

User's Guide
to
the PARI library
(version 2.3.5)

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

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

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

Copyright © 2000–2006 The PARI Group

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions, or translations, of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

PARI/GP is Copyright © 2000–2006 The PARI Group

PARI/GP is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation. It is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY WHATSOEVER.

Table of Contents

Chapter 4: Programming PARI in Library Mode	7
4.1 Introduction: initializations, universal objects	7
4.2 Important technical notes	8
4.2.1 Types	8
4.2.2 Type recursivity	8
4.2.3 Variations on basic functions	9
4.2.4 Portability: 32-bit / 64-bit architectures	9
4.3 Garbage collection	10
4.3.1 Why and how	10
4.3.2 Examples	13
4.3.3 Comments	16
4.4 Creation of PARI objects, assignments, conversions	16
4.4.1 Creation of PARI objects	16
4.4.2 Assignments	18
4.4.3 Copy	19
4.4.4 Clones	20
4.4.5 Conversions	20
4.5 Implementation of the PARI types	20
4.5.1 Type <code>t_INT</code> (integer)	21
4.5.2 Type <code>t_REAL</code> (real number)	23
4.5.3 Type <code>t_INTMOD</code>	23
4.5.4 Type <code>t_FRAC</code> (rational number)	23
4.5.5 Type <code>t_COMPLEX</code> (complex number)	23
4.5.6 Type <code>t_PADIC</code> (p -adic numbers)	23
4.5.7 Type <code>t_QUAD</code> (quadratic number)	24
4.5.8 Type <code>t_POLMOD</code> (polmod)	24
4.5.9 Type <code>t_POL</code> (polynomial)	24
4.5.10 Type <code>t_SER</code> (power series)	25
4.5.11 Type <code>t_RFRAC</code> (rational function)	25
4.5.12 Type <code>t_QFR</code> (indefinite binary quadratic form)	25
4.5.13 Type <code>t_QFI</code> (definite binary quadratic form)	25
4.5.14 Type <code>t_VEC</code> and <code>t_COL</code> (vector)	25
4.5.15 Type <code>t_MAT</code> (matrix)	25
4.5.16 Type <code>t_VECSMALL</code> (vector of small integers)	25
4.5.17 Type <code>t_LIST</code> (list)	26
4.5.18 Type <code>t_STR</code> (character string)	26
4.6 PARI variables	26
4.6.1 Multivariate objects	26
4.6.2 Creating variables	27
4.7 Input and output	28
4.7.1 Input	28
4.7.2 Output	29
4.7.3 Errors	30
4.7.4 Debugging output	31
4.7.5 Timers and timing output	32
4.8 A complete program	32

4.9 Adding functions to PARI	34
4.9.1 Nota Bene	34
4.9.2 The calling interface from gp , parser codes	34
4.9.3 Coding guidelines	36
4.9.4 Integration with gp as a shared module	36
4.9.5 Integration the hard way	37
4.9.6 Example	37
Chapter 5: Technical Reference Guide for Low-Level Functions	39
5.1 Initializing the library	39
5.1.1 General purpose	39
5.1.2 Technical functions	39
5.1.3 Notions specific to the GP interpreter	40
5.2 Handling GENs	41
5.2.1 Length conversions	41
5.2.2 Read type-dependent information	41
5.2.3 Eval type-dependent information	42
5.2.4 Set type-dependent information	43
5.2.5 Type groups	43
5.2.6 Accessors and components	44
5.3 Handling the PARI stack	44
5.3.1 Allocating memory on the stack	44
5.3.2 Garbage collection	45
5.3.3 Copies and clones	47
5.4 Level 0 kernel (operations on ulongs)	47
5.4.1 Micro-kernel	47
5.4.2 Modular kernel	48
5.5 Level 1 kernel (operations on longs , integers and reals)	49
5.5.1 Creation	49
5.5.2 Assignment	50
5.5.3 Copy	50
5.5.4 Conversions	50
5.5.5 Integer parts	51
5.5.6 Valuation and shift	51
5.5.7 Factorization	52
5.5.8 Generic unary operators	52
5.5.9 Comparison operators	53
5.5.10 Generic binary operators	53
5.5.11 Modulo to longs	55
5.5.12 Exact division and divisibility	55
5.5.13 Division with remainder	55
5.5.14 Square root and remainder	56
5.5.15 Pseudo-random integers	56
5.5.16 Modular operations	57
5.5.17 Miscellaneous functions	57
5.6 Level 2 kernel (modular arithmetic)	58
5.6.1 Naming scheme	58
5.6.2 ZX , ZV , ZM	59
5.6.3 FpX	60
5.6.4 FpXQ , Fq	63

5.6.5	FpXX	64
5.6.6	FpXQX, FqX	64
5.6.7	FpV, FpM, FqM	65
5.6.8	Flx	67
5.6.9	Flxq	68
5.6.10	FlxX	68
5.6.11	FlxqX	69
5.6.12	Flv, Flm	69
5.6.13	FlxqV, FlxqM	69
5.6.14	QX	69
5.6.15	RgX	69
5.6.16	Conversions involving single precision objects	71
5.7	Operations on general PARI objects	73
5.7.1	Assignment	73
5.7.2	Conversions	73
5.7.3	Clean Constructors	75
5.7.4	Unclean Constructors	76
5.7.5	Integer parts	77
5.7.6	Valuation and shift	77
5.7.7	Comparison operators	77
5.7.8	Generic unary operators	78
5.7.9	Divisibility, Euclidean division	78
5.7.10	GCD, content and primitive part	79
5.7.11	Generic binary operators	80
5.7.12	Miscellaneous functions	80
5.8	Further type specific functions	80
5.8.1	Vectors and Matrices	80
5.8.2	Low-level vectors and columns functions	81
5.8.3	Function to handle t_VECSMALL	82
5.8.4	Functions to handle bits-vectors	83
5.8.5	Functions to handle vectors of t_VECSMALL	83
Appendix A: A Sample program and Makefile		85
Appendix B: Summary of Available Constants		87
Index		89

Chapter 4:

Programming PARI in Library Mode

The *User's Guide to Pari/GP* gives in three chapters a general presentation of the system, of the `gp` calculator, and detailed explanation of high level PARI routines available through the calculator. The present manual assumes general familiarity with the contents of these chapters and the basics of ANSI C programming, and focuses on the usage of the PARI library. In this chapter, we introduce the general concepts of PARI programming and describe useful general purpose functions. Chapter 5 describes all available public low-level functions.

4.1 Introduction: initializations, universal objects.

To use PARI in library mode, you must write a C program and link it to the PARI library. See the installation guide or the Appendix to the *User's Guide to Pari/GP* on how to create and install the library and include files. A sample Makefile is presented in Appendix A, and a more elaborate one in `examples/Makefile`. 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 with the directive

```
#include <pari.h>
```

In particular, this header defines the fundamental type for all PARI objects: the type **GEN**, which is simply a pointer to `long`.

Before any PARI routine is called, one must initialize the system, and in particular the PARI stack which is both a scratchboard and a repository for computed objects. This is done with a call to the function

```
void pari_init(size_t size, ulong maxprime)
```

The first argument is the number of bytes given to PARI to work with, and the second is the upper limit on a precomputed prime number table; `size` should not reasonably be taken below 500000 but you may set `maxprime = 0`, although the system still needs to precompute all primes up to about 2^{16} .

We have now at our disposal:

- a PARI *stack* containing nothing. It is a big connected chunk of `size` bytes of memory. All your computations take place here. In large computations, unwanted intermediate results quickly clutter up memory 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. PARI takes 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) you should use them.

- the following *universal objects* (by definition, objects which do not belong to the stack): the integers 0, 1, -1 and 2 (respectively called `gen_0`, `gen_1`, `gen_m1` and `gen_2`), 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 (`pol_x[v]`), and the polynomials 1_v (`pol_1[v]`). Here, x_v is the name of variable number v , where $0 \leq v \leq \text{MAXVARN}$. Both `pol_1` and `pol_x` are arrays of GENs, 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 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 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 B.

- 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 get a fatal error when running your program.

4.2 Important technical notes.

4.2.1 Types.

Although PARI objects all have the C type `GEN`, we will freely use the word **type** to refer to PARI dynamic subtypes: `t_INT`, `t_REAL`, etc. The declaration

```
GEN x;
```

declares a C variable of type `GEN`, but its “value” will be said to have type `t_INT`, `t_REAL`, etc. The meaning should always be clear from the context.

4.2.2 Type recursivity.

Conceptually, most PARI types are recursive. But the `GEN` type is a pointer to `long`, not to `GEN`. So special macros must be used to access `GEN`'s components. The simplest one is `gel(V, i)`, where **el** stands for **e**lement, to access component number i of the `GEN` V . This is a valid `lvalue` (may be put on the left side of an assignment), and the following two constructions are exceedingly frequent

```
gel(V, i) = x;
x = gel(V, i);
```

where x and V are GENs. This macro accesses and modifies directly the components of V and do not create a copy of the coefficient, contrary to all the library *functions*.

More generally, to retrieve the values of elements of lists of ... of lists of vectors we have the `gmael` macros (for **m**ultidimensional **a**rray **e**lement). The syntax is `gmael n (V, a1, ..., a n)`, where V is a `GEN`, the a_i are indexes, and n is an integer between 1 and 5. This stands for $x[a_1][a_2] \dots [a_n]$, and returns a `GEN`. The macros `gel` (resp. `gmael`) are synonyms for `gmael1` (resp. `gmael12`).

Finally, the macro `gcoeff(M, i, j)` has exactly the meaning of $M[i, j]$ in GP when M is a matrix. Note that due to the implementation of `t_MATs` as horizontal lists of vertical vectors, `gcoeff(x, y)` is actually equivalent to `gmael(y, x)`. One should use `gcoeff` in matrix context, and `gmael` otherwise.

4.2.3 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 GENs, 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 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 GENs of type `t_INT` (this is not checked).

`GEN addrr(GEN x, GEN y)`: here x and y are GENs of type `t_REAL` (this is not checked).

There also exist functions `addir`, `addri`, `mpadd` (whose two arguments can be of type `t_INT` or `t_REAL`), `addis` (to add a `t_INT` and a `long`) and so on.

All these specialized functions are of course more efficient than the general purpose ones, but note the hidden danger here: the types of the objects involved, if they are themselves results of a previous computation, are not completely predetermined. For instance the multiplication of a `t_REAL` by a `t_INT` *usually* gives a `t_REAL` result, 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 `t_REAL` argument, you are in trouble.

The names are self-explanatory once you know that **i** stands for a `t_INT`, **r** for a `t_REAL`, **mp** for **i** or **r**, **s** for a signed C long integer, **u** for an unsigned C long integer; finally the suffix **z** 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 these low-level functions.

4.2.4 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, 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 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
<code>BITS_IN_LONG</code>	64	32
<code>LONG_IS_64BIT</code>	defined	undefined
<code>DEFAULTPREC</code>	3	4 (≈ 19 decimal digits, see formula below)
<code>MEDDEFAULTPREC</code>	4	6 (≈ 38 decimal digits)
<code>BIGDEFAULTPREC</code>	5	8 (≈ 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

$$\text{bit_accuracy}(\text{prec}) = (\text{prec} - 2) * \text{BITS_IN_LONG}$$

(this is actually the definition of a macro). The corresponding accuracy expressed in decimal digits would be

$$\text{bit_accuracy}(\text{prec}) * \log(2) / \log(10).$$

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 Garbage collection.

4.3.1 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 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

$$(\text{top} - \text{bot}) / \text{sizeof}(\text{long})$$

is (roughly) equal to the `size` argument of `pari_init`. The PARI stack also has a “current stack pointer” called `avma`, which stands for **a**vaila**b**le **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* need to clean up the stack from time to time, when intermediate objects are not 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 do their own garbage collecting, which means that whenever a documented function from the library returns, only its result(s) 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 (e.g. you call few PARI routines, or most of them return `long` integers), then you do not need to do any garbage collecting. This is probably 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. This is equivalent to the above where the initial `av` is computed from `z`.

- 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 simpler 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

```

      bot          avma  lbot          ltop    top
End of stack |-----[+++++[---/--/--/--/--|++++++| Start
              free memory          garbage
```

becomes:

```

      bot          avma  ltop    top
End of stack |-----[+++++[++++++| Start
              free memory
```

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 is suitably updated,
- larger than or equal to `ltop`, in which case it does not change, or
- between `lbot` and `ltop` in which case `gerepile` raises an error (“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 updated address, as in 2). If not, the original address is still valid, and is returned!

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 the objects in the translated zone change after a call to `gerepile`, so you must not access any pointer which previously pointed into the zone below `ltop`. If you need to recover more than one object, use one of the `gerepilemany` functions below.

As a consequence of the preceding explanation, if a 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 GENs created by documented PARI functions are guaranteed to satisfy this. This is because the `gerepile` function knows only about *two connected zones*: the garbage that is erased (between `lbot` and `ltop`) and the significant pointers that are copied and updated. If there is garbage interspersed with your objects, disaster occurs when we try to update them and consider the corresponding “pointers”. In most cases of course the said garbage is in fact a bunch of other GENs, in which case we simply waste time copying and updating them for nothing. But be wary when you allow objects to become disconnected.

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 evaluated after or before the call to `anything()`: it depends on the compiler. If we are out of luck, it is *after* the call, so the result belongs to the garbage zone and the `gerepile` statement becomes equivalent to `avma = ltop`. Thus we 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! Unless mentioned otherwise, documented PARI functions guarantee this.

- Another variant (a special case of `gerepileall` below, where $n = 1$) is

```
GEN gerepilecopy(pari_sp ltop, GEN x))
```

which is functionally 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 `GENs` you want to preserve. It cleans up the most recent part of the stack (between `ltop` and `avma`), updating all the `GENs` 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), hence 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 `GENs` 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 `GENs` 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.

4.3.2 Examples.

4.3.2.1 gerepile

Let `x` and `y` be two preexisting PARI objects and suppose that we want to compute $x^2 + y^2$. This is done using the following program:

```
GEN p1 = gsqr(x);
GEN 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.)

It is not 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:

```
pari_sp ltop = avma; /* remember the current address of the top of the stack */
GEN p1 = gsqr(x);
```

```

GEN p2 = gsqr(y);
pari_sp lbot = avma; /* keep the address of the bottom of the garbage pile */
GEN z = gadd(p1, p2); /* z is now the last object on the stack */
z = gerepile(ltop, lbot, z);

```

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` is 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)));

```

Make sure you understand exactly what has happened before you go on (use the figure from the preceding section).

Remark on assignments and `gerepile`: When the tree structure and the size of the PARI objects which will appear in a computation are under control, one may allocate sufficiently large objects at the beginning, use assignment statements, then simply restore `avma`. Coming back to the above example, note that *if* we know that `x` and `y` are of type real fitting into `DEFAULTPREC` words, we can program without using `gerepile` at all:

```

z = cgetr(DEFAULTPREC); ltop = avma;
gaffect(gadd(gsqr(x), gsqr(y)), z);
avma = ltop;

```

This is often *slower* than a craftily used `gerepile` though, and certainly more cumbersome to use. As a rule, assignment statements should generally be avoided.

