x, flag, prec)$. Also available are **lll**$(x, prec)$ ($flag = 0$), **lllint**(x) ($flag = 1$), and **lllkerim**(x) ($flag = 4$).

3.8.45 qflllgram$(x, \{flag = 0\})$: same as `qflll`, except that the matrix x is the Gram matrix of the lattice vectors, and not the coordinates of the vectors themselves. In particular, x must now be a square symmetric real matrix, corresponding to a positive definite quadratic form. The result is again the transformation matrix T which gives (as columns) the coefficients with respect to the initial basis vectors. The flags have more or less the same meaning, but some are missing. In brief:

flag $= 0$: numerically unstable in the present version 2.2.7.

flag $= 1$: x has integer entries, the computations are all done in integers.

flag $= 4$: x has integer entries, gives the kernel and reduced image.

flag $= 5$: same as 4 for generic x.

The library syntax is **qflllgram0**$(x, flag, prec)$. Also available are **lllgram**$(x, prec)$ (*flag* $= 0$), **lllgramint**(x) (*flag* $= 1$), and **lllgramkerim**(x) (*flag* $= 4$).

3.8.46 qfminim$(x, b, m, \{flag = 0\})$: x being a square and symmetric matrix representing a positive definite quadratic form, this function deals with the minimal vectors of x, depending on *flag*.

If *flag* $= 0$ (default), seeks vectors of square norm less than or equal to b (for the norm defined by x), and at most $2m$ of these vectors. The result is a three-component vector, the first component being the number of vectors, 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).

If *flag* $= 1$, ignores m and returns the first vector whose norm is less than b.

In both these cases, x *is assumed to have integral entries*, and the function searches for the minimal non-zero vectors whenever $b = 0$.

If *flag* $= 2$, x can have non integral real entries, but $b = 0$ is now meaningless (uses Fincke-Pohst algorithm).

The library syntax is **qfminim0**$(x, b, m, flag, prec)$, also available are **minim**(x, b, m) (*flag* $= 0$), **minim2**(x, b, m) (*flag* $= 1$), and finally **fincke_pohst**$(x, b, m, prec)$ (*flag* $= 2$).

3.8.47 qfperfection(x): x being a square and symmetric matrix with integer entries representing a positive definite quadratic form, outputs the perfection rank of the form. That is, gives the rank of the family of the s symmetric matrices $v_i v_i^t$, where s is half the number of minimal vectors and the v_i $(1 \le i \le s)$ are the minimal vectors.

As a side note to old-timers, this used to fail bluntly when x had more than 5000 minimal vectors. Beware that the computations can now be very lengthy when x has many minimal vectors.

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

3.8.48 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, $1 \le i \le B$ is half the number of vectors v such that $q(v) = i$. This routine uses a naive algorithm based on `qfminim`, and will fail if any entry becomes larger than 2^{31}.

The binary digits of *flag* mean:

- 1: count vectors of even norm from 1 to $2B$.
- 2: return a `t_VECSMALL` instead of a `t_GEN`

The library syntax is **qfrep0**$(q, B, flag)$.

3.8.49 qfsign(x): signature of the quadratic form represented by the symmetric matrix x. The result is a two-component vector.

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

3.8.50 setintersect(x, y): intersection of the two sets x and y.

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

3.8.51 setisset(x): returns true (1) if x is a set, false (0) if not. In PARI, a set is simply a row vector whose entries are strictly increasing. To convert any vector (and other objects) into a set, use the function `Set`.

The library syntax is **setisset**(x), and this returns a `long`.

3.8.52 setminus(x, y): difference of the two sets x and y, i.e. set of elements of x which do not belong to y.

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

3.8.53 setsearch($x, y, \{flag = 0\}$): searches if y belongs to the set x. If it does and *flag* is zero or omitted, returns the index j such that $x[j] = y$, otherwise returns 0. If *flag* is non-zero returns the index j where y should be inserted, and 0 if it already belongs to x (this is meant to be used in conjunction with `listinsert`).

This function works also if x is a *sorted* list (see `listsort`).

The library syntax is **setsearch**($x, y, flag$) which returns a `long` integer.

3.8.54 setunion(x, y): union of the two sets x and y.

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

3.8.55 trace(x): this applies to quite general x. If x is not a matrix, it is equal to the sum of x and its conjugate, except for polmods where it is the trace as an algebraic number.

For x a square matrix, it is the ordinary trace. If x is a non-square matrix (but not a vector), an error occurs.

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

3.8.56 vecextract($x, y, \{z\}$): extraction of components of the vector or matrix x according to y. In case x is a matrix, its components are as usual the *columns* of x. The parameter y is a component specifier, which is either an integer, a string describing a range, or a vector.

If y is an integer, it is considered as a mask: the binary bits of y are read from right to left, but correspond to taking the components from left to right. For example, if $y = 13 = (1101)_2$ then the components 1,3 and 4 are extracted.

If y is a vector, which must have integer entries, these entries correspond to the component numbers to be extracted, in the order specified.

If y is a string, it can be

• a single (non-zero) index giving a component number (a negative index means we start counting from the end).

• a range of the form "$a..b$", where a and b are indexes as above. Any of a and b can be omitted; in this case, we take as default values $a = 1$ and $b = -1$, i.e. the first and last components respectively. We then extract all components in the interval $[a, b]$, in reverse order if $b < a$.

In addition, if the first character in the string is `^`, the complement of the given set of indices is taken.

If z is not omitted, x must be a matrix. y is then the *line* specifier, and z the *column* specifier, where the component specifier is as explained above.

```
? v = [a, b, c, d, e];
? vecextract(v, 5)          \\ mask
%1 = [a, c]
? vecextract(v, [4, 2, 1])  \\ component list
%2 = [d, b, a]
? vecextract(v, "2..4")     \\ interval
%3 = [b, c, d]
? vecextract(v, "-1..-3")   \\ interval + reverse order
%4 = [e, d, c]
? vecextract(v, "^2")       \\ complement
%5 = [a, c, d, e]
? vecextract(matid(3), "2..", "..")
%6 =
[0 1 0]

[0 0 1]
```

The library syntax is **extract**(x, y) or **matextract**(x, y, z).

3.8.57 vecsort$(x, \{k\}, \{flag = 0\})$: sorts the vector x in ascending order, using the heapsort method. x must be a vector, and its components integers, reals, or fractions.

If k is present and is an integer, sorts according to the value of the k-th subcomponents of the components of x. k can also be a vector, in which case the sorting is done lexicographically according to the components listed in the vector k. For example, if $k = [2, 1, 3]$, sorting will be done with respect to the second component, and when these are equal, with respect to the first, and when these are equal, with respect to the third.

The binary digits of *flag* mean:

• 1: indirect sorting of the vector x, i.e. if x is an n-component vector, returns a permutation of $[1, 2, \ldots, n]$ which applied to the components of x sorts x in increasing order. For example, `vecextract(x, vecsort(x,,1))` is equivalent to `vecsort(x)`.

• 2: sorts x by ascending lexicographic order (as per the `lex` comparison function).

• 4: use descending instead of ascending order.

The library syntax is **vecsort0**$(x, k, flag)$. To omit k, use `NULL` instead. You can also use the simpler functions

sort(x) (= `vecsort0`$(x, \text{NULL}, 0)$).

indexsort(x) (= `vecsort0`$(x, \text{NULL}, 1)$).

lexsort(x) (= `vecsort0`$(x, \text{NULL}, 2)$).

Also available are **sindexsort** and **sindexlexsort** which return a vector of C-long integers (private type `t_VECSMALL`) v, where $v[1] \dots v[n]$ contain the indices.

3.8.58 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.

The library syntax is **vecteur**(`GEN nmax, entree *ep, char *expr`).

3.8.59 vectorsmall$(n, \{X\}, \{expr = 0\})$: creates a row vector of small integers (type `t_VECSMALL`) with n components whose components are the expression *expr* evaluated at the integer points between 1 and n. If one of the last two arguments is omitted, fill the vector with zeroes.

The library syntax is **vecteursmall**(`GEN nmax, entree *ep, char *expr`).

3.8.60 vectorv$(n, X, expr)$: as `vector`, but returns a column vector (type `t_COL`).

The library syntax is **vvecteur**(`GEN nmax, entree *ep, char *expr`).

3.9 Sums, products, integrals and similar functions.

Although the GP calculator is programmable, it is useful to have preprogrammed a number of loops, including sums, products, and a certain number of recursions. Also, a number of functions from numerical analysis like numerical integration and summation of series will be described here.

One of the parameters in these loops must be the control variable, hence a simple variable name. The last parameter can be any legal PARI expression, including of course expressions using loops. Since it is much easier to program directly the loops in library mode, these functions are mainly useful for GP programming. The use of these functions in library mode is a little tricky and its explanation will be mostly omitted, although the reader can try and figure it out by himself by checking the example given for the `sum` function. In this section we only give the library syntax, with no semantic explanation.

The letter X will always denote any simple variable name, and represents the formal parameter used in the function.

(numerical) integration: A number of Romberg-like integration methods are implemented (see `intnum` as opposed to `intformal` which we already described). 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 1e4000. 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 1e4000.

The integrand may have values belonging to a vector space over the real numbers; in particular, it can be complex-valued or vector-valued.

See also the discrete summation methods below (sharing the prefix `sum`).

3.9.1 intnum($X = a, b, expr, \{flag = 0\}$): numerical integration of *expr* (smooth in $]a, b[$), with respect to X.

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 $a \geq 1$.

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 library syntax is **intnum**(`void *E, GEN (*eval)(GEN,void*), GEN a, GEN b, long flag, long prec`). Where `eval`(x, E) returns the value of the function at x. You may store any additional information required by `eval` in E, or set it to `NULL`.

3.9.2 prod($X = a, b, expr, \{x = 1\}$): product of expression *expr*, initialized at x, the formal parameter X going from a to b. As for `sum`, the main purpose of the initialization parameter x is to force the type of the operations being performed. For example if it is set equal to the integer 1, operations will start being done exactly. If it is set equal to the real 1., they will be done using real numbers having the default precision. If it is set equal to the power series $1 + O(X^k)$ for a certain k, they will be done using power series of precision at most k. These are the three most common initializations.

As an extreme example, compare

```
? prod(i=1, 100, 1 - X^i);  \\ this has degree 5050 !!
time = 3,335 ms.
? prod(i=1, 100, 1 - X^i, 1 + O(X^101))
time = 43 ms.
%2 = 1 - X - X^2 + X^5 + X^7 - X^12 - X^15 + X^22 + X^26 - X^35 - X^40 + \
  X^51 + X^57 - X^70 - X^77 + X^92 + X^100 + O(X^101)
```

The library syntax is **produit**(`entree *ep, GEN a, GEN b, char *expr, GEN x`).

3.9.3 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**(`entree *ep, GEN a, GEN b, char *expr, long prec`).

3.9.4 prodinf($X = a, expr, \{flag = 0\}$): infinite product of expression *expr*, the formal parameter X starting at a. The evaluation stops when the relative error of the expression minus 1 is less than the default precision. The expressions must always evaluate to an element of **C**.

If *flag* $= 1$, do the product of the $(1 +$ *expr*$)$ instead.

The library syntax is **prodinf**(`entree *ep, GEN a, char *expr, long prec`) (*flag* $= 0$), or **prodinf1** with the same arguments (*flag* $= 1$).

3.9.5 solve$(X = a, b, expr)$: find a real root of expression *expr* between a and b, under the condition $expr(X = a) * expr(X = b) \leq 0$. This routine uses Brent's method and can fail miserably if *expr* is not defined in the whole of $[a, b]$ (try `solve(x=1, 2, tan(x))`.

The library syntax is **zbrent**(`void *E, GEN (*eval)(GEN, void*), GEN a, GEN b, long prec`). 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`.

3.9.6 sum$(X = a, b, expr, \{x = 0\})$: sum of expression *expr*, initialized at x, the formal parameter going from a to b. As for `prod`, the initialization parameter x may be given to force the type of the operations being performed.

As an extreme example, compare

```
? sum(i=1, 5000, 1/i); \\ rational number: denominator has 2166 digits.
time = 1,241 ms.
? sum(i=1, 5000, 1/i, 0.)
time = 158 ms.
%2 = 9.094508852984436967261245533
```

The library syntax is **somme**(`entree *ep, GEN a, GEN b, char *expr, GEN x`). This is to be used as follows: `ep` represents the dummy variable used in the expression `expr`

```
/* compute a^2 + ... + b^2 */
{
  /* define the dummy variable "i" */
  entree *ep = is_entry("i");
  /* sum for a <= i <= b */
  return somme(ep, a, b, "i^2", gzero);
}
```

3.9.7 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.

If $flag = 0$, use an algorithm of F. Villegas as modified by D. Zagier. This is much better than Euler-Van Wijngaarden's method which was used formerly.

If $flag = 1$, use a variant with slightly different polynomials. Sometimes faster.

Divergent alternating series can sometimes be summed by this method, as well as series which are not exactly alternating (see for example Section 2.6.4).

The library syntax is **sumalt**(`entree *ep, GEN a, char *expr, long` *flag*`, long prec`).

3.9.8 sumdiv$(n, X, expr)$: sum of expression *expr* over the positive divisors of n.

Arithmetic functions like `sigma` use the multiplicativity of the underlying expression to speed up the computation. In the present version 2.2.7, there is no way to indicate that *expr* is multiplicative in n, hence specialized functions should be preferred whenever possible.

The library syntax is **divsum**(`entree *ep, GEN num, char *expr`).

3.9.9 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. The expressions must always evaluate to a complex number.

The library syntax is **suminf**(`entree *ep, GEN a, char *expr, long prec`).

3.9.10 sumpos($X = a, expr, \{flag = 0\}$): numerical summation of the series *expr*, which must be a series of terms having the same sign, the formal variable X starting at a. The algorithm used is Van Wijngaarden's trick for converting such a series into an alternating one, and is quite slow.

If $flag = 1$, use slightly different polynomials. Sometimes faster.

The library syntax is **sumpos**(`entree *ep, GEN a, char *expr, long` *flag*`, long prec`).

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 felt like suggesting ideas or submitting huge patches for this section of the code. Among these, special thanks go to Klaus-Peter Nischke who suggested the recursive plotting and the forking/resizing stuff under X11, and Ilya Zakharevich who undertook a complete rewrite of the graphic code, so that most of it is now platform-independent and should be relatively easy to port or expand.

These graphic functions are either

• 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 (with somewhat more complicated flags).

• low-level plotting functions, where every drawing primitive (point, line, box, etc.) must be specified by the user. These low-level functions (called *rectplot* functions, sharing the prefix `plot`) work as follows. You have at your disposal 16 virtual windows which are filled independently, and can then be physically ORed on a single window at user-defined positions. These windows are numbered from 0 to 15, and must be initialized before being used by the function `plotinit`, which specifies the height and width of the virtual window (called a *rectwindow* in the sequel). At all times, a virtual cursor (initialized at $[0, 0]$) is associated to the window, and its current value can be obtained using the function `plotcursor`.

A number of primitive graphic objects (called *rect* objects) can then be drawn in these windows, using a default color associated to that window (which can be changed under X11, using the `plotcolor` function, black otherwise) and only the part of the object which is inside the window will be drawn, with the exception of polygons and strings which are drawn entirely (but the virtual cursor can move outside of the window). The ones sharing the prefix `plotr` draw relatively to the current position of the virtual cursor, the others use absolute coordinates. Those having the prefix `plotrecth` put in the rectwindow a large batch of rect objects corresponding to the output of the related `ploth` function.

Finally, the actual physical drawing is done using the function `plotdraw`. Note that the windows are preserved so that further drawings using the same windows at different positions or different windows can be done without extra work. If you want to erase a window (and free the corresponding memory), use the function `plotkill`. It is not possible to partially erase a window. Erase it completely, initialize it again and then fill it with the graphic objects that you want to keep.

In addition to initializing the window, you may want to have a scaled window to avoid unnecessary conversions. For this, use the function `plotscale` below. As long as this function is not called, the scaling is simply the number of pixels, the origin being at the upper left and the y-coordinates going downwards.

Note that in the present version 2.2.7 all these plotting functions (both low and high level) have been written for the X11-window system (hence also for GUI's based on X11 such as Openwindows and Motif) only, though very little code remains which is actually platform-dependent. A Suntools/Sunview, Macintosh, and an Atari/Gem port were provided for previous versions. These *may* be adapted in future releases.

Under X11/Suntools, the physical window (opened by `plotdraw` or any of the `ploth*` functions) is completely separated from GP (technically, a `fork` is done, and the non-graphical memory is immediately freed in the child process), which means you can go on working in the current GP session, without having to kill the window first. Under X11, this window can be closed, enlarged or reduced using the standard window manager functions. No zooming procedure is implemented though (yet).

• Finally, note that in the same way that `printtex` allows you to have a TEX output corresponding to printed results, the functions starting with `ps` allow you to have `PostScript` output of the plots. This will not be absolutely identical with the screen output, but will be sufficiently close. Note that you can use PostScript output even if you do not have the plotting routines enabled. The PostScript output is written in a file whose name is derived from the `psfile` default (`./pari.ps` if you did not tamper with it). Each time a new PostScript output is asked for, the PostScript output is appended to that file. Hence the user must remove this file, or change the value of `psfile`, first if he does not want unnecessary drawings from preceding sessions to appear. On the other hand, in this manner as many plots as desired can be kept in a single file.

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 much 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.

3.10.1 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.

3.10.2 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.

3.10.3 plotclip(w): 'clips' the content of rectwindow w, i.e remove all parts of the drawing that would not be visible on the screen. Together with `plotcopy` this function enables you to draw on a scratchpad before commiting the part you're interested in to the final picture.

3.10.4 plotcolor(w, c): set default color to c in rectwindow w. In present version 2.2.7, this is only implemented for X11 window system, and you only have the following palette to choose from:

1=black, 2=blue, 3=sienna, 4=red, 5=cornsilk, 6=grey, 7=gainsborough.

Note that it should be fairly easy for you to hardwire some more colors by tweaking the files `rect.h` and `plotX.c`. User-defined colormaps would be nice, and *may* be available in future versions.

3.10.5 plotcopy$(w1, w2, dx, dy)$: copy the contents of rectwindow $w1$ to rectwindow $w2$, with offset (dx, dy).

3.10.6 plotcursor(w): give as a 2-component vector the current (scaled) position of the virtual cursor corresponding to the rectwindow w.

3.10.7 plotdraw$(list)$: physically draw the rectwindows given in *list* which must be a vector whose number of components is divisible by 3. If $list = [w1, x1, y1, w2, x2, y2, \ldots]$, the windows $w1$, $w2$, etc. are physically placed with their upper left corner at physical position $(x1, y1)$, $(x2, y2)$,... respectively, and are then drawn together. Overlapping regions will thus be drawn twice, and the windows are considered transparent. Then display the whole drawing in a special window on your screen.

3.10.8 plotfile(s): set the output file for plotting output. The special filename `"-"` redirects to the same place as PARI output. This is only taken into account by the `gnuplot` interface.

3.10.9 ploth$(X = a, b, expr, \{flag = 0\}, \{n = 0\})$: high precision plot of the function $y = f(x)$ represented by the expression *expr*, x going from a to b. This opens a specific window (which is killed whenever you click on it), and returns a four-component vector giving the coordinates of the bounding box in the form $[xmin, xmax, ymin, ymax]$.

Important note: Since this may involve a lot of function calls, it is advised to keep the current precision to a minimum (e.g. 9) before calling this function.

n specifies the number of reference point on the graph (0 means use the hardwired default values, that is: 1000 for general plot, 1500 for parametric plot, and 15 for recursive plot).

If no *flag* is given, *expr* is either a scalar expression $f(X)$, in which case the plane curve $y = f(X)$ will be drawn, or a vector $[f_1(X), \ldots, f_k(X)]$, and then all the curves $y = f_i(X)$ will be drawn in the same window.

The binary digits of *flag* mean:

• 1 = `Parametric`: *parametric plot*. Here *expr* must be a vector with an even number of components. Successive pairs are then understood as the parametric coordinates of a plane curve. Each of these are then drawn.

For instance:

`ploth(X=0,2*Pi,[sin(X),cos(X)],1)` will draw a circle.

`ploth(X=0,2*Pi,[sin(X),cos(X)])` will draw two entwined sinusoidal curves.

`ploth(X=0,2*Pi,[X,X,sin(X),cos(X)],1)` will draw a circle and the line $y = x$.

• 2 = `Recursive`: *recursive plot*. If this flag is set, only *one* curve can be drawn at 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 very fast, and usually more precise than usual plot. Compare the results of

$$\texttt{ploth}(\texttt{X} = -1, 1, \texttt{sin}(1/\texttt{X}), 2) \quad \text{and} \quad \texttt{ploth}(\texttt{X} = -1, 1, \texttt{sin}(1/\texttt{X}))$$

for instance. But beware that if you are extremely unlucky, or choose too few reference points, you may draw some nice polygon bearing little resemblance to the original curve. For instance you should *never* plot recursively an odd function in a symmetric interval around 0. Try

```
ploth(x = -20, 20, sin(x), 2)
```

to see why. Hence, it's usually a good idea to try and plot the same curve with slightly different parameters.

The other values toggle various display options:

• 4 = `no_Rescale`: do not rescale plot according to the computed extrema. This is meant to be used when graphing multiple functions on a rectwindow (as a `plotrecth` call), in conjunction with `plotscale`.

• 8 = `no_X_axis`: do not print the x-axis.

• 16 = `no_Y_axis`: do not print the y-axis.

• 32 = `no_Frame`: do not print frame.

• 64 = `no_Lines`: only plot reference points, do not join them.

• 128 = $\texttt{Points}_\texttt{t}\texttt{oo}$: plot both lines and points.

• 256 = `Splines`: use splines to interpolate the points.

• 512 = `no_X_ticks`: plot no x-ticks.

• 1024 = `no_Y_ticks`: plot no y-ticks.

• 2048 = `Same_ticks`: plot all ticks with the same length.

3.10.10 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`.

3.10.11 plothsizes(): return data corresponding to the output window in the form of a 6-component vector: window width and height, sizes for ticks in horizontal and vertical directions (this is intended for the `gnuplot` interface and is currently not significant), width and height of characters.

3.10.12 plotinit$(w, x, y, \{flag\})$: initialize the rectwindow w, destroying any rect objects you may have already drawn in w. The virtual cursor is set to $(0, 0)$. The rectwindow size is set to width x and height y. If $flag = 0$, x and y represent pixel units. Otherwise, x and y are understood as fractions of the size of the current output device (hence must be between 0 and 1) and internally converted to pixels.

The plotting device imposes an upper bound for x and y, for instance the number of pixels for screen output. These bounds are available through the `plothsizes` function. The following sequence initializes in a portable way (i.e independent of the output device) a window of maximal size, accessed through coordinates in the $[0, 1000] \times [0, 1000]$ range:

```
s = plothsizes();
plotinit(0, s[1]-1, s[2]-1);
plotscale(0, 0,1000, 0,1000);
```

3.10.13 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.

3.10.14 plotlines$(w, X, Y, \{flag = 0\})$: draw on the rectwindow w the polygon such that the (x,y)-coordinates of the vertices are in the vectors of equal length X and Y. For simplicity, the whole polygon is drawn, not only the part of the polygon which is inside the rectwindow. If *flag* is non-zero, close the polygon. In any case, the virtual cursor does not move.

X and Y are allowed to be scalars (in this case, both have to). There, a single segment will be drawn, between the virtual cursor current position and the point (X, Y). And only the part thereof which actually lies within the boundary of w. Then *move* the virtual cursor to (X, Y), even if it is outside the window. If you want to draw a line from $(x1, y1)$ to $(x2, y2)$ where $(x1, y1)$ is not necessarily the position of the virtual cursor, use `plotmove(w,x1,y1)` before using this function.

3.10.15 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.

3.10.16 plotmove(w, x, y): move the virtual cursor of the rectwindow w to position (x, y).

3.10.17 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).

3.10.18 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.

3.10.19 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.

3.10.20 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.

3.10.21 plotrecth$(w, X = a, b, expr, \{flag = 0\}, \{n = 0\})$: writes to rectwindow w the curve output of `ploth`$(w, X = a, b, expr, flag, n)$.

3.10.22 plotrecthraw$(w, data, \{flag = 0\})$: plot graph(s) for *data* in rectwindow w. *flag* has the same significance here as in `ploth`, though recursive plot is no more significant.
data

is a vector of vectors, each corresponding to a list a coordinates. If parametric plot is set, there must be an even number of vectors, each successive pair corresponding to a curve. Otherwise, the first one contains the x coordinates, and the other ones contain the y-coordinates of curves to plot.

3.10.23 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).

3.10.24 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).

3.10.25 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 $(x1 + dx, y1 + dy)$.

3.10.26 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.

3.10.27 plotstring$(w, x, \{flag = 0\})$: draw on the rectwindow w the String x (see Section 2.6.6), 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).

3.10.28 plotterm$(term)$: sets terminal where high resolution plots go (this is currently only taken into account by the `gnuplot` graphical driver). Using the `gnuplot` driver, possible terminals are the same as in gnuplot. If *term* is "?", lists possible values.

Terminal options can be appended to the terminal name and space; terminal size can be put immediately after the name, as in `"gif=300,200"`. Positive return value means success.

3.10.29 psdraw(*list*): same as `plotdraw`, except that the output is a PostScript program appended to the `psfile`.

3.10.30 psploth($X = a, b, expr$): same as `ploth`, except that the output is a PostScript program appended to the `psfile`.

3.10.31 psplothraw(*listx*, *listy*): same as `plothraw`, except that the output is a PostScript program appended to the `psfile`.

3.11 Programming under GP.

3.11.1 Control statements.

A number of control statements are available under 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.

3.11.1.1 break($\{n = 1\}$): interrupts execution of current *seq*, and immediately exits from the n innermost enclosing loops, within the current function call (or the top level loop). n must be bigger than 1. If n is greater than the number of enclosing loops, all enclosing loops are exited.

3.11.1.2 for($X = a, b, seq$): the formal variable X going from a to b, the *seq* is evaluated. Nothing is done if $a > b$. a and b must be in **R**.

3.11.1.3 fordiv(n, X, seq): the formal variable X ranging through the positive divisors of n, the sequence *seq* is evaluated. n must be of type integer.

3.11.1.4 forprime($X = a, b, seq$): the formal variable X ranging over the prime numbers between a to b (including a and b if they are prime), the *seq* is evaluated. 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 **R**.

```
? { forprime(p = 2, 12,
      print(p);
      if (p == 3, p = 6);
    )
  }
2
3
7
11
```

3.11.1.5 forstep$(X = a, b, s, seq)$: the formal variable X going from a to b, in increments of s, the *seq* is evaluated. Nothing is done if $s > 0$ and $a > b$ or if $s < 0$ and $a < b$. s must be in $\mathbf{R}^*$ or a vector of steps $[s_1, \ldots, s_n]$. In the latter case, the successive steps are used in the order they appear in s.

```
? forstep(x=5, 20, [2,4], print(x))
5
7
11
13
17
19
```

3.11.1.6 forsubgroup$(H = G, \{B\}, seq)$: executes *seq* for each subgroup H of the *abelian* group G (given in SNF form or as a vector of elementary divisors), whose index is bounded by B. The subgroups are not ordered in any obvious way, unless G is a p-group in which case Birkhoff's algorithm produces them by decreasing index. A subgroup is given as a matrix whose columns give its generators on the implicit generators of G. For example, the following prints all subgroups of index less than 2 in $G = \mathbf{Z}/2\mathbf{Z}g_1 \times \mathbf{Z}/2\mathbf{Z}g_2$:

```
? G = [2,2]; forsubgroup(H=G, 2, print(H))
[1; 1]
[1; 2]
[2; 1]
[1, 0; 1, 1]
```

The last one, for instance is generated by $(g_1, g_1 + g_2)$. This routine is intended to treat huge groups, when `subgrouplist` is not an option due to the sheer size of the output.

For maximal speed the subgroups have been left as produced by the algorithm. To print them in canonical form (as left divisors of G in HNF form), one can for instance use

```
? G = matdiagonal([2,2]); forsubgroup(H=G, 2, print(mathnf(concat(G,H))))
[2, 1; 0, 1]
[1, 0; 0, 2]
[2, 0; 0, 1]
[1, 0; 0, 1]
```

Note that in this last representation, the index $[G : H]$ is given by the determinant. See `galoissubcyclo` and `galoisfixedfield` for `nfsubfields` applications to Galois theory.

Warning: the present implementation cannot treat a group G, if one of its p-Sylow subgroups has a cyclic factor has more than 2^{31}, resp. 2^{63} elements on a 32-bit, resp. 64-bit architecture.

3.11.1.7 forvec$(X = v, seq, \{flag = 0\})$: v being an n-component vector (where n is arbitrary) of two-component vectors $[a_i, b_i]$ for $1 \le i \le n$, the *seq* is evaluated with the formal variable $X[1]$ going from a_1 to b_1,...,$X[n]$ going from a_n to b_n. The formal variable with the highest index moves the fastest. If *flag* $= 1$, generate only nondecreasing vectors X, and if *flag* $= 2$, generate only strictly increasing vectors X.

3.11.1.8 if$(a, \{seq1\}, \{seq2\})$: if a is non-zero, the expression sequence *seq1* is evaluated, otherwise the expression *seq2* is evaluated. Of course, *seq1* or *seq2* may be empty, so `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, whereas `if (`a`,,`seq`)` evaluates seq if a is equal to zero, and does nothing otherwise. You could get the same result using the `!` (`not`) operator: `if (!`a`,`seq`)`.

Note that the boolean operators `&&` and `||` are evaluated according to operator precedence as explained in Section 2.4, but that, contrary to other operators, the evaluation of the arguments is stopped as soon as the final truth value has been determined. For instance

```
if (reallydoit && longcomplicatedfunction(), ...)%
```

is a perfectly safe statement.

Recall that functions such as `break` and `next` operate on *loops* (such as `for`xxx, `while`, `until`). The `if` statement is *not* a loop (obviously!).

3.11.1.9 next$(\{n = 1\})$: interrupts execution of current seq, resume the next iteration of the innermost enclosing loop, within the current function call (or top level loop). If n is specified, resume at the n-th enclosing loop. If n is bigger than the number of enclosing loops, all enclosing loops are exited.

3.11.1.10 return$(\{x = 0\})$: returns from current subroutine, with result x. If x is omitted, return the `(void)` value (return no result, like `print`).

3.11.1.11 until(a, seq): evaluates expression sequence seq until a is not equal to 0 (i.e. until a is true). If a is initially not equal to 0, seq is evaluated once (more generally, the condition on a is tested *after* execution of the seq, not before as in `while`).

3.11.1.12 while(a, seq): while a is non-zero evaluate the expression sequence seq. The test is made *before* evaluating the seq, hence in particular if a is initially equal to zero the seq will not be evaluated at all.

3.11.2 Specific functions used in GP programming.

In addition to the general PARI functions, it is necessary to have some functions which will be of use specifically for GP, though a few of these can be accessed under library mode. Before we start describing these, we recall the difference between *strings* and *keywords* (see Section 2.6.6): the latter don't get expanded at all, and you can type them without any enclosing quotes. The former are dynamic objects, where everything outside quotes gets immediately expanded.

3.11.2.1 addhelp(S, str): changes the help message for the symbol S. The string str is expanded on the spot and stored as the online help for S. If S is a function *you* have defined, its definition will still be printed before the message str. It is recommended that you document global variables and user functions in this way. Of course GP won't protest if you don't do it.

There's nothing to prevent you from modifying the help of built-in PARI functions (but if you do, we'd like to hear why you needed to do it!).

3.11.2.2 alias$(newkey, key)$: defines the keyword $newkey$ as an alias for keyword key. key must correspond to an existing *function* name. This is different from the general user macros in that alias expansion takes place immediately upon execution, without having to look up any function code, and is thus much faster. A sample alias file `misc/gpalias` is provided with the standard distribution. Alias commands are meant to be read upon startup from the `.gprc` file, to cope with function names you are dissatisfied with, and should be useless in interactive usage.

3.11.2.3 allocatemem({$x = 0$}): this is a very special operation which allows the user to change the stack size *after* initialization. x must be a non-negative integer. If $x! = 0$, a new stack of size $16 * \lceil x/16 \rceil$ bytes will be allocated, all the PARI data on the old stack will be moved to the new one, and the old stack will be discarded. If $x = 0$, the size of the new stack will be twice the size of the old one.

Although it is a function, this must be the *last* instruction in any GP sequence. The technical reason is that this routine usually moves the stack, so objects from the current sequence might not be correct anymore. Hence, to prevent such problems, this routine terminates by a `longjmp` (just as an error would) and not by a return.

The library syntax is **allocatemoremem**(x), where x is an unsigned long, and the return type is void. GP uses a variant which ends by a `longjmp`.

3.11.2.4 default({*key*}, {*val*}, {*flag*}): sets the default corresponding to keyword *key* to value *val*. *val* is a string (which of course accepts numeric arguments without adverse effects, due to the expansion mechanism). See Section 2.1 for a list of available defaults, and Section 2.2 for some shortcut alternatives. Typing `default()` (or `\d`) yields the complete default list as well as their current values.

If *val* is omitted, prints the current value of default *key*. If *flag* is set, returns the result instead of printing it.

3.11.2.5 error({*str*}$*$): outputs its argument list (each of them interpreted as a string), then interrupts the running GP program, returning to the input prompt.

Example: `error("n = ", n, " is not squarefree !")`.

Note that, due to the automatic concatenation of strings, you could in fact use only one argument, just by suppressing the commas.

UNIX: **3.11.2.6 extern**(*str*): the string *str* is the name of an external command (i.e. one you would type from your UNIX shell prompt). This command is immediately run and its input fed into GP, just as if read from a file.

3.11.2.7 getheap(): returns a two-component row vector giving the number of objects on the heap and the amount of memory they occupy in long words. Useful mainly for debugging purposes.

The library syntax is **getheap**().

3.11.2.8 getrand(): returns the current value of the random number seed. Useful mainly for debugging purposes.

The library syntax is **getrand**(), returns a C long.

3.11.2.9 getstack(): returns the current value of `top` − `avma`, i.e. the number of bytes used up to now on the stack. Should be equal to 0 in between commands. Useful mainly for debugging purposes.

The library syntax is **getstack**(), returns a C long.

3.11.2.10 gettime(): returns the time (in milliseconds) elapsed since either the last call to `gettime`, or to the beginning of the containing GP instruction (if inside GP), whichever came last.

The library syntax is **gettime**(), returns a C long.

3.11.2.11 global(*list of variables*): declares the corresponding variables to be global. From now on, you will be forbidden to use them as formal parameters for function definitions or as loop indexes. This is especially useful when patching together various scripts, possibly written with different naming conventions. For instance the following situation is dangerous:

```
p = 3    \\ fix characteristic
...
forprime(p = 2, N, ...)
f(p) = ...
```

since within the loop or within the function's body (even worse: in the subroutines called in that scope), the true global value of `p` will be hidden. If the statement `global(p = 3)` appears at the beginning of the script, then both expressions will trigger syntax errors.

Calling `global` without arguments prints the list of global variables in use. In particular, `eval(global)` will output the values of all global variables.

3.11.2.12 input(): reads a string, interpreted as a GP expression, from the input file, usually standard input (i.e. the keyboard). If a sequence of expressions is given, the result is the result of the last expression of the sequence. When using this instruction, it is useful to prompt for the string by using the `print1` function. Note that in the present version 2.19 of `pari.el`, when using GP under GNU Emacs (see Section 2.10) one *must* prompt for the string, with a string which ends with the same prompt as any of the previous ones (a `"?  "` will do for instance).

UNIX: **3.11.2.13 install**(*name*, *code*, {*gpname*}, {*lib*}): loads from dynamic library *lib* the function *name*. Assigns to it the name *gpname* in this GP session, with argument code *code* (see Section 4.9.2 for an explanation of those). If *lib* is omitted, uses `libpari.so`. If *gpname* is omitted, uses *name*.

This function is useful for adding custom functions to the GP interpreter, or picking useful functions from unrelated libraries. For instance, it makes the function `system` obsolete:

```
? install(system, vs, sys, "libc.so")
? sys("ls gp*")
gp.c            gp.h            gp_rl.c
```

But it also gives you access to all (non static) functions defined in the PARI library. For instance, the function `GEN addii(GEN x, GEN y)` adds two PARI integers, and is not directly accessible under GP (it's eventually called by the `+` operator of course):

```
? install("addii", "GG")
? addii(1, 2)
%1 = 3
```

Re-installing a function will print a Warning, and update the prototype code if needed, but will reload a symbol from the library, even it the latter has been recompiled.

Caution: This function may not work on all systems, especially when GP has been compiled statically. In that case, the first use of an installed function will provoke a Segmentation Fault, i.e. a major internal blunder (this should never happen with a dynamically linked executable). Hence, if you intend to use this function, please check first on some harmless example such as the ones above that it works properly on your machine.

3.11.2.14 kill(s): kills the present value of the variable, alias or user-defined function s. The corresponding identifier can now be used to name any GP object (variable or function). This is the only way to replace a variable by a function having the same name (or the other way round), as in the following example:

```
? f = 1
%1 = 1
? f(x) = 0
  ***   unused characters: f(x)=0
                             ^----
? kill(f)
? f(x) = 0
? f()
%2 = 0
```

When you kill a variable, all objects that used it become invalid. You can still display them, even though the killed variable will be printed in a funny way (following the same convention as used by the library function `fetch_var`, see Section 4.6). For example:

```
? a^2 + 1
%1 = a^2 + 1
? kill(a)
? %1
%2 = #<1>^2 + 1
```

If you simply want to restore a variable to its "undefined" value (monomial of degree one), use the quote operator: `a = 'a`. Predefined symbols (`x` and GP function names) cannot be killed.

3.11.2.15 print({*str*}$*$): outputs its (string) arguments in raw format, ending with a newline.

3.11.2.16 print1({*str*}$*$): outputs its (string) arguments in raw format, without ending with a newline (note that you can still embed newlines within your strings, using the `\n` notation !).

3.11.2.17 printp({*str*}$*$): outputs its (string) arguments in prettyprint (beautified) format, ending with a newline.

3.11.2.18 printp1({*str*}$*$): outputs its (string) arguments in prettyprint (beautified) format, without ending with a newline.

3.11.2.19 printtex({*str*}$*$): outputs its (string) arguments in TEX format. This output can then be used in a TEX manuscript. The printing is done on the standard output. If you want to print it to a file you should use `writetex` (see there).

Another possibility is to enable the `log` default (see Section 2.1). You could for instance do:

```
default(logfile, "new.tex");
default(log, 1);
printtex(result);
```

(You can use the automatic string expansion/concatenation process to have dynamic file names if you wish).

3.11.2.20 quit(): exits GP.

3.11.2.21 read({*str*}): reads in the file whose name results from the expansion of the string *str*. If *str* is omitted, re-reads the last file that was fed into GP. The return value is the result of the last expression evaluated. If a GP `binary file` is read using this command (see Section 3.11.2.30), the file is loaded and the last object in the file is returned.

3.11.2.22 reorder({$x = []$}): x must be a vector. If x is the empty vector, this gives the vector whose components are the existing variables in increasing order (i.e. in decreasing importance). Killed variables (see `kill`) will be shown as `0`. If x is non-empty, it must be a permutation of variable names, and this permutation gives a new order of importance of the variables, *for output only*. For example, if the existing order is `[x,y,z]`, then after `reorder([z,x])` the order of importance of the variables, with respect to output, will be `[z,y,x]`. The internal representation is unaffected.

3.11.2.23 setrand(n): reseeds the random number generator to the value n. The initial seed is $n = 1$.

The library syntax is **setrand**(n), where n is a `long`. Returns n.

UNIX: **3.11.2.24 system**(*str*): *str* is a string representing a system command. This command is executed, its output written to the standard output (this won't get into your logfile), and control returns to the PARI system. This simply calls the C `system` command.

3.11.2.25 trap({e}, {*rec*}, {*seq*}): tries to execute *seq*, trapping error e, that is effectively preventing it from aborting computations in the usual way; the recovery sequence *rec* is executed if the error occurs and the evaluation of *rec* becomes the result of the command. If e is omitted, all exceptions are trapped. Note in particular that hitting `^C` (Control-C) raises an exception. See Section 2.7.2 for an introduction to error recovery under GP.

```
? \\ trap division by 0
? inv(x) = trap (gdiver, INFINITY, 1/x)
? inv(2)
%1 = 1/2
? inv(0)
%2 = INFINITY
```

If *seq* is omitted, defines *rec* as a default action when catching exception e, provided no other trap as above intercepts it first. The error message is printed, as well as the result of the evaluation of *rec*, and control is given back to the GP prompt. In particular, current computation is then lost.

The following error handler prints the list of all user variables, then stores in a file their name and their values:

```
? { trap( ,
      print(reorder);
      writebin("crash")) }
```

If no recovery code is given (*rec* is omitted) a *break loop* will be started (see Section 2.7.3). In particular

```
? trap()
```

by itself installs a default error handler, that will start a break loop whenever an exception is raised.

If *rec* is the empty string `""` the default handler (for that error if e is present) is disabled.

Note: The interface is currently not adequate for trapping individual exceptions. In the current version 2.2.7, the following keywords are recognized, but the name list will be expanded and changed in the future (all library mode errors can be trapped: it's a matter of defining the keywords to GP, and there are currently far too many useless ones):

`accurer`: accuracy problem

`gdiver`: division by 0

`invmoder`: impossible inverse modulo

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

`siginter`: SIGINT received (usually from Control-C)

`talker`: miscellaneous error

`typeer`: wrong type

`errpile`: the PARI stack overflows

3.11.2.26 type($x, \{t\}$): this is useful only under GP. If t is not present, returns the internal type number of the PARI object x. Otherwise, makes a copy of x and sets its type equal to type t, which can be either a number or, preferably since internal codes may eventually change, a symbolic name such as `t_FRACN` (you can skip the `t_` part here, so that `FRACN` by itself would also be all right). Check out existing type names with the metacommand `\t`.

GP will not let you create meaningless objects in this way where the internal structure does not match the type. This function can be useful to create reducible rationals (type `t_FRACN`) or rational functions (type `t_RFRACN`). In fact it's the only way to do so in GP. In this case, the created object, as well as the objects created from it, will not be reduced automatically, making some operations a bit faster.

There is no equivalent library syntax, since the internal functions `typ` and `settyp` are available. Note that `settyp` does *not* create a copy of `x`, contrary to most PARI functions. It also doesn't check for consistency. `settyp` just changes the type in place and returns nothing. `typ` returns a C long integer. Note also the different spellings of the internal functions (`set`)`typ` and of the GP function `type`, which is due to the fact that `type` is a reserved identifier for some C compilers.

3.11.2.27 whatnow(*key*): if keyword *key* is the name of a function that was present in GP version 1.39.15 or lower, outputs the new function name and syntax, if it changed at all (387 out of 560 did).

3.11.2.28 write(*filename*, {*str*$*$}): writes (appends) to *filename* the remaining arguments, and appends a newline (same output as `print`).

3.11.2.29 write1(*filename*, {*str*$*$}): writes (appends) to *filename* the remaining arguments without a trailing newline (same output as `print1`).

3.11.2.30 writebin(*filename*, {x}): writes (appends) to *filename* the object x in binary format. This format is not human readable, but contains the exact internal structure of x, and is much faster to save/load than a string expression, as would be produced by `write`. The binary file format includes a magic number, so that such a file can be recognized and correctly input by the regular `read` or `\r` function. If saved objects refer to (polynomial) variables that are not defined in the new session, they will be displayed in a funny way (see Section 3.11.2.14).

If x is omitted, saves all user variables from the session, together with their names. Reading such a "named object" back in a GP session will set the corresponding user variable to the saved value. E.g after

```
x = 1; writebin("log")
```

reading `log` into a clean session will set `x` to 1. The relative variables priorities (see Section 2.6.2) of new variables set in this way remain the same (preset variables retain their former priority, but are set to the new value). In particular, reading such a session log into a clean session will restore all variables exactly as they were in the original one.

User functions, installed functions and history objects can not be saved via this function. Just as a regular input file, a binary file can be compressed using `gzip`, provided the file name has the standard `.gz` extension.

In the present implementation, the binary files are architecture dependent and compatibility with future versions of GP is not guaranteed. Hence binary files should not be used for long term storage (also, they are larger and harder to compress than text files).

3.11.2.31 writetex(*filename*, {*str*∗}): as `write`, in TEX format.

Chapter 4:
Programming PARI in Library Mode

4.1 Introduction: initializations, universal objects.

To be able to use PARI in library mode, you must write a C program and link it to the PARI library. See the installation guide (in Appendix A) on how to create and install the library and include files. A sample Makefile is presented in Appendix B.

Probably the best way to understand how programming is done is to work through a complete example. We will write such a program in Section 4.8. Before doing this, a few explanations are in order.

First, one must explain to the outside world what kind of objects and routines we are going to use. This is done simply with the statement

```
#include <pari.h>
```

This file `pari.h` imports all the necessary constants, variables and functions, defines some important macros, and also defines the fundamental type for all PARI objects: the type **GEN**, which is simply a pointer to `long`.

Technical note: we would have liked to define a type `GEN` to be a pointer to itself. This unfortunately is not possible in C, except by using structures, but then the names become unwieldy. The result of this is that when we use a component of a PARI object, it will be a `long`, hence will need to be typecast to a `GEN` again if we want to avoid warnings from the compiler. This will sometimes be quite tedious, but of course is trivially done. See the discussion on typecasts in the next section.

After declaring the use of the file `pari.h`, the first executable statement of a main program should be to initialize the PARI system, and in particular the PARI stack which will be both a scratchboard and a repository for computed objects. This is done with a call to the function

`void` **pari_init**`(long size, ulong maxprime))`

The first argument is the number of bytes given to PARI to work with (it should not reasonably be taken below 500000), and the second is the upper limit on a precomputed prime number table. If you do not want prime numbers, just put `maxprime` = 2. Be careful because lots a PARI functions need this table (certainly all the ones of interest to number theorists). If you wind up with the error message "not enough precomputed primes", try to increase this value.

We have now at our disposal:

• a large PARI *stack* containing nothing. It is a big connected chunk of memory whose size you chose when invoking `pari_init`. All your computations are going to take place here. When doing large computations, unwanted intermediate results clutter up memory very fast so some kind of garbage collecting is needed. Most large systems do garbage collecting when the memory is getting scarce, and this slows down the performance. In PARI we have taken a different approach: you must do your own cleaning up when the intermediate results are not needed anymore. Special purpose routines have been written to do this; we will see later how (and when, if at all) you should use them.

• the following *universal objects* (by definition, objects which do not belong on the stack): the integers 0, 1 and 2 (respectively called **gzero**, **gun**, and **gdeux**), the fraction $\frac{1}{2}$ (**ghalf**), the complex number i (**gi**). All of these are of type `GEN`.

In addition, space is reserved for the polynomials x_v (**polx**`[`v`]`), and the polynomials 1_v (**polun**`[`v`]`). Here, x_v is the name of variable number v, where $0 \leq v \leq$ `MAXVARN` (the exact value of which depends on your machine, at least 16383 in any case). Both `polun` and `polx` are arrays of `GEN`s, the index being the polynomial variable number.

However, except for the ones corresponding to variables 0 and `MAXVARN`, these polynomials are *not* created upon initialization. It is the programmer's responsibility to fill them before use. We will see how this is done in Section 4.6 (*never* through direct assignment).

• a *heap* which is just a linked list of permanent universal objects. For now, it contains exactly the ones listed above. You will probably very rarely use the heap yourself; and if so, only as a collection of individual copies of objects taken from the stack (called clones in the sequel). Thus you need not bother with its internal structure, which may change as PARI evolves. Some complex PARI functions may create clones for special garbage collecting purposes, usually destroying them when returning.

• a table of primes (in fact of *differences* between consecutive primes), called **diffptr**, of type `byteptr` (pointer to `unsigned char`). Its use is described in appendix C.

• access to all the built-in functions of the PARI library. These are declared to the outside world when you include `pari.h`, but need the above things to function properly. So if you forget the call to `pari_init`, you will immediately get a fatal error when running your program.

4.2 Important technical notes.

4.2.1 Typecasts.

We have seen that, due to the non-recursiveness of the PARI types from the compiler's point of view, many typecasts will be necessary when programming in PARI. To take an example, a vector V of dimension 2 (two components) will be represented by a chunk of memory pointed to by the `GEN` `V`. `V[0]` contains coded information, in particular about the type of the object, its length, etc. `V[1]` and `V[2]` contain pointers to the two components of `V`. Those coefficients `V[i]` themselves are in chunks of memory whose complexity depends on their own types, and so on. This is where typecasting will be necessary: a priori, `V[i]` (for `i` $= 1, 2$) is a `long`, but we will want to use it as a `GEN`. The following two constructions will be exceedingly frequent (`x` and `V` are `GEN`s):

```
V[i] = (long) x;
x = (GEN) V[i];
```

Note that a typecast is not a valid lvalue (cannot be put on the left side of an assignment), so `(GEN)V[i] = x` would be incorrect, though some compilers may accept it.

Due to this annoyance, the PARI functions and variables that occur most frequently have analogues which are macros including the typecast. The complete list can be found in the file `paricast.h` (which is included by `pari.h` and can be found at the same place). For instance you can abbreviate:

```
(long) gzero     ----->  zero
(long) gun       ----->  un
```

```
(long) polx[v]   -----> lpolx[v]
(long) gadd(x,y) -----> ladd(x,y)
```

In general, replacing a leading `g` by an `l` in the name of a PARI function will typecast the result to `long`. Note that `ldiv` is an ANSI C function which is hidden in PARI by a macro of the same name representing `(long)gdiv`.

The macro **coeff**(x, m, n) exists with exactly the meaning of `x[m,n]` under GP when `x` is a matrix. This is a purely syntactical trick to reduce the number of typecasts and thus does not create a copy of the coefficient (contrary to all the library *functions*). It can be put on the left side of an assignment statement, and its value, of type `long` integer, is a pointer to the desired coefficient object. The macro `gcoeff` is a synonym for `(GEN) coeff`, hence cannot be put on the left side of an assignment.

To retrieve the values of elements of lists of ... of lists of vectors, without getting infuriated by gigantic lists of typecasts, we have the **mael** macros (for **m**ultidimensional **a**rray **el**ement). The syntax is **mael**$n(x, a_1, \ldots, a_n)$, where x is a `GEN`, the a_i are indexes, and n is an integer between 2 and 5 (with a standalone `mael` as a synonym for `mael2`). This stands for $x[a_1][a_2]\ldots[a_n]$ (with all the necessary typecasts), and returns a `long` (i.e. they are valid lvalues). The `gmael`n macros are synonyms for `(GEN)mael`n. Note that due to the implementation of matrix types in PARI (i.e. as horizontal lists of vertical vectors), `coeff(x,y)` is actually completely equivalent to `mael(y,x)`. It is suggested that you use `coeff` in matrix context, and `mael` otherwise.

4.2.2 Variations on basic functions. In the library syntax descriptions in Chapter 3, we have only given the basic names of the functions. For example `gadd`(x, y) assumes that x and y are PARI objects (of type `GEN`), and *creates* the result $x + y$ on the PARI stack. For most of the basic operators and functions, many other variants are available. We give some examples for `gadd`, but the same is true for all the basic operators, as well as for some simple common functions (a more complete list is given in Chapter 5):

`GEN` **gaddgs**`(GEN x, long y)`

`GEN` **gaddsg**`(long x, GEN y)`

In the following three, `z` is a preexisting `GEN` and the result of the corresponding operation is put into `z`. The size of the PARI stack does not change:

`void` **gaddz**`(GEN x, GEN y, GEN z)`

`void` **gaddgsz**`(GEN x, long y, GEN z)`

`void` **gaddsgz**`(GEN x, GEN y, GEN z)`

There are also low level functions which are special cases of the above:

`GEN` **addii**`(GEN x, GEN y)`: here x and y are `GEN`s of type `t_INT` (this is not checked).

`GEN` **addrr**`(GEN x, GEN y)`: here x and y are `GEN` reals (type `t_REAL`).

There also exist functions **addir**, **addri**, **mpadd** (whose two arguments can be of type integer or real), **addis** (to add a `t_INT` and a `long`) and so on.

All these functions can of course be called by the user but we feel that the few microseconds lost in calling more general functions (in this case `gadd`) are compensated by the fact that one needs to remember a much smaller number of functions, and also because there is a hidden danger here: the types of the objects that you use, if they are themselves results of a previous computation, are not

completely predetermined. For instance the multiplication of a type real `t_REAL` by a type integer `t_INT` *usually* gives a result of type real, except when the integer is 0, in which case according to the PARI philosophy the result is the exact integer 0. Hence if afterwards you call a function which specifically needs a real type argument, you are going to be in trouble.

If you really want to use these functions, their names are self-explanatory once you know that **i** stands for a PARI integer, **r** for a PARI real, **mp** for i or r, **s** for an ordinary signed long, whereas **z** (as a suffix) means that the result is not created on the PARI stack but assigned to a preexisting GEN object passed as an extra argument.

For completeness, Chapter 5 gives a description of all these low-level functions.

Please note that in the present version 2.2.7 the names of the functions are not always consistent. This will be changed. Hence anyone programming in PARI must be aware that the names of almost all functions that he uses might be subject to change. If the need arises (i.e. if there really are people out there who delve into the innards of PARI), updated versions with no name changes will be released.

4.2.3 Portability: 32-bit / 64-bit architectures.

PARI supports both 32-bit and 64-bit based machines, but not simultaneously! The library will have been compiled assuming a given architecture (a priori following a guess by the `Configure` program, see Appendix A), and some of the header files you include (through `pari.h`) will have been modified to match the library.

Portable macros are defined to bypass most machine dependencies. If you want your programs to run identically on 32-bit and 64-bit machines, you will have to use these, and not the corresponding numeric values, whenever the precise size of your `long` integers might matter. Here are the most important ones:

	64-bit	32-bit	
`BITS_IN_LONG`	64	32	
`LONG_IS_64BIT`	defined	undefined	
`DEFAULTPREC`	3	4	($\approx$ 19 decimal digits, see formula below)
`MEDDEFAULTPREC`	4	6	($\approx$ 38 decimal digits)
`BIGDEFAULTPREC`	5	8	($\approx$ 57 decimal digits)

For instance, suppose you call a transcendental function, such as

`GEN` **`gexp`**`(GEN x, long prec)`.

The last argument `prec` is only used if `x` is an exact object, otherwise the relative precision is determined by the precision of `x`. But since `prec` sets the size of the inexact result counted in (`long`) *words* (including codewords), the same value of `prec` will yield different results on 32-bit and 64-bit machines. Real numbers have two codewords (see Section 4.5), so the formula for computing the bit accuracy is

$$\texttt{bit\_accuracy}(\texttt{prec}) = (\texttt{prec} - 2) * \texttt{BITS\_IN\_LONG}$$

(this is actually the definition of a macro). The corresponding accuracy expressed in decimal digits would be

$$\texttt{bit\_accuracy}(\texttt{prec}) * \log(2)/\log(10).$$

For example if the value of `prec` is 5, the corresponding accuracy for 32-bit machines is $(5-2) * \log(2^{32})/\log(10) \approx 28$ decimal digits, while for 64-bit machines it is $(5-2) * \log(2^{64})/\log(10) \approx 57$ decimal digits.

Thus, you must take care to change the `prec` parameter you are supplying according to the bit size, either using the default precisions given by the various `DEFAULTPREC`s, or by using conditional constructs of the form:

```
#ifndef LONG_IS_64BIT
  prec = 4;
#else
  prec = 6;
#endif
```

which is in this case equivalent to the statement `prec = MEDDEFAULTPREC;`.

Note that for parity reasons, half the accuracies available on 32-bit architectures (the odd ones) have no precise equivalents on 64-bit machines.

4.3 Creation of PARI objects, assignments, conversions.

4.3.1 Creation of PARI objects. The basic function which creates a PARI object is the function **cgetg** whose prototype is:

`GEN` **`cgetg`**`(long length, long type).`

Here `length` specifies the number of longwords to be allocated to the object, and type is the type number of the object, preferably in symbolic form (see Section 4.5 for the list of these). The precise effect of this function is as follows: it first creates on the PARI *stack* a chunk of memory of size `length` longwords, and saves the address of the chunk which it will in the end return. If the stack has been used up, a message to the effect that "the PARI stack overflows" will be printed, and an error raised. Otherwise, it sets the type and length of the PARI object. In effect, it fills correctly and completely its first codeword (`z[0]` or `*z`). Many PARI objects also have a second codeword (types `t_INT`, `t_REAL`, `t_PADIC`, `t_POL`, and `t_SER`). In case you want to produce one of those from scratch (this should be exceedingly rare), *it is your responsibility to fill this second codeword*, either explicitly (using the macros described in Section 4.5), or implicitly using an assignment statement (using `gaffect`).