Variations on a theme: it is often necessary to do several `gerepiles` 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 $[x^2 + y, y^2 + x]$. 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);
gel(z, 1) = p2;
gel(z, 2) = 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);
gel(z, 1) = gadd(p1,y);
gel(z, 2) = gadd(p3,x); z = gerepile(ltop,lbot,z);

```

Finishing by `z = gerepileupto(ltop, z)` would be ok as well. Beware that

```

ltop = avma; p1 = gadd(gsqr(x), y); p3 = gadd(gsqr(y), x);

```

```

z = cgetg(3, t_VEC);
gel(z, 1) = p1;
gel(z, 2) = p3; z = gerepileupto(ltop,z); /* WRONG */

```

is a disaster since `p1` and `p3` are created before `z`, so the call to `gerepileupto` overwrites them, leaving `gel(z, 1)` and `gel(z, 2)` pointing at random data! On the other hand

```

ltop = avma; z = cgetg(3, t_VEC);
gel(z, 1) = gadd(gsqr(x), y);
gel(z, 2) = gadd(gsqr(y), x); z = gerepileupto(ltop,z); /* INEFFICIENT */

```

leaves the results of `gsqr(x)` and `gsqr(y)` on the stack (and lets `gerepileupto` update them for naught). Finally, the most elegant and efficient version (with respect to time and memory use) is as follows

```

z = cgetg(3, t_VEC);
ltop = avma; gel(z, 1) = gerepileupto(ltop, gadd(gsqr(x), y));
ltop = avma; gel(z, 2) = gerepileupto(ltop, gadd(gsqr(y), x));

```

which avoids updating the container `z` and cleans up its components individually, as soon as they are computed.

One last example. Let us compute the product of two complex numbers x and y , using the $3M$ method which requires 3 multiplications instead of the obvious 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 as follows:

```

ltop = avma;
p1 = gmul(gel(x,1), gel(y,1));
p2 = gmul(gel(x,2), gel(y,2));
p3 = gmul(gadd(gel(x,1), gel(x,2)), gadd(gel(y,1), gel(y,2)));
p4 = gadd(p1,p2);
lbot = avma; z = cgetg(3, t_COMPLEX);
gel(z, 1) = gsub(p1,p2);
gel(z, 2) = gsub(p3,p4); z = gerepile(ltop,lbot,z);

```

Exercise. Write a function which multiplies a matrix by a column vector. Hint: start with a `cgetg` of the result, and use `gerepile` whenever a coefficient of the result vector is computed. You can look at the answer in `src/basemath/gen1.c:MC_mul()`.

4.3.2.2 gerepileall

Let us now see why we may need the `gerepileall` variants. Although it is not an infrequent occurrence, we do 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 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 is cleaned, the pointers fixed up, but we have lost the address of `z2`. This is where we need the `gerepileall` function:

```

gerepileall(ltop, 2, &z1, &z2)

```

copies `z1` and `z2` to new locations, cleans the stack from `ltop` to the old `avma`, and updates the pointers `z1` and `z2`. 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 is usually faster since we do not need the initial copy (on the other hand, it is less cache friendly).

A most 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 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 $n = 2$).

4.3.3 Comments.

First, `gerepile` has turned out to be a flexible and fast garbage collector for number-theoretic computations, which compares favorably with more sophisticated methods used in other systems. Our benchmarks indicate that the price paid for using `gerepile` and `gerepile`-related copies, when properly used, is usually less than 1 percent of the total running time, which is quite acceptable!

Second, it is of course harder on the programmer, and quite error-prone if you do not stick to a consistent PARI programming style. If all seems lost, just use `gerepilecopy` (or `gerepileall`) to fix up the stack for you. You can always optimize later when you have sorted out exactly which routines are crucial and what objects need to be preserved and their usual sizes.

If you followed us this far, congratulations, and rejoice: the rest is much easier.

4.4 Creation of PARI objects, assignments, conversions.

4.4.1 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” is printed, and an error raised. Otherwise, it sets the type and length of the PARI object. In effect, it fills 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, which 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_COMPLEX`, `t_POLMOD`, `t_RFRAC`, 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 create efficiently a constant object, or to prepare for later assignments (see Section 4.4.2). 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. `t_INT` or `t_REAL`,

```
GEN cgeti(long length)
```

```
GEN cgetr(long length)
```

should be used instead of `cgetg(length, t_INT)` and `cgetg(length, t_REAL)` respectively. Finally

```
GEN cgetc(long prec)
```

creates a `t_COMPLEX` whose real and imaginary part are `t_REALs` allocated by `cgetr(prec)`.

Examples: 1) Both `z = cgeti(DEFAULTPREC)` and `cgetg(DEFAULTPREC, t_INT)` create a `t_INT` 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 is *not* 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);
gel(z, 1) = cgetr(MEDDEFAULTPREC);
gel(z, 2) = cgetr(MEDDEFAULTPREC);
```

or simply `z = cgetc(MEDDEFAULTPREC)`.

3) To create a matrix object for 4×3 matrices:

```
z = cgetg(4, t_MAT);
for(i=1; i<4; i++) gel(z, i) = cgetg(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) creates 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 mkintn(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 **sizeof(long)** is irrelevant, the behaviour is also as above on 64-bit machines).

```
mkintn(3, a2, a1, a0);
```

returns $a_2 2^{64} + a_1 2^{32} + a_0$.

GEN mkpoln(long *n*, ...) Returns the **t_POL** whose *n* coefficients (**GEN**) follow, in order of decreasing degree.

```
mkpoln(3, gen_1, gen_2, gen_0);
```

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, hence *one more* than the degree.

GEN mkvecn(long *n*, ...) returns the **t_VEC** whose *n* coefficients (**GEN**) follow.

GEN mkcoln(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 their arguments, nor do they produce an object a priori suitable for **gerepileupto**. For instance

```
/* gerepile-safe: components are universal objects */
z = mkvecn(3, gen_1, gen_0, gen_2);

/* not OK for gerepileupto: stoi(3) creates component before root */
z = mkvecn(3, stoi(3), gen_0, gen_2);

/* NO! First vector component x is destroyed */
x = gclone(gen_1);
z = mkvecn(3, x, gen_0, gen_2);
gunclone(x);
```

The following function is also available as a special case of **mkintn**:

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.

4.4.2 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) also changes the other one, which is usually not desirable.

Very important note: Using the functions described in this paragraph is inefficient and often awkward: one of the **gerepile** functions (see Section 4.3) should be preferred. See the paragraph end for one exception 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 a **t_INT** into a **t_REAL**, 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**. For instance, let **y** be the zero integer with length equal to 2; then **y** is too small to accommodate any non-zero **t_INT**. In general common sense tells you what is possible, keeping in mind the PARI philosophy which says that if it makes sense it is valid. 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 4.2.3) implicitly use this function. In fact what they exactly do is record **avma** (see Section 4.3), 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. There is one exception: for simple objects (e.g. leaves) whose size is controlled, they can be easier to use than **gerepile**, and about as efficient.

Coercion. It is often useful to coerce an inexact object to a given precision. For instance at the beginning of a routine where precision can be kept to a minimum; otherwise the precision of the input is used in all subsequent computations, which is inefficient if the latter is known to thousands of digits. One may use the **gaffect** function for this, but it is easier and more efficient to call

GEN gtofp(GEN x, long prec) converts the complex number **x** (**t_INT**, **t_REAL**, **t_FRAC**, **t_QUAD** or **t_COMPLEX**) to either a **t_REAL** or **t_COMPLEX** whose components are **t_REAL** of length **prec**.

4.4.3 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**. Its syntax is:

```
GEN gcopy(GEN x)
```

and the effect is to create a new copy of **x** on the PARI stack.

Sometimes, on the contrary, a quick copy of the skeleton of **x** is enough, leaving pointers to the original data in **x** for the sake of speed instead of making a full recursive copy. Use **GEN shallowcopy(GEN x)** for this. Note that the result is not suitable for **gerepileupto** !

Make sure at this point that you understand the difference between **y = x**, **y = gcopy(x)**, **y = shallowcopy(x)** and **gaffect(x,y)**.

4.4.4 Clones. Sometimes, it is more efficient to create a *persistent* copy of a PARI object. This is not created on the stack but on the heap, hence unaffected by `gerepile` and friends. The function which does this is called `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!

4.4.5 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 t_INT.
```

```
GEN dbltor(double s): C double to 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 are 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).
```

4.5 Implementation of the PARI types.

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` is treated as the C pointer it is, on the other, as PARI's handle on some mathematical entity, so we will shamelessly write `z ≠ 0` to indicate that the *value* thus represented is nonzero (in which case the *pointer* `z` is certainly 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 depends 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 or not (see Section 4.4.4). 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).

`long isclone(GEN z)` is `z` a clone?

`void setisclone(GEN z)` sets the *clone* bit.

`void unsetisclone(GEN z)` unsets the *clone* bit.

Remark. The clone bit is there so that `gunclone` can check it is deleting an object which was allocated by `gclone`. Miscellaneous vector entries are often cloned by `gp` so that a GP statement like `v[1] = x` does not involve copying the whole of `v`: the component `v[1]` is deleted if its clone bit is set, and is replaced by a clone of `x`. Don't set/unset yourself the clone bit unless you know what you are doing: in particular *never* set the clone bit of a vector component when the said vector is scheduled to be uncloned. Hackish code may abuse the clone bit to tag objects for reasons unrelated to the above instead of using proper data structures. Don't do that.

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])` is accepted by your compiler, as well as the more proper `typ(gel(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.

Some types have a second codeword, used differently by each type, 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 most significant mantissa longword 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]`.

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. Now assume that $z \neq 0$, and let

$$|z| = \sum_{i=0}^n z_i B^i, \quad \text{where } z_n \neq 0 \text{ and } B = 2^{\text{BITS_IN_LONG}}.$$

With these notations, n is `lgefint(z) - 3`, and the mantissa of `z` may be manipulated via the following interface:

`GEN int_MSW(GEN z)` returns a pointer to the most significant word of `z`, z_n .

`GEN int_LSW(GEN z)` returns a pointer to the least significant word of `z`, z_0 .

GEN int_W(GEN z, long i) returns the i -th significant word of z , z_i . Accessing the i -th significant word for $i > n$ yields unpredictable results.

GEN int_precW(GEN z) returns the previous (less significant) word of z , z_{i-1} assuming z points to z_i .

GEN int_nextW(GEN z) returns the next (more significant) word of z , z_{i+1} assuming z points to z_i .

Unnormalized integers, such that z_n is possibly 0, are explicitly forbidden. To enforce this, one may write an arbitrary mantissa then call

```
void int_normalize(GEN z, long known0)
```

normalizes in place a non-negative integer (such that z_n is possibly 0), assuming at least the first **known0** words are zero.

For instance a binary **and** could be implemented in the following way:

```
GEN AND(GEN x, GEN y) {
    long i, lx, ly, lout;
    long *xp, *yp, *outp; /* mantissa pointers */
    GEN out;

    if (!signe(x) || !signe(y)) return gen_0;
    lx = lgefint(x); xp = int_LSW(x);
    ly = lgefint(y); yp = int_LSW(y); lout = min(lx,ly); /* > 2 */
    out = cgeti(lout); out[1] = evalsigne(1) | evallgefint(lout);
    outp = int_LSW(out);
    for (i=2; i < lout; i++)
    {
        *outp = (*xp) & (*yp);
        outp = int_nextW(outp);
        xp = int_nextW(xp);
        yp = int_nextW(yp);
    }
    if ( !*int_MSW(out) ) out = int_normalize(out, 1);
    return out;
}
```

This low-level interface is mandatory in order to write portable code since PARI can be compiled using various multiprecision kernels, for instance the native one or GNU MP, with incompatible internal structures (for one thing, the mantissa is oriented in different directions).

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, $\text{mod}n(x) := \text{mod}n(|x|)$ 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 GENs, 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 a `t_INT`, and a binary exponent. This exponent is 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 ??.

`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 `z` 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], ..., z[lg(z) - 1]`.

More precisely, let m be the integer $(z[2], \dots, z[lg(z)-1])$ in base $2^{\text{BITS_IN_LONG}}$; here, `z[2]` is the most significant longword and is normalized, i.e. its most significant bit is 1. Then we have $M := m \cdot 2^{1-\text{bit_accuracy}(\lg(z))}$.

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`: `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 `t_INT`s, satisfying the inequality $0 \leq z[2] < z[1]$.

It is good practice to keep the modulus object on the heap, so that new `t_INTMOD`s 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` (rational number): `z[1]` points to the numerator n , and `z[2]` to the denominator d . Both must be of type `t_INT` such that $d \neq 0$, $n > 0$ and $(n, d) = 1$ (see `gred_frac2`).

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 (mostly `t_INT`, `t_REAL` and `t_FRAC`).

4.5.6 Type `t_PADIC` (p -adic numbers): this type has a second codeword `z[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 ??.

`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 at_INT representing the p -adic unit associated to z modulo $z[3]$ (and to zero if z is zero). To summarize, if $z \neq 0$, we have the equality:

$$z = p^{\text{valp}(z)} * (z[4] + O(z[3])), \quad \text{where} \quad z[3] = O(p^{\text{precp}(z)}).$$

4.5.7 Type t_QUAD (quadratic number): $z[1]$ points to the canonical polynomial P defining the quadratic field (as output by `quadpoly`), $z[2]$ to the “real part” and $z[3]$ to the “imaginary part”. The latter are of type t_INT , t_FRAC , t_INTMOD , or t_PADIC and are to be taken as the coefficients of z with respect to the canonical basis $(1, X)$ or $\mathbf{Q}[X]/(P(X))$, see Section ???. Exact complex numbers may be implemented as quadratics, but t_COMPLEX is in general more versatile (t_REAL components are allowed) and more efficient.

Operations involving a t_QUAD and t_COMPLEX are implemented by converting the t_QUAD to a t_REAL (or t_COMPLEX with t_REAL components) to the accuracy of the t_COMPLEX . As a consequence, operations between t_QUAD and *exact* t_COMPLEX s are not allowed.

4.5.8 Type t_POLMOD (polmod): as for t_INTMOD s, $z[1]$ 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, satisfying the inequality $\deg z[2] < \deg z[1]$. However, $z[2]$ is allowed to be a simplification of such a polynomial, e.g. a scalar. This is tricky considering the hierarchical structure of the variables; in particular, a polynomial in variable of *lesser* priority (see Section ??) than the modulus variable is valid, since it is considered as the constant term of a polynomial of degree 0 in the correct variable. On the other hand a variable of *greater* priority is not acceptable; see Section ?? for the problems which may arise.

4.5.9 Type t_POL (polynomial): this type has a second codeword. It contains a “*sign*”: 0 if the polynomial is equal to 0, and 1 if not (see however the important remark below) and a *variable number* (e.g. 0 for x , 1 for y , etc...).

These data can be handled with the following macros: **signe** and **setsigne** as for t_INT and t_REAL ,

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 ???. 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`, which has lower priority than any other variable number.

The components $z[2], z[3], \dots, z[\text{lg}(z)-1]$ point to the coefficients of the polynomial *in ascending order*, with $z[2]$ being the constant term and so on.

For an object of type t_POL , **leading_term**, **constant_term**, **degpol** return a pointer to the leading term (with respect to the main variable of course), constant term, and degree of the polynomial (with the convention $\deg(0) = -1$). Applied to any other type, the result is unspecified. Note that no copy is made on the pari stack so the returned value is not safe for a basic **gerepile** call. Note that $\text{degpol}(z) = \text{lg}(z) - 3$.

The leading term is not allowed to be an exact 0 (*unnormalized polynomial*). To prevent this, one may use

GEN `normalizepol`(GEN `x`) applied to an unnormalized `t_POL` `x` (with all coefficients correctly set except that `leading_term(x)` might be zero), normalizes `x` correctly in place and returns `x`. For internal use.