Note that the argument `length` is predetermined for a number of types: 3 for types `t_INTMOD`, `t_FRAC`, `t_FRACN`, `t_COMPLEX`, `t_POLMOD`, `t_RFRAC` and `t_RFRACN`, 4 for type `t_QUAD` and `t_QFI`, and 5 for type `t_PADIC` and `t_QFR`. However for the sake of efficiency, no checking is done in the function `cgetg`, so disasters will occur if you give an incorrect length.

Notes: 1) The main use of this function is to prepare for later assignments (see Section 4.3.2). Most of the time you will use `GEN` objects as they are created and returned by PARI functions. In this case you do not need to use `cgetg` to create space to hold them.

2) For the creation of leaves, i.e. integers or reals, which is very common,

`GEN` **`cgeti`**`(long length)`

`GEN` **`cgetr`**`(long length)`

should be used instead of `cgetg(length, t_INT)` and `cgetg(length, t_REAL)` respectively.

3) The macros **lgetg**, **lgeti**, **lgetr** are predefined as `(long)cgetg`, `(long)cgeti`, `(long)cgetr`, respectively.

4) Finally, there are two low-level routines to allocate space on the PARI stack:

`GEN` **`new_chunk`**`(size_t n)`

allocates a `GEN` with n components, *without* filling the required code words. This is the low-level constructor underlying `cgetg`, which calls `new_chunk`, then sets the first code word. It works by simply returning the address `((GEN)avma) - n`, after checking that it is larger than `(GEN)bot`.

`char*` **`stackmalloc`**`(size_t n)`

allocates memory on the stack for n chars (*not* n `GEN`s). This is faster than using `malloc`, and easier to use in most situations when temporary storage is needed. In particular there is no need to `free` individually all variables thus allocated: a simple `avma = oldavma` might be enough. On the other hand, beware that this is not permanent independant storage, but part of the PARI stack.

Note that objects allocated through these two functions cannot be `gerepile`'d. They are not valid `GEN`s since they have no PARI type.

Examples: 1) `z = cgeti(DEFAULTPREC)` and `cgetg(DEFAULTPREC, t_INT)` create an integer object whose "precision" is `bit_accuracy(DEFAULTPREC)` $= 64$. This means `z` can hold rational integers of absolute value less than 2^{64}. Note that in both cases, the second codeword will *not* be filled. Of course we could use numerical values, e.g. `cgeti(4)`, but this would have different meanings on different machines as `bit_accuracy(4)` equals 64 on 32-bit machines, but 128 on 64-bit machines.

2) The following creates a *complex number* whose real and imaginary parts can hold real numbers of precision `bit_accuracy`(`MEDDEFAULTPREC`) $= 96$ bits:

```
z = cgetg(3, t_COMPLEX);
z[1] = lgetr(MEDDEFAULTPREC);
z[2] = lgetr(MEDDEFAULTPREC);
```

3) To create a matrix object for 4×3 matrices:

```
z = cgetg(4, t_MAT);
for(i=1; i<4; i++) z[i] = lgetg(5, t_COL);
```

If one wishes to create space for the matrix elements themselves, one has to follow this with a double loop to fill each column vector.

These last two examples illustrate the fact that since PARI types are recursive, all the branches of the tree must be created. The function **cgetg** creates only the "root", and other calls to `cgetg` must be made to produce the whole tree. For matrices, a common mistake is to think that `z =`

`cgetg(4, t_MAT)` (for example) will create the root of the matrix: one needs also to create the column vectors of the matrix (obviously, since we specified only one dimension in the first `cgetg`!). This is because a matrix is really just a row vector of column vectors (hence a priori not a basic type), but it has been given a special type number so that operations with matrices become possible.

Finally, to facilitate input of constant objects when speed is not paramount, there are four `varargs` functions:

`GEN` **coefs_to_int**`(long n, ...)` returns the non-negative `t_INT` whose development in base 2^{32} is given by the following n words (`unsigned long`). It is assumed that all such arguments are less than 2^{32} (the actual word size is irrelevant, the behaviour is also as above on 64-bit machines).

```
coefs_to_int(3, a2, a1, a0);
```

returns $a_2 2^{64} + a_1 2^{32} + a_0$.

`GEN` **coefs_to_pol**`(long n, ...)` Returns the `t_POL` whose n coefficients (`GEN`) follow, in order of decreasing degree.

```
coefs_to_pol(3, gun, gdeux, gzero);
```

returns the polynomial $X^2 + 2X$ (in variable 0, use `setvarn` if you want other variable numbers). Beware that n is the number of coefficients, *not* the degree.

`GEN` **coefs_to_vec**`(long n, ...)` returns the `t_VEC` whose n coefficients (`GEN`) follow.

`GEN` **coefs_to_col**`(long n, ...)` returns the `t_COL` whose n coefficients (`GEN`) follow.

Warning Contrary to the policy of general PARI functions, the latter three functions do *not* copy of its arguments, nor do they produce an object a priori suitable for `gerepileupto`. For instance

```
/* gerepile-safe: components are universal objects */
z = coefs_to_vec(3, gun, gzero, gdeux);

/* NOT OK for gerepileupto: stoi(1) creates component before vector root */
z = coefs_to_vec(3, stoi(1), gzero, gdeux);

/* NO! First vector component is destroyed */
x = gclone(gun);
z = coefs_to_vec(3, x, gzero, gdeux);
gunclone(x);
```

The following function is also available as a special case of `coefs_to_int`:

`GEN` **u2toi**`(ulong a, ulong b)`

Returns the `GEN` equal to $2^{32}a + b$, *assuming* that $a, b < 2^{32}$. This does not depend on `sizeof(long)`: the behaviour is as above on both 32 and 64-bit machines.

4.3.2 Assignments. Firstly, if `x` and `y` are both declared as `GEN` (i.e. pointers to something), the ordinary C assignment `y = x` makes perfect sense: we are just moving a pointer around. However, physically modifying either `x` or `y` (for instance, `x[1] = 0`) will also change the other one, which is usually not desirable.

Very important note: Using the functions described in this paragraph is very inefficient and often awkward: one of the `gerepile` functions (see Section 4.4) should be preferred. See the paragraph end for some exceptions to this rule.

The general PARI assignment function is the function **gaffect** with the following syntax:

`void` **gaffect**`(GEN x, GEN y)`

Its effect is to assign the PARI object `x` into the *preexisting* object `y`. This copies the whole structure of `x` into `y` so many conditions must be met for the assignment to be possible. For instance it is allowed to assign an integer into a real number, but the converse is forbidden. For that, you must use the truncation or rounding function of your choice (see section 3.2). It can also happen that `y` is not large enough or does not have the proper tree structure to receive the object `x`. As an extreme example, assume `y` is the zero integer with length equal to 2. Then all assignments of a non-zero integer into `y` will result in an error message since `y` is not large enough to accommodate a non-zero integer. In general common sense will tell you what is possible, keeping in mind the PARI philosophy which says that if it makes sense it is legal. For instance, the assignment of an imprecise object into a precise one does *not* make sense. However, a change in precision of imprecise objects is allowed.

All functions ending in "`z`" such as **gaddz** (see Section 4.2.2) implicitly use this function. In fact what they exactly do is record **avma** (see Section 4.4), perform the required operation, **gaffect** the result to the last operand, then restore the initial `avma`.

You can assign ordinary C long integers into a PARI object (not necessarily of type `t_INT`). Use the function **gaffsg** with the following syntax:

`void` **gaffsg**`(long s, GEN y)`

Note: due to the requirements mentioned above, it is usually a bad idea to use `gaffect` statements. Two exceptions:

• for simple objects (e.g. leaves) whose size is controlled, they can be easier to use than gerepile, and about as efficient.

• to coerce an inexact object to a given precision. For instance

```
gaffect(x, (tmp=cgetr(3))); x = tmp;
```

at the beginning of a routine where precision can be kept to a minimum (otherwise the precision of `x` will be used in all subsequent computations, which will be a disaster if `x` is known to thousands of digits).

4.3.3 Copy. It is also very useful to copy a PARI object, not just by moving around a pointer as in the `y = x` example, but by creating a copy of the whole tree structure, without pre-allocating a possibly complicated `y` to use with `gaffect`. The function which does this is called **gcopy**, with the predefined macro **lcopy** as a synonym for `(long)gcopy`. Its syntax is:

`GEN` **gcopy**`(GEN x)`

and the effect is to create a new copy of x on the PARI stack. Beware that universal objects which occur in specific components of certain types (mainly moduli for types `t_INTMOD` and `t_PADIC`) are not copied, as they are assumed to be permanent. In this case, `gcopy` only copies the pointer. Use `GEN` **forcecopy**`(GEN x)` if you want a complete copy.

Please be sure at this point that you really understand the difference between `y = x`, `y = gcopy(x)`, and `gaffect(x,y)`: this will save you from many obvious mistakes later on.

4.3.4 Clones. Sometimes, it may be more efficient to create a *permanent* copy of a PARI object. This will not be created on the stack but on the heap. The function which does this is called **gclone**, with the predefined macro **lclone** as a synonym for `(long)gclone`. Its syntax is:

`GEN` **gclone**`(GEN x)`

A clone can be removed from the heap (thus destroyed) using

`void` **gunclone**`(GEN x)`

No PARI object should keep references to a clone which has been destroyed. If you want to copy a clone back to the stack then delete it, use `forcecopy` and not `gcopy`, otherwise some components might not be copied (moduli of `t_INTMOD`s and `t_POLMOD`s for instance).

4.3.5 Conversions. The following functions convert C objects to PARI objects (creating them on the stack as usual):

`GEN` **stoi**`(long s)`: C `long` integer ("small") to PARI integer (`t_INT`)

`GEN` **dbltor**`(double s)`: C `double` to PARI real (`t_REAL`). The accuracy of the result is 19 decimal digits, i.e. a type `t_REAL` of length `DEFAULTPREC`, although on 32-bit machines only 16 of them will be significant.

We also have the converse functions:

`long` **itos**`(GEN x)`: `x` must be of type `t_INT`,

`double` **rtodbl**`(GEN x)`: `x` must be of type `t_REAL`,

as well as the more general ones:

`long` **gtolong**`(GEN x)`,

`double` **gtodouble**`(GEN x)`.

4.4 Garbage collection.

4.4.1 Why and how.

As we have seen, the `pari_init` routine allocates a big range of addresses, the *stack*, that are going to be used throughout. Recall that all PARI objects are pointers. Except for a few universal objects, they will all point at some part of the stack.

The stack starts at the address `bot` and ends just before `top`. This means that the quantity

$$(\texttt{top} - \texttt{bot}) \,/\, \texttt{sizeof(long)}$$

is equal to the `size` argument of `pari_init`. The PARI stack also has a "current stack pointer" called **avma**, which stands for **av**ailable **m**emory **a**ddress. These three variables are global (declared by `pari.h`). They are of type `pari_sp`, which means *pari stack pointer*.

The stack is oriented upside-down: the more recent an object, the closer to `bot`. Accordingly, initially `avma` = `top`, and `avma` gets *decremented* as new objects are created. As its name indicates, `avma` always points just *after* the first free address on the stack, and `(GEN)avma` is always (a pointer to) the latest created object. When `avma` reaches `bot`, the stack overflows, aborting all computations, and an error message is issued. To avoid this *you* will need to clean up the stack

from time to time, when some bunch of intermediate objects will not be needed anymore. This is called "*garbage collecting*."

We are now going to describe briefly how this is done. We will see many concrete examples in the next subsection.

• First, PARI routines will do their own garbage collecting, which means that whenever a documented function from the library returns, only its result(s) will have been added to the stack (non-documented ones may not do this). In particular, a PARI function that does not return a `GEN` does not clutter the stack. Thus, if your computation is small enough (i.e. you call few PARI routines, or most of them return `long` integers), then you do not need to do any garbage collecting. This will probably be the case in many of your subroutines. Of course the objects that were on the stack *before* the function call are left alone. Except for the ones listed below, PARI functions only collect their own garbage.

• It may happen that all objects that were created after a certain point can be deleted — for instance, if the final result you need is not a `GEN`, or if some search proved futile. Then, it is enough to record the value of `avma` just *before* the first garbage is created, and restore it upon exit:

```
pari_sp av = avma; /* record initial avma */

garbage ...
avma = av; /* restore it */
```

All objects created in the `garbage` zone will eventually be overwritten: they should not be accessed anymore once `avma` has been restored.

• If you want to destroy (i.e. give back the memory occupied by) the *latest* PARI object on the stack (e.g. the latest one obtained from a function call), you can use the function

`void` **cgiv**`(GEN z)`

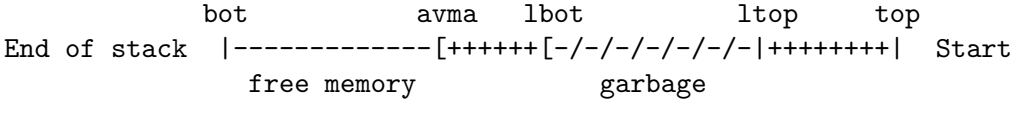
where `z` is the object you want to give back.

• Unfortunately life is not so simple, and sometimes you will want to give back accumulated garbage *during* a computation without losing recent data. For this you need the `gerepile` function (or one of its variants described hereafter):

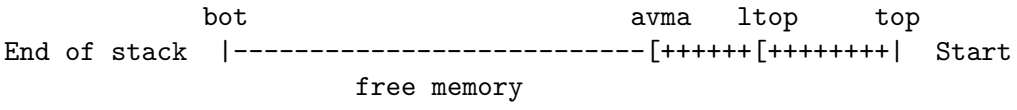
`GEN` **gerepile**`(pari_sp ltop, pari_sp lbot, GEN q)`

This function cleans up the stack between `ltop` and `lbot`, where `lbot` $<$ `ltop`, and returns the updated object `q`. This means:

1) we translate (copy) all the objects in the interval [`avma`, `lbot`[, so that its right extremity abuts the address `ltop`. Graphically



becomes:



where `++` denote significant objects, `--` the unused part of the stack, and `-/-` the garbage we remove.

2) The function then inspects all the PARI objects between `avma` and `lbot` (i.e. the ones that we want to keep and that have been translated) and looks at every component of such an object which is not a codeword. Each such component is a pointer to an object whose address is either

— between `avma` and `lbot`, in which case it will be suitably updated,

— larger than or equal to `ltop`, in which case it will not change, or

— between `lbot` and `ltop` in which case `gerepile` will scream an error message at you ("significant pointers lost in gerepile").

3) **avma** is updated (we add `ltop` − `lbot` to the old value).

4) We return the (possibly updated) object `q`: if `q` initially pointed between `avma` and `lbot`, we return the translated address, as in 2). If not, the original address is still valid (and we return it!).

As stated above, no component of the remaining objects (in particular `q`) should belong to the erased segment [`lbot`, `ltop`[, and this is checked within `gerepile`. But beware as well that the addresses of all the objects in the translated zone will have changed after a call to `gerepile`: every pointer you may have kept around elsewhere, outside the stack objects, which previously pointed into the zone below `ltop` must be discarded. If you need to recover more than one object, use one of the `gerepilemany` functions below.

As a consequence of the preceding explanation, we must now state the most important law about programming in PARI:

If a given PARI object is to be relocated by gerepile then, apart from universal objects, the chunks of memory used by its components should be in consecutive memory locations. All `GEN`s created by documented PARI function are guaranteed to satisfy this.

This is because the `gerepile` function knows only about *two connected zones*: the garbage that will be erased (between `lbot` and `ltop`) and the significant pointers that will be copied and updated. If there is garbage interspersed with your objects, disasters will occur when we try to update them and consider the corresponding "pointers". So be *very* wary when you allow objects to become disconnected. Have a look at the examples, it is not as complicated as it seems.

In practice this is achieved by the following programming idiom:

```
ltop=avma; garbage(); lbot=avma; q=anything();
return gerepile(ltop, lbot, q); /* returns the updated q */
```

Beware that

```
ltop=avma; garbage();
return gerepile(ltop, avma, anything())
```

might work, but should be frowned upon. We cannot predict whether `avma` is going to be evaluated after or before the call to `anything()`: it depends on the compiler. If we are out of luck, it will be *after* the call, so the result will belong to the garbage zone and the `gerepile` statement becomes equivalent to `avma = ltop`. Thus we would return a pointer to random garbage.

• A simple variant is

`GEN` **gerepileupto**`(pari_sp ltop, GEN q)`

which cleans the stack between `ltop` and the *connected* object `q` and returns `q` updated. For this to work, `q` must have been created *before* all its components, otherwise they would belong to the

garbage zone! Documented PARI functions guarantee this. If you stumble upon one that does not, consider it a bug worth reporting.

• Another variant (a special case of `gerepilemany` below, where $n = 1$) is

`GEN` **gerepilecopy**`(pari_sp ltop, GEN x))`

which is functionnally equivalent to `gerepileupto(ltop, gcopy(x))` but more efficient. In this case, the `GEN` parameter `x` need not satisfy any property before the garbage collection (it may be disconnected, components created before the root and so on). Of course, this is less efficient than either `gerepileupto` or `gerepile`, because `x` has to be copied to a clean stack zone first.

• To cope with complicated cases where many objects have to be preserved, you can use

`void` **gerepileall**`(pari_sp ltop, int n, ...)`

where the routine expects n further arguments, which are the *addresses* of the `GEN`s you want to preserve. It cleans up the most recent part of the stack (between `ltop` and `avma`), updating all the `GEN`s added to the argument list. A copy is done just before the cleaning to preserve them, so they do not need to be connected before the call. With `gerepilecopy`, this is the most robust of the `gerepile` functions (the less prone to user error), but also the slowest.

An alternative syntax, obsolete but kept for backward compatibility, is given by

`void` **gerepilemany**`(pari_sp ltop, GEN *gptr[], int n)`

which works exactly as above, except that the preserved `GEN`s are the elements of the array `gptr` (of length n): `gptr[0]`, `gptr[1]`, ..., `gptr[`n`-1]`.

• More efficient, but tricky to use is

`void` **gerepilemanysp**`(pari_sp ltop, pari_sp lbot, GEN *gptr[], int n)`

which cleans the stack between `lbot` and `ltop` and updates the `GEN`s pointed at by the elements of `gptr` without doing any copying. This is subject to the same restrictions as `gerepile`, the only difference being that more than one address gets updated.

4.4.2 Examples.

4.4.2.1 gerepile

Let `x` and `y` be two preexisting PARI objects and suppose that we want to compute $\texttt{x}^2+\texttt{y}^2$. This can trivially be done using the following program (we skip the necessary declarations; everything in sight is a `GEN`):

```
p1 = gsqr(x);
p2 = gsqr(y); z = gadd(p1,p2);
```

The `GEN z` indeed points at the desired quantity. However, consider the stack: it contains as unnecessary garbage `p1` and `p2`. More precisely it contains (in this order) `z`, `p2`, `p1`. (Recall that, since the stack grows downward from the top, the most recent object comes first.) We need a way to get rid of this garbage (in this case it causes no harm except that it occupies memory space, but in other cases it could disconnect other PARI objects and this is dangerous).

It would not have been possible to get rid of `p1`, `p2` before `z` is computed, since they are used in the final operation. We cannot record `avma` before `p1` is computed and restore it later, since this would destroy `z` as well. It is not possible either to use the function `cgiv` since `p1` and `p2` are not at the bottom of the stack and we do not want to give back `z`.

But using `gerepile`, we can give back the memory locations corresponding to `p1`, `p2`, and move the object `z` upwards so that no space is lost. Specifically:

```
ltop = avma; /* remember the current address of the top of the stack */
p1 = gsqr(x); p2 = gsqr(y);
lbot = avma; /* keep the address of the bottom of the garbage pile */
z = gadd(p1, p2); /* z is now the last object on the stack */
z = gerepile(ltop, lbot, z); /* garbage collecting */
```

Of course, the last two instructions could also have been written more simply:

```
z = gerepile(ltop, lbot, gadd(p1,p2));
```

In fact `gerepileupto` is even simpler to use, because the result of `gadd` will be the last object on the stack and `gadd` is guaranteed to return an object suitable for `gerepileupto`:

```
ltop = avma;
z = gerepileupto(ltop, gadd(gsqr(x), gsqr(y)));
```

As you can see, in simple conditions the use of `gerepile` is not really difficult. However make sure you understand exactly what has happened before you go on (use the figure from the preceding section).

Important remark: as we will see presently it is often necessary to do several `gerepile`s during a computation. However, the fewer the better. The only condition for `gerepile` to work is that the garbage be connected. If the computation can be arranged so that there is a minimal number of connected pieces of garbage, then it should be done that way.

For example suppose we want to write a function of two `GEN` variables `x` and `y` which creates the vector $[\mathtt{x}^2 + \mathtt{y}, \mathtt{y}^2 + \mathtt{x}]$. Without garbage collecting, one would write:

```
p1 = gsqr(x); p2 = gadd(p1, y);
p3 = gsqr(y); p4 = gadd(p3, x);
z = cgetg(3,t_VEC);
z[1] = (long)p2;
z[2] = (long)p4;
```

This leaves a dirty stack containing (in this order) `z`, `p4`, `p3`, `p2`, `p1`. The garbage here consists of `p1` and `p3`, which are separated by `p2`. But if we compute `p3` *before* `p2` then the garbage becomes connected, and we get the following program with garbage collecting:

```
ltop = avma; p1 = gsqr(x); p3 = gsqr(y); lbot = avma;
z = cgetg(3,t_VEC);
z[1] = ladd(p1,y);
z[2] = ladd(p3,x);
z = gerepile(ltop,lbot,z);
```

Finishing by `z = gerepileupto(ltop, z)` would be ok as well. But when you have the choice, it is usually clearer to brace the garbage between `ltop` / `lbot` pairs.

Beware that

```
ltop = avma; p1 = gadd(gsqr(x), y); p3 = gadd(gsqr(y), x);
z = cgetg(3,t_VEC);
z[1] = (long)p1;
z[2] = (long)p3
```

```
z = gerepileupto(ltop,z); /* WRONG !!! */
```

would be a disaster since `p1` and `p3` would be created before `z`, so the call to `gerepileupto` would overwrite them, leaving `z[1]` and `z[2]` pointing at random data!

We next want to write a program to compute the product of two complex numbers x and y, using the $3M$ method (which takes only 3 multiplications instead of 4). Let $z = x * y$, and set $x = x_r + i*x_i$ and similarly for y and z. We compute $p_1 = x_r*y_r$, $p_2 = x_i*y_i$, $p_3 = (x_r+x_i)*(y_r+y_i)$, and then we have $z_r = p_1 - p_2$, $z_i = p_3 - (p_1 + p_2)$. The program is essentially as follows:

```
ltop = avma;
p1 = gmul(x[1],y[1]);
p2 = gmul(x[2],y[2]);
p3 = gmul(gadd(x[1],x[2]), gadd(y[1],y[2]));
p4 = gadd(p1,p2); lbot = avma;
z = cgetg(3,t_COMPLEX);
z[1] = lsub(p1,p2);
z[2] = lsub(p3,p4);
z = gerepile(ltop,lbot,z);
```

"Essentially", because for instance `x[1]` is a `long` and not a `GEN`, so we need to insert many annoying typecasts: `p1 = gmul((GEN)x[1], (GEN)y[1])` and so on.

Let us now look at a less trivial example where more than one `gerepile` is needed in practice. (At the expense of efficiency, one can always use only one using `gerepilecopy`.) Suppose that we want to write a function which multiplies a line vector by a matrix. Such a function is of course already part of `gmul`, but let us ignore this for a moment. Then the most natural way is to do a `cgetg` of the result immediately, and then a `gerepile` for each coefficient of the result vector to get rid of the garbage which has accumulated while this particular coefficient was computed. We leave the details to the reader, who can look at the answer in the file `basemath/gen1.c`, in the function `gmul`, case `t_VEC` times case `t_MAT`. It would theoretically be possible to have a single connected piece of garbage, but it would be a much less natural and unnecessarily complicated program, which would in fact be slower.

4.4.2.2 gerepilemany

Let us now see why we may need the `gerepilemany` variants. Although it is not an infrequent occurrence, we will not give a specific example but a general one: suppose that we want to do a computation (usually inside a larger function) producing more than one PARI object as a result, say two for instance. Then even if we set up the work properly, before cleaning up we will have a stack which has the desired results `z1`, `z2` (say), and then connected garbage from lbot to ltop. If we write

```
z1 = gerepile(ltop, lbot, z1);
```

then the stack will be cleaned, the pointers fixed up, but we will have lost the address of `z2`. This is where we need one of the gerepilemany functions: we declare

```
GEN *gptr[2]; /* Array of pointers to GENs */
gptr[0] = &z1; gptr[1] = &z2;
```

and now the call `gerepilemany(ltop, gptr, 2)` copies `z1` and `z2` to new locations, cleans the stack from `ltop` to the old `avma`, and updates the pointers `z1` and `z2`.

An equivalent, slighly slower, formulation is to simply write

```
gerepileall(ltop, 2, &z1, &z2)
```

so that the array `gptr` is in fact not needed.

Here we do not assume anything about the stack: the garbage can be disconnected and `z1`, `z2` need not be at the bottom of the stack. If all of these assumptions are in fact satisfied, then we can call `gerepilemanysp` instead, which will usually be faster since we do not need the initial copy (on the other hand, it is less cache friendly).

Another important usage is "random" garbage collection during loops whose size requirements we cannot (or do not bother to) control in advance:

```
pari_sp ltop = avma, limit = stack_lim(avma, 1);
GEN x, y;

while (...)
{
  garbage(); x = anything();
  garbage(); y = anything()
  garbage();
  if (avma < limit) /* memory is running low (half spent since entry) */
    gerepileall(ltop, 2, &x, &y);
}
```

Here we assume that only `x` and `y` are needed from one iteration to the next. As it would be too costly to call gerepile once for each iteration, we only do it when it seems to have become necessary.

The macro `stack_lim(avma,`n`)` denotes an address where $2^{n-1}/(2^{n-1}+1)$ of the remaining stack space is exhausted (1/2 for $n = 1$, 2/3 for $n = 2$).

4.4.3 Some hints and tricks. In this section, we give some indications on how to avoid most problems connected with garbage collecting:

First, although it looks complicated, `gerepile` has turned out to be a very flexible and fast garbage collector, which compares very favorably with much more sophisticated methods used in other systems. Our benchmarks indicate that the price paid for using `gerepile`, when properly used, is usually around 1 or 2 percents of the total running time, which is quite acceptable!

Secondly, in many cases, in particular when the tree structure and the size of the PARI objects which will appear in a computation are under control, one can avoid `gerepile` altogether by creating sufficiently large objects at the beginning (using **cgetg**), and then using assignment statements and operations ending with z (such as **gaddz**). Coming back to our first example, note that if we know that x and y are of type real and of length less than or equal to 5, we can program without using `gerepile` at all:

```
z = cgetr(5); ltop = avma;
p1 = gsqr(x); p2 = gsqr(y); gaddz(p1,p2,z);
avma = ltop;
```

This practice will usually be *slower* than a craftily used `gerepile` though, and is certainly more cumbersome to use. As a rule, assignment statements should generally be avoided.

Thirdly, the philosophy of `gerepile` is the following: keep the value of the stack pointer `avma` at the beginning, and just *before* the last operation. Afterwards, it would be too late since the lower

end address of the garbage zone would have been lost. Of course you can always use `gerepileupto`, but you will have to assume that the object was created *before* its components.

Lastly, if all seems lost, just use `gerepilecopy` (or `gerepileall`) to fix up the stack for you.

If you followed us this far, congratulations, and rejoice: the rest is much easier.

4.5 Implementation of the PARI types.

Although it is a little tedious, we now go through each type and explain its implementation. Let `z` be a `GEN`, pointing at a PARI object. In the following paragraphs, we will constantly mix two points of view: on the one hand, `z` will be treated as the C pointer it is (in the context of program fragments like `z[1]`), on the other, as PARI's handle on (the internal representation of) some mathematical entity, so we will shamelessly write $z \neq 0$ to indicate that the *value* thus represented is nonzero (in which case the *pointer* `z` certainly will be non-`NULL`). We offer no apologies for this style. In fact, you had better feel comfortable juggling both views simultaneously in your mind if you want to write correct PARI programs.

Common to all the types is the first codeword `z[0]`, which we do not have to worry about since this is taken care of by `cgetg`. Its precise structure will depend on the machine you are using, but it always contain the following data: the *internal type number* associated to the symbolic type name, the *length* of the root in longwords, and a technical bit which indicates whether the object is a clone (see below) or not. This last one is used by GP for internal garbage collecting, you will not have to worry about it.

These data can be handled through the following *macros*:

`long` **typ**`(GEN z)` returns the type number of `z`.

`void` **settyp**`(GEN z, long n)` sets the type number of `z` to `n` (you should not have to use this function if you use `cgetg`).

`long` **lg**`(GEN z)` returns the length (in longwords) of the root of `z`.

`long` **setlg**`(GEN z, long l)` sets the length of `z` to `l` (you should not have to use this function if you use `cgetg`; however, see an advanced example in Section 4.8).

(If you know enough about PARI to need to access the "clone" bit, then you will be able to find usage hints in the code. It *is* technical after all.)

These macros are written in such a way that you do not need to worry about type casts when using them: i.e. if `z` is a `GEN`, `typ(z[2])` will be accepted by your compiler, without your having to explicitly type `typ((GEN)z[2])`. Note that for the sake of efficiency, none of the codeword-handling macros check the types of their arguments even when there are stringent restrictions on their use.

The possible second codeword is used differently by the different types, and we will describe it as we now consider each of them in turn:

4.5.1 Type `t_INT` (integer): this type has a second codeword `z[1]` which contains the following information:

the sign of `z`: coded as 1, 0 or -1 if `z` > 0, `z` $= 0$, `z` < 0 respectively.

the *effective length* of `z`, i.e. the total number of significant longwords. This means the following: apart from the integer 0, every integer is "normalized", meaning that the first mantissa longword (i.e. `z[2]`) is non-zero. However, the integer may have been created with a longer length. Hence the "length" which is in `z[0]` can be larger than the "effective length" which is in `z[1]`. Accessing `z[i]` for `i` larger than or equal to the effective length will yield random results.

This information is handled using the following macros:

`long` **signe**`(GEN z)` returns the sign of `z`.

`void` **setsigne**`(GEN z, long s)` sets the sign of `z` to `s`.

`long` **lgefint**`(GEN z)` returns the effective length of `z`.

`void` **setlgefint**`(GEN z, long l)` sets the effective length of `z` to `l`.

The integer 0 can be recognized either by its sign being 0, or by its effective length being equal to 2. When `z` $\neq 0$, the word `z[2]` exists and is non-zero, and the absolute value of `z` is (`z[2]`,`z[3]`,...,`z[lgefint(z)-1]`) in base `2^BITS_IN_LONG`, where as usual in this notation `z[2]` is the highest order longword.

The following further macros are available:

`long` **mpodd**`(GEN x)` which is 1 if `x` is odd, and 0 otherwise.

`long` **mod2**`(GEN x)`, **mod4**`(x)`, and so on up to **mod64**`(x)`, which give the residue class of `x` modulo the corresponding power of 2, for *positive* `x`. By definition, `mod`$n(x) :=$ `mod`$n(|x|)$ for $x < 0$ (the macros disregard the sign), and the result is undefined if $x = 0$.

These macros directly access the binary data and are thus much faster than the generic modulo functions. Besides, they return long integers instead of `GEN`s, so they do not clutter up the stack.

4.5.2 Type `t_REAL` (real number): this type has a second codeword z[1] which also encodes its sign (obtained or set using the same functions as for the integers), and a binary exponent. This exponent can be handled using the following macros:

`long` **expo**`(GEN z)` returns the exponent of `z`. This is defined even when `z` is equal to zero, see Section 1.2.6.3.

`void` **setexpo**`(GEN z, long e)` sets the exponent of `z` to `e`.

Note the functions:

`long` **gexpo**`(GEN z)` which tries to return an exponent for `z`, even if `z` is not a real number.

`long` **gsigne**`(GEN z)` which returns a sign for `z`, even when `z` is neither real nor integer (a rational number for instance).

The real zero is characterized by having its sign equal to 0. If `z` is not equal to 0, then is is represented as $2^e M$, where e is the exponent, and $M \in [1, 2[$ is the mantissa of z, whose digits are stored in `z`$[2], \ldots,$ `z`$[$`lg(z)` $- 1]$.

More precisely, let m be the integer (z[2],..., z[lg(z)-1]) in base 2^BITS_IN_LONG; here, z[2] is the most significant longword and is normalized, i.e. its most significant bit is 1. Then we have $M := m \cdot 2^{1-\texttt{bit\_accuracy(lg(z))}}$.

Thus, the real number 3.5 to accuracy bit_accuracy(lg(z)) is represented as z[0] (encoding type = t_REAL, lg(z)), z[1] (encoding sign = 1, expo = 1), z[2] = 0xe0000000, z[3] = ... = z[lg(z) − 1] = 0x0.

4.5.3 Type t_INTMOD (integermod): z[1] points to the modulus, and z[2] at the number representing the class z. Both are separate GEN objects, and both must be of type integer, satisfying the inequality $0 \leq \texttt{z}[2] < \texttt{z}[1]$.

It is good practice to keep the modulus object on the heap, so that new integermods resulting from operations can point at this common object, instead of carrying along their own copies of it on the stack. The library functions implement this practice almost by default.

4.5.4 Type t_FRAC and t_FRACN (rational number): z[1] points to the numerator, and z[2] to the denominator. Both must be of type integer. In principle $\texttt{z}[2] > 0$, but this rule does not have to be strictly obeyed. Note that a type t_FRACN rational number can be converted to irreducible form using the function GEN **gred**(GEN x).

4.5.5 Type t_COMPLEX (complex number): z[1] points to the real part, and z[2] to the imaginary part. A priori z[1] and z[2] can be of any type, but only certain types are useful and make sense.

4.5.6 Type t_PADIC (p-adic numbers): this type has a second codeword [1] which contains the following information: the p-adic precision (the exponent of p modulo which the p-adic unit corresponding to z is defined if z is not 0), i.e. one less than the number of significant p-adic digits, and the exponent of z. This information can be handled using the following functions:

long **precp**(GEN z) returns the p-adic precision of z.

void **setprecp**(GEN z, long l) sets the p-adic precision of z to l.

long **valp**(GEN z) returns the p-adic valuation of z (i.e. the exponent). This is defined even if z is equal to 0, see Section 1.2.6.3.

void **setvalp**(GEN z, long e) sets the p-adic valuation of z to e.

In addition to this codeword, z[2] points to the prime p, z[3] points to $p^{\text{precp(z)}}$, and z[4] points to an integer representing the p-adic unit associated to z modulo z[3] (and points to zero if z is zero). To summarize, if $z \neq 0$, we have the equality:

$$\texttt{z} = p^{\text{valp(z)}} * (\texttt{z}[4] + O(\texttt{z}[3])) = p^{\text{valp(z)}} * (\texttt{z}[4] + O(p^{\text{precp(z)}}))$$

4.5.7 Type t_QUAD (quadratic number): z[1] points to the polynomial defining the quadratic field, z[2] to the "real part" and z[3] to the "imaginary part", which are to be taken as the coefficients of z with respect to the "canonical" basis $(1, w)$, see Section 1.2.3. Complex numbers are a particular case of quadratics but deserve a separate type.

4.5.8 Type `t_POLMOD` (polmod): exactly as for integermods, `z[1]` points to the modulus, and `z[2]` to a polynomial representing the class of `z`. Both must be of type `t_POL` in the same variable. However, `z[2]` is allowed to be a simplification of such a polynomial, e.g a scalar. This is quite tricky considering the hierarchical structure of the variables; in particular, a polynomial in variable of *lesser* priority (see Section 2.6.2) than the modulus variable is valid, since it can be considered as the constant term of a polynomial of degree 0 in the correct variable. On the other hand a variable of *greater* priority would not be acceptable; see Section 2.6.2 for the problems which may arise.

4.5.9 Type `t_POL` (polynomial): this type has a second codeword which is analogous to the one for integers. It contains a "*sign*": 0 if the polynomial is equal to 0, and 1 if not (see however the important remark below), a *variable number* (e.g. 0 for x, 1 for y, etc...), and an *effective length*.

These data can be handled with the following macros:

signe and **setsigne** as for reals and integers.

`long` **lgef**`(GEN z)` returns the effective length of `z`.

`void` **setlgef**`(GEN z, long l)` sets the effective length of `z` to `l`.

`long` **varn**`(GEN z)` returns the variable number of the object `z`.

`void` **setvarn**`(GEN z, long v)` sets the variable number of `z` to `v`.

The variable numbers encode the relative priorities of variables as discussed in Section 2.6.2. We will give more details in Section 4.6. Note also the function `long` **gvar**`(GEN z)` which tries to return a variable number for `z`, even if `z` is not a polynomial or power series. The variable number of a scalar type is set by definition equal to `BIGINT`.

The components `z[2]`, `z[3]`,...`z[lgef(z)-1]` point to the coefficients of the polynomial *in ascending order*, with `z[2]` being the constant term and so on. Note that the degree of the polynomial is equal to its effective length minus three. The function

`long` **degree**`(GEN x)` returns the degree of `x` with respect to its main variable even when `x` is not a polynomial (a rational function for instance). By convention, the degree of 0 is -1.

Important remark. A zero polynomial can be characterized by the fact that its sign is 0. However, its effective length may be equal to 2, or greater than 2. If it is greater than 2, this means that all the coefficients of the polynomial are equal to zero (as they should for a zero polynomial), but not all of these zeros are exact zeros, and more precisely the leading term `z[lgef(z)-1]` is not an exact zero.

4.5.10 Type `t_SER` (power series): This type also has a second codeword, which encodes a "*sign*", i.e. 0 if the power series is 0, and 1 if not, a *variable number* as for polynomials, and an *exponent*. This information can be handled with the following functions: **signe**, **setsigne**, **varn**, **setvarn** as for polynomials, and **valp**, **setvalp** for the exponent as for p-adic numbers. Beware: do *not* use **expo** and **setexpo** on power series.

If the power series is non-zero, `z[2]`, `z[3]`,...`z[lg(z)-1]` point to the coefficients of `z` in ascending order, `z[2]` being the first non-zero coefficient. Note that the exponent of a power series can be negative, i.e. we are then dealing with a Laurent series (with a finite number of negative terms).

4.5.11 Type t_RFRAC and t_RFRACN (rational function): z[1] points to the numerator, and z[2] on the denominator. The denominator must be of type polynomial. Note that a type t_RFRACN rational function can be converted to irreducible form using the function **gred**.

4.5.12 Type t_QFR (indefinite binary quadratic form): z[1], z[2], z[3] point to the three coefficients of the form and should be of type integer. z[4] is Shanks's distance function, and should be of type real.

4.5.13 Type t_QFI (definite binary quadratic form): z[1], z[2], z[3] point to the three coefficients of the form. All three should be of type integer.

4.5.14 Type t_VEC and t_COL (vector): z[1], z[2],...z[lg(z)-1] point to the components of the vector.

4.5.15 Type t_MAT (matrix): z[1], z[2],...z[lg(z)-1] point to the column vectors of z, i.e. they must be of type t_COL and of the same length.

The next two types were introduced for specific GP use, and you will be much better off using the standard malloc'ed C constructs when programming in library mode. We quote them just for completeness, advising you not to use them:

4.5.16 Type t_LIST (list): This one has a second codeword which contains an effective length (handled through **lgef** / **setlgef**). z[2],..., z[lgef(z)-1] contain the components of the list.

4.5.17 Type t_STR (character string): char * **GSTR**(z) (= (z+1)) points to the first character of the (NULL-terminated) string.

Implementation note: for the types including an exponent (or a valuation), we actually store a biased non-negative exponent (bit-ORing the biased exponent to the codeword), obtained by adding a constant to the true exponent: either HIGHEXPOBIT (for t_REAL) or HIGHVALPBIT (for t_PADIC and t_SER). Of course, this is encapsulated by the exponent/valuation-handling macros and need not concern the library user.

4.6 PARI variables.

4.6.1 Multivariate objects

We now consider variables and formal computations, and give the technical details corresponding to the general discussion in Section 2.6.2. As we have seen in Section 4.5, the codewords for types t_POL and t_SER encode a "variable number". This is an integer, ranging from 0 to MAXVARN. The lower it is, the higher the variable priority.

In fact, the way an object will be considered in formal computations depends entirely on its "principal variable number" which is given by the function

long **gvar**(GEN z)

which returns a variable number for z, even if z is not a polynomial or power series. The variable number of a scalar type is set by definition equal to BIGINT which is bigger than any legal variable number. The variable number of a recursive type which is not a polynomial or power series is the minimal variable number of its components. But for polynomials and power series only the

"outermost" number counts (we directly access `varn`(x) in the codewords): the representation is not symmetrical at all.

Under GP, one need not worry too much since the interpreter will define the variables as it sees them* and do the right thing with the polynomials produced (however, have a look at the remark in Section 2.3.8).

But in library mode, they are tricky objects if you intend to build polynomials yourself (and not just let PARI functions produce them, which is usually less efficient). For instance, it does not make sense to have a variable number occur in the components of a polynomial whose main variable has a higher number (lower priority), even though there is nothing PARI can do to prevent you from doing it; see Section 2.6.2 for a discussion of possible problems in a similar situation.

4.6.2 Creating variables A basic difficulty is to "create" a variable. As we have seen in Section 4.1, a plethora of objects is associated to variable number v. Here is the complete list: **polun**[v] and **polx**[v], which you can use in library mode and which represent, respectively, the monic monomials of degrees 0 and 1 in v; **varentries**[v], and **polvar**[v]. The latter two are only meaningful to GP, but they have to be set nevertheless. All of them must be properly defined before you can use a given integer as a variable number.

Initially, this is done for 0 (the variable `x` under GP), and `MAXVARN`, which is there to address the need for a "temporary" new variable in library mode and cannot be input under GP. No documented library function can create from scratch an object involving `MAXVARN` (of course, if the operands originally involve `MAXVARN`, the function will abide). We call the latter type a "temporary variable". The regular variables meant to be used in regular objects, are called "user variables".

4.6.2.1 User variables: When the program starts, `x` is the only user variable (number 0). To define new ones, use

`long` **fetch_user_var**`(char *`s`)`

which inspects the user variable named s (creating it if needed), and returns its variable number.

```
long v = fetch_user_var("y");
GEN gy = polx[v];
```

This function raises an error if s is already known as a function name to the interpreter.

Caveat: it is possible to use **flissexpr** (see Section 4.7.1) to execute a GP command and create GP variables on the fly as needed:

```
GEN gy = flissexpr("y"); /* supposedly returns polx[v], for some v */
long v = gvar(gy);
```

This is dangerous, especially when programming functions that will be used under GP. The code above reads the value of `y`, as it is currently known by the GP interpreter (possibly creating it in the process). All is well and good if `y` has not been tampered with in previous GP commands. But if `y` has been modified (e.g `y = 1`), then the value of `gy` is not what you expected it to be and corresponds instead to the current value of the GP variable (e.g `gun`).

* More precisely, the first time a given identifier is read by the GP parser (and is not immediately interpreted as a function) a new variable is created, and it is assigned a strictly lower priority than any variable in use at this point. On startup, before any user input has taken place, 'x' is defined in this way and will thus always have maximal priority (and variable number 0).

4.6.2.2 Temporary variables: MAXVARN is available, but is better left to pari internal functions (some of which do not check that MAXVARN is free for them to use, which can be considered a bug). You can create more temporary variables using

`long` **fetch_var**`()`

This returns a variable number which is guaranteed to be unused by the library at the time you get it and as long as you do not delete it (we will see how to do that shortly). This has *lower* number (i.e. *higher* priority) than any temporary variable produced so far (MAXVARN is assumed to be the first such). This call updates all the aforementioned internal arrays. In particular, after the statement `v = fetch_var()`, you can use `polun[v]` and `polx[v]`. The variables created in this way have no identifier assigned to them though, and they will be printed as #<number>, except for MAXVARN which will be printed as #. You can assign a name to a temporary variable, after creating it, by calling the function

`void` **name_var**`(long n, char *s)`

after which the output machinery will use the name `s` to represent the variable number `n`. The GP parser will *not* recognize it by that name, however, and calling this on a variable known to GP will raise an error. Temporary variables are meant to be used as free variables, and you should never assign values or functions to them as you would do with variables under GP. For that, you need a user variable.

All objects created by `fetch_var` are on the heap and not on the stack, thus they are not subject to standard garbage collecting (they will not be destroyed by a `gerepile` or `avma = ltop` statement). When you do not need a variable number anymore, you can delete it using

`long` **delete_var**`()`

which deletes the *latest* temporary variable created and returns the variable number of the previous one (or simply returns 0 if you try, in vain, to delete MAXVARN). Of course you should make sure that the deleted variable does not appear anywhere in the objects you use later on. Here is an example:

```
{
  long first = fetch_var();
  long n1 = fetch_var();
  long n2 = fetch_var(); /* prepare three variables for internal use */
  ...
  /* delete all variables before leaving */
  do { num = delete_var(); } while (num && num <= first);
}
```

The (dangerous) statement

```
while (delete_var()) /* empty */;
```

removes all temporary variables that were in use, except MAXVARN which cannot be deleted.

4.7 Input and output.

Two important aspects have not yet been explained which are specific to library mode: input and output of PARI objects.

4.7.1 Input.

For input, PARI provides you with two powerful high level functions which enables you to input your objects as if you were under GP. In fact, the second one *is* essentially the GP syntactical parser, hence you can use it not only for input but for (most) computations that you can do under GP. These functions are called **flisexpr** and **flisseq**. The first one has the following syntax:

`GEN` **flisexpr**`(char *s)`

Its effect is to analyze the input string s and to compute the result as in GP. However it is limited to one expression. If you want to read and evaluate a sequence of expressions, use

`GEN` **flisseq**`(char *s)`

In fact these two functions start by *filtering* out all spaces and comments in the input string (that is what the initial `f` stands for). They then call the underlying basic functions, the GP parser proper: `GEN` **lisexpr**`(char *s)` and `GEN` **lisseq**`(char *s)`, which are slightly faster but which you probably do not need.

To read a `GEN` from a file, you can use the simpler interface

`GEN` **lisGEN**`(FILE *file)`

which reads a character string of arbitrary length from the stream `file` (up to the first newline character), applies `flisexpr` to it, and returns the resulting `GEN`. This way, you will not have to worry about allocating buffers to hold the string. To interactively input an expression, use `lisGEN(stdin)`. This function returns `NULL` if `EOF` is encountered before a complete expression could be read.

Once in a while, it may be necessary to evaluate a GP expression sequence involving a call to a function you have defined in C. This is easy using **install** which allows you to manipulate quite an arbitrary function (GP knows about pointers!). The syntax is

`void` **install**`(void *f, char *name, char *code)`

where `f` is the (address of) the function (cast to the C type `void*`), `name` is the name by which you want to access your function from within your GP expressions, and `code` is a character string describing the function call prototype (see Section 4.9.2 for the precise description of prototype strings). In case the function returns a `GEN`, it should satisfy `gerepileupto` assumptions (see Section 4.4).

4.7.2 Output.

For output, there exist essentially three different functions (with variants), corresponding to the three main GP output formats (as described in Section 2.1.16), plus three extra ones, respectively devoted to TEX output, string output, and (advanced) debugging.

• "raw" format, obtained by using the function **brute** with the following syntax:

`void` **brute**`(GEN obj, char x, long n)`

This prints the PARI object `obj` in format `x0.n`, using the notations from Section 2.1.9. Recall that here `x` is either `'e'`, `'f'` or `'g'` corresponding to the three numerical output formats, and `n` is the number of printed significant digits, and should be set to -1 if all of them are wanted (these arguments only affect the printing of real numbers). Usually you will not need that much flexibility, so most of the time you will get by with the function

`void` **outbrute**`(GEN obj)`, which is equivalent to `brute(x,'g',-1)`,

or even better, with

`void` **output**`(GEN obj)` which is equivalent to `outbrute(obj)` followed by a newline and a buffer flush. This is especially nice during debugging. For instance using `dbx` or `gdb`, if `obj` is a `GEN`, typing `print output(obj)` will enable you to see the content of `obj` (provided the optimizer has not put it into a register, but it is rarely a good idea to debug optimized code).

• "prettymatrix" format: this format is identical to the preceding one except for matrices. The relevant functions are:

`void` **matbrute**`(GEN obj, char x, long n)`

`void` **outmat**`(GEN obj)`, which is followed by a newline and a buffer flush.

• "prettyprint" format: the basic function has an additional parameter `m`, corresponding to the (minimum) field width used for printing integers:

`void` **sor**`(GEN obj, char x, long n, long m)`

The simplified version is

`void` **outbeaut**`(GEN obj)` which is equivalent to `sor(obj,'g',-1,0)` followed by a newline and a buffer flush.

• The first extra format corresponds to the **texprint** function of GP, and gives a TEX output of the result. It is obtained by using:

`void` **exe**`(GEN obj, char x, long n)`

• The second one is the function **GENtostr** which converts a PARI `GEN` to an ASCII string. The syntax is

`char*` **GENtostr**`(GEN obj)`, wich returns a `malloc`'ed character string (which you should `free` after use).

• The third and final one outputs the hexadecimal tree corresponding to the GP command `\x` using the function

`void` **voir**`(GEN obj, long nb)`, which will only output the first `nb` words corresponding to leaves (very handy when you have a look at big recursive structures). If you set this parameter to -1 all significant words will be printed. Usually this last type of output would only be used for debugging purposes.

Remark. Apart from **GENtostr**, all PARI output is done on the stream **outfile**, which by default is initialized to **stdout**. If you want that your output be directed to another file, you should use the function `void switchout(char *name)` where `name` is a character string giving the name of the file you are going to use. The output will be *appended* at the end of the file. In order to close the file, simply call `switchout(NULL)`.

Similarly, errors are sent to the stream **errfile** (**stderr** by default), and input is done on the stream **infile**, which you can change using the function **switchin** which is analogous to **switchout**.

(Advanced) Remark. All output is done according to the values of the **pariOut** / **pariErr** global variables which are pointers to structs of pointer to functions. If you really intend to use these, this probably means you are rewriting GP. In that case, have a look at the code in `language/es.c` (`init80()` or `GENtostr()` for instance).

4.7.3 Errors.

If you want your functions to issue error messages, you can use the general error handling routine **err**. The basic syntax is

```
err(talker, "error message");
```

This will print the corresponding error message and exit the program (in library mode; go back to the GP prompt otherwise). You can also use it in the more versatile guise

```
err(talker, format, ...);
```

where `format` describes the format to use to write the remaining operands, as in the **printf** function (however, see the next section). The simple syntax above is just a special case with a constant format and no remaining arguments.

The general syntax is

`void err(numerr,...)`

where `numerr` is a codeword which indicates what to do with the remaining arguments and what message to print. The list of valid keywords is in `language/errmessages.c` together with the basic corresponding message. For instance, `err(typeer,"matexp")` will print the message:

```
*** incorrect type in matexp.
```

Among the codewords are *warning* keywords (all those which start with the prefix `warn`). In that case, **err** does *not* abort the computation, just print the requested message and go on. The basic example is

```
err(warner, "Strategy 1 failed.  Trying strategy 2")
```

which is the exact equivalent of `err(talker,...)` except that you certainly do not want to stop the program at this point, just inform the user that something important has occurred (in particular, this output would be suitably highlighted under GP, whereas a simple `printf` would not).

4.7.4 Debugging output.

The global variables **DEBUGLEVEL** and **DEBUGMEM** (corresponding to the default **debug** and **debugmem**, see Section 2.1) are used throughout the PARI code to govern the amount of diagnostic and debugging output, depending on their values. You can use them to debug your own functions, especially after having made them accessible under GP through the command **install** (see Section 3.11.2.13).

For debugging output, you can use `printf` and the standard output functions (**brute** or **output** mainly), but also some special purpose functions which embody both concepts, the main one being

`void` **fprintferr**`(char *pariformat, ...)`

Now let us define what a PARI format is. It is a character string, similar to the one `printf` uses, where `%` characters have a special meaning. It describes the format to use when printing the remaining operands. But, in addition to the standard format types, you can use `%Z` to denote a `GEN` object (we would have liked to pick `%G` but it was already in use!). For instance you could write:

```
err(talker, "x[%d] = %Z is not invertible!", i, x[i])
```

since the **err** function accepts PARI formats. Here `i` is an `int`, `x` a `GEN` which is not a leaf and this would insert in raw format the value of the `GEN` `x[i]`.

4.7.5 Timers and timing output.

To profile your functions, you can use the PARI timer. The functions `long` **timer**`()` and `long` **timer2**`()` return the elapsed time since the last call of the same function (in milliseconds). Two different functions (identical except for their independent time-of-last-call memories!) are provided so you can have both global timing and fine tuned profiling.

You can also use `void` **msgtimer**`(char *format,...)`, which prints prints `Time`, then the remaining arguments as specified by `format` (which is a PARI format), then the output of `timer2`.

This mechanism is simple to use but not foolproof. If some other function uses these timers, and many PARI functions do use `timer2` when `DEBUGLEVEL` is high enough, the timings will be meaningless. To handle timing in a reentrant way, PARI defines a dedicated datatype, `pari_timer`. The functions

`long` **TIMER**`(pari_timer *T)`

`long` **msgTIMER**`(pari_timer *T, char *format,...)`

are equivalent to `timer` and `msgtimer` respectively, except they use a unique timer `T` containing all the information needed, so that no other function can mess with your timings. They are used as follows:

```
pari_timer T;
(void)TIMER(&T); /* initialize timer */
...
printf("Total time: %ld\n", TIMER(&T));
```

or

```
pari_timer T;
long i;
```

```
GEN L;
(void)TIMER(&T); /* initialize timer */
for (i = 1; i < 10; i++) {
  ...
  msgTIMER(&T, "for i = %ld (L[i] = %Z)", i, L[i]);
}
```

4.8 A complete program.

Now that the preliminaries are out of the way, the best way to learn how to use the library mode is to work through a detailed non-trivial example of a main program. We will write a program which computes the exponential of a square matrix x. The complete listing is given in Appendix B, but each part of the program will be produced and explained here. We will use an algorithm which is not optimal but is not far from the one used for the PARI function **gexp** (in fact embodied in the function `mpexp1`). This consists in calculating the sum of the series:

$$e^{x/(2^n)} = \sum_{k=0}^{\infty} \frac{(x/(2^n))^k}{k!}$$

for a suitable positive integer n, and then computing e^x by repeated squarings. First, we will need to compute the L^2-norm of the matrix x, i.e. the quantity:

$$z = \|x\|_2 = \sqrt{\sum x_{i,j}^2}.$$

We will then choose the integer n such that the L^2-norm of $x/(2^n)$ is less than or equal to 1, i.e.

$$n = \lceil \ln(z)/\ln(2) \rceil$$

if $z \geq 1$, and $n = 0$ otherwise. Then the series will converge at least as fast as the usual one for e^1, and the cutoff error will be easy to estimate. In fact a larger value of n would be preferable, but this is slightly machine dependent and more complicated, and will be left to the reader.

Let us start writing our program. So as to be able to use it in other contexts, we will structure it in the following way: a main program which will do the input and output, and a function which we shall call `matexp` which does the real work. The main program is easy to write. It can be something like this:

```
#include <pari.h>
GEN matexp(GEN x, long prec);
int
main()
{
  long d, prec = 3;
  GEN x;
  /* take a stack of 10^6 bytes, no prime table */
  pari_init(1000000, 2);
  printf("precision of the computation in decimal digits:\n");
  d = itos(lisGEN(stdin));
```

```
  if (d > 0) prec = (long) (d*pariK1+3);
  printf("input your matrix in GP format:\n");
  x = matexp(lisGEN(stdin), prec);
  sor(x, 'g', d, 0);
  exit(0);
}
```

The variable `prec` represents the length in longwords of the real numbers used. **pariK1** is a constant (defined in `paricom.h`) equal to $\ln(10)/(\ln(2) * \texttt{BITS\_IN\_LONG})$, which allows us to convert from a number of decimal digits to a number of longwords, independently of the actual bit size of your long integers. The function **lisGEN** reads an expression (here from standard input) and converts it to a `GEN`, like the GP parser itself would. This means it takes care of whitespace etc. in the input, and can do computations (e.g. `matid(2)` or `[1,0; 0,1]` are equally valid inputs).