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 length may be greater than 2, meaning that all the coefficients of the polynomial are equal to zero, but the leading term `z[lg(z)-1]` is an inexact zero. More precisely, `gcmp0(x)` is true for all coefficients `x` of the polynomial, an `isexactzero(x)` is false for the leading coefficient. The same remark applies to `t_SERs`.

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.

The coefficients `z[2]`, `z[3]`, ... `z[lg(z)-1]` point to the coefficients of `z` in ascending order. As for polynomials (see remark there), the sign of a `t_SER` is 0 if and only if the leading coefficient of the series is an inexact 0. (It cannot be an exact 0.)

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` (rational function): `z[1]` points to the numerator n , and `z[2]` on the denominator d . The denominator must be of type `t_POL`, with variable of higher priority than the numerator. The numerator n is not an exact 0 and $(n, d) = 1$ (see `gred_rfac2`).

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 are of type `t_INT`. `z[4]` is Shanks’s distance function, and must be of type `t_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 are of type `t_INT`.

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.

4.5.16 Type `t_VECSMALL` (vector of small integers): `z[1]`, `z[2]`, ... `z[lg(z)-1]` are ordinary signed long integers. This type is used instead of a `t_VEC` of `t_INTs` for efficiency reasons, for instance to implement efficiently permutations, polynomial arithmetic and linear algebra over small finite fields, etc.

The next two types were introduced for specific `gp` use, and you are better off using the standard malloc’ed C constructs when programming in library mode. We quote them for completeness, advising you not to use them:

4.5.17 Type `t_LIST` (list): This one has a second codeword which contains an effective length (handled through `lgeflist` / `setlgeflist`). `z[2], ..., z[lgeflist(z)-1]` contain the components of the list.

4.5.18 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 `HIGHEXPBIT` (for `t_REAL`) or `HIGHVALPBIT` (for `t_PADIC` and `t_SER`). Of course, this is encapsulated by the exponent/valuation-handling macros and needs 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 ???. 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`. Relative priorities may be ascertained using

```
int varncmp(long v, long w)
```

which is > 0 , $= 0$, < 0 whenever v has lower, resp. same, resp. higher priority than w .

The way an object is 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 has lower priority than any valid variable number. The variable number of a recursive type which is not a polynomial or power series is the variable number with highest priority among 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 needs not worry too much since the interpreter defines the variables as it sees them* and do the right thing with the polynomials produced (however, have a look at the remark in Section ??).

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 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 lower priority, even though PARI cannot prevent you from doing it; see Section ?? for a discussion of possible problems in a similar situation.

* 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 has initially maximal priority (and variable number 0).

4.6.2 Creating variables A basic difficulty is to “create” a variable. As we have seen in Section 4.1, a number of objects is associated to variable number v . Here is the complete list: `pol_1[v]` and `pol_x[v]`, which you can use in library mode and which represent, respectively, the monic monomials of degrees 0 and 1 in v ; `varetries[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 abides). 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 = pol_x[v];
```

This function raises an error if `s` is already registered as a function name.

Caveat: you can use `gp_read_str` (see Section 4.7.1) to execute a GP command and create GP variables on the fly as needed:

```
GEN gy = gp_read_str("'y"); /* returns pol_x[v], for some v */
long v = varn(gy);
```

But please note the quote `'y` in the above. Using `gp_read_str("y")` might work, but is dangerous, especially when programming functions to be used under `gp`. The latter reads the value of `y`, as *currently* known by the `gp` interpreter, possibly creating it in the process. But if `y` has been modified by previous `gp` commands (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 `gen_1`).

Technical remark If you are rewriting the `gp` interpreter, you may use the lower level

```
entree * fetch_named_var(char *s)
```

which returns an `entree*` suitable for inclusion in the interpreter hashlists of symbols.

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 *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 `pol_1[v]` and `pol_x[v]`. The variables created in this way have no identifier assigned to them though, and they is printed as `#<number>`, except for `MAXVARN` which

is 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` raises 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 are not 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 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 one powerful high level function which enables you to input your objects as if you were under `gp`. In fact, it *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`. It has the following syntax:

```
GEN gp_read_str(char *s)
```

In fact this function starts by *filtering* out all spaces and comments in the input string. They it calls the underlying basic function, the GP parser proper: `GEN gp_read_str(char *s)`, which is slightly faster but which you probably do not need.

To read a `GEN` from a file, you can use the simpler interface

GEN gp_read_stream(FILE *file)

which reads a character string of arbitrary length from the stream `file` (up to the first complete expression sequence), applies `gp_read_str` to it, and returns the resulting **GEN**. This way, you do not have to worry about allocating buffers to hold the string. To interactively input an expression, use `gp_read_stream(stdin)`.

Finally, you can read in a whole file, as in GP's `read` statement

GEN gp_read_file(char *name)

As usual, the return value is that of the last non-empty expression evaluated. Note that `gp`'s metacommands are not recognized.

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 must satisfy `gerepileupto` assumptions (see Section 4.3).

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 ??), plus three extra ones, respectively devoted to \TeX output, string output, and 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 ??. 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 one does not need that much flexibility, and gets 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)` enables 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** GP function, and gives a \TeX output of the result. It is obtained by using:

```
void texe(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)`, which 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` metacommand `\x` using the function

`void voir(GEN obj, long nb)`, which only outputs 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 are printed. This last type of output is only 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 is *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 `pari_err`. The basic syntax is

```
pari_err(talker, "error message");
```

This prints 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

```
pari_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 pari_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, `pari_err(typeer,"extgcd")` prints the message:

```
*** incorrect type in extgcd.
```

To issue a warning, use

`void pari_warn(warnerr,...)` In that case, of course, we do *not* abort the computation, just print the requested message and go on. The basic example is

```
pari_warn(warner, "Strategy 1 failed. Trying strategy 2")
```

which is the exact equivalent of `pari_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).

The valid *warning* keywords are `warner` (general), `warnprec` (increasing precision), `warnmem` (garbage collecting) and `warnfile` (error in file operation), used as follows:

```
pari_warn(warnprec, "bnfinit", newprec);
pari_warn(warnmem, "bnfinit");
pari_warn(warnfile, "close", "log"); /* error when closing "log" */
```

4.7.4 Debugging output.

The global variables `DEBUGLEVEL` and `DEBUGMEM` (corresponding to the default `debug` and `debugmem`, see Section ??) 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 ??).

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:

```
pari_err(talker, "x[%d] = %Z is not invertible!", i, x[i])
```

since the `pari_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 data type, `pari_timer`. The functions

```
void TIMERstart(pari_timer *T)
long TIMER(pari_timer *T)
long msgTIMER(pari_timer *T, char *format,...)
```

provide an equivalent to `timer` and `msgtimer`, 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;
TIMERstart(&T); /* initialize timer */
...
printf("Total time: %ld\n", TIMER(&T));
```

or

```
pari_timer T;
TIMERstart(&T);
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 study a detailed example. We want to write a program which computes the gcd of two integers, together with the Bezout coefficients. We shall use the standard quadratic algorithm which is not optimal but is not too far from the one used in the PARI function `bezout`.

Let x, y two integers and initially $\begin{pmatrix} s_x & s_y \\ t_x & t_y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, so that

$$\begin{pmatrix} s_x & s_y \\ t_x & t_y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}.$$

To apply the ordinary Euclidean algorithm to the right hand side, multiply the system from the left by $\begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$, with $q = \text{floor}(x/y)$. Iterate until $y = 0$ in the right hand side, then the first line of the system reads

$$s_x x + s_y y = \gcd(x, y).$$

In practice, there is no need to update s_y and t_y since $\gcd(x, y)$ and s_x are enough to recover s_y . The following program is now straightforward. A couple of new functions appear in there, whose description can be found in the technical reference manual in Chapter 5.

```
#include <pari/pari.h>
/*
GP;install("extgcd", "GG&&", "gcdex", "./libextgcd.so");
*/
/* return d = gcd(a,b), sets u, v such that au + bv = gcd(a,b) */
GEN
extgcd(GEN A, GEN B, GEN *U, GEN *V)
{
    pari_sp av = avma;
    GEN ux = gen_1, vx = gen_0, a = A, b = B;
    if (typ(a) != t_INT || typ(b) != t_INT) pari_err(typeer, "extgcd");
    if (signe(a) < 0) { a = negi(a); ux = negi(ux); }
    while (!gcmp0(b))
    {
        GEN r, q = dvmdii(a, b, &r), v = vx;
        vx = subii(ux, mulii(q, vx));
        ux = v;
        a = b; b = r;
    }
    *U = ux;
    *V = diviixact( subii(a, mulii(A,ux)), B );
    gerepileall(av, 3, &a, U, V); return a;
}

int
main()
{
    GEN x, y, d, u, v;
    pari_init(1000000,2);
    printf("x = "); x = gp_read_stream(stdin);
    printf("y = "); y = gp_read_stream(stdin);
    d = extgcd(x, y, &u, &v);
    pariprintf("gcd = %Z\nu = %Z\nv = %Z\n", d,u,v);
    return 0;
}
```

Note that, for simplicity, the inner loop does not include any garbage collection, hence memory use is quadratic in the size of the inputs instead of linear.

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 your improvements are incorporated into the next release.

We classify changes to PARI into four rough classes, where changes of the first three types are almost certain to be accepted. The first type includes all improvements to the documentation, in a broad sense. This includes correcting typos or inaccuracies 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 fine points unduly omitted in the present manual.

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 <code>int</code> with <code>long</code>)
S	symbol (i.e. GP identifier name). Function expects a <code>*entree</code>
V	variable (as <code>S</code> , but rejects symbols associated to functions)
n	variable, expects a variable number (a <code>long</code> , not an <code>*entree</code>)
I	string containing a sequence of GP statements (a <i>seq</i>), to be processed by <code>gp_read_str</code> (useful for control statements)
E	string containing a <i>single</i> GP statement (an <i>expr</i>), to be processed by <code>readexpr</code>
r	raw input (treated as a string without quotes). Quoted args are copied as strings Stops at first unquoted <code>'</code> ' or <code>,</code> . Special chars can be quoted using <code>'\'</code> Example: <code>aa"b\n)"c</code> yields the string <code>"aab\n)c"</code>
s	expanded string. Example: <code>Pi"x"2</code> yields <code>"3.142x2"</code> Unquoted components can be of any PARI type (converted following current output format)
- Optional arguments:
 - s*** any number of strings, possibly 0 (see `s`)
 - Dxxx** 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 ??), 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 sends a `GEN *g = {&a, &b, &c, NULL}`, where *a*, *b*, *c* are `GENs` of type `t_STR` (content `"xx"`), `t_INT` (equal to 1) and `t_STR` (content `"yy"`).

The format to indicate a default value (atom starts with a `D`) is `"Dvalue,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 is converted to a `long`, and is 0 by default”. So if the user-given argument reads `1 + 3` at this point, `4L` is sent to the function; and `0L` if the argument is omitted. The following special syntaxes are available:

<code>DG</code>	optional <code>GEN</code> , send <code>NULL</code> if argument omitted.
<code>D&</code>	optional <code>*GEN</code> , send <code>NULL</code> if argument omitted.
<code>DV</code>	optional <code>*entree</code> , send <code>NULL</code> if argument omitted.
<code>DI</code>	optional <code>*char</code> , send <code>NULL</code> if argument omitted.
<code>Dn</code>	optional variable number, <code>-1</code> if omitted.

- Automatic arguments:

<code>f</code>	Fake <code>*long</code> . C function requires a pointer but we do not use the resulting <code>long</code>
<code>p</code>	real precision (default <code>realprecision</code>)
<code>P</code>	series precision (default <code>seriesprecision</code> , global variable <code>preccl</code> for the library)

- Return type: `GEN` by default, otherwise the following can appear at the start of the code string:

<code>i</code>	return <code>int</code>
<code>l</code>	return <code>long</code>
<code>v</code>	return <code>void</code>

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 if 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:

<code>GEN name(GEN x, GEN y, long prec)</code>	----> <code>"GGp"</code>
<code>void name(GEN x, GEN y, long prec)</code>	----> <code>"vGGp"</code>
<code>void name(GEN x, long y, long prec)</code>	----> <code>"vGLp"</code>
<code>long name(GEN x)</code>	----> <code>"lG"</code>
<code>int name(long x)</code>	----> <code>"iL"</code>

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 `paridecl.h`.

Remark: If you need to implement complicated control statements (probably for some improved summation functions), you need to know about the **entree** type, which is not documented. Check the comment at the end of `language/init.c` and the source code in `language/sumiter.c`.

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, since otherwise successive calls to the function clutters the stack with unnecessary garbage, and stack overflow occurs 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 4.3).

If error messages or warnings are to be generated in your function, use **pari_err** and **pari_warn** respectively. Recall that **pari_err** does not return but ends with a **longjmp** statement. As well, instead of explicit **printf** / **fprintf** statements, use the following encapsulated variants:

void pariflush(): flush output stream.

void pariputc(char c): write character *c* to the output stream.

void pariputs(char *s): write *s* to the output stream.

void fprintferr(char *s): write *s* to the 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 want to 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 `extgcd` 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 figure out the correct parser code corresponding to the function prototype, as explained above (Section 4.9.2).

Now, assuming your Operating System is supported by `install`, write a GP script like the following:

```
install(libname, code, gpname, library)
addhelp(gpname, "some help text")
```

(see Section ?? and ??). 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 ??), and that's it. You can now 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, 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
```

(If the help text does not fit on a single line, continuation lines must start by a whitespace character.) A GP2C-related **Description** field is also available to improve the code GP2C generates when compiling scripts involving your function. See the GP2C documentation for details.

At this point you can recompile `gp`, which will first rebuild the functions database.

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 is in this case 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 describe all public low-level functions of the PARI library. These essentially include functions for handling all the PARI types. Higher level functions, such as arithmetic or transcendental functions, are described in Chapter 3 of the GP user's manual. A general introduction to the major concepts of PARI programming can be found in Chapter 4.

Many other undocumented functions can be found throughout the source code. These private functions are more efficient than the library wrappers, but sloppier on argument checking and damage control. Use them at your own risk!

Important advice: generic routines eventually call lower level functions. Optimize your algorithms first, not overhead and conversion costs between PARI routines. For generic operations, use generic routines first, don't waste time looking for the most specialized one available unless you identify a genuine bottleneck. The PARI source code is part of the documentation; look for inspiration there.

We let `BIL` abbreviate `BITS_IN_LONG`. The type `long` denotes a BIL-bit signed long integer. The type `ulong` is defined as `unsigned long`. The word *stack* always refer to the PARI stack, allocated through an initial `pari_init` call. Refer to Chapters 1–2 and 4 for general background.

5.1 Initializing the library.

The following functions enable you to start using the PARI functions in a program, and cleanup without exiting the whole program.

5.1.1 General purpose

`void pari_init(size_t size, ulong maxprime)` initialize the library, with a stack of `size` bytes and a prime table up to the maximum of `maxprime` and 2^{16} . Unless otherwise mentionned, no PARI function will function properly before such an initialization.

`void pari_close(void)` stop using the library (assuming it was initialized with `pari_init`) and frees all allocated objects.

5.1.2 Technical functions

`void pari_init_opts(size_t size, ulong maxprime, ulong opts)` as `pari_init`, more flexible. `opts` is a mask of flags among the following:

`INIT_JMPm`: install pari error handler. When an exception is raised, the program is terminated with `exit(1)`.

`INIT_SIGm`: install pari signal handler.

`INIT_DFTm`: initialize the `GP_DATA` environment structure. This one *must* be enabled once. If you close pari, then restart it, you need not reinitialize `GP_DATA`; if you do not, then old values are restored.

`void pari_close_opts(ulong init_opts)` as `pari_close`, for a library initialized with a mask of options using `pari_init_opts`. `opts` is a mask of flags among

`INIT_SIGm`: restore `SIG_DFL` default action for signals tampered with by `pari` signal handler.

`INIT_DFTm`: frees the `GP_DATA` environment structure.

`void pari_sig_init(void (*f)(int))` install the signal handler `f` (see `signal(2)`): the signals `SIGBUS`, `SIGFPE`, `SIGINT`, `SIGBREAK`, `SIGPIPE` and `SIGSEGV` are concerned.

5.1.3 Notions specific to the GP interpreter

An **entree** is the generic object associated to an identifier (a name) in GP's interpreter, be it a built-in or user function, or a variable. For a function, it has at least the following fields:

`char *name` : the name under which the interpreter knows us.

`ulong valence` : obsolete, set it to 1.

`void *value` : a pointer to the C function to call.

`long menu` : an integer from 1 to 11 (to which group of function help do we belong).

`char *code` : the prototype code.

`char *help` : the help text for the function.

A routine in GP is described to the analyzer by an **entree** structure. Built-in `pari` routines are grouped in *modules*, which are arrays of **entree** structs, the last of which satisfy `name = NULL` (sentinel).

There are currently six modules in GP: general functions (`functions_basic`), gp-specific functions (`functions_fp`), gp-specific highlevel functions (`functions_highlevel`), member functions, and two modules of obsolete functions. The function `pari_init` initializes the interpreter and declares all symbols in `functions_basic`. You may declare further functions on a case by case basis or as a whole module using

`void pari_add_function(entree *ep)` adds a single routine to the table of symbols in the interpreter. It assumes `pari_init` has been called.

`void pari_add_module(entree *mod)` adds all the routines in module `mod` to the table of symbols in the interpreter. It assumes `pari_init` has been called.

For instance, `gp` implements a number of private routines, which it adds to the default set via the call

```
pari_add_module(functions_gp);
pari_add_module(functions_highlevel);
```


5.2 Handling GENs.

Almost all these functions are either macros or inlined. Unless mentioned otherwise, they do not evaluate their arguments twice. Most of them are specific to a set of types, although no consistency checks are made: e.g. one may access the `sign` of a `t_PADIC`, but the result is meaningless.

5.2.1 Length conversions

`long ndec2nlong(long x)` converts a number of decimal digits to a number of words. Returns $1 + \text{floor}(x \times \text{BIL} \log_2 10)$.

`long ndec2prec(long x)` converts a number of decimal digits to a number of codewords. This is equal to $2 + \text{ndec2nlong}(x)$.

`long prec2ndec(long x)` converts a number of of codewords to a number of decimal digits.

`long nbits2nlong(long x)` converts a number of bits to a number of words. Returns the smallest word count containing x bits, i.e $\text{ceil}(x/\text{BIL})$.

`long nbits2prec(long x)` converts a number of bits to a number of codewords. This is equal to $2 + \text{nbits2nlong}(x)$.

`long nchar2nlong(long x)` converts a number of bytes to number of words. Returns the smallest word count containing x bytes, i.e $\text{ceil}(x/\text{sizeof}(\text{long}))$.

`long bit_accuracy(long x)` converts a `t_REAL` length into a number of significant bits. Returns $(x - 2)\text{BIL}$. The macro `bit_accuracy_mul(x,y)` computes the same thing multiplied by y .

5.2.2 Read type-dependent information

`long typ(GEN x)` returns the type number of x . The header files included through `pari.h` define symbolic constants for the GEN types: `t_INT` etc. Never use their actual numerical values. E.g to determine whether x is a `t_INT`, simply check

```
if (typ(x) == t_INT) { }
```

The types are internally ordered and this simplifies the implementation of commutative binary operations (e.g addition, gcd). Avoid using the ordering directly, as it may change in the future; use type grouping macros instead (Section 5.2.5).

`long lg(GEN x)` returns the length of x in BIL-bit words.

`long lgfint(GEN x)` returns the effective length of the `t_INT` x in BIL-bit words.

`long signe(GEN x)` returns the sign (-1 , 0 or 1) of x . Can be used for `t_INT`, `t_REAL`, `t_POL` and `t_SER` (for the last two types, only 0 or 1 are possible).

`long gsigne(GEN x)` same as `signe`, but also valid for `t_FRAC` (and marginally less efficient for the other types). Raise a type error if `typ(x)` is not among those three.

`long expi(GEN x)` returns the binary exponent of the real number equal to the `t_INT` x . This is a special case of `gexpo`.

`long expo(GEN x)` returns the binary exponent of the `t_REAL` x .

`long gexpo(GEN x)` same as `expo`, but also valid when x is not a `t_REAL` (returns the largest exponent found among the components of x). When x is an exact 0 , this returns `-HIGHEXPOBIT`, which is lower than any valid exponent.

long valp(GEN *x*) returns the p -adic valuation (for a **t_PADIC**) or X -adic valuation (for a **t_SER**, taken with respect to the main variable) of *x*.

long precp(GEN *x*) returns the precision of the **t_PADIC** *x*.

long varn(GEN *x*) returns the variable number of the **t_POL** or **t_SER** *x* (between 0 and **MAXVARN**).

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.

long degpol(GEN *x*) returns the degree of **t_POL** *x*, *assuming* its leading coefficient is non-zero (an exact 0 is impossible, but an inexact 0 is allowed). By convention the degree of an exact 0 polynomial is -1 . If the leading coefficient of *x* is 0, the result is undefined.

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).

int gprecision(GEN *x*) as **precision** for scalars; returns the lowest precision encountered among the components otherwise.

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*.

5.2.3 Eval type-dependent information. These routines convert type-dependant information to bitmask to fill the codewords of GEN objects (see Section 4.5). E.g for a **t_REAL** *z*:

```
z[1] = evalsigne(-1) | evalexpo(2)
```

Compatible components of a codeword for a given type can be OR-ed as above.

ulong evaltyp(long *x*) convert type *x* to bitmask (first codeword of all GENs)

long evallg(long *x*) convert length *x* to bitmask (first codeword of all GENs). Raise overflow error if *x* is so large that the corresponding length cannot be represented

long _evallg(long *x*) as **evallg** *without* the overflow check.

ulong evalvarn(long *x*) convert variable number *x* to bitmask (second codeword of **t_POL** and **t_SER**)

long evalsigne(long *x*) convert sign *x* (in $-1, 0, 1$) to bitmask (second codeword of **t_INT**, **t_REAL**, **t_POL**, **t_SER**)

long evalprecp(long *x*) convert p -adic (X -adic) precision *x* to bitmask (second codeword of **t_PADIC**, **t_SER**)

long evalvalp(long *x*) convert p -adic (X -adic) valuation *x* to bitmask (second codeword of **t_PADIC**, **t_SER**). Raise overflow error if *x* is so large that the corresponding valuation cannot be represented

long _evalvalp(long *x*) same as **evalvalp** *without* the overflow check.

long evalexpo(long *x*) convert exponent *x* to bitmask (second codeword of **t_REAL**). Raise overflow error if *x* is so large that the corresponding exponent cannot be represented

long _evalexpo(long *x*) same as **evalexpo** *without* the overflow check.

`long evallgefint(long x)` convert effective length `x` to bitmask (second codeword `t_INT`). This should be less or equal than the length of the `t_INT`, hence there is no overflow check for the effective length.

`long evallgeflist(long x)` convert effective length `x` to bitmask (second codeword `t_LIST`). This should be less or equal than the length of the `t_LIST`, hence there is no overflow check for the effective length.

5.2.4 Set type-dependent information. Use these macros with extreme care since usually the corresponding information is set otherwise, and the components and further codeword fields (which are left unchanged) may not be compatible with the new information.

`void settyp(GEN x, long s)` sets the type number of `x` to `s`.

`void setlg(GEN x, long s)` sets the length of `x` to `s`. This is an efficient way of truncating vectors, matrices or polynomials.

`void setlgefint(GEN x, long s)` sets the effective length of the `t_INT` `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 a `t_INT` or `t_REAL`, `s` must be equal to `-1`, `0` or `1`, and if `x` is a `t_POL` or `t_SER`, `s` must be equal to `0` or `1`.

`void setexpo(GEN x, long s)` sets the binary exponent of the `t_REAL` `x` to `s`. The 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 `t_PADIC` or a `t_SER`, respectively.

`void setprec(GEN x, long s)` sets the p -adic precision of the `t_PADIC` `x` to `s`.

`void setvarn(GEN x, long s)` sets the variable number of the `t_POL` or `t_SER` `x` to `s` (where $0 \leq s \leq \text{MAXVARN}$).

5.2.5 Type groups . In the following macros, `t` denotes the type of a `GEN`. Some of these macros may evaluate their argument twice. Always use them as in

```
long tx = typ(x);
if (is_intreal_t(tx)) { }
```

`int is_recursive_t(long t)` true iff `t` is a recursive type (the recursive types are `t_INT`, `t_REAL`, `t_STR` or `t_VECSMALL`).

`int is_intreal_t(long t)` true iff `t` is `t_INT` or `t_REAL`.

`int is_rational_t(long t)` true iff `t` is `t_INT` or `t_FRAC`.

`int is_vec_t(long t)` true iff `t` is `t_VEC` or `t_COL`.

`int is_matvec_t(long t)` true iff `t` is `t_MAT`, `t_VEC` or `t_COL`.

`int is_scalar_t(long t)` true iff `t` is a scalar, i.e a `t_INT`, `t_REAL`, `t_INTMOD`, `t_FRAC`, `t_COMPLEX`, `t_PADIC`, `t_QUAD`, or `t_POLMOD`.

`int is_extscalar_t(long t)` true iff `t` is a scalar (see `is_scalar_t`) or `t` is `t_POL`.

`int is_const_t(long t)` true iff `t` is a scalar which is not `t_POLMOD`.

5.2.6 Accessors and components. The first two functions return GEN components as copies on the stack:

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, *t_POL* or *t_SER*, and otherwise of the *n*-th component of *x*.

On the contrary, the following routines return the address of a GEN component. No copy is made on the stack:

GEN **constant_term**(GEN *x*) returns the address the constant term of *t_POL* *x*. By convention, a 0 polynomial (whose *sign* is 0) has *gen_0* constant term.

GEN **leading_term**(GEN *x*) returns the address the leading term of *t_POL* *x*. This may be an inexact 0.

GEN **gel**(GEN *x*, long *i*) returns the address of the *x*[*i*] entry of *x*. (*el* stands for element.)

GEN **gcoeff**(GEN *x*, long *i*, long *j*) returns the address of the *x*[*i*,*j*] entry of *t_MAT* *x*, i.e. the coefficient at row *i* and column *j*.

GEN **gmael**(GEN *x*, long *i*, long *j*) returns the address of the *x*[*i*][*j*] entry of *x*. (*mael* stands for multidimensional array element.)

GEN **gmael2**(GEN *A*, long *x1*, long *x2*) is an alias for *gmael*. Similar macros *gmael3*, *gmael4*, *gmael5* are available.

5.3 Handling the PARI stack.

5.3.1 Allocating memory on the stack

GEN **cgetg**(long *n*, long *t*) allocates memory on the stack for an object of length *n* and type *t*, and initializes its first codeword.

GEN **cgeti**(long *n*) allocates memory on the stack for a *t_INT* of length *n*, and initializes its first codeword. Identical to *cgetg*(*n*,*t_INT*).

GEN **cgetr**(long *n*) allocates memory on the stack for a *t_REAL* of length *n*, and initializes its first codeword. Identical to *cgetg*(*n*,*t_REAL*).

GEN **cgetc**(long *n*) allocates memory on the stack for a *t_COMPLEX*, whose real and imaginary parts are *t_REAL*s of length *n*.

GEN **cgetp**(GEN *x*) creates space sufficient to hold the *t_PADIC* *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 **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* GENs). 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 independent storage, but part of the stack.

Objects allocated through these last two functions cannot be **gerepile**'d. They are not valid GENs since they have no PARI type.

5.3.2 Garbage collection. See Section 4.3 for a detailed explanation and many examples.

`void cgiv(GEN x)` frees object `x` if it is the last created on the stack (otherwise nothing happens).

`GEN gerepile(pari_sp p, pari_sp q, GEN x)` general garbage collector for the stack.

`void gerepileall(pari_sp av, int n, ...)` cleans up the stack from `av` on (i.e from `avma` to `av`), preserving the `n` objects which follow in the argument list (of type `GEN*`). E.g: `gerepileall(av, 2, &x, &y)` preserves `x` and `y`.

`void gerepileallsp(pari_sp av, pari_sp ltop, int n, ...)` cleans up the stack between `av` and `ltop`, updating the `n` elements which follow `n` in the argument list (of type `GEN*`). Check that the elements of `g` have no component between `av` and `ltop`, and assumes that no garbage is present between `avma` and `ltop`. Analogous to (but faster than) `gerepileall` otherwise.

`GEN gerepilecopy(pari_sp av, GEN x)` cleans up the stack from `av` on, preserving the object `x`. Special case of `gerepileall` (case `n = 1`), except that the routine returns the preserved GEN instead of updating its adress through a pointer.

`void gerepilemany(pari_sp av, GEN* g[], int n)` alternative interface to `gerepileall`

`void gerepilemanysp(pari_sp av, pari_sp ltop, GEN* g[], int n)` alternative interface to `gerepileallsp`.

`void gerepilecoeffs(pari_sp av, GEN x, int n)` cleans up the stack from `av` on, preserving `x[0], ..., x[n-1]` (which are GENs).

`void gerepilecoeffssp(pari_sp av, pari_sp ltop, GEN x, int n)` cleans up the stack from `av` to `ltop`, preserving `x[0], ..., x[n-1]` (which are GENs). Same assumptions as in `gerepilemanysp`, of which this is a variant. For instance

```
z = cgetg(3, t_COMPLEX);
av = avma; garbage(); ltop = avma;
z[1] = fun1();
z[2] = fun2();
gerepilecoeffssp(av, ltop, z + 1, 2);
return z;
```

cleans up the garbage between `av` and `ltop`, and connects `z` and its two components. This is marginally more efficient than the standard

```
av = avma; garbage(); ltop = avma;
z = cgetg(3, t_COMPLEX);
z[1] = fun1();
z[2] = fun2(); return gerepile(av, ltop, z);
```

`GEN gerepileupto(pari_sp av, GEN q)` analogous to (but faster than) `gerepilecopy`. Assumes that `q` is connected and that its root was created before any component.

GEN **gerepileuptoint**(*pari_sp* av, GEN q) analogous to (but faster than) **gerepileupto**. Assumes further that q is a *t_INT*. The length and effective length of the resulting *t_INT* are equal.

GEN **gerepileuptoleaf**(*pari_sp* av, GEN q) analogous to (but faster than) **gerepileupto**. Assumes further that q is a leaf, i.e a non-recursive type (**is_recursive_t**(*typ*(q)) is non-zero). Contrary to **gerepileuptoint**, **gerepileuptoleaf** leaves length and effective length of a *t_INT* unchanged.

void **stackdummy**(*pari_sp* av, *pari_sp* ltop) inhibits the memory area between av *included* and ltop *excluded* with respect to **gerepile**, in order to avoid a call to **gerepile**(av, ltop,...). The stack space is not reclaimed though.