Finally, `sor` is the general output routine. We have chosen to give `d` significant digits since this is what was asked for. Note that there is a trick hidden here: if a negative `d` was input, then the computation will be done in precision 3 (i.e. about 9.7 decimal digits for 32-bit machines and 19.4 for 64-bit machines) and in the function `sor`, giving a negative third argument outputs all the significant digits, which is entirely appropriate. Now let us attack the main course, the function `matexp`:

```
GEN
matexp(GEN x, long prec)
{
  pari_sp lbot, ltop = avma;
  long lx=lg(x),i,k,n;
  GEN y,r,s,p1,p2;
  /* check that x is a square matrix */
  if (typ(x) != t_MAT) err(talker,"this expression is not a matrix");
  if (lx == 1) return cgetg(1, t_MAT);
  if (lx != lg(x[1])) err(talker,"not a square matrix");
  /* compute the L_2 norm of x */
  s = gzero;
  for (i=1; i<lx; i++)
    s = gadd(s, gnorml2((GEN)x[i]));
  if (typ(s) == t_REAL) setlg(s,3);
  s = gsqrt(s,3); /* we do not need much precision on s */
  /* if s < 1, we are happy */
  k = expo(s);
  if (k < 0) { n = 0; p1 = x; }
  else { n = k+1; p1 = gmul2n(x,-n); setexpo(s,-1); }
```

Before continuing, several remarks are in order.

First, before starting this computation which will produce garbage on the stack, we have carefully saved the value of the stack pointer `avma` in `ltop`. Note that we are going to assume throughout that the garbage does not overflow the currently available stack. If it ever did, we would have several options — allocate a larger stack in the main program (for instance change 1000000 into 2000000), do some `gerepile`ing along the way, or (if you know what you are doing) use `allocatemoremem`.

Secondly, the **err** function is the general error handler for the PARI library. This will abort the program after printing the required message.

Thirdly, notice how we handle the special case `lx` = 1 (empty matrix) *before* accessing `lx(x[1])`. Doing it the other way round could produce a fatal error. Indeed, if `x` is of length 1, then `x[1]` is not a component of `x`. It is just the contents of the memory cell which happens to follow the one pointed to by `x`, and thus has no reason to be a valid `GEN`. Now recall that none of the codeword handling macros do any kind of type checking (see Section 4.5), thus **lg** would consider `x[1]` as a valid address, and try to access `*((GEN)x[1])` (the first codeword) which is unlikely to be a legal memory address.

In the fourth place, to compute the square of the L^2-norm of `x` we just add the squares of the L^2-norms of the column vectors which we obtain using the library function **gnorml2**. Had this function not existed, the norm computation would of course have been just as easy to write, but we would have needed a double loop.

We then take the square root of `s`, in precision 3 (the smallest possible). The `prec` argument of transcendental functions (here 3) is only taken into account when the arguments are *exact* objects, and thus no a priori precision can be determined from the objects themselves. To cater for this possibility, if `s` is of type `t_REAL`, we use the function **setlg** which effectively sets the precision of `s` to the required value. Note that here, since we are using a numeric value for a `cget` function, the program will run slightly differently on 32-bit and 64-bit machines: we want to use the smallest possible bit accuracy, and this is equal to `BITS_IN_LONG`.

Note that the matrix `x` is allowed to have complex entries, but the function `gnorml2` guarantees that `s` is a non-negative real number (not necessarily of type `t_REAL` of course). If we had not known this fact, we would simply have added the instruction `s = greal(s);` just after the `for` loop.

Note also that the function `gnorml2` works as desired on matrices, so we really did not need this loop at all (`s = gnorml2(x)` would have been enough), but we wanted to give examples of function usage. Similarly, it is of course not necessary to take the square root for testing whether the norm exceeds 1.

In the fifth place, note that we initialized the sum `s` to **gzero**, which is an exact zero. This is logical, but has some disadvantages: if all the entries of the matrix are integers (or rational numbers), the computation will take rather long, about twice as long as with real numbers of the same length. It would be better to initialize `s` to a real zero, using for instance the instructions:

```
s = cgetr(prec+1); gaffsg(0,s);
```

This raises the question: which real zero does this produce (have a look at Section 1.2.6.3)? In fact, the following choice has been made: it will give you the zero with exponent equal to −`BITS_IN_LONG` times the number of longwords in the mantissa, i.e. −`bit_accuracy`(`lg`(`s`)). Instead of the above idiom, you can also use the function `GEN` **realzero**`(long prec)`, which simply returns a real zero to accuracy −`bit_accuracy`(*prec*).

The sixth remark here is about how to determine the approximate size of a real number. The fastest way to do this is to look at its binary exponent. Hence we need to have `s` actually represented as a real number, and not as an integer or a rational number. The result of transcendental functions is guaranteed to be of type `t_REAL`, or complex with `t_REAL` components, thus this is indeed the case after the call to `gsqrt` since its argument is a nonnegative (real) number.

Finally, note the use of the function **gmul2n**. It has the following syntax:

`GEN` **gmul2n**`(GEN x, long n)`

and the effect is simply to multiply `x` by $2^{\mathtt{n}}$, where `n` can be positive or negative. This is much faster than `gmul` or `gmulgs`.

There is another function **gshift** with exactly the same syntax. When `n` is non-negative, the effects of these two functions are the same. However, when `n` is negative, `gshift` acts like a right shift of `-n`, hence does not normally perform an exact division on integers. The function `gshift` is the PARI analogue of the C or GP operators `<<` and `>>`.

We now come to the heart of the function. We have a `GEN p1` which points to a certain matrix of which we want to take the exponential. We will want to transform this matrix into a matrix with real (or complex of real) entries before starting the computation. To do this, we simply multiply by the real number 1 in precision `prec` + 1 (to be on the side of safety). To sum the series, we will use three variables: a variable `p2` which at stage k will contain $\mathtt{p1}^k/k!$, a variable `y` which will contain $\sum_{i=0}^{k} \mathtt{p1}^i/i!$, and a variable `r` which will contain the size estimate $\mathtt{s}^k/k!$. Note that we do not use Horner's rule. This is simply because we are lazy and do not want to compute in advance the number of terms that we need. We leave this modification (and many other improvements!) to the reader. The program continues as follows:

```
  /* initializations before the loop */
  r = cgetr(prec+1); gaffsg(1,r); p1 = gmul(r,p1);
  y = gscalmat(r,lx-1); /* creates scalar matrix with r on diagonal */
  p2 = p1; r = s; k = 1;
  y = gadd(y,p2);

  /* now the main loop */
  while (expo(r) >= -BITS_IN_LONG*(prec-1))
  {
    k++; p2 = gdivgs(gmul(p2,p1),k);
    r = gdivgs(gmul(s,r),k); y = gadd(y,p2);
  }

  /* now square back n times if necessary */
  if (!n) { lbot = avma; y = gcopy(y); }
  else
  {
    for (i=0; i<n; i++) { lbot = avma; y = gsqr(y); }
  }
  return gerepile(ltop,lbot,y);
}
```

A few remarks once again. First note the use of the function **gscalmat** with the following syntax:

`GEN` **gscalmat**`(GEN x, long m)`

The effect of this function is to create the $\mathtt{m} \times \mathtt{m}$ scalar matrix whose diagonal entries are `x`. Hence the length of the matrix including the codeword will in fact be `m+1`. There is a corresponding function **gscalsmat** which takes a long as a first argument.

If we refer to what has been said above, the main loop should be self-evident.

When we do the final squarings, according to the fundamental dogma on the use of **gerepile**, we keep the value of `avma` in `lbot` just *before* the squaring, so that if it is the last one, `lbot` will indeed be the bottom address of the garbage pile, and `gerepile` will work. Note that it takes a completely negligible time to do this in each loop compared to a matrix squaring. However, when

`n` is initially equal to 0, no squaring has to be done, and we have our final result ready but we lost the address of the bottom of the garbage pile. Hence we use the trick of copying `y` again to the top of the stack. This is inefficient, but does the trick. If we wanted to avoid this using only `gerepile`, the best thing to do would be to put the instruction `lbot=avma` just before both occurrences of the instruction `y=gadd(p2,y)`. Of course, we could also rewrite the last block as follows:

```
/* now square back n times */
for (i=0; i<n; i++) y = gsqr(y);
return gerepileupto(ltop,y);
```

because it does not matter to **gerepileupto** that we have lost the address just before the final result (note that the loop is not executed if `n` is 0). It is safe to use `gerepileupto` here as `y` will have been created by either `gsqr` or `gadd`, both of which are guaranteed to return suitable objects.

Remarks. As such, the program should work most of the time if `x` is a square matrix with real or complex entries. Indeed, since essentially the first thing that we do is to multiply by the real number 1, the program should work for integer, real, rational, complex or quadratic entries. This is in accordance with the behavior of transcendental functions.

Furthermore, since this program is intended to be only an illustrative example, it has been written a little sloppily. In particular many error checks have been omitted, and the efficiency is far from optimal. An evident improvement would be the use of `gerepileupto` mentioned above. Another improvement is to multiply the matrix x by the real number 1 right at the beginning, speeding up the computation of the L^2-norm in many cases. These improvements are included in the version given in Appendix B. Still another improvement would come from a better choice of `n`. If the reader takes a look at the implementation of the function `mpexp1` in the file `basemath/trans1.c`, he can make the necessary changes himself. Finally, there exist other algorithms of a different nature to compute the exponential of a matrix.

4.9 Adding functions to PARI.

4.9.1 Nota Bene. As mentioned in the `COPYING` file, modified versions of the PARI package can be distributed under the conditions of the GNU General Public License. If you do modify PARI, however, it is certainly for a good reason, hence we would like to know about it, so that everyone can benefit from it. There is then a good chance that the modifications that you have made will be incorporated into the next release.

(Recall the e-mail address: `pari@math.u-bordeaux.fr`, or use the mailing lists).

Roughly four types of modifications can be made. The first type includes all improvements to the documentation, in a broad sense. This includes correcting typos or inaccurracies of course, but also items which are not really covered in this document, e.g. if you happen to write a tutorial, or pieces of code exemplifying some fine points that you think were unduly omitted.

The second type is to expand or modify the configuration routines and skeleton files (the `Configure` script and anything in the `config/` subdirectory) so that compilation is possible (or easier, or more efficient) on an operating system previously not catered for. This includes discovering and removing idiosyncrasies in the code that would hinder its portability.

The third type is to modify existing (mathematical) code, either to correct bugs, to add new functionalities to existing functions, or to improve their efficiency.

Finally the last type is to add new functions to PARI. We explain here how to do this, so that in particular the new function can be called from GP.

4.9.2 The calling interface from GP, parser codes. A parser code is a character string describing all the GP parser needs to know about the function prototype. It contains a sequence of the following atoms:

• Syntax requirements, used by functions like `for`, `sum`, etc.:

`=` separator `=` required at this point (between two arguments)

• Mandatory arguments, appearing in the same order as the input arguments they describe:

`G` `GEN`

`&` `*GEN`

`L` long (we implicitly identify `int` with `long`)

`S` symbol (i.e. GP identifier name). Function expects a `*entree`

`V` variable (as `S`, but rejects symbols associated to functions)

`n` variable, expects a variable number (a `long`, not an `*entree`)

`I` string containing a sequence of GP statements (a *seq*), to be processed by `lisseq` (useful for control statements)

`E` string containing a *single* GP statement (an *expr*), to be processed by `lisexpr`

`r` raw input (treated as a string without quotes). Quoted args are copied as strings
Stops at first unquoted `')'` or `','`. Special chars can be quoted using '\'
Example: `aa"b\n)"c` yields the string `"aab\n)c"`

`s` expanded string. Example: `Pi"x"2` yields `"3.142x2"`
Unquoted components can be of any PARI type (converted following current output format)

• Optional arguments:

`s*` any number of strings, possibly 0 (see `s`)

`D`*xxx* argument has a default value

The `s*` code is technical and you probably do not need it, but we give its description for completeness. It reads all remaining arguments in *string context* (see Section 2.6.6), and sends a (`NULL`-terminated) list of `GEN*` pointing to these. The automatic concatenation rules in string context are implemented so that adjacent strings are read as different arguments, as if they had been comma-separated. For instance, if the remaining argument sequence is: `"xx" 1, "yy"`, the `s*` atom will send a `GEN *g = {&a, &b, &c, NULL}`, where a, b, c are `GEN`s of type `t_STR` (content `xx`), `t_INT` and `t_STR` (content `yy`).

The format to indicate a default value (atom starts with a `D`) is "`D`*value*`,`*type*`,`", where *type* is the code for any mandatory atom (previous group), *value* is any valid GP expression which is converted according to *type*, and the ending comma is mandatory. For instance `D0,L,` stands for "this optional argument will be converted to a `long`, and is `0` by default". So if the user-given argument reads `1 + 3` at this point, `(long)4` is sent to the function (via `itos`()); and `(long)0` if the argument is omitted. The following special syntaxes are available:

`DG` optional `GEN`, send `NULL` if argument omitted.

`D&` optional `*GEN`, send `NULL` if argument omitted.

`DV` optional `*entree`, send `NULL` if argument omitted.

`DI` optional `*char`, send `NULL` if argument omitted.

`Dn` optional variable number, -1 if omitted.

• Automatic arguments:

`f` Fake `*long`. C function requires a pointer but we do not use the resulting `long`

`p` real precision (default `realprecision`)

`P` series precision (default `seriesprecision`, global variable `precdl` for the library)

• Return type: `GEN` by default, otherwise the following can appear at the start of the code string:

`i`	return `int`
`l`	return `long`
`v`	return `void`

No more than 8 arguments can be given (syntax requirements and return types are not considered as arguments). This is currently hardcoded but can trivially be changed by modifying the definition of `argvec` in `anal.c:identifier()`. This limitation should disappear in future versions.

When the function is called under GP, the prototype is scanned and each time an atom corresponding to a mandatory argument is met, a user-given argument is read (GP outputs an error message it the argument was missing). Each time an optional atom is met, a default value is inserted if the user omits the argument. The "automatic" atoms fill in the argument list transparently, supplying the current value of the corresponding variable (or a dummy pointer).

For instance, here is how you would code the following prototypes (which do not involve default values):

```
GEN name(GEN x, GEN y, long prec)   ----> "GGp"
void name(GEN x, GEN y, long prec)  ----> "vGGp"
void name(GEN x, long y, long prec) ----> "vGLp"
long name(GEN x)                    ----> "lG"
int name(long x)                    ----> "iL"
```

If you want more examples, GP gives you easy access to the parser codes associated to all GP functions: just type `\h` *function*. You can then compare with the C prototypes as they stand in the code.

Remark: If you need to implement complicated control statements (probably for some improved summation functions), you will need to know about the **entree** type, which is not documented. Check the comment before the function list at the end of `language/init.c` and the source code in `language/sumiter.c`. You should be able to make something of it.

4.9.3 Coding guidelines. Code your function in a file of its own, using as a guide other functions in the PARI sources. One important thing to remember is to clean the stack before exiting your main function (usually using `gerepile`), since otherwise successive calls to the function will clutter the stack with unnecessary garbage, and stack overflow will occur sooner. Also, if it returns a `GEN` and you want it to be accessible to GP, you have to make sure this `GEN` is suitable for `gerepileupto` (see Section 4.4).

If error messages are to be generated in your function, use the general error handling routine `err` (see Section 4.7.3). Recall that, apart from the `warn` variants, this function does not return but ends with a `longjmp` statement. As well, instead of explicit `printf` / `fprintf` statements, use the following encapsulated variants:

`void` **pariputs**`(char *s)`: write `s` to the GP output stream.

`void` **fprintferr**`(char *s)`: write `s` to the GP error stream (this function is in fact much more versatile, see Section 4.7.4).

Declare all public functions in an appropriate header file, if you expect somebody will access them from C. For example, if dynamic loading is not available, you may need to modify PARI to access these functions, so put them in `paridecl.h`. The other functions should be declared `static` in your file.

Your function is now ready to be used in library mode after compilation and creation of the library. If possible, compile it as a shared library (see the `Makefile` coming with the `matexp` example in the distribution). It is however still inaccessible from GP.

4.9.4 Integration with GP as a shared module

To tell GP about your function, you must do the following. First, find a name for it. It does not have to match the one used in library mode, but consistency is nice. It has to be a valid GP identifier, i.e. use only alphabetic characters, digits and the underscore character (_), the first character being alphabetic.

Then you have to figure out the correct parser code corresponding to the function prototype. This has been explained above (Section 4.9.2).

Now, assuming your Operating System is supported by `install`, simply write a GP script like the following:

```
install(libname, code, gpname, library)
addhelp(gpname, "some help text")
```

(see Section 3.11.2.1 and 3.11.2.13). The addhelp part is not mandatory, but very useful if you want others to use your module. `libname` is how the function is named in the library, usually the same name as one visible from C.

Read that file from your GP session (from your preferences file for instance, see Section 2.9), and that's it, you can use the new function *gpname* under GP (and we would very much like to hear about it!).

4.9.5 Integration the hard way

If `install` is not available for your Operating System, things are more complicated: you have to hardcode your function in the GP binary (or install Linux). Here is what needs to be done:

You need to choose a section and add a file `functions/`*section*`/`*gpname* containing the following, keeping the notation above:

`Function:` *gpname*
`Section:` *section*
`C-Name:` *libname*
`Prototype:` *code*
`Help:` *some help text*

At this point you can rebuild the database by running `make Def` in the directory `desc`. Then, you can recompile GP.

4.9.6 Example. A complete description could look like this:

```
{
  install(bnfinit0, GD0,L,DGp, ClassGroupInit, "libpari.so");
  addhelp(ClassGroupInit, "ClassGroupInit(P,{flag=0},{data=[]}):
    compute the necessary data for ...");
}
```

which means we have a function `ClassGroupInit` under GP, which calls the library function `bnfinit0` . The function has one mandatory argument, and possibly two more (two `'D'` in the code), plus the current real precision. More precisely, the first argument is a `GEN`, the second one is converted to a `long` using `itos` (`0` is passed if it is omitted), and the third one is also a `GEN`, but we pass `NULL` if no argument was supplied by the user. This matches the C prototype (from `paridecl.h`):

```
GEN bnfinit0(GEN P, long flag, GEN data, long prec)
```

This function is in fact coded in `basemath/buch2.c`, and will in this case be completely identical to the GP function `bnfinit` but GP does not need to know about this, only that it can be found somewhere in the shared library `libpari.so`.

Important note: You see in this example that it is the function's responsibility to correctly interpret its operands: `data = NULL` is interpreted *by the function* as an empty vector. Note that since `NULL` is never a valid `GEN` pointer, this trick always enables you to distinguish between a default value and actual input: the user could explicitly supply an empty vector!

Note: If `install` is not available, we have to add a file

```
functions/number_fields/ClassGroupInit
```

containing the following:

```
Function: ClassGroupInit
Section: number_fields
C-Name: bnfinit0
Prototype: GD0,L,DGp
Help: ClassGroupInit(P,{flag=0},{tech=[]}): this routine does ...
```

Chapter 5:
Technical Reference Guide for Low-Level Functions

In this chapter, we give a description all public low-level functions of the PARI system. These essentially include functions for handling all the PARI types. Higher level functions, such as arithmetic or transcendental functions, are described fully in Chapter 3 of this manual.

Many other undocumented functions can be found throughout the source code. These private functions are more efficient than the library functions that call them, but much sloppier on argument checking and damage control. Use them at your own risk!

5.1 Level 0 kernel (operations on unsigned longs).

Level 0 operations simulate basic operations of the 68020 processor on which PARI was originally implemented. The type `ulong` is defined in the file `parigen.h` as `unsigned long`. Note that in the prototypes below a `ulong` is sometimes implicitly typecast to `int` or `long`.

The global `ulong` variables `overflow` (which will contain only 0 or 1) and `hiremainder` used to be declared in the file `pariinl.h`. However, for certain architectures they are no longer needed, and/or have been replaced with local variables for efficiency; and the 'functions' mentioned below are really chunks of assembler code which will be inlined at each invocation by the compiler. If you really need to use these lowest-level operations directly, make sure you know your way through the PARI kernel sources, and understand the architecture dependencies.

To make the following descriptions valid both for 32-bit and 64-bit machines, we will set `BIL` to be equal to 32 (resp. 64), an abbreviation of `BITS_IN_LONG`, which is what is actually used in the source code.

`ulong` **addll**`(ulong x, ulong y)` adds the ulongs `x` and `y`, returns the lower `BIL` bits and puts the carry bit into `overflow`.

`ulong` **addllx**`(int x, ulong y)` adds `overflow` to the sum of the ulongs `x` and `y`, returns the lower `BIL` bits and puts the carry bit into `overflow`.

`ulong` **subll**`(ulong x, ulong y)` subtracts the ulongs `x` and `y`, returns the lower `BIL` bits and put the carry (borrow) bit into `overflow`.

`ulong` **subllx**`(ulong x, ulong y)` subtracts `overflow` from the difference of the ulongs `x` and `y`, returns the lower `BIL` bits and puts the carry (borrow) bit into `overflow`.

`ulong` **shiftl**`(ulong x, ulong y)` shifts the ulong `x` left by `y` bits, returns the lower `BIL` bits and stores the high-order `BIL` bits into `hiremainder`. We must have $1 \leq$ `y` $\leq$ `BIL`. In particular, `y` must be non-zero; the caller is responsible for testing this.

`ulong` **shiftlr**`(ulong x, ulong y)` shifts the ulong `x << BIL` right by `y` bits, returns the higher `BIL` bits and stores the low-order `BIL` bits into `hiremainder`. We must have $1 \leq$ `y` $\leq$ `BIL`. In particular, `y` must be non-zero.

`int` **bfffo**`(ulong x)` returns the number of leading zero bits in the ulong `x` (i.e. the number of bit positions by which it would have to be shifted left until its leftmost bit first becomes equal to 1, which can be between 0 and `BIL` -1 for nonzero `x`). When `x` is 0, `BIL` is returned.

`ulong` **mulll**`(ulong x, ulong y)` multiplies the ulong `x` by the ulong `y`, returns the lower `BIL` bits and stores the high-order `BIL` bits into `hiremainder`.

`ulong` **addmul**`(ulong x, ulong y)` adds `hiremainder` to the product of the ulongs `x` and `y`, returns the lower `BIL` bits and stores the high-order `BIL` bits into `hiremainder`.

`ulong` **divll**`(ulong x, ulong y)` returns the Euclidean quotient of (`hiremainder << BIL`) + `x` and the ulong divisor `y` and stores the remainder into `hiremainder`. An error occurs if the quotient cannot be represented by a ulong, i.e. if `hiremainder` $\geq$ `y` initially.

The following routines are not part of the level 0 kernel per se, but implement modular operations on words in terms of the above. They are written so that no overflow may occur. Let $m \geq 1$ be the modulus; all operands representing classes modulo m are assumed to belong to $[0, m-1[$ (the result may be wrong for a number of reasons otherwise: it may not be reduced, overflow can occur, etc.).

`ulong` **adduumod**`(ulong x, ulong y, ulong m)` returns the smallest positive representative of $x + y$ modulo m.

`ulong` **subuumod**`(ulong x, ulong y, ulong m)` returns the smallest positive representative of $x - y$ modulo m.

`ulong` **muluumod**`(ulong x, ulong y, ulong m)` returns the smallest positive representative of xy modulo m.

`ulong` **invumod**`(ulong x, ulong m)` returns the smallest positive representative of x^{-1} modulo m. If x is not invertible mod m, return 0.

`long` **invsmod**`(long x, long m)` returns the smallest positive representative of x^{-1} modulo m. If x is not invertible mod m, return 0. In this routine no specific assumptions are made about the size or sign of x (m is still assumed to be positive). Consequently, it is a little slower than `invumod`.

`ulong` **divuumod**`(ulong x, ulong y, ulong m)` returns the smallest positive representative of xy^{-1} modulo m. Ix y is not invertible mod m, return 0.

`ulong` **powuumod**`(ulong x, ulong n, ulong m)` returns the smallest positive representative of x^n modulo m.

`ulong` **powusmod**`(ulong x, long n, ulong m)` as `powuumod`, but n is allowed to be negative, in which case it is assumed that x is invertible modulo m (otherwise, 0 is returned).

5.2 Level 1 kernel (operations on longs, integers and reals).

In this section as elsewhere, `long` denotes a `BIL`-bit signed C-integer, "integer" denotes a PARI multiprecise integer (type `t_INT`), "real" denotes a PARI multiprecise real (type `t_REAL`). Refer to Chapters 1–2 and 4 for general background.

Note: Many functions consist of an elementary operation, immediately followed by an assignment statement. All such functions are obtained using macros (see the file `paricom.h`), hence you can easily extend the list. Below, they will be introduced like in the following example:

`GEN` **gadd**[**z**]`(GEN x, GEN y[, GEN z])` followed by the explicit description of the function

```
GEN gadd(GEN x, GEN y)
```

which creates its result on the stack, returning a `GEN` pointer to it, and the parts in brackets indicate that there exists also a function

```
void gaddz(GEN x, GEN y, GEN z)
```

which assigns its result to the pre-existing object `z`, leaving the stack unchanged.

5.2.1 Basic unit and subunit handling functions

`long` **typ**`(GEN x)` returns the type number of `x`. (The header files included through `pari.h` will give you access to the symbolic constants `t_INT` etc., so you should never need to know the actual numerical values.)

`long` **lg**`(GEN x)` returns the length of `x` in `BIL`-bit words.

`long` **lgef**`(GEN x)` returns the effective length of the polynomial `x` in `BIL`-bit words.

`long` **lgefint**`(GEN x)` returns the effective length of the integer `x` in `BIL`-bit words.

`long` **signe**`(GEN x)` returns the sign (-1, 0 or 1) of `x`. Can be used for integers, reals, polynomials and power series (for the last two types, only 0 or 1 are possible).

`long` **gsigne**`(GEN x)` same as `signe`, but also valid for rational numbers (and marginally less efficient for the other types).

`long` **expo**`(GEN x)` returns the unbiased binary exponent of the real number `x`.

`long` **gexpo**`(GEN x)` same as `expo`, but also valid when `x` is not a real number. When `x` is an exact 0, this returns `-HIGHEXPOBIT`.

`long` **expi**`(GEN x)` returns the binary exponent of the real number equal to the integer `x`. This is a special case of `gexpo` above, covering the case where `x` is of type `t_INT`.

`long` **valp**`(GEN x)` returns the unbiased 16-bit p-adic valuation (for a p-adic) or X-adic valuation (for a power series, taken with respect to the main variable) of `x`.

`long` **precp**`(GEN x)` returns the precision of the p-adic `x`.

`long` **varn**`(GEN x)` returns the variable number of `x` (between 0 and `MAXVARN`). Should be used only for polynomials and power series.

`long` **gvar**`((GEN x))` returns the main variable number when any variable at all occurs in the composite object `x` (the smallest variable number which occurs), and `BIGINT` otherwise.

`void` **settyp**`(GEN x, long s)` sets the type number of `x` to `s`. This should be used with extreme care since usually the type is set otherwise, and the components and further codeword fields (which are left unchanged) may not match the PARI conventions for the new type.

`void` **setlg**`(GEN x, long s)` sets the length of `x` to `s`. Again this should be used with extreme care since usually the length is set otherwise, and increasing the length joins previously unrelated

memory words to the root node of `x`. This is, however, an extremely efficient way of truncating vectors or polynomials.

`void` **setlgef**`(GEN x, long s)` sets the effective length of `x` to `s`, where `x` is a polynomial. The number `s` must be less than or equal to the length of `x`.

`void` **setlgefint**`(GEN x, long s)` sets the effective length of the integer `x` to `s`. The number `s` must be less than or equal to the length of `x`.

`void` **setsigne**`(GEN x, long s)` sets the sign of `x` to `s`. If `x` is an integer or real, `s` must be equal to -1, 0 or 1, and if `x` is a polynomial or a power series, `s` must be equal to 0 or 1.

`void` **setexpo**`(GEN x, long s)` sets the binary exponent of the real number `x` to `s`, after adding the appropriate bias. The unbiased value `s` must be a 24-bit signed number.