More precisely, this routine assumes that av is recorded earlier than ltop, then marks the specified stack segment as a non-recursive type of the correct length. Thus **gerepile** will not inspect the zone, at most copy it. To be used in the following situation:

```
av0 = avma; z = cgetg(t_VEC, 3);
gel(z,1) = HUGE(); av = avma; garbage(); ltop = avma;
gel(z,2) = HUGE(); stackdummy(av, ltop);
```

Compared to the orthodox

```
gel(z,2) = gerepile(av, ltop, gel(z,2));
```

or even more wasteful

```
z = gerepilecopy(av0, z);
```

we temporarily lose (av – ltop) words but save a costly **gerepile**. In principle, a garbage collection higher up the call chain should reclaim this later anyway.

Without the **stackdummy**, if the [av, ltop] zone is arbitrary (not even valid GENs as could happen after direct truncation via **setlg**), we would leave dangerous data in the middle of z, which would be a problem for a later

```
gerepile(..., ... , z);
```

And even if it were made of valid GENs, inhibiting the area makes sure **gerepile** will not inspect their components, saving time.

Another natural use in low-level routines is to “shorten” an existing GEN z to its first $l - 1$ components:

```
setlg(z, l);
stackdummy((pari_sp)(z + lg(z)), (pari_sp)(z + 1));
```

or to its last l components:

```
long L = lg(z) - l;
stackdummy((pari_sp)(z + L), (pari_sp)z);
z += L; setlg(z, L);
```

5.3.3 Copies and clones

GEN gclone(GEN *x*) creates a new permanent copy of the object *x* on the heap. The *clone bit* of the result is set.

void gunclone(GEN *x*) delete the clone *x* (created by **gclone**). Fatal error if *x* not a clone.

GEN gcopy(GEN *x*) creates a new copy of the object *x* on the stack.

int isonstack(GEN *x*) true iff *x* belongs to the stack. This is a macro whose argument is evaluated several times.

void copyifstack(GEN *x*, GEN *y*) sets *y* = **gcopy**(*x*) if *x* belongs to the stack, and *y* = *x* otherwise. This macro evaluates its arguments once, contrary to

```
y = isonstack(x)? gcopy(x): x;
```

void icopyifstack(GEN *x*, GEN *y*) as **copyifstack** assuming *x* is a **t_INT**.

long taille(GEN *x*) returns the total number of BIL-bit words occupied by the tree representing *x*.

void traverseheap(void(*f)(GEN, void *), void *data) this applies **f**(*x*, *data*) to each object *x* on the PARI heap, most recent first. Mostly for debugging purposes.

GEN getheap() a simple wrapper around **traverseheap**. Returns a two-component row vector giving the number of objects on the heap and the amount of memory they occupy in long words.

5.4 Level 0 kernel (operations on ulongs).

5.4.1 Micro-kernel. Level 0 operations simulate basic operations of the 68020 processor on which PARI was originally implemented. They need “global” **ulong** variables **overflow** (which will contain only 0 or 1) and **hiremainder** to function properly. However, for certain architectures these are replaced with local variables for efficiency; and the ‘functions’ mentioned below are really chunks of inlined assembler code. So, a routine using one of these lowest-level functions where the description mentions either **hiremainder** or **overflow** must declare the corresponding

```
LOCAL_HIREMAINDER;  
LOCAL_OVERFLOW;
```

in a declaration block. Variables **hiremainder** and **overflow** then become available in the enclosing block. For instance a loop over the powers of an **ulong** *p* protected from overflows could read

```
while (pk < lim)  
{  
    LOCAL_HIREMAINDER;  
    ...  
    pk = mulll(pk, p); if (hiremainder) break;  
}
```

ulong addll(ulong *x*, ulong *y*) adds *x* and *y*, returns the lower BIL bits and puts the carry bit into **overflow**.

ulong addllx(ulong *x*, ulong *y*) adds **overflow** to the sum of the *x* and *y*, returns the lower BIL bits and puts the carry bit into **overflow**.

ulong subll(ulong x, ulong y) subtracts 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 x and y , returns the lower BIL bits and puts the carry (borrow) bit into **overflow**.

int bfffo(ulong x) returns the number of leading zero bits in x . That is, 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 $\text{BIL} - 1$ for nonzero x . When x is 0, the result is undefined.

ulong mulll(ulong x, ulong y) multiplies x by 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 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 $(\text{hiremainder} \ll \text{BIL}) + x$ by y and stores the remainder into **hiremainder**. An error occurs if the quotient cannot be represented by an **ulong**, i.e. if initially $\text{hiremainder} \geq y$.

5.4.2 Modular kernel. 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 Fl.add(ulong x, ulong y, ulong m) returns the smallest positive representative of $x + y$ modulo m .

ulong Fl.neg(ulong x, ulong m) returns the smallest positive representative of $-x$ modulo m .

ulong Fl.sub(ulong x, ulong y, ulong m) returns the smallest positive representative of $x - y$ modulo m .

long Fl.center(ulong x, ulong m, ulong mo2) returns the representative in $] - m/2, m/2]$ of x modulo m . Assume $0 \leq x < m$ and $\text{mo2} = m > 1$.

ulong Fl.mul(ulong x, ulong y, ulong m) returns the smallest positive representative of xy modulo m .

ulong Fl.inv(ulong x, ulong m) returns the smallest positive representative of x^{-1} modulo m . If x is not invertible mod m , raise an exception.

ulong Fl.div(ulong x, ulong y, ulong m) returns the smallest positive representative of xy^{-1} modulo m . If y is not invertible mod m , raise an exception.

ulong Fl.pow(ulong x, ulong n, ulong m) returns the smallest positive representative of x^n modulo m .

ulong Fl.sqrt(ulong x, ulong p) returns the square root of x modulo p (smallest positive representative). Assumes p to be prime, and x to be a square modulo p .

ulong gener_Fl(ulong p) returns a primitive root modulo p , assuming p is prime.

ulong gener_Fl.local(ulong p, GEN L), see **gener_Fp.local**, L is an **Flv**.

long krouu(ulong x, ulong y) returns the Kronecker symbol $(x|y)$, i.e. $-1, 0$ or 1 . Assumes y is non-zero. If y is an odd prime, this is the Legendre symbol.

5.5 Level 1 kernel (operations on longs, integers and reals).

Note: Many functions consist of an elementary operation, immediately followed by an assignment statement. They will be introduced as 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. All such functions are obtained using macros (see the file `paricom.h`), hence you can easily extend the list. These assignment variants are inefficient; don't use them.

5.5.1 Creation

GEN **cgeti**(long n) allocates memory on the PARI stack for a `t_INT` 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 `t_REAL` of length n, and initializes its first codeword. Identical to `cgetg(n,t_REAL)`.

GEN **cgetc**(long n) allocates memory on the PARI stack for a `t_COMPLEX`, whose real and imaginary parts are `t_REALs` of length n.

GEN **real_1**(long prec) create a `t_REAL` equal to 1 to `prec` words of accuracy.

GEN **real_m1**(long prec) create a `t_REAL` equal to -1 to `prec` words of accuracy.

GEN **real_0_bit**(long bit) create a `t_REAL` equal to 0 with exponent $-\text{bit}$.

GEN **real_0**(long prec) is a shorthand for

```
real_0_bit( -bit_accuracy(prec) )
```

GEN **int2n**(long n) creates a `t_INT` equal to $1 \ll n$ (i.e 2^n if $n \geq 0$, and 0 otherwise).

GEN **int2u**(ulong n) creates a `t_INT` equal to 2^n .

GEN **real2n**(long n, long prec) create a `t_REAL` equal to 2^n to `prec` words of accuracy.

GEN **stoi**(char *s) convert the character string s to a `t_INT`.

GEN **stror**(char *s, long prec) convert the character string s to a `t_REAL` of precision `prec`.

5.5.2 Assignment. In this section, the `z` argument in the `z`-functions must be of type `t_INT` or `t_REAL`.

`void mpaff(GEN x, GEN z)` assigns `x` into `z` (where `x` and `z` are `t_INT` or `t_REAL`). Assumes that $\lg(z) > 2$.

`void affii(GEN x, GEN z)` assigns the `t_INT` `x` into the `t_INT` `z`.

`void affir(GEN x, GEN z)` assigns the `t_INT` `x` into the `t_REAL` `z`. Assumes that $\lg(z) > 2$.

`void affiz(GEN x, GEN z)` assigns `t_INT` `x` into `t_INT` or `t_REAL` `z`. Assumes that $\lg(z) > 2$.

`void affsi(long s, GEN z)` assigns the `long` `s` into the `t_INT` `z`. Assumes that $\lg(z) > 2$.

`void affsr(long s, GEN z)` assigns the `long` `s` into the `t_REAL` `z`. Assumes that $\lg(z) > 2$.

`void affsz(long s, GEN z)` assigns the `long` `s` into the `t_INT` or `t_REAL` `z`. Assumes that $\lg(z) > 2$.

`void affui(ulong u, GEN z)` assigns the `ulong` `u` into the `t_INT` `z`. Assumes that $\lg(z) > 2$.

`void affur(ulong u, GEN z)` assigns the `ulong` `u` into the `t_REAL` `z`. Assumes that $\lg(z) > 2$.

`void affrr(GEN x, GEN z)` assigns the `t_REAL` `x` into the `t_REAL` `z`.

The function `affrs` and `affri` do not exist. So don't use them.

5.5.3 Copy

`GEN icopy(GEN x)` copy relevant words of the `t_INT` `x` on the stack: the length and effective length of the copy are equal.

`GEN rcopy(GEN x)` copy the `t_REAL` `x` on the stack.

`GEN mpcopy(GEN x)` copy the `t_INT` or `t_REAL` `x` on the stack. Contrary to `icopy`, `mpcopy` preserves the original length of a `t_INT`.

5.5.4 Conversions

`GEN itor(GEN x, long prec)` converts the `t_INT` `x` to a `t_REAL` of length `prec` and return the latter. Assumes that $\text{prec} > 2$.

`long itos(GEN x)` converts the `t_INT` `x` to a `long` if possible, otherwise raise an exception.

`long itos_or_0(GEN x)` converts the `t_INT` `x` to a `long` if possible, otherwise return 0.

`ulong itou(GEN x)` converts the `t_INT` `|x|` to an `ulong` if possible, otherwise raise an exception.

`long itou_or_0(GEN x)` converts the `t_INT` `|x|` to an `ulong` if possible, otherwise return 0.

`GEN stoi(long s)` creates the `t_INT` corresponding to the `long` `s`.

`GEN stor(long s, long prec)` converts the `long` `s` into a `t_REAL` of length `prec` and return the latter. Assumes that $\text{prec} > 2$.

`GEN utoi(ulong s)` converts the `ulong` `s` into a `t_INT` and return the latter.

`GEN utoipos(ulong s)` converts the *non-zero* `ulong` `s` into a `t_INT` and return the latter.

`GEN utoineg(ulong s)` converts the *non-zero* `ulong` `s` into a `t_INT` and return the latter.

GEN **utor**(ulong s, long prec) converts the ulong s into a t_REAL of length prec and return the latter. Assumes that $\text{prec} > 2$.

GEN **rtor**(GEN x, long prec) converts the t_REAL x to a t_REAL of length prec and return the latter. If $\text{prec} < \lg(x)$, round properly. If $\text{prec} > \lg(x)$, padd with zeroes. Assumes that $\text{prec} > 2$.

The following function is also available as a special case of **mkintn**:

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.

5.5.5 Integer parts

GEN **ceilr**(GEN x) smallest integer larger or equal to the t_REAL x (i.e. the **ceil** function).

GEN **floorr**(GEN x) largest integer smaller or equal to the t_REAL x (i.e. the **floor** function).

GEN **roundr**(GEN x) rounds the t_REAL x to the nearest integer (towards $+\infty$).

GEN **truncr**(GEN x) truncates the t_REAL x (not the same as **floorr** if x is and negative).

GEN **mpceil**[z](GEN x[, GEN z]) as **ceilr** except that x may be a t_INT.

GEN **ceil_safe**(GEN x), x being a real number (not necessarily a t_REAL) returns an integer which is larger than any possible incarnation of x. (Recall that a t_REAL represents an interval of possible values.)

GEN **mpfloor**[z](GEN x[, GEN z]) as **floorr** except that x may be a t_INT.

GEN **mpround**[z](GEN x[, GEN z]) as **roundr** except that x may be a t_INT.

GEN **mptrunc**[z](GEN x[, GEN z]) as **truncr** except that x may be a t_INT.

GEN **diviiround**(GEN x, GEN y) if x and y are t_INTs, 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 **roundr**).

5.5.6 Valuation and shift

long **vals**(long s) 2-adic valuation of the long s. Returns -1 if s is equal to 0.

long **vali**(GEN x) 2-adic valuation of the t_INT x. Returns -1 if x is equal to 0.

GEN **mpshift**[z](GEN x, long n[, GEN z]) shifts the t_INT or t_REAL x by n. If n is positive, this is a left shift, i.e. multiplication by 2^n . If n is negative, it is a right shift by $-n$, which amounts to the truncation of the quotient of x by 2^{-n} .

GEN **shifti**(GEN x, long n) shifts the t_INT x by n.

GEN **shiftr**(GEN x, long n) shifts the t_REAL x by n.

long **Z_pvalrem**(GEN x, GEN p, GEN *r) applied to t_INTs $x \neq 0$ and p, $|p| > 1$, returns the highest exponent e such that p^e divides x. The quotient x/p^e is returned in *r. In particular, if p is a prime, this returns the valuation at p of x, and *r is the prime-to-p part of x.

long **Z_pval**(GEN x, GEN p) as **Z_pvalrem** but only returns the “valuation”.

long **Z_lvalrem**(GEN x, ulong p, GEN *r) as **Z_pvalrem**, except that p is an ulong ($p > 1$).

`long Z_lval(GEN x, ulong p)` as `Z_pval`, except that `p` is an `ulong` ($p > 1$).

`long u_lvalrem(ulong x, ulong p, ulong *r)` as `Z_pvalrem`, except the inputs/outputs are now `ulongs`.

`long u_pvalrem(ulong x, GEN p, ulong *r)` as `Z_pvalrem`, except `x` and `r` are now `ulongs`.

`long u_lval(ulong x, ulong p)` as `Z_pval`, except the inputs/outputs are now `ulongs`.

5.5.7 Factorization

`GEN Z_factor(GEN n)` factors the `t_INT` `n`. The “primes” in the factorization are actually strong pseudoprimes.

`long Z_issquarefree(GEN x)` returns 1 if the `t_INT` `n` is square-free, and 0 otherwise.

`long Z_issquare(GEN n)` returns 1 if `t_INT` `n` is a square, and 0 otherwise. This is tested first modulo small prime powers, then `sqrtemi` is called.

`long Z_issquarerem(GEN n, GEN *sqrtn)` as `Z_issquare`. If `n` is indeed a square, set `sqrtn` to its integer square root.

`int isprime(GEN n)`, returns 1 if the `t_INT` `n` is a (fully proven) prime number and 0 otherwise.

`int uisprime(ulong p)`, returns 1 if `p` is a prime number and 0 otherwise.

`long Z_issquarerem(GEN n, GEN *sqrtn)` as `Z_issquare`. If `n` is indeed a square, set `sqrtn` to its integer square root.

`long uissquarerem(ulong n, ulong *sqrtn)` as `Z_issquarerem`, for an `ulong` operand `n`.

5.5.8 Generic unary operators. Let “*op*” be a unary operation among

op=**neg**: negation ($-x$).

op=**abs**: absolute value ($|x|$).

The names and prototypes of the low-level functions corresponding to *op* are as follows. The result is of the same type as `x`.

`GEN mpop(GEN x)` creates the result of *op* applied to the `t_INT` or `t_REAL` `x`.

`GEN opi(GEN x)` creates the result of *op* applied to the `t_INT` `x`.

`GEN opr(GEN x)` creates the result of *op* applied to the `t_REAL` `x`.

`GEN mpoz(GEN x, GEN z)` assigns the result of applying *op* to the `t_INT` or `t_REAL` `x` into the `t_INT` or `t_REAL` `z`.

Remark: it has not been considered useful to include functions `void opsz(long,GEN)`, `void opiz(GEN,GEN)` and `void oprz(GEN, GEN)`.

5.5.9 Comparison operators

`int mpcmp(GEN x, GEN y)` compares the `t_INT` or `t_REAL` `x` to the `t_INT` or `t_REAL` `y`. The result is the sign of $x - y$.

`int cmpii(GEN x, GEN y)` compares the `t_INT` `x` to the `t_INT` `y`.

`int cmpir(GEN x, GEN y)` compares the `t_INT` `x` to the `t_REAL` `y`.

`int cmpis(GEN x, long s)` compares the `t_INT` `x` to the `long` `s`.

`int cmpsi(long s, GEN x)` compares the `long` `s` to the `t_INT` `x`.

`int cmpsr(long s, GEN x)` compares the `long` `s` to the `t_REAL` `x`.

`int cmpri(GEN x, GEN y)` compares the `t_REAL` `x` to the `t_INT` `y`.

`int cmprr(GEN x, GEN y)` compares the `t_REAL` `x` to the `t_REAL` `y`.

`int cmprs(GEN x, long s)` compares the `t_REAL` `x` to the `long` `s`.

`int equalii(GEN x, GEN y)` compares the `t_INT`s `x` and `y`. The result is 1 if $x = y$, 0 otherwise.

`int equalsi(long s, GEN x)`

`int equalis(GEN x, long s)` compare the `t_INT` `x` and the `long` `s`. The result is 1 if $x = y$, 0 otherwise.

`int equalui(ulong s, GEN x)`

`int equaliu(GEN x, ulong s)` compare the `t_INT` `x` and the `ulong` `s`. The result is 1 if $|x| = y$, 0 otherwise.

`int absi_cmp(GEN x, GEN y)` compares the `t_INT`s `x` and `y`. The result is the sign of $|x| - |y|$.

`int absi_equal(GEN x, GEN y)` compares the `t_INT`s `x` and `y`. The result is 1 if $|x| = |y|$, 0 otherwise.

`int absr_cmp(GEN x, GEN y)` compares the `t_REAL`s `x` and `y`. The result is the sign of $|x| - |y|$.

5.5.10 Generic binary operators. Let “*op*” be a binary operation among

op=**add**: addition ($x + y$). The result is a `t_REAL` unless both `x` and `y` are `t_INT`s (or `long`s).

op=**sub**: subtraction ($x - y$). The result is a `t_REAL` unless both `x` and `y` are `t_INT` (or `long`s).

op=**mul**: multiplication ($x * y$). The result is a `t_REAL` unless both `x` and `y` are `t_INT`s (or `long`s), or if `x` or `y` is an exact 0.

op=**div**: division (x / y). In the case where `x` and `y` are both `t_INT`s or `long`s, the result is the Euclidean quotient, where the remainder has the same sign as the dividend `x`. It is the ordinary division otherwise. If one of `x` or `y` is a `t_REAL`, the result is a `t_REAL` unless `x` is an exact 0. A division-by-0 error occurs if `y` is equal to 0.

op=**rem**: remainder (“ $x \% y$ ”). This operation is defined only when `x` and `y` are `long`s or `t_INT`. The result is the Euclidean remainder corresponding to **div**, i.e. its sign is that of the dividend `x`. The result is always a `t_INT`.

op=mod: true remainder ($x \% y$). This operation is defined only when x and y are longs or `t_INTs`. The result is the true Euclidean remainder, i.e. non-negative and less than the absolute value of y .

The names and prototypes of the low-level functions corresponding to *op* are as follows. In this section, the z argument in the z -functions must be of type `t_INT` or `t_REAL`. `t_INT` is only allowed when no ‘r’ appears in the argument code (no `t_REAL` operand is involved).

GEN **mpop**[z](GEN x , GEN y [, GEN z]) applies *op* to the `t_INT` or `t_REAL` x and y .

GEN **opsi**[z](long s , GEN x [, GEN z]) applies *op* to the long s and the `t_INT` x .

GEN **opsr**[z](long s , GEN x [, GEN z]) applies *op* to the long s and the `t_REAL` x .

GEN **opss**[z](long s , long t [, GEN z]) applies *op* to the longs s and t .

GEN **opii**[z](GEN x , GEN y [, GEN z]) applies *op* to the `t_INTs` x and y .

GEN **opir**[z](GEN x , GEN y [, GEN z]) applies *op* to the `t_INT` x and the `t_REAL` y .

GEN **opis**[z](GEN x , long s [, GEN z]) applies *op* to the `t_INT` x and the long s .

GEN **opri**[z](GEN x , GEN y [, GEN z]) applies *op* to the `t_REAL` x and the `t_INT` y .

GEN **oprr**[z](GEN x , GEN y [, GEN z]) applies *op* to the `t_REALs` x and y .

GEN **oprs**[z](GEN x , long s [, GEN z]) applies *op* to the `t_REAL` x and the long s .

Some miscellaneous routines whose meaning should be clear from their names:

GEN **muluu**(ulong x , ulong y)

GEN **mului**(ulong x , GEN y)

GEN **muliu**(GEN x , ulong y)

GEN **sqri**(GEN x) squares the `t_INT` x

GEN **truedivii**(GEN x , GEN y) returns the true Euclidean quotient (with non-negative remainder less than $|y|$).

GEN **truedivis**(GEN x , long y) returns the true Euclidean quotient (with non-negative remainder less than $|y|$).

GEN **centermodii**(GEN x , GEN y , GEN $y2$), given `t_INTs` x , y , returns z congruent to x modulo y , such that $-y/2 \leq z < y/2$. Assumes that $y2 = \text{shifti}(y, -1)$. the representative of $ssquares$ the `t_INT` x

5.5.11 Modulo to longs. The following variants of `modii` do not clutter the stack:

`long smodis(GEN x, long y)` computes the true Euclidean remainder of the `t_INT` `x` by the `long` `y`. This is the non-negative remainder, not the one whose sign is the sign of `x` as in the `div` functions.

`long smodsi(long x, GEN y)` computes the true Euclidean remainder of the `long` `x` by a `t_INT` `y`.

`long smodss(long x, long y)` computes the true Euclidean remainder of the `long` `x` by a `t_long` `y`.

`ulong umodiu(GEN x, ulong y)` computes the true Euclidean remainder of the `t_INT` `x` by the `ulong` `y`.

`ulong umodui(ulong x, GEN y)` computes the true Euclidean remainder of the `ulong` `x` by the `t_INT` `|y|`.

The routine `smodsi` does not exist, since it would not always be defined: for a *negative* `x`, its result `x + |y|` would in general not fit into a `long`. Use either `umodui` or `modsi`.

5.5.12 Exact division and divisibility

`void diviexact(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 diviueexact(GEN x, ulong y)` returns the Euclidean quotient `|x|/y` (note the absolute value!), assuming `y` divides `x` and `y` is non-zero.

`int dvdii(GEN x, GEN y)` if the `t_INT` `y` divides the `t_INT` `x`, returns 1 (true), otherwise returns 0 (false).

`int dvdiiz(GEN x, GEN y, GEN z)` if the `t_INT` `y` divides the `t_INT` `x`, assigns the quotient to the `t_INT` `z` and returns 1 (true), otherwise returns 0 (false).

`int dvdisz(GEN x, long y, GEN z)` if the `t_long` `y` divides the `t_INT` `x`, assigns the quotient to the `t_INT` `z` and returns 1 (true), otherwise returns 0 (false).

`int dvdiuz(GEN x, ulong y, GEN z)` if the `t_ulong` `y` divides the `t_INT` `x`, assigns the quotient `|x|/y` to the `t_INT` `z` and returns 1 (true), otherwise returns 0 (false).

5.5.13 Division with remainder. The following functions return two objects, unless specifically asked for only one of them — a quotient and a remainder. The quotient is returned and the remainder is returned through the variable whose address is passed as the `r` argument. The term *true Euclidean remainder* refers to the non-negative one (`mod`), and *Euclidean remainder* by itself to the one with the same sign as the dividend (`rem`). All `GEN`s, whether returned directly or through a pointer, are created on the stack.

`GEN dvmdii(GEN x, GEN y, GEN *r)` returns the Euclidean quotient of the `t_INT` `x` by a `t_INT` `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`. If the remainder is 0 set `r = gen_0`.

`void dvmdiiz(GEN x, GEN y, GEN z, GEN t)` assigns the Euclidean quotient of the `t_INT`s `x` and `y` into the `t_INT` or `t_REAL` `z`, and the Euclidean remainder into the `t_INT` or `t_REAL` `t`.

Analogous routines `dvmdis[z]`, `dvmdsi[z]`, `dvmdss[z]` are available, where `s` denotes a `long` argument. But the following routines are in general more flexible:

`long sdivss_rem(long s, long t, long *r)` computes the Euclidean quotient and remainder of the `longs` `s` and `t`. Puts the remainder into `*r`, and returns the quotient. The remainder is of the sign of the dividend `s`, and has strictly smaller absolute value than `t`.

`long sdivsi_rem(long s, GEN x, long *r)` computes the Euclidean quotient and remainder of the `long` `s` by the `t_INT` `x`. As `sdivss_rem` otherwise.

`long sdivsi(long s, GEN x)` as `sdivsi_rem`, without remainder.

`GEN divis_rem(GEN x, long s, long *r)` computes the Euclidean quotient and remainder of the `t_INT` `x` by the `long` `s`. As `sdivss_rem` otherwise.

`GEN diviu_rem(GEN x, ulong s, long *r)` computes the Euclidean quotient and remainder of the `t_INT` `x` by the `ulong` `s`. As `sdivss_rem` otherwise.

`GEN divsi_rem(long s, GEN y, long *r)` computes the Euclidean quotient and remainder of the `t_long` `s` by the `GEN` `y`. As `sdivss_rem` otherwise.

`GEN divss_rem(long x, long y, long *r)` computes the Euclidean quotient and remainder of the `t_long` `x` by the `long` `y`. As `sdivss_rem` otherwise.

`GEN truedvmdii(GEN x, GEN y, GEN *r)`, as `dvmdii` but with a non-negative remainder.

5.5.14 Square root and remainder

`GEN sqrtremi(GEN N, GEN *r)`, returns the integer square root S of the non-negative `t_INT` `N` (rounded towards 0) and puts the remainder R into `*r`. Precisely, $N = S^2 + R$ with $0 \leq R \leq 2S$. If `r` is equal to `NULL`, the remainder is not created. In the generic case, the remainder is created after the quotient and can be disposed of individually with `cgiv(R)`. If the remainder is 0 set `R = gen_0`.

Uses a divide and conquer algorithm (discrete variant of Newton iteration) due to Paul Zimmermann (“Karatsuba Square Root”, INRIA Research Report 3805 (1999)).

`GEN sqrti(GEN N)`, returns the integer square root S of the non-negative `t_INT` `N` (rounded towards 0). This is identical to `sqrtremi(N, NULL)`.

5.5.15 Pseudo-random integers

`long random_bits(long k)` returns a random $0 \leq x < 2^k$. Assumes that $0 \leq k < 31$.

`long pari_rand31(long k)` as `random_bits` with $k = 31$.

`GEN randomi(GEN n)` returns a random `t_INT` between 0 and `n - 1`. The result is pasted from successive calls to `pari_rand31`.

5.5.16 Modular operations. In this subsection, all GENs are `t_INT`.

`ulong Fp_powu(GEN x, ulong n, GEN m)` raises x to the n -th power modulo p (smallest non-negative residue).

`GEN Fp_pow(GEN x, GEN n, GEN m)` returns x^n modulo p (smallest non-negative residue).

`GEN Fp_inv(GEN a, GEN m)` returns an inverse of a modulo m (smallest non-negative residue). Raise an error if a is not invertible.

`GEN Fp_invsafe(GEN a, GEN m)` as `Fp_inv`, but return `NULL` if a is not invertible.

`int invmod(GEN a, GEN m, GEN *g)`, return 1 if a modulo m is invertible, else return 0 and set $g = \gcd(a, m)$.

`GEN Fp_sqrt(GEN x, GEN p)` returns a square root of x modulo p (the smallest non-negative residue), where x, p are `t_INTs`, and p is assumed to be prime. Return `NULL` if x is not a quadratic residue modulo p .

`GEN Fp_sqrtn(GEN x, GEN n, GEN p, GEN *zn)` returns an n -th root of x modulo p (smallest non-negative residue), where x, n, p are `t_INTs`, and p is assumed to be prime. Return `NULL` if x is not an n -th power residue. Otherwise, if zn is non-`NULL` set it to a primitive n -th root of 1.

`long kross(long x, long y)` returns the Kronecker symbol $(x|y)$, i.e. $-1, 0$ or 1 . If y is an odd prime, this is the Legendre symbol. (Contrary to `krouu`, `kross` also supports $y = 0$)

`long krois(GEN x, long y)` returns the Kronecker symbol $(x|y)$ of `t_INT` x and `long` y . As `kross` otherwise.

`long krosi(long x, GEN y)` returns the Kronecker symbol $(x|y)$ of `long` x and `t_INT` y . As `kross` otherwise.

`long kronecker(GEN x, GEN y)` returns the Kronecker symbol $(x|y)$ of `t_INTs` x and y . As `kross` otherwise.

`GEN gener_Fp(GEN p)` returns a primitive root modulo p , assuming p is prime.

`GEN gener_Fp_local(GEN p, GEN L)`, L being a vector of primes dividing $p - 1$, returns an integer x which is a generator of the ℓ -Sylow of \mathbf{F}_p^* for every ℓ in L . In other words, $x^{(p-1)/\ell} \neq 1$ for all such ℓ . In particular, returns `Fp_gener(p)` if L contains all primes dividing $p - 1$.

5.5.17 Miscellaneous functions

`void addumului(ulong a, ulong b, GEN x)` return $a + b|X|$.

`long cgcd(long x, long y)`, returns the GCD of the `t_longs` x and y .

`long cbezout(long a, long b, long *u, long *v)`, returns the GCD d of a and b and sets u, v to the Bezout coefficients such that $au + bv = d$.

`GEN bezout(GEN a, GEN b, GEN *u, GEN *v)`, returns the GCD d of `t_INTs` a and b and sets u, v to the Bezout coefficients such that $au + bv = d$.

`GEN factoru(ulong n)`, returns the factorization of n . The result is a 2-component vector $[P, E]$, where P and E are `t_VECSMALL` containing the prime divisors of n , and the $v_p(n)$.

`GEN factoru_pow(ulong n)`, returns the factorization of n . The result is a 3-component vector $[P, E, C]$, where P, E and C are `t_VECSMALL` containing the prime divisors of n , the $v_p(n)$ and the $p^{v_p(n)}$.

GEN **gcdii**(GEN *x*, GEN *y*), returns the GCD of the *t_INT*s *x* and *y*.

GEN **lcmii**(GEN *x*, GEN *y*), returns the LCM of the *t_INT*s *x* and *y*.

long **maxss**(long *x*, long *y*), return the largest of *x* and *y*.

long **minss**(long *x*, long *y*), return the smallest of *x* and *y*.