`void` **setvalp**`(GEN x, long s)` sets the p-adic or X-adic valuation of `x` to `s`, if `x` is a p-adic or a power series, respectively.

`void` **setprecp**`(GEN x, long s)` sets the p-adic precision of the p-adic number `x` to `s`.

`void` **setvarn**`(GEN x, long s)` sets the variable number of the polynomial or power series `x` to `s` (where $0 \leq$ `s` $\leq$ `MAXVARN`).

5.2.2 Memory allocation on the PARI stack

`GEN` **cgetg**`(long n, long t)` allocates memory on the PARI stack for an object of length `n` and type `t`, and initializes its first codeword.

`GEN` **cgeti**`(long n)` allocates memory on the PARI stack for an integer of length `n`, and initializes its first codeword. Identical to `cgetg(n,t_INT)`.

`GEN` **cgetr**`(long n)` allocates memory on the PARI stack for a real of length `n`, and initializes its first codeword. Identical to `cgetg(n,t_REAL)`.

`void` **cgiv**`(GEN x)` frees object `x` if it is the last created on the PARI stack (otherwise nothing happens).

`GEN` **gerepile**`(pari_sp p, pari_sp q, GEN x)` general garbage collector for the PARI stack. See Section 4.4 for a detailed explanation and many examples.

5.2.3 Assignments, conversions and integer parts

`void` **mpaff**`(GEN x, GEN z)` assigns `x` into `z` (where `x` and `z` are integers or reals).

`void` **affsz**`(long s, GEN z)` assigns the long `s` into the integer or real `z`.

`void` **affsi**`(long s, GEN z)` assigns the long `s` into the integer `z`.

`void` **affsr**`(long s, GEN z)` assigns the long `s` into the real `z`.

`void` **affii**`(GEN x, GEN z)` assigns the integer `x` into the integer `z`.

`void` **affir**`(GEN x, GEN z)` assigns the integer `x` into the real `z`.

`void` **affrs**`(GEN x, long s)` assigns the real `x` into the long `s`... not. This is a forbidden assignment in PARI, so an error message is issued.

`void` **affri**`(GEN x, GEN z)` assigns the real `x` into the integer `z`... no it doesn't. This is a forbidden assignment in PARI, so an error message is issued.

`void` **affrr**`(GEN x, GEN z)` assigns the real `x` into the real `z`.

`GEN` **itor**`(GEN x, long prec)` assigns the `t_INT` `x` into a `t_REAL` of length `prec` and return the latter.

`long` **itos**`(GEN x)` converts the PARI integer `x` to a C long (if possible, otherwise an error message is issued).

`GEN` **stoi**`(long s)` creates the PARI integer corresponding to the long `s`.

`GEN` **stor**`(long s, long prec)` assigns the long `s` into a `t_REAL` of length `prec` and return the latter.

`GEN` **mptrunc**[**z**]`(GEN x[, GEN z])` truncates the integer or real `x` (not the same as the integer part if `x` is non-integer and negative).

`GEN` **mpent**[**z**]`(GEN x[, GEN z])` true integer part of the integer or real `x` (i.e. the `floor` function).

5.2.4 Valuation and shift

`long` **vals**`(long s)` 2-adic valuation of the long `s`. Returns -1 if `s` is equal to 0, with no error.

`long` **vali**`(GEN x)` 2-adic valuation of the integer `x`. Returns -1 if `s` is equal to 0, with no error.

`GEN` **mpshift**[**z**]`(GEN x, long n[, GEN z])` shifts the real or integer `x` by `n`. If `n` is positive, this is a left shift, i.e. multiplication by $2^{\mathtt{n}}$. If `n` is negative, it is a right shift by $-$`n`, which amounts to the truncation of the quotient of `x` by $2^{-\mathtt{n}}$.

`GEN` **shifts**`(long s, long n)` converts the long `s` into a PARI integer and shifts the value by `n`.

`GEN` **shifti**`(GEN x, long n)` shifts the integer `x` by `n`.

`GEN` **shiftr**`(GEN x, long n)` shifts the real `x` by `n`.

5.2.5 Unary operations

Let "*op*" be some unary operation of type `GEN (*)(GEN)`. The names and prototypes of the low-level functions corresponding to *op* will be as follows.

`GEN` **mp***op*`(GEN x)` creates the result of *op* applied to the integer or real `x`.

`GEN` *op***s**`(long s)` creates the result of *op* applied to the long `s`.

`GEN` *op***i**`(GEN x)` creates the result of *op* applied to the integer `x`.

`GEN` *op***r**`(GEN x)` creates the result of *op* applied to the real `x`.

`GEN` **mp***op***z**`(GEN x, GEN z)` assigns the result of applying *op* to the integer or real `x` into the integer or real `z`.

Remark: it has not been considered useful to include the functions `void` *op*`sz(long,GEN)`, `void` *op*`iz(GEN,GEN)` and `void` *op*`rz(GEN, GEN)`.

The above prototype schemes apply to the following operators:

op=**neg**: negation (−`x`). The result is of the same type as `x`.

op=**abs**: absolute value (|`x`|). The result is of the same type as `x`.

In addition, there exist the following special unary functions with assignment:

`void` **mpinvz**`(GEN x, GEN z)` assigns the inverse of the integer or real `x` into the real `z`. The inverse is computed as a quotient of real numbers, not as a Euclidean division.

`void` **mpinvsr**`(long s, GEN z)` assigns the inverse of the long `s` into the real `z`.

`void` **mpinvir**`(GEN x, GEN z)` assigns the inverse of the integer `x` into the real `z`.

`void` **mpinvrr**`(GEN x, GEN z)` assigns the inverse of the real `x` into the real `z`.

5.2.6 Comparison operators

`int` **mpcmp**`(GEN x, GEN y)` compares the integer or real `x` to the integer or real `y`. The result is the sign of `x` − `y`.

`int` **cmpsi**`(long s, GEN x)` compares the long `s` to the integer `x`.

`int` **cmpsr**`(long s, GEN x)` compares the long `s` to the real `x`.

`int` **cmpis**`(GEN x, long s)` compares the integer `x` to the long `s`.

`int` **cmpii**`(GEN x, GEN y)` compares the integer `x` to the integer `y`.

`int` **cmpir**`(GEN x, GEN y)` compares the integer `x` to the real `y`.

`int` **cmprs**`(GEN x, long s)` compares the real `x` to the long `s`.

`int` **cmpri**`(GEN x, GEN y)` compares the real `x` to the integer `y`.

`int` **cmprr**`(GEN x, GEN y)` compares the real `x` to the real `y`.

`int` **egalii**`(GEN x, GEN y)` compares the integers `x` and `y`. The result is 1 if `x` = `y`, 0 otherwise.

`int` **absi_cmp**`(GEN x, GEN y)` compares the integers `x` and `y`. The result is the sign of |`x`| − |`y`|.

`int` **absi_equal**`(GEN x, GEN y)` compares the integers `x` and `y`. The result is 1 if |`x`| = |`y`|, 0 otherwise.

`int` **absr_cmp**`(GEN x, GEN y)` compares the reals `x` and `y`. The result is the sign of |`x`| − |`y`|.

5.2.7 Binary operations

Let "*op*" be some operation of type `GEN (*)(GEN,GEN)`. The names and prototypes of the low-level functions corresponding to *op* will be as follows. In this section, the `z` argument in the `z`-functions must be of type `t_INT` or `t_REAL`.

`GEN` **mp***op*[**z**]`(GEN x, GEN y[, GEN z])` applies *op* to the integer-or-reals `x` and `y`.

`GEN` *op***ss**[**z**]`(long s, long t[, GEN z])` applies *op* to the longs `s` and `t`.

`GEN` *op***si**[**z**]`(long s, GEN x[, GEN z])` applies *op* to the long `s` and the integer `x`.

`GEN` *op***sr**[**z**]`(long s, GEN x[, GEN z])` applies *op* to the long `s` and the real `x`.

`GEN` *op***is**[**z**]`(GEN x, long s[, GEN z])` applies *op* to the integer `x` and the long `s`.

`GEN` *op***ii**[**z**]`(GEN x, GEN y[, GEN z])` applies *op* to the integers `x` and `y`.

`GEN` *op***ir**[**z**]`(GEN x, GEN y[, GEN z])` applies *op* to the integer `x` and the real `y`.

`GEN` *op***rs**[**z**]`(GEN x, long s[, GEN z])` applies *op* to the real `x` and the long `s`.

`GEN` *op***ri**[**z**]`(GEN x, GEN y[, GEN z])` applies *op* to the real `x` and the integer `y`.

`GEN` *op***rr**[**z**]`(GEN x, GEN y[, GEN z])` applies *op* to the reals `x` and `y`.

Each of the above can be used with the following operators.

op=**add**: addition (`x + y`). The result is real unless both `x` and `y` are integers (or longs).

op=**sub**: subtraction (`x - y`). The result is real unless both `x` and `y` are integers (or longs).

op=**mul**: multiplication (`x * y`). The result is real unless both `x` and `y` are integers (or longs), OR if `x` or `y` is the integer or long zero.

op=**div**: division (`x / y`). In the case where `x` and `y` are both integers or longs, the result is the Euclidean quotient, where the remainder has the same sign as the dividend `x`. If one of `x` or `y` is real, the result is real unless `x` is the integer or long zero. A division-by-zero error occurs if `y` is equal to zero.

op=**res**: remainder ("`x % y`"). This operation is defined only when `x` and `y` are longs or integers. The result is the Euclidean remainder corresponding to `div`, i.e. its sign is that of the dividend `x`. The result is always an integer.

op=**mod**: remainder (`x % y`). This operation is defined only when `x` and `y` are longs or integers. The result is the true Euclidean remainder, i.e. non-negative and less than the absolute value of `y`.

5.2.8 Division with remainder: the following functions return two objects, unless specifically asked for only one of them — a quotient and a remainder. The remainder will be created on the stack, and a `GEN` pointer to this object will be returned through the variable whose address is passed as the `r` argument.

`GEN` **dvmdss**`(long s, long t, GEN *r)` creates the Euclidean quotient and remainder of the longs `s` and `t`. If `r` is not `NULL` or `ONLY_REM`, this puts the remainder into `*r`, and returns the quotient. If `r` is equal to `NULL`, only the quotient is returned. If `r` is equal to `ONLY_REM`, the remainder is returned instead of the quotient. In the generic case, the remainder is created after the quotient and can be disposed of individually with a `cgiv(r)`. The remainder is always of the sign of the dividend `s`.

`GEN` **dvmdsi**`(long s, GEN x, GEN *r)` creates the Euclidean quotient and remainder of the long `s` by the integer `x`. Obeys the same conventions with respect to `r`.

`GEN` **dvmdis**`(GEN x, long s, GEN *r)` create the Euclidean quotient and remainder of the integer x by the long s.

`GEN` **dvmdii**`(GEN x, GEN y, GEN *r)` returns the Euclidean quotient of the integer `x` by the integer `y` and puts the remainder into `*r`. If `r` is equal to `NULL`, the remainder is not created, and if `r` is equal to `ONLY_REM`, only the remainder is created and returned. In the generic case, the remainder is created after the quotient and can be disposed of individually with a `cgiv(r)`. The remainder is always of the sign of the dividend `x`.

`GEN` **truedvmdii**`(GEN x, GEN y, GEN *r)`, as `dvmdii` but with a non-negative remainder.

`void` **mpdvmdz**`(GEN x, GEN y, GEN z, GEN *r)` assigns the Euclidean quotient of the integers `x` and `y` into the integer or real `z`, putting the remainder into `*r` (unless `r` is equal to `NULL` or `ONLY_REM` as above).

`void` **dvmdssz**`(long s, long t, GEN z, GEN *r)` assigns the Euclidean quotient of the longs `s` and `t` into the integer or real `z`, putting the remainder into `*r` (unless `r` is equal to `NULL` or `ONLY_REM` as above).

`void` **dvmdsiz**`(long s, GEN x, GEN z, GEN *r)` assigns the Euclidean quotient of the long `s` and the integer `x` into the integer or real `z`, putting the remainder into `*r` (unless `r` is equal to `NULL` or `ONLY_REM` as above).

`void` **dvmdisz**`(GEN x, long s, GEN z, GEN *r)` assigns the Euclidean quotient of the integer `x` and the long `s` into the integer or real `z`, putting the remainder into `*r` (unless `r` is equal to `NULL` or `ONLY_REM` as above).

`void` **dvmdiiz**`(GEN x, GEN y, GEN z, GEN *r)` assigns the Euclidean quotient of the integers `x` and `y` into the integer or real `z`, putting the address of the remainder into `*r` (unless `r` is equal to `NULL` or `ONLY_REM` as above).

`void` **diviiexact**`(GEN x, GEN y)` returns the Euclidean quotient `x/y`, assuming `y` divides `x`. Uses Jebelean algorithm (Jebelean-Krandick bidirectional exact division is not implemented).

`void` **diviuexact**`(GEN x, ulong y)` returns the Euclidean quotient `x/y`, assuming `y` divides `x` and `y` is *odd*.

5.2.9 Miscellaneous functions

`void` **addsii**`(long s, GEN x, GEN z)` assigns the sum of the long `s` and the integer `x` into the integer `z` (essentially identical to `addsiz` except that `z` is specifically an integer).

`long` **divise**`(GEN x, GEN y)` if the integer `y` divides the integer `x`, returns 1 (true), otherwise returns 0 (false).

`long` **divisii**`(GEN x, long s, GEN z)` assigns the Euclidean quotient of the integer `x` and the long `s` into the integer `z`, and returns the remainder as a long.

`long` **mpdivis**`(GEN x, GEN y, GEN z)` if the integer `y` divides the integer `x`, assigns the quotient to the integer `z` and returns 1 (true), otherwise returns 0 (false).

`void` **mulsii**`(long s, GEN x, GEN z)` assigns the product of the long `s` and the integer `x` into the integer `z` (essentially identical to `mulsiz` except that `z` is specifically an integer).

`void` **addumului**`(ulong a, ulong b, GEN x)` return $a + b|X|$.

5.3 Level 2 kernel (operations on general PARI objects).

The functions available to handle subunits are the following.

`GEN` **compo**`(GEN x, long n)` creates a copy of the `n`-th true component (i.e. not counting the codewords) of the object `x`.

`GEN` **truecoeff**`(GEN x, long n)` creates a copy of the coefficient of degree `n` of `x` if `x` is a scalar, polynomial or power series, and otherwise of the `n`-th component of `x`.

The remaining two are macros, NOT functions (see Section 4.2.1 for a detailed explanation):

`long` **coeff**`(GEN x, long i, long j)` applied to a matrix `x` (type `t_MAT`), this gives the address of the coefficient at row `i` and column `j` of `x`.

`long` **mael$n$**`(GEN x, long` a_1`, ..., long` a_n`)` stands for `x[`a_1`][`a_2`]...[`a_n`]`, where $2 \leq n \leq 5$, with all the necessary typecasts.

5.3.1 Copying and conversion

`GEN` **cgetp**`(GEN x)` creates space sufficient to hold the p-adic `x`, and sets the prime p and the p-adic precision to those of `x`, but does not copy (the p-adic unit or zero representative and the modulus of) `x`.

`GEN` **gcopy**`(GEN x)` creates a new copy of the object `x` on the PARI stack. For permanent subobjects, only the pointer is copied.

`GEN` **forcecopy**`(GEN x)` same as **copy** except that even permanent subobjects are copied onto the stack.

`long` **taille**`(GEN x)` returns the total number of `BIL`-bit words occupied by the tree representing `x`.

`GEN` **gclone**`(GEN x)` creates a new permanent copy of the object `x` on the heap.

`GEN` **greffe**`(GEN x, long l, int use_stack)` applied to a polynomial `x` (type `t_POL`), creates a power series (type `t_SER`) of length `l` starting with `x`, but without actually copying the coefficients, just the pointers. If `use_stack` is zero, this is created through malloc, and must be freed after use. Intended for internal use only.

`double` **rtodbl**`(GEN x)` applied to a real `x` (type `t_REAL`), converts `x` into a C double if possible.

`GEN` **dbltor**`(double x)` converts the C double `x` into a PARI real.

`double` **gtodouble**`(GEN x)` if `x` is a real number (but not necessarily of type `t_REAL`), converts `x` into a C double if possible.

`long` **gtolong**`(GEN x)` if `x` is an integer (not a C long, but not necessarily of type `t_INT`), converts `x` into a C long if possible.

`GEN` **gtopoly**`(GEN x, long v)` converts or truncates the object `x` into a polynomial with main variable number `v`. A common application would be the conversion of coefficient vectors.

`GEN` **gtopolyrev**`(GEN x, long v)` converts or truncates the object `x` into a polynomial with main variable number `v`, but vectors are converted in reverse order.

`GEN` **gtoser**`(GEN x, long v)` converts the object `x` into a power series with main variable number `v`.

`GEN` **gtovec**`(GEN x)` converts the object `x` into a (row) vector.

`GEN` **co8**`(GEN x, long l)` applied to a quadratic number `x` (type `t_QUAD`), converts `x` into a real or complex number depending on the sign of the discriminant of `x`, to precision `l` `BIL`-bit words.

`GEN` **gcvtop**`(GEN x, GEN p, long l)` converts `x` into a `p`-adic number of precision `l`.

`GEN` **gmodulcp**`(GEN x, GEN y)` creates the object **Mod**`(x,y)` on the PARI stack, where `x` and `y` are either both integers, and the result is an integermod (type `t_INTMOD`), or `x` is a scalar or a polynomial and `y` a polynomial, and the result is a polymod (type `t_POLMOD`).

`GEN` **gmodulgs**`(GEN x, long y)` same as **gmodulcp** except `y` is a `long`.

`GEN` **gmodulss**`(long x, long y)` same as **gmodulcp** except both `x` and `y` are `long`s.

`GEN` **gmodulo**`(GEN x, GEN y)` same as **gmodulcp** except that the modulus `y` is copied onto the heap and not onto the PARI stack.

`long` **gexpo**`(GEN x)` returns the binary exponent of `x` or the maximal binary exponent of the coefficients of `x`. Returns `-HIGHEXPOBIT` if `x` has no components or is an exact zero.

`long` **gsigne**`(GEN x)` returns the sign of `x` (-1, 0 or 1) when `x` is an integer, real or (irreducible or reducible) fraction. Raises an error for all other types.

`long` **gvar**`(GEN x)` returns the main variable of `x`. If no component of `x` is a polynomial or power series, this returns `BIGINT`.

`int` **precision**`(GEN x)` If `x` is of type `t_REAL`, returns the precision of `x` (the length of `x` in `BIL`-bit words if `x` is not zero, and a reasonable quantity obtained from the exponent of `x` if `x` is numerically equal to zero). If `x` is of type `t_COMPLEX`, returns the minimum of the precisions of the real and imaginary part. Otherwise, returns 0 (which stands in fact for infinite precision).

`long` **sizedigit**`(GEN x)` returns 0 if `x` is exactly 0. Otherwise, returns **gexpo**`(x)` multiplied by $\log_{10}(2)$. This gives a crude estimate for the maximal number of decimal digits of the components of `x`.

5.3.2 Comparison operators and valuations

`int` **gcmp0**`(GEN x)` returns 1 (true) if `x` is equal to 0, 0 (false) otherwise.

`int` **isexactzero**`(GEN x)` returns 1 (true) if `x` is exactly equal to 0, 0 (false) otherwise. Note that many PARI functions will return a pointer to **gzero** when they are aware that the result they return is an exact zero, so it is almost always faster to test for pointer equality first, and call **isexactzero** (or **gcmp0**) only when the first test fails.

`int` **gcmp1**`(GEN x)` returns 1 (true) if `x` is equal to 1, 0 (false) otherwise.

`int` **gcmp_1**`(GEN x)` returns 1 (true) if `x` is equal to -1, 0 (false) otherwise.

`long` **gcmp**`(GEN x, GEN y)` comparison of `x` with `y` (returns the sign of $x - y$).

`long` **gcmpsg**`(long s, GEN x)` comparison of the long `s` with `x`.

`long` **gcmpgs**`(GEN x, long s)` comparison of `x` with the long `s`.

`long` **lexcmp**`(GEN x, GEN y)` comparison of `x` with `y` for the lexicographic ordering.

`long` **gegal**`(GEN x, GEN y)` returns 1 (true) if `x` is equal to `y`, 0 otherwise.

`long` **gegalsg**`(long s, GEN x)` returns 1 (true) if the long `s` is equal to `x`, 0 otherwise.

`long` **gegalgs**`(GEN x, long s)` returns 1 (true) if `x` is equal to the long `s`, 0 otherwise.

`long` **iscomplex**`(GEN x)` returns 1 (true) if `x` is a complex number (of component types embeddable into the reals) but is not itself real, 0 if `x` is a real (not necessarily of type `t_REAL`), or raises an error if `x` is not embeddable into the complex numbers.

`long` **ismonome**`(GEN x)` returns 1 (true) if `x` is a non-zero monomial in its main variable, 0 otherwise.

`long` **ggval**`(GEN x, GEN p)` returns the greatest exponent e such that $\texttt{p}^e$ divides `x`, when this makes sense.

`long` **gval**`(GEN x, long v)` returns the highest power of the variable number `v` dividing the polynomial `x`.

`int` **pvaluation**`(GEN x, GEN p, GEN *r)` applied to non-zero integers `x` and `p`, returns the highest exponent e such that $\texttt{p}^e$ divides `x`, creates the quotient $\texttt{x}/\texttt{p}^e$ and returns its address in `*r`. In particular, if `p` is a prime, this returns the valuation at `p` of `x`, and `*r` will obtain the prime-to-`p` part of `x`.

5.3.3 Assignment statements

`void` **gaffsg**`(long s, GEN x)` assigns the long `s` into the object `x`.

`void` **gaffect**`(GEN x, GEN y)` assigns the object `x` into the object `y`.

5.3.4 Unary operators

`GEN` **gneg**[**z**]`(GEN x[, GEN z])` yields $-$`x`.

`GEN` **gabs**[**z**]`(GEN x[, GEN z])` yields |`x`|.

`GEN` **gsqr**`(GEN x)` creates the square of `x`.

`GEN` **ginv**`(GEN x)` creates the inverse of `x`.

`GEN` **gfloor**`(GEN x)` creates the floor of `x`, i.e. the (true) integral part.

`GEN` **gfrac**`(GEN x)` creates the fractional part of `x`, i.e. `x` minus the floor of `x`.

`GEN` **gceil**`(GEN x)` creates the ceiling of `x`.

`GEN` **ground**`(GEN x)` rounds the components of `x` to the nearest integers. Exact half-integers are rounded towards $+\infty$.

`GEN` **grndtoi**`(GEN x, long *e)` same as **round**, but in addition puts minus the number of significant binary bits left after rounding into `*e`. If `*e` is positive, all significant bits have been lost. This kind of situation raises an error message in **ground** but not in **grndtoi**.

`GEN` **gtrunc**`(GEN x)` truncates `x`. This is the (false) integer part if `x` is an integer (i.e. the unique integer closest to `x` among those between 0 and `x`). If `x` is a series, it will be truncated to a polynomial; if `x` is a rational function, this takes the polynomial part.

`GEN` **gcvtoi**`(GEN x, long *e)` same as **grndtoi** except that rounding is replaced by truncation.

`GEN` **gred**[**z**]`(GEN x[, GEN z])` reduces `x` to lowest terms if `x` is a fraction or rational function (types `t_FRAC`, `t_FRACN`, `t_RFRAC` and `t_RFRACN`), otherwise creates a copy of `x`.

`GEN` **content**`(GEN x)` creates the GCD of all the components of `x`.

`GEN` **normalize**`(GEN x)` applied to an unnormalized power series `x` (i.e. type `t_SER` with all coefficients correctly set except that `x[2]` might be zero), normalizes `x` correctly in place. Returns `x`. For internal use.

`GEN` **normalizepol**`(GEN x)` applied to an unnormalized polynomial `x` (i.e. type `t_POL` with all coefficients correctly set except that `x[2]` might be zero), normalizes `x` correctly in place and returns `x`. For internal use.

5.3.5 Binary operators

`GEN` **gmax**[**z**]`(GEN x, GEN y[, GEN z])` yields the maximum of the objects `x` and `y` if they can be compared.

`GEN` **gmaxsg**[**z**]`(long s, GEN x[, GEN z])` yields the maximum of the long `s` and the object `x`.

`GEN` **gmaxgs**[**z**]`(GEN x, long s[, GEN z])` yields the maximum of the object `x` and the long `s`.

`GEN` **gmin**[**z**]`(GEN x, GEN y[, GEN z])` yields the minimum of the objects `x` and `y` if they can be compared.

`GEN` **gminsg**[**z**]`(long s, GEN x[, GEN z])` yields the minimum of the long `s` and the object `x`.

`GEN` **gmings**[**z**]`(GEN x, long s[, GEN z])` yields the minimum of the object `x` and the long `s`.

`GEN` **gadd**[**z**]`(GEN x, GEN y[, GEN z])` yields the sum of the objects `x` and `y`.

`GEN` **gaddsg**[**z**]`(long s, GEN x[, GEN z])` yields the sum of the long `s` and the object `x`.

GEN **gaddgs**[**z**](GEN x, long s[, GEN z]) yields the sum of the object x and the long s.

GEN **gsub**[**z**](GEN x, GEN y[, GEN z]) yields the difference of the objects x and y.

GEN **gsubgs**[**z**](GEN x, long s[, GEN z]) yields the difference of the object x and the long s.

GEN **gsubsg**[**z**](long s, GEN x[, GEN z]) yields the difference of the long s and the object x.

GEN **gmul**[**z**](GEN x, GEN y[, GEN z]) yields the product of the objects x and y.

GEN **gmulsg**[**z**](long s, GEN x[, GEN z]) yields the product of the long s with the object x.

GEN **gmulgs**[**z**](GEN x, long s[, GEN z]) yields the product of the object x with the long s.

GEN **gshift**[**z**](GEN x, long n[, GEN z]) yields the result of shifting (the components of) x left by n (if n is non-negative) or right by $-$n (if n is negative). Applies only to integers, reals and vectors/matrices of such. For other types, it is simply multiplication by 2^n.

GEN **gmul2n**[**z**](GEN x, long n[, GEN z]) yields the product of x and 2^n. This is different from gshift when n is negative and x is of type t_INT: **gshift** truncates, while **gmul2n** creates a fraction if necessary.

GEN **gdiv**[**z**](GEN x, GEN y[, GEN z]) yields the quotient of the objects x and y.

GEN **gdivgs**[**z**](GEN x, long s[, GEN z]) yields the quotient of the object x and the long s.

GEN **gdivsg**[**z**](long s, GEN x[, GEN z]) yields the quotient of the long s and the object x.

GEN **gdivent**[**z**](GEN x, GEN y[, GEN z]) yields the true Euclidean quotient of x and the integer or polynomial y.

GEN **gdiventsg**[**z**](long s, GEN x[, GEN z]) yields the true Euclidean quotient of the long s by the integer x.

GEN **gdiventgs**[**z**](GEN x, long s[, GEN z]) yields the true Euclidean quotient of the integer x by the long s.

GEN **gdiventres**(GEN x, GEN y) creates a 2-component vertical vector whose components are the true Euclidean quotient and remainder of x and y.

GEN **gdivmod**(GEN x, GEN y, GEN *r) If r is not equal to NULL or ONLY_REM, creates the (false) Euclidean quotient of x and y, and puts (the address of) the remainder into *r. If r is equal to NULL, do not create the remainder, and if r is equal to ONLY_REM, create and output only the remainder. The remainder is created after the quotient and can be disposed of individually with a cgiv(r).

GEN **poldivres**(GEN x, GEN y, GEN *r) same as **gdivmod** but specifically for polynomials x and y.

GEN **gdeuc**(GEN x, GEN y) creates the Euclidean quotient of the polynomials x and y.

GEN **gdivround**(GEN x, GEN y) if x and y are integers, returns the quotient x/y of x and y, rounded to the nearest integer. If x/y falls exactly halfway between two consecutive integers, then it is rounded towards $+\infty$ (as for **round**). If x and y are not both integers, the result is the same as that of **gdivent**.

GEN **gmod**[**z**](GEN x, GEN y[, GEN z]) yields the true remainder of x modulo the integer or polynomial y.

`GEN` **gmodsg**[**z**]`(long s, GEN x[, GEN z])` yields the true remainder of the long `s` modulo the integer `x`.