GEN **powuu**(ulong *n*, ulong *k*), returns n^k .

GEN **powiu**(GEN *n*, ulong *k*), assumes *n* is a *t_INT* and returns n^k .

ulong **upowuu**(ulong *n*, ulong *k*), returns n^k modulo 2^{BIL} . This is meant to be used for tiny *k*, where in fact n^k fits into an *ulong*.

void **rdivii**(GEN *x*, GEN *y*, long *prec*), assuming *x* and *y* are both of type *t_INT*, return the quotient *x/y* as a *t_REAL* of precision *prec*.

void **rdivis**(GEN *x*, long *y*, long *prec*), assuming *x* is of type *t_INT*, return the quotient *x/y* as a *t_REAL* of precision *prec*.

void **rdivsi**(long *x*, GEN *y*, long *prec*), assuming *y* is of type *t_INT*, return the quotient *x/y* as a *t_REAL* of precision *prec*.

void **rdivss**(long *x*, long *y*, long *prec*), return the quotient *x/y* as a *t_REAL* of precision *prec*.

5.6 Level 2 kernel (modular arithmetic).

These routines implement univariate polynomial arithmetic and linear algebra over finite fields, in fact over finite rings of the form $(\mathbf{Z}/p\mathbf{Z})[X]/(T)$, where *p* is not necessarily prime and $T \in (\mathbf{Z}/p\mathbf{Z})[X]$ is possibly reducible; and finite extensions thereof. All this can be emulated with *t_INTMOD* and *t_POLMOD* coefficients and using generic routines, at a considerable loss of efficiency. Also, some specialized routines are available that have no obvious generic equivalent.

5.6.1 Naming scheme. A function name is built in the following way: $A_1 \dots A_n \text{fun}$ for an operation *fun* with *n* arguments of class A_1, \dots, A_n . A class name is given by a base ring followed by a number of code letters. Base rings are among

F_l: $\mathbf{Z}/l\mathbf{Z}$ where $l < 2^{\text{BIL}}$ is not necessarily prime. Implemented using *ulongs*

F_p: $\mathbf{Z}/p\mathbf{Z}$ where *p* is a *t_INT*, not necessarily prime. Implemented as *t_INT*s *z*, preferably satisfying $0 \leq z < p$. More precisely, any *t_INT* can be used as an **F_p**, but reduced inputs are treated more efficiently. Outputs from **Fpxxx** routines are reduced.

F_q: $\mathbf{Z}[X]/(p, T(X))$, *p* a *t_INT*, *T* a *t_POL* with **F_p** coefficients or NULL (in which case no reduction modulo *T* is performed). Implemented as *t_POL*s *z* with **F_p** coefficients, $\deg(z) < \deg T$.

Z: the integers \mathbf{Z} , implemented as *t_INT*s.

z: the integers \mathbf{Z} , implemented using (signed) *long*s.

Q: the rational numbers \mathbf{Q} , implemented as *t_INT*s and *t_FRAC*s.

Rg: a commutative ring, whose elements can be *gadd*-ed, *gmul*-ed, etc.

Possible letters are:

X: polynomial in X (**t_POL** in a fixed variable), e.g. **FpX** means $\mathbf{Z}/p\mathbf{Z}[X]$

Y: polynomial in $Y \neq X$. E.g. **FpXY** means $((\mathbf{Z}/p\mathbf{Z})[Y])[X]$

V: vector (**t_VEC** or **t_COL**), treated as a line vector (independantly of the actual type). E.g. **ZV** means \mathbf{Z}^k for some k .

C: vector (**t_VEC** or **t_COL**), treated as a column vector (independantly of the actual type). The difference with **V** is purely semantic.

M: matrix (**t_MAT**). E.g. **QM** means a matrix with rational entries

Q: representative (**t_POL**) of a class in a polynomial quotient ring. E.g. an **FpXQ** belongs to $(\mathbf{Z}/p\mathbf{Z})[X]/(T(X))$, **FpXQV** means a vector of such elements, etc.

x, **m**, **v**, **c**, **q**: as their uppercase counterpart, but coefficient arrays are implemented using **t_VECSMALLs**, which coefficient understood as **ulongs**.

x (and **q**) are implemented by a **t_VECSMALL** whose first coefficient is used as a code-word and the following are the coefficients, similarly to a **t_POL**. This is known as a 'POLSMALL'.

m are implemented by a **t_MAT** whose components (columns) are **t_VECSMALLs**. This is known as a 'MATSMALL'.

v and **c** are regular **t_VECSMALLs**. Difference between the two is purely semantic.

Omitting the letter means the argument is a scalar in the base ring. Standard functions *fun* are

add: add

sub: subtract

mul: multiply

sqr: square

div: divide (Euclidean quotient)

rem: Euclidean remainder

divrem: return Euclidean quotient, store remainder in a pointer argument.

gcd: GCD

extgcd: return GCD, store Bezout coefficients in pointer arguments

pow: exponentiate

compo: composition

5.6.2 ZX, ZV, ZM. A **ZV** (resp. a **ZM**, resp. a **ZX**) is a **t_VEC** or **t_COL** (resp. **t_MAT**, resp. **t_POL**) with **t_INT** coefficients.

5.6.2.1 ZV

GEN ZV_add(**GEN x**, **GEN y**) adds **x** and **y**.

GEN ZV_sub(**GEN x**, **GEN y**) subtracts **x** and **y**.

5.6.2.2 ZX

GEN **ZX_renormalize**(GEN *x*, long *l*), as **normalizepol**, where $l = \lg(x)$, in place.

GEN **ZX_add**(GEN *x*, GEN *y*) adds *x* and *y*.

GEN **ZX_sub**(GEN *x*, GEN *y*) subtracts *x* and *y*.

GEN **ZX_neg**(GEN *x*, GEN *p*) returns $-x$.

GEN **ZX_Z_add**(GEN *x*, GEN *y*) adds the integer *y* to the polynomial *x*.

GEN **ZX_Z_mul**(GEN *x*, GEN *y*) multiplies the polynomial *x* by the integer *y*.

GEN **ZX_mul**(GEN *x*, GEN *y*) multiplies *x* and *y*.

GEN **ZX_sqr**(GEN *x*, GEN *p*) returns x^2 .

GEN **ZX_caract**(GEN *T*, GEN *A*, long *v*) returns the characteristic polynomial of $\text{Mod}(A, T)$, where *T* is a ZX, *A* is a ZX. More generally, *A* is allowed to be a QX, hence possibly has rational coefficients, *assuming* the result is a ZX, i.e. the algebraic number $\text{Mod}(A, T)$ is integral over \mathbb{Z} .

GEN **ZX_disc**(GEN *T*) returns the discriminant of the ZX *T*.

int **ZX_is_squarefree**(GEN *T*) returns 1 if the ZX *T* is squarefree, 0 otherwise.

GEN **ZX_resultant**(GEN *A*, GEN *B*) returns the resultant of the ZX *A* and *B*.

GEN **ZX_QX_resultant**(GEN *A*, GEN *B*) returns the resultant of the ZX *A* and the QX *B*, *assuming* the result is an integer.

GEN **ZY_ZXY_resultant**(GEN *A*, GEN *B*) under the assumption that *A* in $\mathbb{Z}[Y]$, *B* in $\mathbb{Q}[Y][X]$, and $R = \text{Res}_Y(A, B) \in \mathbb{Z}[X]$, returns the resultant *R*.

GEN **ZY_ZXY_rnfequation**(GEN *A*, GEN *B*, long **lambda*), assume *A* in $\mathbb{Z}[Y]$, *B* in $\mathbb{Q}[Y][X]$, and $R = \text{Res}_Y(A, B) \in \mathbb{Z}[X]$. If *lambda* = NULL, returns *R* as in **ZY_ZXY_resultant**. Otherwise, *lambda* must point to some integer, e.g. 0 which is used as a seed. The function then finds a small $\lambda \in \mathbb{Z}$ (starting from **lambda*) such that $R_\lambda(X) := \text{Res}_Y(A, B(X + \lambda Y))$ is squarefree, resets **lambda* to the chosen value and returns R_λ .

GEN **ZM_inv**(GEN *M*, GEN *d*) if *M* is a ZM and *d* is a t_INT such that $M' := dM^{-1}$ is integral, return M' . It is allowed to set *d* = NULL, in which case, the determinant of *M* is computed and used instead.

GEN **QM_inv**(GEN *M*, GEN *d*) as above, with *M* a QM. We still assume that M' has integer coefficients.

5.6.3 FpX. Let *p* an understood t_INT, to be given in the function arguments; in practice *p* is not assumed to be prime, but be wary. An Fp object is a t_INT belonging to $[0, p-1]$, an FpX is a t_POL in a fixed variable whose coefficients are Fp objects. Unless mentioned otherwise, all outputs in this section are FpXs. All operations are understood to take place in $(\mathbb{Z}/p\mathbb{Z})[X]$.

5.6.3.1 Basic operations. In what follows p is always a `t_INT`, not necessarily prime.

`GEN Rg_to_Fp(GEN z, GEN p)`, z a scalar which can be mapped to $\mathbf{Z}/p\mathbf{Z}$: a `t_INT`, a `t_INTMOD` whose modulus is divisible by p , a `t_FRAC` whose denominator is coprime to p , or a `t_PADIC` with underlying prime p . Returns `lift(z * Mod(1,p))`, normalized.

`GEN RgX_to_FpX(GEN z, GEN p)`, z a `t_POL`, returns the `FpX` obtained by applying `Rg_to_Fp` coefficientwise.

`GEN RgV_to_FpV(GEN z, GEN p)`, z a `t_VEC` or `t_COL`, returns the `FpV` (as a `t_VEC`) obtained by applying `Rg_to_Fp` coefficientwise.

`GEN RgC_to_FpC(GEN z, GEN p)`, z a `t_VEC` or `t_COL`, returns the `FpC` (as a `t_COL`) obtained by applying `Rg_to_Fp` coefficientwise.

`GEN FpX_to_mod(GEN z, GEN p)`, z a `ZX`. Returns `z * Mod(1,p)`, normalized. Hence the returned value has `t_INTMOD` coefficients.

`GEN FpX_red(GEN z, GEN p)`, z a `ZX`, returns `lift(z * Mod(1,p))`, normalized.

`GEN FpXV_red(GEN z, GEN p)`, z a `t_VEC` of `ZX`. Applies `FpX_red` componentwise and returns the result (and we obtain a vector of `FpXs`).

Now, except for p , the operands and outputs are all `FpX` objects. Results are undefined on other inputs.

`GEN FpX_add(GEN x, GEN y, GEN p)` adds x and y .

`GEN FpX_neg(GEN x, GEN p)` returns $-x$.

`GEN FpX_renormalize(GEN x, long l)`, as `normalizepol`, where $l = \lg(x)$, in place.

`GEN FpX_sub(GEN x, GEN y, GEN p)` subtracts y from x .

`GEN FpX_mul(GEN x, GEN y, GEN p)` multiplies x and y .

`GEN FpX_sqr(GEN x, GEN p)` returns x^2 .

`GEN FpX_divrem(GEN x, GEN y, GEN p, GEN *pr)` returns the quotient of x by y , and sets `pr` to the remainder.

`GEN FpX_div(GEN x, GEN y, GEN p)` returns the quotient of x by y .

`GEN FpX_div_by_X_x(GEN A, GEN a, GEN p, GEN *r)` returns the quotient of the `FpX` A by $(X - a)$, and sets `r` to the remainder $A(a)$.

`GEN FpX_rem(GEN x, GEN y, GEN p)` returns the remainder $x \bmod y$

`GEN FpX_gcd(GEN x, GEN y, GEN p)` returns a (not necessarily monic) greatest common divisor of x and y .

`GEN FpX_extgcd(GEN x, GEN y, GEN p, GEN *u, GEN *v)` returns $d = \text{GCD}(x, y)$, and sets `*u`, `*v` to the Bezout coefficients such that $*ux + *vy = d$.

`GEN FpX_center(GEN z, GEN p)` returns the polynomial whose coefficient belong to the symmetric residue system (clean version of `centermod`, which assumes the coefficients already belong to $[0, p - 1]$).

5.6.3.2 Miscellaneous operations

GEN FpX_normalize(GEN z, GEN p) divides the FpX z by its leading coefficient. If the latter is 1, z itself is returned, not a copy. If not, the inverse remains uncollected on the stack.

GEN FpX_Fp_add(GEN y, GEN x, GEN p) add the Fp x to the FpX y, possibly modifying the argument y (thus the operation uses constant time instead of linear linear). This function is not suitable for gerepileupto nor for gerepile.

GEN FpX_Fp_mul(GEN y, GEN x, GEN p) multiplies the FpX y by the Fp x.

GEN FpX_rescale(GEN P, GEN h, GEN p) returns $h^{\deg(P)}P(x/h)$. P is an FpX and h is a non-zero Fp (the routine would work with any non-zero t_INT but is not efficient in this case).

GEN FpX_eval(GEN x, GEN y, GEN p) evaluates the FpX x at the Fp y. The result is an Fp.

GEN FpXV_FpC_mul(GEN V, GEN W, GEN p) multiplies a line vector of FpX by a column vector of Fp (scalar product). The result is an FpX.

GEN FpXV_prod(GEN V, GEN p), V being a vector of FpX, returns their product.

GEN FpV_roots_to_pol(GEN V, GEN p, long v), V being a vector of INTs, returns the monic FpX $\prod_i (\text{pol_x}[v] - V[i])$.

GEN FpX_chinese_coprime(GEN x, GEN y, GEN Tx, GEN Ty, GEN Tz, GEN p) returns an FpX, congruent to x mod Tx and to y mod Ty. Assumes Tx and Ty are coprime, and Tz = Tx * Ty or NULL (in which case it is computed within).

GEN FpV_polint(GEN x, GEN y, GEN p) returns the FpX interpolation polynomial with value y[i] at x[i]. Assumes lengths are the same, components are t_INTs, and the x[i] are distinct modulo p.

long FpX_is_squarefree(GEN f, GEN p) returns 1 if the FpX f is squarefree, 0 otherwise.

long FpX_is_irred(GEN f, GEN p) returns 1 if the FpX f is irreducible, 0 otherwise. Assumes that p is prime. If f has few factors, **FpX_nbfact**(f,p) == 1 is much faster.

long FpX_is_totally_split(GEN f, GEN p) returns 1 if the FpX f splits into a product of distinct linear factors, 0 otherwise. Assumes that p is prime.

GEN FpX_factor(GEN f, GEN p), factors the FpX f. Assumes that p is prime. The returned value v has two components: v[1] is a vector of distinct irreducible (FpX) factors, and v[2] is a t_VECSMALL of corresponding exponents. The order of the factors is deterministic (the computation is not).

long FpX_nbfact(GEN f, GEN p), assuming the FpX f is squarefree, returns the number of its irreducible factors. Assumes that p is prime.

long FpX_degfact(GEN f, GEN p), as **FpX_factor**, but the degrees of the irreducible factors are returned instead of the factors themselves (as a t_VECSMALL). Assumes that p is prime.

long FpX_nbroots(GEN f, GEN p) returns the number of distinct roots in $\mathbf{Z}/p\mathbf{Z}$ of the FpX f. Assumes that p is prime.

GEN FpX_roots(GEN f, GEN p) returns the roots in $\mathbf{Z}/p\mathbf{Z}$ of the FpX f (without multiplicity, as a vector of Fps). Assumes that p is prime.

GEN FpX_rand(long d, long v, GEN p) returns a random FpX in variable v, of degree less than d.