`GEN` **gmodgs**[**z**]`(GEN x, long s[, GEN z])` yields the true remainder of the integer `x` modulo the long `s`.

`GEN` **gres**`(GEN x, GEN y)` creates the Euclidean remainder of the polynomial `x` divided by the polynomial `y`.

`GEN` **ginvmod**`(GEN x, GEN y)` creates the inverse of `x` modulo `y` when it exists. `y` must be of type `t_INT` (in which case `x` is of type `t_INT`) or `t_POL` (in which case `x` is either a scalar type or of type `t_POL`).

`GEN` **gpow**`(GEN x, GEN y, long l)` creates $\mathtt{x}^{\mathtt{y}}$. The precision `l` is taken into account only if `y` is not an integer and `x` is an exact object. If `y` is an integer, binary powering is done. Otherwise, the result is $\exp(\mathtt{y} * \log(\mathtt{x}))$ computed to precision `l`.

`GEN` **ggcd**`(GEN x, GEN y)` creates the GCD of `x` and `y`.

`GEN` **glcm**`(GEN x, GEN y)` creates the LCM of `x` and `y`.

`GEN` **subres**`(GEN x, GEN y)` creates the resultant of the polynomials `x` and `y` computed using the subresultant algorithm.

`GEN` **gpowgs**`(GEN x, long n)` creates $\mathtt{x}^{\mathtt{n}}$ using binary powering.

`GEN` **gsubst**`(GEN x, long v, GEN y)` substitutes the object `y` into `x` for the variable number `v`.

`int` **gdivise**`(GEN x, GEN y)` returns 1 (true) if `y` divides `x`, 0 otherwise.

`GEN` **gbezout**`(GEN x,GEN y, GEN *u,GEN *v)` creates the GCD of `x` and `y`, and puts (the addresses of) objects u and v such that $u\mathtt{x} + v\mathtt{y} = \gcd(\mathtt{x}, \mathtt{y})$ into `*u` and `*v`.

Appendix A:
Installation Guide for the UNIX Versions

1. Required tools.

We assume that you have either an `ANSI C` or a `C++` compiler available. If your machine does not have one, we strongly suggest that you obtain the `gcc/g++` compiler from the Free Software Foundation or by anonymous `ftp`. As for all GNU software mentioned afterwards, you can find the most convenient site to fetch `gcc` at the address

`http://www.gnu.ai.mit.edu/order/ftp.html`

You can certainly compile PARI with a different compiler, but the PARI kernel takes advantage of some optimizations provided by `gcc` if it is available. This results in at least 20% speedup on most architectures*.

1.1. Optional packages: The following programs and libraries are useful in conjunction with GP, but not mandatory. They're probably already installed somewhere on your system (with the possible exception of `readline`, which we think is really worth a try). In any case, get them before proceeding if you want the functionalities they provide. All of them are free (though you ought to make a small donation to the FSF if you use (and like) GNU wares).

• GNU `MP` library. This provides an alternative multiprecision kernel, which is faster than PARI's native one. To enable detection of GMP, use `Configure --with-gmp`. *This is an experimental feature, use at your own risk!*

• GNU `readline` library. This provides line editing under GP, an automatic context-dependent completion, and an editable history of commands. Note that it is incompatible with SUN commandtools (yet another reason to dump Suntools for X Windows). A recent readline (version number at least 2.2) is preferred, but older versions should be usable.

• GNU `gzip/gunzip/gzcat` package enables GP to read compressed data.

• GNU `emacs`. GP can be run in an Emacs buffer, with all the obvious advantages if you are familiar with this editor. Note that `readline` is still useful in this case since it provides a much better automatic completion than is provided by Emacs GP-mode.

• `perl` provides extended online help (full text from this manual) about functions and concepts, which can be used under GP or independently (`http://www.perl.com` will direct you to the nearest `CPAN` archive site).

• A colour-capable `xterm`, which enables GP to use different (user configurable) colours for its output. All `xterm` programs which come with current X11 distributions satisfy this requirement.

* One notable exception is the native AIX C compiler on IBM RS/6000 workstations, which generates fast code even without any special help from the PARI kernel sources.

2. Compiling the library and the GP calculator.

2.1. Basic configuration: First, have a look at the `MACHINES` file to see if anything funny applies to your architecture or operating system. Then, type

```
./Configure
```

in the toplevel directory. This attempts to configure GP/PARI without outside help. Note that if you want to install the end product in some nonstandard place, you can use the `--prefix` option, as in

```
./Configure --prefix=/an/exotic/directory
```

(the default prefix is `/usr/local`). For example, to build a package for a Linux distribution, you may want to use

```
./Configure --prefix=/usr
```

This phase extracts some files and creates a directory `O`*xxx* where the object files and executables will be built. The *xxx* part depends on your architecture and operating system, thus you can build GP for several different machines from the same source tree (the builds are completely independent, so can be done simultaneously).

Technical note: The precise default destinations are as follows: the `gp` binary, the scripts `gphelp` and `tex2mail` go to `$prefix/bin`. The pari library goes to `$prefix/lib` and include files to `$prefix/include/pari`. Other system-dependant data go to `$prefix/lib/pari`.

As for architecture independent files, they go to various subdirectories of `$share_prefix`, which defaults to `$prefix/share`, and can be specified via the `--share-prefix` argument. Man pages go into `$share_prefix/man`, Emacs files into `$share_prefix/emacs/site-lisp/pari`, and other system-independant data to various subdirectories of `$share_prefix/pari`: documentation, sample GP scripts and C code, extra packages like `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).

Technical note 2: `Configure` lets the following environment variable override the defaults if set:

`AS`: Assembler.

`CC`: C compiler.

`DLLD`: Dynamic library linker.

`LD`: Static linker.

The contents of the following variables are *appended* to the values computed by `Configure`:

`CFLAGS`: Flags for `CC`.

`LDFLAGS`: Flags for `LD`.

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`.

Technical note 3: `Configure` accepts many other flags besides the ones mentionned above. See `Configure --help` for a complete list. In particular, there are sets of flags related to GNU MP (`--with-gmp*`) and GNU readline library (`--with-readline*`). Note that autodetection of GMP is *disabled* by default.

Technical note 4: The multiprecision 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 ix86 ppc sparcv7 sparcv8_micro sparcv8_super`

L1: `auto`, `none` and `gmp`

`auto` means to use the auto-detected value. L0=`none` means to use the portable C kernel (no assembler), L1=`none` means to use the PARI L1 kernel.

• A fully qualified kernel name *fqkn* is of the form L_0-L_1.

• A *name* not containing a dash '-' is an alias. An alias stands for *name*-`none`, but `gmp` stand for `auto-gmp`.

• The default kernel is `auto-none`.

2.2. Troubleshooting and fine tuning: 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 probably report genuine problems. If anything should have been found and was not, consider that `Configure` failed and follow the instructions below. Look especially for the `readline` and `X11` libraries, and the `perl` and `gunzip` (or `zcat`) binaries.

In case the default `Configure` run fails miserably, try

`./Configure -a`

(interactive mode) and answer all the questions (there aren't that many). Of course, `Configure` will still provide defaults for each answer but if you accept them all, it will fail just the same, so be wary. In any case, we would appreciate a bug report including the complete output from `Configure` and the file `O`*xxx*`/pari.cfg` that was produced in the process.

2.3. 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 won't find them. Nonetheless, it first searches the distribution toplevel for a `readline` directory. Thus, if you just want to give `readline` a try, as you probably should, you can get the source and compile it there (you do not need to install it). You can also use this feature together with a symbolic link, named `readline`, in the PARI toplevel directory if you have compiled the readline library somewhere else, without installing it to one of its standard locations.

You can also invoke `Configure` with one of the following arguments:

`--with-readline[=`*prefix to* `lib/libreadline.`*xx and* `include/readline.h]`

`--with-readline-lib=`*path to* `libreadline.`*xx*

`--with-readline-include=`*path to* `readline.h`

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 will 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).

Technical note: Configure can build GP on different architectures simultaneously from the same toplevel sources. Instead of the `readline` link alluded above, you can create `readline-`*osname-arch*, using the same naming conventions as for the `O`*xxx* directory, e.g `readline-linux-i686`.

2.4. Debugging/profiling: If you also want to debug the PARI library,

```
Configure -g
```

will create a directory `O`*xxx*`.dbg` containing a special `Makefile` ensuring that the GP and PARI library built there will be suitable for debugging (if your compiler doesn't use standard flags, e.g. `-g` you may have to tweak that `Makefile`). 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.

2.5. Compilation and tests: To compile the GP binary and build the documentation, type

```
make all
```

To only compile the GP binary, type

```
make gp
```

in the distribution directory. If your `make` program supports parallel make, you can speed up the process by going to the `O`*xxx* directory that `Configure` created and doing a parallel make here (for instance `make -j4` with GNU make). It may even work from the toplevel directory.

The GP binary built above 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.

2.5.1. Testing

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 (user time) on modern machines. See the file `MACHINES` to get an idea of how much time comparable systems need. (We would appreciate a short note in the same format in case your system is not listed and you nevertheless have a working GP executable.)

If a `[BUG]` message shows up, something went wrong. Probably with the installation procedure, but it may be a bug in the Pari system, in which case we would appreciate a report (including the relevant `*.dif` file in the `O`*xxx* directory and the file `pari.cfg`).

Known problems:

• `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 (e.g Illegal Instruction on startup). It doesn't affect `gp-sta`.

2.5.2. Some more testing [*Optional*]

You can test GP in compatibility mode with `make test-compat`. If you want to test the graphic routines, use `make test-ploth`. You will have to click on the mouse button after seeing each image. There will be eight of them, probably shown twice (try to resize at least one of them as a further test). More generaly, typing `make` without argument will print the list of available extra tests among all available targets.

The `make bench` and `make test-compat` runs produce a Postscript file `pari.ps` in `O`*xxx* which you can send to a Postscript printer. The output should bear some similarity to the screen images.

Finally, `make test-kernel` is only useful to developpers, and should only be applied to an optimized gp build (not a debugging one): it checks whether the inline assembler kernel seems to work, and provides simple diagnostics if it does not.

3. 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're an avid GP fan, don't forget to delete the old pari libraries once in a while).

This installs in the directories chosen at `Configure` time the default GP executable (probably `gp-dyn`) under the name `gp`, the default PARI library (probably `libpari.so`), the necessary include files, the manual pages, the documentation and help scripts and emacs macros.

To save on disk space, you can manually `gzip` some of the documentation files if you wish: `usersch*.tex` and all `dvi` files (assuming your `xdvi` knows how to deal with compressed files); the online-help system will 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 much 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's no risk of breaking them by installing a new pari library.

3.1. Extra packages: The following optional packages endow PARI with some extra capabilities (a single package for now!).

• `galdata`: The default `polgalois` function can only compute Galois groups of polynomials of degree less or equal to 7. Install this package if you want to handle polynomials of degree bigger than 7 (and less than 11).

To install package *pack*, you need to fetch the separate archive: *pack*`.tgz` which you can download from the `pari` server. Copy the archive in the PARI toplevel directory, then extract its contents; these will go to `data/`*pack*`/`. Typing `make install-data` will then install all such packages.

3.2. The `GPRC` file: Copy the file `misc/gprc.dft` (or `gprc.dos` if you're using `GP.EXE`) to `$HOME/.gprc`. Modify it to your liking. For instance, if you're 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`.

4. Getting Started.

4.1. 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: `users.dvi` (manual with a table of contents and an index), `tutorial.dvi` (a short tutorial), and `refcard.dvi` (a reference card for GP). You can send these files to your favourite printer in the usual way, probably via `dvips`. The reference card is also provided as a `PostScript` document, which may be easier to print than its `dvi` equivalent (it is in Landscape orientation and assumes A4 paper size).

If the `pdftex` package is part of your TEX setup, you can produce these documents in PDF format, which may be more convenient for online browsing (the manual is complete with hyperlinks); type

```
make docpdf
```

All these documents are available online from PARI home page and on

```
ftp://pari.math.u-bordeaux.fr/pub/pari
```

in any case.

4.2. 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 `mattrans.c` program, which is commented in the manual.

4.3. 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 (much worse than the built-in `bnfinit`!). See the file `examples/EXPLAIN` for some explanations.

4.4. EMACS: If you want to use `gp` under GNU Emacs, read the file `emacs/pariemacs.txt`. If you are familiar with Emacs, we suggest that you do so.

4.5. The PARI Community: There are three mailing lists devoted to the PARI/GP package (run courtesy of Dan Bernstein), and most feedback should be directed to those. They are:

• `pari-announce`: to announce major version changes. You can't 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.

• `pari-users`: for everything else.

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`

The PARI home page (maintained by Gerhard Niklasch) at the address

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

maintains an archive of all discussions as well as a download area. If don't want to subscribe to those lists, you can write to us at the address

`pari@math.u-bordeaux.fr`

At the very least, we will forward you mail to the lists above and correct faulty behaviour, if necessary. 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 2.2.7}",
    year         = 2002,
    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!

Appendix B:
A Sample program and Makefile

We assume that you have installed the PARI library and include files as explained in Appendix A or in the installation guide. If you chose differently any of the directory names, change them accordingly in the Makefiles.

If the program example that we have given is in the file `matexp.c` (say as the first of several matrix transcendental functions), then a sample Makefile might look as follows. Note that the actual file `examples/Makefile` is much more elaborate and you should have a look at it if you intend to use `install()` on custom made functions, see Section 3.11.2.13.

```
CC = cc
INCDIR = /usr/local/include/pari
LIBDIR = /usr/local/lib
CFLAGS = -O -I$(INCDIR) -L$(LIBDIR)

all:  matexp

matexp:  matexp.c
      $(CC) $(CFLAGS) -o matexp matexp.c -lpari -lm
```

We then give the listing of the program `examples/matexp.c` seen in detail in Section 4.8, with the slight modifications explained at the end of that section.

```
/* Id: matexp.c,v 1.4 2002/10/02 15:28:56 karim Exp */
#include "pari.h"

GEN
matexp(GEN x,long prec)
{
  pari_sp ltop = avma;
  long lx=lg(x),i,k,n;
  GEN y,r,s,p1,p2;

  /* check that x is a square matrix */
  if (typ(x) != t_MAT) err(typeer,"matexp");
  if (lx == 1) return cgetg(1, t_MAT);
  if (lx != lg(x[1])) err(talker,"not a square matrix");

  /* convert x to real or complex of real and compute its L_2 norm */
  s = gzero; r = cgetr(prec+1); affsr(1,r); x = gmul(r,x);
  for (i=1; i<lx; i++)
    s = gadd(s, gnorml2((GEN)x[i]));
  if (typ(s) == t_REAL) setlg(s,3);
  s = gsqrt(s,3); /* we do not need much precision on s */

  /* if s < 1 we are happy */
  k = expo(s);
  if (k < 0) { n = 0; p1 = x; }
  else { n = k+1; p1 = gmul2n(x,-n); setexpo(s,-1); }

  /* initializations before the loop */
```

```
  y = gscalmat(r,lx-1); /* creates scalar matrix with r on diagonal */
  p2 = p1; r = s; k = 1;
  y = gadd(y,p2);
  /* the main loop */
  while (expo(r) >= -BITS_IN_LONG*(prec-1))
  {
    k++; p2 = gdivgs(gmul(p2,p1),k);
    r = gdivgs(gmul(s,r),k); y = gadd(y,p2);
  }
  /* square back n times if necessary */
  for (i=0; i<n; i++) y = gsqr(y);
  return gerepileupto(ltop,y);
}

int
main()
{
  long d, prec = 3;
  GEN x;
  /* take a stack of 10^6 bytes, no prime table */
  pari_init(1000000, 2);
  printf("precision of the computation in decimal digits:\n");
  d = itos(lisGEN(stdin));
  if (d > 0) prec = (long)(d*pariK1+3);
  printf("input your matrix in GP format:\n");
  x = matexp(lisGEN(stdin), prec);
  sor(x, 'g', d, 0);
  exit(0);
}
```

Appendix C:
Summary of Available Constants

In this appendix we give the list of predefined constants available in the PARI library. All of them are in the heap and *not* on the PARI stack. We start by recalling the universal objects introduced in Section 4.1:

```
t_INT: gzero (zero), gun (un), gdeux (deux)
t_FRAC: ghalf (lhalf)
t_COMPLEX: gi
t_POL: polun[..] (lpolun[..]), polx[..] (lpolx[..])
```

Only polynomials in the variables `0` and `MAXVARN` are defined initially. Use `fetch_var()` (see Section 4.6.2.2) to create new ones.

The other objects are not initialized by default:

`bern`(i). This is the $2i$-th Bernoulli number ($B_0 = 1$, $B_2 = 1/6$, $B_4 = -1/30$, etc...). To initialize them, use the function:

`void` **mpbern**`(long n, long prec)`

This creates the even numbered Bernoulli numbers up to B_{2n-2} *as real numbers* of precision `prec`. They can then be used with the macro `bern(i)`. Note that this is not a function but simply an abbreviation, hence care must be taken that `i` is inside the right bounds (i.e. $0 \le$ `i` $\le n-1$) before using it, since no checking is done by PARI itself.

`geuler`. This is Euler's constant. It is initialized by the first call to `mpeuler` (see Section 3.3.2).

`gpi`. This is the number π. It is initialized by the first call to `mppi` (see Section 3.3.4).

The use of both `geuler` and `gpi` is deprecated since it's always possible that some library function increases the precision of the constant *after* you've computed it, hence modifying the computation accuracy without your asking for it and increasing your running times for no good reason. You should always use `mpeuler` and `mppi` (note that only the first call will actually compute the constant, unless a higher precision is required).

In addition, some single or double-precision real numbers (like `PI`) are predefined, and their list is in the file `paricom.h`.

Finally, one has access to a table of (differences of) primes through the pointer `diffptr`. This is used as follows: when

`void` **pari_init**`(long size, ulong maxprime)`

is called, this table is initialized with the successive differences of primes up to (just a little beyond) `maxprime` (see Section 4.1). The prime table will occupy roughly `maxprime`/ log(`maxprime`) bytes in memory, so be sensible when choosing `maxprime` (it is 500000 by default under `gp`). In any case, the implementation requires that `maxprime` < 4294965248 (resp. 18446744073709549568) on 32-bit (resp. 64-bit) machines, whatever memory is available.

The largest prime computable using this table is available as the output of

`ulong` **maxprime**`()`

After the following initializations (the names p and *ptr* are arbitrary of course)

```
byteptr ptr = diffptr;
ulong p = 0;
```

calling the macro `NEXT_PRIME_VIADIFF_CHECK(p, ptr)` repeatedly will assign the successive prime numbers to p. Overrunning the prime table boundary will raise the error `primer1`, which will just print the error message:

```
*** not enough precomputed primes
```

and then abort the computations. The alternative macro `NEXT_PRIME_VIADIFF` operates in the same way, but will omit that check, and is slightly faster. It should be used in the following way:

```
byteptr ptr = diffptr;
ulong p = 0;
if (maxprime() < goal) err(primer1); /* not enough primes */
while (p <= goal) /* run through all primes up to goal */
{
  NEXT_PRIME_VIADIFF(p, ptr);
  ...
}
```

Here, we use the general error handling function `err` (see Section 4.7.3), with the codeword `primer1`, raising the "not enough primes" error.

You can use the function `initprimes` from the file `arith2.c` to compute a new table on the fly and assign it to `diffptr` or to a similar variable of your own. Beware that before changing `diffptr`, you should really free the (`malloc`ed) precomputed table first, and then all pointers into the old table will become invalid.

PARI currently guarantees that the first 6547 primes, up to and including 65557, will be present in the table, even if you set `maxnum` to zero.

SomeWord refers to PARI-GP concepts.
`SomeWord` is a PARI-GP keyword.
SomeWord is a generic index entry.

A

Abelian extension 127, 133
`abs` 69
`absi_cmp` 212
`absi_equal` 212
`absr_cmp` 212
accuracy 9
`acos` 70
`acosh` 70
`addell` 90
addhelp 204
`addhelp` 42, 164
`addii` 173
`addir` 173
`addis` 173
`addll` 207
`addllx` 207
`addmul` 207
`addprimes` 43, 76, 126
`addri` 173
`addrr` 173
`addsii` 214
`addumului` 215
`adduumod` 208
`adj` 144
adjoint matrix 143
`affii` 210
`affir` 210
`affri` 210
`affrr` 210
`affrs` 210
`affsi` 210
`affsr` 210
`affsz` 210
`agm` 70
`akell` 90
`algdep` 141
`algdep0` 141
algebraic dependence 141
`algtobasis` 117
`alias` 42, 164
`allocatemem` 19, 164
`allocatemoremem` 165, 198
alternating series 155
and 57
`and` 61
`anell` 90
`apell` 90
`apell2` 90
`apprgen` 136
`apprgen9` 136
`area` 89
`arg` 70
Artin L-function 107
Artin root number 107
`asin` 70
`asinh` 70
assignment 177
`assmat` 144
`atan` 70
`atanh` 70
automatic simplification 20
available commands 23
`avma` 178, 179

B

backslash character 32
`base` 117
`base2` 117
`basistoalg` 117
Berlekamp 81
`bern` 231
`bernfrac` 70
Bernoulli numbers 70, 71, 76
`bernreal` 70, 71
`bernvec` 71
`besselh1` 71
`besselh2` 71
`besseli` 71
`besselj` 71
`besseljh` 71
`besselk` 71
`besseln` 71
`bestappr` 76, 77
`bestappr0` 77
`bezout` 77
`bezoutres` 77
`bfffo` 207
`BIGDEFAULTPREC` 174
`BIGINT` 189, 190
`bigomega` 77
`bilhell` 91

binaire . . . 61
binary file . . . 170
binary file . . . 24, 168
binary flag . . . 53
binary quadratic form . . . 7, 27, 60
binary . . . 61
binome . . . 77
binomial coefficient . . . 77
binomial . . . 77
Birch and Swinnerton-Dyer conjecture . . . 91
bitand . . . 57, 61
bitneg . . . 61
bitnegimply . . . 62
bitor . . . 57, 62
BITS_IN_LONG . . . 174
bittest . . . 62
bitwise and . . . 57, 61
bitwise exclusive or . . . 62
bitwise inclusive or . . . 62
bitwise negation . . . 61
bitwise or . . . 57
bitxor . . . 62
bit_accuracy . . . 174, 199
bnf . . . 96
bnf . . . 39, 98
bnfcertify . . . 99
bnfclassgrouponly . . . 100
bnfclassunit . . . 99
bnfclassunit0 . . . 100
bnfclgp . . . 100
bnfdecodemodule . . . 100
bnfinit . . . 96, 100
bnfinit0 . . . 102
bnfisintnorm . . . 102
bnfisnorm . . . 102
bnfisprincipal . . . 101, 102
bnfissunit . . . 102
bnfisunit . . . 103
bnfmake . . . 103
bnfnarrow . . . 87, 103
bnfreg . . . 104
bnfsignunit . . . 103
bnfsunit . . . 104
bnfunit . . . 104
bnr . . . 96
bnr . . . 39
bnrclass . . . 105
bnrclass0 . . . 105
bnrclassno . . . 105
bnrclassnolist . . . 105
bnrconductor . . . 105
bnrconductorofchar . . . 106
bnrdisc . . . 106
bnrdisc0 . . . 106
bnrdisclist . . . 106
bnrdisclist0 . . . 107
bnrinit . . . 105, 106, 107
bnrinit0 . . . 107
bnrisconductor . . . 107
bnrisprincipal . . . 101, 107
bnrL1 . . . 104, 105
bnrrootnumber . . . 107
bnrstark . . . 88, 107, 108, 134
boolean operators . . . 57
brace characters . . . 32
break loop . . . 44, 168
break . . . 44, 162
Breuil . . . 90
brute . . . 193, 195
buchfu . . . 104
buchimag . . . 87
Buchmann . . . 98, 99, 115, 136
Buchmann-McCurley . . . 87
buchnarrow . . . 103
buchreal . . . 87
buffersize . . . 15

C

Cantor-Zassenhaus . . . 81
caract . . . 142
caradj . . . 142
carhess . . . 142
case distinction . . . 31
ceil . . . 62
centerlift . . . 62, 63
centerlift0 . . . 63
certifybuchall . . . 99
cgetg . . . 175, 176, 185, 210
cgeti . . . 175, 210
cgetp . . . 215
cgetr . . . 175, 210
cgiv . . . 180, 210
changevar . . . 34, 63
character string . . . 28, 190
character . . . 104, 106, 107
characteristic polynomial . . . 141
charpoly . . . 141

charpoly0 . . . 142
Chebyshev . . . 139
chinese . . . 77
chinois . . . 77
classno . . . 85
classno2 . . . 85
clgp . . . 98
CLISP . . . 46
clone . . . 172, 178
cmdtool . . . 20
cmpii . . . 212
cmpir . . . 212
cmpis . . . 212
cmpri . . . 212
cmprr . . . 212
cmprs . . . 212
cmpsi . . . 212
cmpsr . . . 212
co8 . . . 216
code words . . . 63
codiff . . . 98
coeff . . . 63, 173, 215
coefs_to_col . . . 177
coefs_to_int . . . 177
coefs_to_pol . . . 177
coefs_to_vec . . . 177
colors . . . 16
column vector . . . 7, 27, 190
comparison operators . . . 57
compatible . . . 16
completion . . . 50
complex number . . . 7, 8, 25, 188
compo . . . 63, 215
component . . . 63
components . . . 63
composition . . . 85, 86
compositum . . . 124
compraw . . . 85
compress . . . 24
concat . . . 40, 142, 143
conductor . . . 106
conj . . . 64
conjvec . . . 64
Conrad . . . 90
content . . . 35, 77, 218
contfrac . . . 78
contfrac0 . . . 78
contfracpnqn . . . 78
continued fraction . . . 78
Control statements . . . 162
conversions . . . 179
convol . . . 140
coordch . . . 91
copy . . . 178
core . . . 78
core0 . . . 78
core2 . . . 78
coredisc . . . 78
coredisc0 . . . 78
coredisc2 . . . 78
cos . . . 71
cosh . . . 71
cotan . . . 72
CPU time . . . 21
creation . . . 175
cyc . . . 98
cyclo . . . 137
Cygwin . . . 13

D

datapath . . . 17
dbltor . . . 179, 215
debug . . . 17, 23, 195
debugfiles . . . 17, 23
debugging . . . 195
debuglevel . . . 81
DEBUGLEVEL . . . 195, 196
debugmem . . . 17, 23
DEBUGMEM . . . 195
debugmem . . . 195
decodemodule . . . 100
decomposition into squares . . . 148
Dedekind . . . 72, 108, 128, 135
default precision . . . 9
default . . . 42, 165
DEFAULTPREC . . . 174
defaults . . . 14, 23
definite binary quadratic form . . . 190
degree . . . 189
degree . . . 137, 189
delete_var . . . 192
denom . . . 64
denominator . . . 35, 64
deriv . . . 135, 136
derivpol . . . 136
destruction . . . 180
det . . . 144

det2 . . . 144
detint . . . 144
diagonal . . . 144
Diamond . . . 90
diff . . . 98
difference . . . 54
diffptr . . . 172, 231
dilog . . . 72
dirdiv . . . 78
direuler . . . 78, 79
Dirichlet series . . . 78, 79, 108
dirmul . . . 79
dirzetak . . . 108
disc . . . 89, 98
discf . . . 118
discsr . . . 137
diviiexact . . . 214
divise . . . 214
divisii . . . 214
divisors . . . 79
diviuexact . . . 214
divll . . . 208
divrem . . . 35, 56
divsum . . . 155
divuumod . . . 208
dvi . . . 51
dvmdii . . . 214
dvmdiiz . . . 214
dvmdis . . . 213
dvmdisz . . . 214
dvmdsi . . . 213
dvmdsiz . . . 214
dvmdss . . . 213
dvmdssz . . . 214

E

echo . . . 17, 23
ECM . . . 76, 81
editing characters . . . 31
effective length . . . 187, 189
egalii . . . 212
eigen . . . 144
eint1 . . . 72
element_div . . . 118
element_divmodpr . . . 118
element_mul . . . 118
element_mulmodpr . . . 118
element_pow . . . 118
element_powmodpr . . . 118
element_reduce . . . 119
element_val . . . 119
ell . . . 39
elladd . . . 90
ellak . . . 90
ellan . . . 90
ellap . . . 90
ellap0 . . . 90
ellbil . . . 90
ellchangecurve . . . 91
ellchangepoint . . . 91
elleisnum . . . 91
elleta . . . 91
ellglobalred . . . 91
ellheight . . . 91
ellheight0 . . . 91
ellheightmatrix . . . 92
ellinit . . . 89, 92
ellinit0 . . . 93
ellisoncurve . . . 93
ellj . . . 93
elllocalred . . . 93
elllseries . . . 93
ellminimalmodel . . . 91, 93, 94
ellorder . . . 94
ellordinate . . . 94
ellpointtoz . . . 94
ellpow . . . 94
ellrootno . . . 94
ellsigma . . . 94
ellsub . . . 94
elltaniyama . . . 95
elltors . . . 95
elltors0 . . . 95
ellwp . . . 95
ellwp0 . . . 95
ellzeta . . . 95
ellztopoint . . . 95
Emacs . . . 49
EMX . . . 13
entree . . . 39, 54, 203
environment expansion . . . 61
environment expansion . . . 15
environment variable . . . 61
erfc . . . 72
err . . . 195, 196, 198
errfile . . . 195
error handler . . . 45

error recovery . . . 43
error trapping . . . 44
error . . . 42, 195
`error` . . . 42, 44, 165
`eta` . . . 72, 89
Euclid . . . 82
Euclidean quotient . . . 55
Euclidean remainder . . . 55
Euler product . . . 78, 85, 154
Euler totient function . . . 76, 79
Euler . . . 155
`Euler` . . . 31, 33, 69
Euler-Maclaurin . . . 76
`eulerphi` . . . 76, 79
`eval` . . . 60, 136
exact object . . . 8
`exe` . . . 194
`exp` . . . 72
`expi` . . . 209
`expo` . . . 187, 189, 209
expression sequence . . . 35
expression . . . 35
`extern` . . . 20, 42, 165
external prettyprint . . . 19
`extract` . . . 152

F

`factcantor` . . . 81
`factmod` . . . 82
`factmod9` . . . 81
`factor` . . . 79, 80
`factor0` . . . 80
`factorback` . . . 80
`factorback0` . . . 80
`factorcantor` . . . 81
`factoredbase` . . . 117
`factoredpolred` . . . 126
`factorff` . . . 79, 81
`factorial` . . . 81
`factorint` . . . 79, 81
`factormod` . . . 79, 81, 82
`factornf` . . . 79, 80, 108
`factorpadic` . . . 136
`factorpadic4` . . . 136
`fetch_user_var` . . . 191
`fetch_var` . . . 191
`ffinit` . . . 82
`fibo` . . . 82
`fibonacci` . . . 82
field discriminant . . . 117
filename . . . 15
filter . . . 193
`fincke_pohst` . . . 150
finite field . . . 26
fixed floating point format . . . 17
flag . . . 53
`flisexpr` . . . 192, 193
`flisseq` . . . 192, 193
`flissexpr` . . . 191
`floor` . . . 64
`foo` . . . 53
`for` . . . 162
`forcecopy` . . . 178, 179, 215
Ford . . . 117
`fordiv` . . . 162
formal integration . . . 136
format . . . 193, 195
`format` . . . 17
`forprime` . . . 162
`forstep` . . . 162
`forsubgroup` . . . 134, 163
`forvec` . . . 163
`FpM_ker` . . . 146
`fprintferr` . . . 196, 203
`frac` . . . 64
FreeBSD . . . 13
`fu` . . . 98
fundamental units . . . 88, 98, 99
`fundunit` . . . 88
`futu` . . . 98

G

`gabs` . . . 70
`gach` . . . 70
`gacos` . . . 70
`gadd` . . . 54
`gaddgs` . . . 173
`gaddgsz` . . . 173
`gaddgs[z]` . . . 218
`gaddsg` . . . 173
`gaddsgz` . . . 173
`gaddsg[z]` . . . 218
`gaddz` . . . 173, 178, 185
`gadd[z]` . . . 208, 218
`gaffect` . . . 177, 178, 217
`gaffsg` . . . 178, 217

Galois 102, 119, 120, 123, 125, 132, 163
galois 126
galoisapply 120
galoisconj 121
galoisconj0 121
galoisconj2 121
galoisconj4 121
galoisexport 108, 109
galoisfixedfield 109, 163
galoisidentify 109
galoisinit 108, 109, 110
galoisisabelian 110
galoispermtopol 110, 111
galoissubcyclo 107, 111, 139, 163
galoissubfields 111, 112, 123
galoissubgroups 112
gamma 72
gammah 72
gand 57
garbage collecting 179
garg 70
gash 70
gasin 70
gatan 70
gath 70
gauss 148
gaussmodulo 148
gaussmodulo2 148
gbezout 77, 220
gbitand 61
gbitneg 62
gbitnegimply 62
gbitor 62
gbittest 62
gbittest3 62
gbitxor 62
gboundcf 78
gcarrecomplet 83
gcarreparfait 83
gcd 82
gceil 62, 217
gcf 78
gcf2 78
gch 71
gclone 178, 179, 215
gcmp 57, 217
gcmp0 . . . . . 57, 216
gcmp1 . . . . . 57, 216
gcmpgs 217
gcmpsg 217
gcmp_1 217
gcmp_1 57
gconj 64
gcopy 178, 179, 215
gcos 71
gcotan 72
gcvtoi 67, 218
gcvtop 216
gdeuc 219
gdeux 171
gdiv 55
gdivent 55
gdiventgs[z] 219
gdiventres 56, 219
gdiventsg[z] 219
gdivent[z] 219
gdivgs[z] 219
gdivise 220
gdivmod 219
gdivround 55, 219
gdivsg[z] 219
gdiv[z] 219
gegal 57, 217
gegalgs 217
gegalsg 217
gen (member function) 98
GEN 7, 171
gener 89
generic matrix 41
genrand 66
GENtostr 60, 194
geq 57
gerepile 177, 180, 182, 200, 210
gerepileall 182, 185
gerepilecopy 182, 185
gerepilemany 184
gerepilemany 182
gerepilemanysp 182
gerepileupto 177, 181, 182, 201
getheap 165
getrand 165
getstack 165
gettime 165
geuler 231
geval 136
gexp 72, 197
gexpo 187, 209, 216
gfloor 64, 217

gfrac 64, 217
ggamd 72
ggamma 72
ggcd 82, 220
gge 57
ggprecision 66
ggrandocp 135
ggt 57
ggval 68, 217
ghalf 171
ghell 91
ghell2 91
gi 171
gimag 64
ginv 217
ginvmod 220
gisfundamental 82
gisirreducible 137
gisprime 83
gispseudoprime 83
gissquarefree 83
glambdak 135
glcm 84, 220
gle 57
glength 64
glngamma 73
global 39
global 33, 165
globalreduction 91
glog 73
glogagm 73
glt 57
gmax 58
gmaxgs[z] 218
gmaxsg[z] 218
gmax[z] 218
gmin 58
gmings[z] 218
gminsg[z] 218
gmin[z] 218
gmod 56
gmodgs[z] 219
gmodsg[z] 219
gmodulcp 59, 216
gmodulgs 216
gmodulo 59, 216
gmodulss 216
gmod[z] 219
gmul 55
gmul2n 57, 199
gmul2n[z] 219
gmulgs[z] 219
gmulsg[z] 219
gmul[z] 219
gne 57
gneg 54
gnil 33
gnorm 65
gnorml2 . . . . . 65, 199
gnot 57
gor 57
GP 13
gphelp 23
gpi 231
gpolvar 68
gpow 57, 69, 220
gpowgs 220
gprc 13, 15, 19
GPRC 47
gprc 47
gprec 66
gpsi 74
greal 66
gred 188, 189
gred[z] 218
greffe 215
gres 220
GRH 87, 98, 99, 102, 132, 141
grndtoi 67, 218
ground 67, 217
gscalmat 145, 200
gscalsmat 145, 200
gsh 74
gshift 57, 200
gshift3 57
gshift[z] 219
gsigne 58, 187, 209, 216
gsin 74
gsqr 55, 74, 217
gsqrt 74
gsqrtn 75
GSTR 190
gsub 54
gsubgs[z] 218
gsubsg[z] 218
gsubst 140, 220
gsubst0 140
gsub[z] 218

gsumdivk 88
gtan 75
gth 75
gtodouble 179, 215
gtolong 179, 216
gtomat 58
gtopoly 59, 216
gtopolyrev 59, 216
gtoser 60, 216
gtoset 60
gtovec 61, 216
gtovecsmall 61
gtrace 151
gtrans 148
gtrunc 67, 218
gun 171
gunclone 179
gval 217
gvar 189, 190, 209, 216
gzero 171, 199
gzeta 76
gzetak 135
gzip 24
gzip 170

H

Hadamard product 140
hashing function 39
hashtable 39
hbessel1 71
hbessel2 71
hclassno 85
heap 24, 231
hell 91
help 18
Hermite normal form . . . 80, 96, 97, 113, 115, 121, 123, 134, 144, 145, 163
hess 144
hexadecimal tree 194
hil 82
Hilbert class field 87
Hilbert matrix 144
Hilbert symbol 82, 121
hilbert 82
histsize 18
hnf 145
hnfall 145
hnfmod 145
hnfmodid 145
hqfeval 136
Hurwitz class number 85
hyperu 72

I

I 25, 31, 69
ibessel 71
ideal list 97
ideal 96
idealadd 112
idealaddtoone 112
idealaddtoone0 112
idealappr 112
idealappr0 112
idealchinese 112
idealcoprime 112
idealdiv 112, 113
idealdiv0 112
idealdivexact 113
idealfactor 113
idealhermite 113
idealhnf 113, 129
idealhnf0 113
idealintersect 113, 146
idealinv 113, 122
ideallist 113
ideallist0 113
ideallistarch 113
ideallistarch0 113
idealllred 115
ideallog 114
idealmin 114
idealmul 114
idealmulred 114
idealnorm 114
idealpow 114
idealpowred 114
idealpows 114
idealprimedec 114
idealprincipal 115
idealred 115
idealstar 115
idealstar0 115
idealtwoelt 116
idealval 116
ideal_two_elt0 116
idele 96

ideleprincipal 116
idmat 145
if 163
imag 64
image 145
imagecompl 146
imag_i 64
imprecise object 8
incgam 72, 73
incgam0 73
incgam1 73
incgam2 73
incgamc 73
inclusive or 57
indefinite binary quadratic form 189
indexrank 146
indexsort 152
infile 195
infinite product 154
infinite sum 155
infinity 153
initell 93
initzeta 135
input 192
input 166
install 42, 46, 166, 193, 195, 204
integ 136
integer 7, 25, 186
integermod 7, 25, 188
integral basis 117
internal longword format 24
internal representation 24
interpolating polynomial 137
intersect 146
intformal 136
intnum 154
inverseimage 146
invsmod 208
invumod 208
iscomplex 217
isdiagonal 146
isexactzero 57, 216
isfundamental 82
isideal 122
ismonome 217
isprime 82, 83
isprincipalall 102
isprincipalrayall 107
ispseudoprime 82, 83, 84
issquare 83
issquarefree 76, 83
isunit 103
itor 210
itos 179, 202, 210

J

j 89
jacobi 149
jbesselh 71
jell 93

K

kbessel 71
kbessel2 71
ker 146
keri 146
kerint 146
keyword 40
kill 166
Kodaira 93
Kronecker symbol 83
kronecker 83, 84

L

laplace 140
lclone 178
lcm 84
lcopy 178
leadingcoeff 138
leaves 8
leaves 7
Legendre polynomial 138
Legendre symbol 83
legendre 138
length 64
Lenstra 81, 136
lex 57
lexcmp 58, 217
lexsort 152
lg 186, 199, 209
lgef 189, 190, 209
lgefint 187, 209
lgetg 175
lgeti 175
lgetr 175
library mode 171
LiDIA 81

lift 62, 64, 65
lift0 65
limit 38
lindep 143
lindep0 143
line editor 50
linear dependence 143
lines 18
Linux 13, 204
lisexpr 193
lisGEN 193, 198
Lisp 46
lisseq 193
list 7, 28, 190
List 58
listcreate 143
listinsert 143
listkill 143
listput 143
listsort 143
LLL 115, 120, 141, 143, 145, 146, 149
lll 149
lllgram 150
lllgramint 150
lllgramkerim 150
lllint 149
lllkerim 149
lngamma 73
local 33, 36, 39
localreduction 93
log 18, 22, 23, 54, 73, 167
logfile 167
logfile 18
LONG_IS_64BIT 174
lpolx 173
lseriesell 93

M

MACHINES 13
mael 63, 173
mael$n$ 215
makebigbnf 103
Mat 27, 58, 142
matadjoint 143
matalgtobasis 116
matbasistoalg 116
matbrute 194
matcompanion 144
matdet 144
matdetint 144
matdiagonal 144
mateigen 144
matextract 152
mathell 92
mathess 144
mathilbert 144
mathnf 141, 144
mathnf0 145
mathnfmod 145
mathnfmodid 145
matid 145
matimage 145
matimage0 145
matimagecompl 145
matindexrank 146
matintersect 146
matinverseimage 146
matisdiagonal 146
matker 146
matker0 146
matkerint 146
matkerint0 146
matmuldiagonal 146
matmultodiagonal 146
matpascal 147
matqpascal 147
matrank 147
matrice 147
matrix 7, 8, 27, 41, 190
matrix 147
matrixqz 147
matrixqz0 147
matsize 147
matsnf 147
matsnf0 148
matsolve 148
matsolvemod 148
matsolvemod0 148
matsupplement 148
mattranspose 148
max 58
maxprime 171, 231
MAXVARN 172, 191
MEDDEFAULTPREC 174
member functions 39, 89, 98
min 58
minideal 114

minim 150
minim2 150
minimal model 91, 93
minimal polynomial 141
Mod 59
Mod0 59
mod2 187
mod4 187
mod64 187
modpr 123
modreverse 116
modulargcd 82
module 97
Moebius 76, 83, 84
moebius 76, 84
Mordell-Weil group 92, 94
mpadd 173
mpaff 210
mpbern 231
mpcmp 212
mpdivis 215
mpdvmdz 214
mpent[z] 211
mpeuler 69, 231
mpfact 81
mpfactr 81
mpinvir 211
mpinvrr 212
mpinvsr 211
mpinvz 211
mpodd 187
mppi 69, 231
MPQS 76, 81
mpshift[z] 211
mptrunc[z] 211
msgTIMER 196
msgtimer 196
mu 84
mulll 207
mulsii 215
multivariate polynomial 38
muluumod 208

N

name_var 192
nbessel 71
newtonpoly 116
new_chunk 175
new_galois_format 17, 18, 125, 126
next 44, 164
nextprime 84
NEXT_PRIME_VIADIFF 232
NEXT_PRIME_VIADIFF_CHECK(p, ptr) . . 232
nf 96
nf 39, 98
nfalgtobasis 117
nfbasis 117, 122
nfbasis0 117
nfbasistoalg 117
nfdetint 117
nfdisc 117
nfdiscf0 117
nfdiveuc 118
nfdivres 118
nfeltdiv 118
nfeltdiveuc 118
nfeltdivmodpr 118
nfeltdivrem 118
nfeltmod 118
nfeltmul 118
nfeltmulmodpr 118
nfeltpow 118
nfeltpowmodpr 118
nfeltreduce 119
nfeltreducemodpr 119
nfeltval 119
nffactor 80, 108, 119, 122
nffactormod 119
nfgaloisapply 119
nfgaloisconj 109, 120
nfhermite 121
nfhermitemod 121
nfhilbert 121
nfhnf 121
nfhnfmod 121
nfinit 96, 109, 121, 126
nfinit0 122
nfisideal 122
nfisincl 122, 123
nfisisom 123
nfkermodpr 123
nfmod 118
nfmodprinit 118, 123
nfnewprec 122, 123
nfreducemodpr 119
nfroots 123
nfrootsof1 123

nfsmith 124
nfsnf 124
nfsolvemodpr 124
nfsubfield 109
nfsubfields 123, 163
no 98
norm 65
normalize 218
normalizepol 218
norml2 65
not 57
nucomp 86
nudupl 86
numbdiv 84
number field 26
numbpart 84
numdiv 84
numer 65
numerator 35, 65
numerical derivation 29
numerical integration 153
numtoperm 65, 66
nupow 86
Néron-Tate height 91

O

O 31, 135
omega 87
omega 84, 89
oncurve 93
operator 28
or 57
or 62
ordell 94
order 89
orderell 94
ordred 127
outbeaut 194
outbrute 194
outfile 194
outmat 194
output formats 14
output 193
output 18, 23, 194, 195

P

p-adic number 7, 25, 188
padicappr 136
padicprec 66
parametric plot 158
pari.h 171
pariErr 195
pariK1 198
pariOut 195
PariPerl 46
pariputs 203
PariPython 46
parisize 19
pari_init 171, 172, 231
pari_sp 179
pari_timer 196
parser code 201, 204
Pascal triangle 147
path 19
Pauli 117
perf 150
Perl 46
permtonum 65, 66
phi 79
Pi 31, 69
plot 157
plotbox 157
plotclip 157
plotcolor 157
plotcopy 157, 158
plotcursor 158
plotdraw 158
plotfile 158
ploth 54, 158
plothraw 159
plothsizes 159
plotinit 159
plotkill 160
plotlines 160
plotlinetype 160
plotmove 160
plotpoints 160
plotpointsize 160
plotpointtype 160
plotrbox 160
plotrecth 159, 161
plotrecthraw 161
plotrline 161
plotrmove 161
plotrpoint 161
plotscale 159, 161
plotstring 42, 161

plotterm 42, 161
pnqn 78
pointch 91
pointell 96
pointer 54
pointer 54
Pol 59
polcoeff 63, 136
polcoeff0 137
polcompositum 124
polcompositum0 125
polcyclo 137
poldegree 137
poldisc 137
poldisc0 137
poldiscreduced 137
poldivres 219
poleval 136
polfnf 108
polgalois 18, 125
polhensellift 137
polint 137
polinterpolate 137
polisirreducible 137
Pollard Rho 76, 81
pollead 138
pollegendre 138
polmod 7, 26, 188
polmodrecip 116
polrecip 138
polred 126
polred0 126
polredabs 126
polredabs0 127
polredord 127
polresultant 138
polresultant0 138
Polrev 59
polroots 138
polrootsmod 138
polrootspadic 139
polsturm 139
polsubcyclo 139
polsylvestermatrix 139
polsym 139
poltchebi 139
poltschirnhaus 127
polun 172, 191
polvar 191
polx 172, 191
polylog 73
polylog0 74
polynomial 7, 8, 27, 189
polzag 139
polzagier 139
polzagreel 139
PostScript 157
powell 94
power series 7, 8, 27, 189
powering 56, 69
powraw 86
powusmod 208
powuumod 208
precdl 69
precision 68
precision 66, 216
precision0 66
precp 188, 209
precprime 84
preferences file 13, 14, 47, 204
prettymatrix format 19
prettyprint format 19
prettyprinter 19
prime 84
primedec 115
primeform 86
primelimit 19, 126
primer1 232
primes 84
principal ideal 115
principalideal 115
principalidele 116
print 40, 42, 167
print1 167
printf 195
printp 167
printp1 167
printtex 167
priority 28
prod 154
prodeuler 154
prodinf 154
prodinf1 154
product 55
produit 154
programming 32, 162
prompt 19
prompt_cont 20

psdraw 161
pseudo-basis 97
pseudo-matrix 97
psfile 20, 157
psi 74
psploth 161
psplothraw 162
pvaluation 217
Python 46

Q

Qfb 60
Qfb0 60
qfbclassno 84, 85
qfbclassno0 85
qfbcompraw 85
qfbhclassno 85
qfbnucomp 86
qfbnupow 86
qfbpowraw 86
qfbprimeform 86
qfbred 86
qfbred0 86
qfbsolve 86, 87
qfeval 136
qfgaussred 148
qfi 60
qfjacobi 149
qflll 141, 149
qflll0 149
qfllllgram 149
qfllllgram0 150
qfminim 150
qfminim0 150
qfperfection 150
qfr 60
qfrep 150
qfrep0 150
qfsign 150
quadclassunit 87
quadclassunit0 87
quaddisc 87
quadgen 26, 87
quadhilbert 87
quadpoly 87
quadpoly0 87
quadratic number 7, 8, 26, 188
quadray 88
quadregulator 88
quadunit 88
quit 24, 167
quote 167
quotient 55

R

racine 88
random 66
rank 147
rational function 7, 27, 189
rational number 7, 25, 188
raw format 18
rayclassno 105
rayclassnolist 105
read 24
read 21, 42, 168, 169
readline 50
readline 20
real number 7, 25, 187
real 66
realprecision 20, 24
realzero 199
real_i 66
recip 140
recursion depth 38
recursion 38
recursive plot 158
recursiveness 6
redimag 86
redreal 86
redrealnod 86
reduceddiscsmith 137
reduction 85, 86
reference card 22
reg 98
regula 88
regulator 104
removeprimes 88
reorder 34, 63, 168
resultant2 138
return 44, 164
rhoreal 86
rhorealnod 86
Riemann zeta-function 37, 76
rnf 96
rnfalgtobasis 127
rnfbasis 127

rnfbasistoalg 127
rnfcharpoly 127
rnfconductor 127, 128
rnfdedekind 128
rnfdet 128
rnfdisc 128
rnfdiscf 128
rnfelementabstorel 128
rnfelementdown 128
rnfelementreltoabs 128
rnfelementup 129
rnfeltabstorel 128
rnfeltdown 128
rnfeltreltoabs 128
rnfeltup 129
rnfequation 129
rnfequation0 129
rnfhermitebasis 129
rnfhnfbasis 129
rnfidealabstorel 129
rnfidealdown 129
rnfidealhermite 130
rnfidealhnf 129
rnfidealmul 130
rnfidealnormabs 130
rnfidealnormrel 130
rnfidealreltoabs 130
rnfidealtwoelement 130
rnfidealtwoelt 130
rnfidealup 130, 131
rnfinit 131
rnfinitalg 132
rnfisfree 132
rnfisnorm 132
rnfisnorminit 132
rnfkummer 132, 133, 134
rnflllgram 133
rnfnormgroup 133
rnfpolred 133
rnfpolredabs 133
rnfpseudobasis 127, 134
rnfsteinitz 134
Roblot 117
rootmod 138
rootmod2 138
rootpadic 139
roots 89, 98, 138
rootsof1 123
rootsold 138
round 2 117
round 4 . . . . . 117, 136
round 66
row vector 7, 27, 190
RSX 13
rtodbl 179, 215

S

scalar product 55
scalar type 8
Schönage 138
scientific format 17
secure 20
Ser 60
serconvol 140
seriesprecision 20, 24
serlaplace 140
serreverse 140
Set 60
setexpo 187, 189, 210
setintersect 151
setisset 151
setlg 186, 199, 209
setlgef 189, 190, 209
setlgefint 187, 209
setminus 151
setprecp 188, 210
setrand 168
setsearch 151
setsigne 187, 189, 210
settyp 186, 209
setunion 151
setvalp 188, 189, 210
setvarn 177, 189, 210
Shanks SQUFOF 76, 81
Shanks 60, 84, 85, 86
shift 57
shifti 211
shiftl 207
shiftlr 207
shiftmul 57
shiftr 211
shifts 211
sigma 88, 155
sign 58
sign 58, 98
signat 151
signe 187, 189, 209

signunits . . . 103
simplefactmod . . . 82
simplify . . . 20, 22, 67
sin . . . 74
sindexlexsort . . . 152
sindexsort . . . 152
sinh . . . 74
sizebyte . . . 67
sizedigit . . . 67, 216
smallfact . . . 80
smallinitell . . . 93
Smith normal form . . 98, 101, 103, 115, 124, 134, 147, 163
smith . . . 148
solve . . . 154
somme . . . 155
sor . . . 194
sort . . . 152
sqr . . . 74
sqred . . . 149
sqrt . . . 74
sqrtint . . . 88
sqrtn . . . 74
srgcd . . . 82
stack . . . 171
stack . . . 24, 231
stackmalloc . . . 176
stacksize . . . 38
stack_lim . . . 185
Stark units . . . 87, 107
startup . . . 47
stderr . . . 195
stdout . . . 194
Steinitz class . . . 134
stoi . . . 179, 211
stor . . . 211
Str . . . 40, 42, 60
Strchr . . . 60, 61
Strexpand . . . 61
strftime . . . 15, 19
strictmatch . . . 21
string context . . . 40, 202
string . . . 7, 28, 40
Strtex . . . 61
strtoGEN . . . 60
sturm . . . 139
sturmpart . . . 139
subcyclo . . . 139
subell . . . 94
subfield . . . 123
subfields . . . 123
subgroup . . . 163
subgrouplist . . . 134, 163
subgrouplist0 . . . 134
subll . . . 207
subllx . . . 207
subres . . . 138, 220
subresext . . . 77
subresultant algorithm . . . 82, 137, 138
subst . . . 140
subuumod . . . 208
sum . . . 54
sum . . . 153, 155
sumalt . . . 155
sumdiv . . . 88, 155
suminf . . . 155
sumpos . . . 155, 156
suppl . . . 148
switchin . . . 195
switchout . . . 194, 195
sylvestermatrix . . . 139
symmetric powers . . . 139
system . . . 20, 42, 166, 168

T

t2 . . . 98
taille . . . 67, 215
taille2 . . . 67
talker . . . 195
Tamagawa number . . . 91, 93
tan . . . 75
tanh . . . 75
taniyama . . . 95
Taniyama-Weil conjecture . . . 90
Tate . . . 89
tate . . . 89
tayl . . . 141
Taylor series . . . 55
Taylor . . . 90
taylor . . . 140
tchebi . . . 139
teich . . . 75
teichmuller . . . 75
tex2mail . . . 19
texprint . . . 194
TeXstyle . . . 18, 21
theta . . . 75

thetanullk 75
thue 141
thueinit 141
time expansion 15
timer 21
TIMER 196
timer 196
timer2 196
trace 151
Trager 108
trap 42, 44, 168
truecoeff 63, 137, 215
truedvmdii 214
truncate 64, 67
tschirnhaus 127
tu 98
tufu 98
tutorial 22
typ 186, 209
type number 186
type 42, 169
typecast 172
types 6
t_COL 7, 27, 190
t_COMPLEX 7, 25, 188
t_FRAC 7, 25, 188
t_FRACN 7, 25, 188
t_INT 7, 25, 186
t_INTMOD 7, 25, 188
t_LIST 7, 28, 190
t_MAT 7, 27, 190
t_PADIC 7, 25, 188
t_POL 7, 27, 189
t_POLMOD 7, 26, 188
t_QFI 7, 27, 190
t_QFR 7, 27, 189
t_QUAD 7, 26, 188
t_REAL 7, 25, 187
t_RFRAC 7, 27, 189
t_RFRACN 7, 27, 189
t_SER 7, 27, 189
t_STR 7, 28, 190
t_VEC 7, 27, 190
t_VECSMALL 28

U

u2toi 177
ulimit 38
ulong 207
un 173
universal object 231
until 164
user defined functions 36

V

vali 211
valp 188, 189, 209
vals 211
valuation 67
van Hoeij 79, 108
varargs 177
varentries 191
variable (priority) 26, 34, 190
variable (temporary) 191
variable (user) 191
variable number 189, 190, 202
variable 26, 27, 29, 32, 172
variable 68
varn 189, 190, 209
Vec 27, 61
vecbezout 77
vecbezoutres 77
veceint1 72
vecextract 146, 151
vecmax 58
vecmin 58
Vecsmall 61
vecsort 152
vecsort0 152
vecteur 153
vecteursmall 153
vector 8
vector 153
vectorsmall 153
vectorv 153
version number 24
Vi 50
voir 194
vvecteur 153

W

w 89
weber 75
weber0 76
Weierstrass $\wp$-function 95
Weierstrass equation 89

Weil curve 95
weipell 95
wf 76
wf1 76
wf2 76
whatnow 42, 169
while 164
Wiles 90
write 21, 24, 42, 169
write1 169
writebin 168
writebin 169
writetex 170

X

x[,n] 63
x[m,n] 63
x[m,] 63
x[n] 63

Z

Zassenhaus 81, 136
zbrent 155
zell 94
zero 8
zero 173
zeta function 37
zeta 76
zetak 135
zetakinit 135
zideallog 114
zk 98
zkst 98
znlog 88
znorder 89
znprimroot 89
znstar 89