GEN **FpX_resultant**(GEN x, GEN y, GEN p) returns the resultant of x and y, both FpX. The result is a t_INT belonging to $[0, p - 1]$.

GEN **FpY_FpXY_resultant**(GEN a, GEN b, GEN p), a a t_POL of t_INTs (say in variable Y), b a t_POL (say in variable X) whose coefficients are either t_POLs in $\mathbf{Z}[Y]$ or t_INTs. Returns $\text{Res}_Y(a, b)$, which is an FpX. The function assumes that Y has lower priority than X.

5.6.4 FpXQ, Fq. Let p a t_INT and T an FpX for p, both to be given in the function arguments; an FpXQ object is an FpX whose degree is strictly less than the degree of T. An Fq is either an FpXQ or an Fp. Both represent a class in $(\mathbf{Z}/p\mathbf{Z})[X]/(T)$, in which all operations below take place. In addition, Fq routines also allow T = NULL, in which case no reduction mod T is performed on the result.

For efficiency, the routines in this section may leave small unused objects behind on the stack (their output is still suitable for **gerepileupto**). Besides T and p, arguments are either FpXQ or Fq depending on the function name. (All Fq routines accept FpXQs by definition, not the other way round.)

GEN **Rg_to_FpXQ**(GEN z, GEN T, GEN p), z a GEN which can be mapped to $\mathbf{F}_p[X]/(T)$: anything Rg_to_Fp can be applied to, a t_POL to which RgX_to_FpX can be applied to, a t_POLMOD whose modulus is divisible by T (once mapped to a FpX), a suitable t_RFRAC. Returns z as an FpXQ, normalized.

GEN **RgX_to_FpXQX**(GEN z, GEN T, GEN p), z a t_POL, returns the FpXQ obtained by applying Rg_to_FpXQ coefficientwise.

GEN **RgX_to_FqX**(GEN z, GEN T, GEN p), z a t_POL, returns the FpXQ obtained by applying Rg_to_FpXQ coefficientwise and simplifying scalars to t_INTs.

GEN **Fq_red**(GEN x, GEN T, GEN p), x a ZX or t_INT, reduce it to an Fq (T = NULL is allowed iff x is a t_INT).

GEN **FqX_red**(GEN x, GEN T, GEN p), x a t_POL whose coefficients are ZXs or t_INTs, reduce them to Fqs. (If T = NULL, as FpXX_red(x, p).)

GEN **FqV_red**(GEN x, GEN T, GEN p), x a vector of ZXs or t_INTs, reduce them to Fqs. (If T = NULL, only reduce components mod p to FpXs or Fps.)

GEN **FpXQ_mul**(GEN y, GEN x, GEN T, GEN p)

GEN **FpXQ_sqr**(GEN y, GEN T, GEN p)

GEN **FpXQ_div**(GEN x, GEN y, GEN T, GEN p)

GEN **FpXQ_inv**(GEN x, GEN T, GEN p) computes the inverse of x

GEN **FpXQ_invsafe**(GEN x, GEN T, GEN p), as FpXQ_inv, returning NULL if x is not invertible.

GEN **FpXQ_pow**(GEN x, GEN n, GEN T, GEN p) computes x^n .

GEN **Fq_add**(GEN x, GEN y, GEN T/*unused*/, GEN p)

GEN **Fq_sub**(GEN x, GEN y, GEN T/*unused*/, GEN p)

GEN **Fq_mul**(GEN x, GEN y, GEN T, GEN p)

GEN **Fq_neg**(GEN x, GEN T, GEN p)

GEN **Fq_neg_inv**(GEN x, GEN T, GEN p) computes $-x^{-1}$

GEN Fq_inv(GEN x, GEN pol, GEN p) computes x^{-1} , raising an error if x is not invertible.
GEN Fq_invsafe(GEN x, GEN pol, GEN p) as **Fq_inv**, but returns NULL if x is not invertible.
GEN Fq_pow(GEN x, GEN n, GEN pol, GEN p) returns x^n .
GEN FpXQ_charpoly(GEN x, GEN T, GEN p) returns the characteristic polynomial of x
GEN FpXQ_minpoly(GEN x, GEN T, GEN p) returns the minimal polynomial of x
GEN FpXQ_powers(GEN x, long n, GEN T, GEN p) returns $[x^0, \dots, x^n]$ as a **t_VEC** of **FpXQs**.
GEN FpX_FpXQ_compo(GEN f, GEN x, GEN T, GEN p) returns $f(x)$.
GEN FpX_FpXQV_compo(GEN f, GEN V, GEN T, GEN p) returns $f(x)$, assuming that V was computed by **FpXQ_powers**(x, n, T, p).

5.6.5 FpXX. Contrary to what the name implies, an **FpXQX** is a **t_POL** whose coefficients are either **t_INTs** or **t_FpXs**. This reduce memory overhead at the expense of consistency.

GEN FpXX_add(GEN x, GEN y, GEN p) adds x and y.
GEN FpXX_red(GEN z, GEN p), z a **t_POL** whose coefficients are either **ZXs** or **t_INTs**. Returns the **t_POL** equal to z with all components reduced modulo p.
GEN FpXX_renormalize(GEN x, long l), as **normalizepol**, where $l = \lg(x)$, in place.

5.6.6 FpXQX, FqX. Contrary to what the name implies, an **FpXQX** is a **t_POL** whose coefficients are **Fqs**. So the only difference between **FqX** and **FpXQX** routines is that **T = NULL** is not allowed in the latter. (It was thought more useful to allow **t_INT** components than to enforce strict consistency, which would not imply any efficiency gain.)

5.6.6.1 Basic operations

GEN FqX_mul(GEN x, GEN y, GEN T, GEN p)
GEN FqX_Fq_mul(GEN P, GEN U, GEN T, GEN p) multiplies the **FqX** y by the **Fq** x.
GEN FqX_normalize(GEN z, GEN T, GEN p) divides the **FqX** z by its leading term.
GEN FqX_sqr(GEN x, GEN T, GEN p)
GEN FqX_divrem(GEN x, GEN y, GEN T, GEN p, GEN *z)
GEN FqX_div(GEN x, GEN y, GEN T, GEN p)
GEN FqX_rem(GEN x, GEN y, GEN T, GEN p)
GEN FqX_gcd(GEN P, GEN Q, GEN T, GEN p)
GEN FpXQX_red(GEN z, GEN T, GEN p) z a **t_POL** whose coefficients are **ZXs** or **t_INTs**, reduce them to **FpXQs**.
GEN FpXQX_mul(GEN x, GEN y, GEN T, GEN p)
GEN FpXQX_sqr(GEN x, GEN T, GEN p)
GEN FpXQX_divrem(GEN x, GEN y, GEN T, GEN p, GEN *pr)
GEN FpXQX_gcd(GEN x, GEN y, GEN T, GEN p)
GEN FpXQX_extgcd(GEN x, GEN y, GEN T, GEN p, GEN *ptu, GEN *ptv)

GEN FpXQYQ_pow(GEN x, GEN n, GEN S, GEN T, GEN p), x and T being FpXQXs, returns x^n modulo S.

GEN FpXQXV_prod(GEN V, GEN T, GEN p), V being a vector of FpXQX, returns their product.

GEN FqV_roots_to_pol(GEN V, GEN T, GEN p, long v), V being a vector of Fqs, returns the monic FqX $\prod_i (\text{pol_x}[v] - V[i])$.

5.6.6.2 Miscellaneous operations

GEN init_Fq(GEN p, long n, long v) returns an irreducible polynomial of degree n over \mathbf{F}_p , in variable v.

long FqX_is_squarefree(GEN P, GEN T, GEN p)

GEN FqX_factor(GEN x, GEN T, GEN p) same output convention as **FpX_factor**. Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

GEN FpX_factorff_irred(GEN P, GEN T, GEN p). Assumes p prime and T irreducible in $\mathbf{F}_p[X]$. P being an *irreducible* FpX, factors it over the finite field $\mathbf{F}_p[Y]/(T(Y))$ and returns the vector of irreducible FqXs factors (the exponents, being all equal to 1, are not included).

GEN FpX_ffisom(GEN P, GEN Q, GEN p). Assumes p prime, P, Q are ZXs, both irreducible mod p, and $\deg(P) \mid \deg(Q)$. Outputs a monomorphism between $\mathbf{F}_p[X]/(P)$ and $\mathbf{F}_p[X]/(Q)$, as a polynomial R such that $Q \mid P(R)$ in $\mathbf{F}_p[X]$. If P and Q have the same degree, it is of course an isomorphism.

void FpX_ffintersect(GEN P, GEN Q, long n, GEN p, GEN *SP, GEN *SQ, GEN MA, GEN MB) Assumes p is prime, P, Q are ZXs, both irreducible mod p, and n divides both the degree of P and Q. Compute SP and SQ such that the subfield of $\mathbf{F}_p[X]/(P)$ generated by SP and the subfield of $\mathbf{F}_p[X]/(Q)$ generated by SQ are isomorphic of degree n. The polynomials P and Q do not need to be of the same variable. If MA (resp. MB) is not NULL, it must be the matrix of the Frobenius map in $\mathbf{F}_p[X]/(P)$ (resp. $\mathbf{F}_p[X]/(Q)$).

GEN FpXQ_ffisom_inv(GEN S, GEN T, GEN p). Assumes p is prime, T a ZX, which is irreducible modulo p, S a ZX representing an automorphism of $\mathbf{F}_q := \mathbf{F}_p[X]/(T)$. (S(X) is the image of X by the automorphism.) Returns the inverse automorphism of S, in the same format, i.e. an FpX H such that $H(S) \equiv X$ modulo (T, p).

long FqX_nbfact(GEN u, GEN T, GEN p). Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

long FqX_nbroots(GEN f, GEN T, GEN p) Assumes p is prime and T irreducible in $\mathbf{F}_p[X]$.

5.6.7 FpV, FpM, FqM. A ZV (resp. a ZM) is a t_VEC or t_COL (resp. t_MAT) with t_INT coefficients. An FpV or FpM, with respect to a given t_INT p, is the same with Fp coordinates; operations are understood over $\mathbf{Z}/p\mathbf{Z}$. An FqM is a matrix with Fq coefficients (with respect to given T, p), not necessarily reduced (i.e arbitrary t_INTs and ZXs in the same variable as T).

5.6.7.1 Basic operations

GEN FpC_to_mod(GEN z, GEN p), z a ZC. Returns $\text{Col}(z) * \text{Mod}(1, p)$, hence a t_COL with t_INTMOD coefficients.

GEN FpV_to_mod(GEN z, GEN p), z a ZV. Returns $\text{Vec}(z) * \text{Mod}(1, p)$, hence a t_VEC with t_INTMOD coefficients.

GEN FpM_to_mod(GEN z, GEN p), z a ZM. Returns $z * \text{Mod}(1, p)$, hence with t_INTMOD coefficients.

GEN FpC_red(GEN *z*, GEN *p*), *z* a ZC. Returns `lift(Col(z) * Mod(1,p))`, hence a `t_COL`.

GEN FpV_red(GEN *z*, GEN *p*), *z* a ZV. Returns `lift(Vec(z) * Mod(1,p))`, hence a `t_VEC`

GEN FpM_red(GEN *z*, GEN *p*), *z* a ZM. Returns `lift(z * Mod(1,p))`, which is an `FpM`.

GEN FpC_Fp_mul(GEN *x*, GEN *y*, GEN *p*) multiplies the ZC *x* (seen as a column vector) by the `t_INT` *y* and reduce modulo *p* to obtain an `FpC`.

GEN FpC_FpV_mul(GEN *x*, GEN *y*, GEN *p*) multiplies the ZC *x* (seen as a column vector) by the ZV *y* (seen as a row vector, assumed to have compatible dimensions), and reduce modulo *p* to obtain an `FpM`.

GEN FpM_mul(GEN *x*, GEN *y*, GEN *p*) multiplies the two ZMs *x* and *y* (assumed to have compatible dimensions), and reduce modulo *p* to obtain an `FpM`.

GEN FpM_FpC_mul(GEN *x*, GEN *y*, GEN *p*) multiplies the ZM *x* by the ZC *y* (seen as a column vector, assumed to have compatible dimensions), and reduce modulo *p* to obtain an `FpC`.

GEN FpV_FpC_mul(GEN *x*, GEN *y*, GEN *p*) multiplies the ZV *x* (seen as a row vector) by the ZC *y* (seen as a column vector, assumed to have compatible dimensions), and reduce modulo *p* to obtain an `Fp`.

5.6.7.2 Fp-linear algebra. The implementations are not asymptotically efficient ($O(n^3)$ standard algorithms).

GEN FpM_deplin(GEN *x*, GEN *p*) returns a non-trivial kernel vector, or NULL if none exist.

GEN FpM_gauss(GEN *a*, GEN *b*, GEN *p*) as `gauss`

GEN FpM_image(GEN *x*, GEN *p*) as `image`

GEN FpM_intersect(GEN *x*, GEN *y*, GEN *p*) as `intersect`

GEN FpM_inv(GEN *x*, GEN *p*) returns the inverse of *x*, or NULL if *x* is not invertible.

GEN FpM_invimage(GEN *m*, GEN *v*, GEN *p*) as `inverseimage`

GEN FpM_ker(GEN *x*, GEN *p*) as `ker`

long FpM_rank(GEN *x*, GEN *p*) as `rank`

GEN FpM_indexrank(GEN *x*, GEN *p*) as `indexrank` but returns a `t_VECSMALL`

GEN FpM_suppl(GEN *x*, GEN *p*) as `suppl`

5.6.7.3 Fq-linear algebra

GEN FqM_gauss(GEN *a*, GEN *b*, GEN *T*, GEN *p*) as `gauss`

GEN FqM_ker(GEN *x*, GEN *T*, GEN *p*) as `ker`

GEN FqM_suppl(GEN *x*, GEN *T*, GEN *p*) as `suppl`

5.6.8 Flx Let p an understood `ulong`, assumed to be prime, to be given the the function arguments; an `F1` is an `ulong` belonging to $[0, p - 1]$, an `Flx` z is a `t_VECSMALL` representing a polynomial with small integer coefficients. Specifically $z[0]$ is the usual codeword, $z[1] = \text{evalvarn}(v)$ for some variable v , then the coefficients by increasing degree. An `FlxX` is a `t_POL` whose coefficients are `Flxs`.

In the following, an argument called `sv` is of the form `evalvarn(v)` for some variable number v .

5.6.8.1 Basic operations

`ulong Rg_to_F1(GEN z, ulong p)`, z which can be mapped to $\mathbf{Z}/p\mathbf{Z}$: a `t_INT`, a `t_INTMOD` whose modulus is divisible by p , a `t_FRAC` whose denominator is coprime to p , or a `t_PADIC` with underlying prime p . Returns `lift(z * Mod(1,p))`, normalized, as an `F1`.

`GEN Flx_red(GEN z, ulong p)` converts from zx with non-negative coefficients to `Flx` (by reducing them mod p).

`GEN Flx_add(GEN x, GEN y, ulong p)`

`GEN Flx_neg(GEN x, ulong p)`

`GEN Flx_neg_inplace(GEN x, ulong p)`, same as `Flx_neg`, in place (x is destroyed).

`GEN Flx_sub(GEN x, GEN y, ulong p)`

`GEN Flx_mul(GEN x, GEN y, ulong p)`

`GEN Flx_sqr(GEN x, ulong p)`

`GEN Flx_divrem(GEN x, GEN y, ulong p, GEN *pr)`

`GEN Flx_div(GEN x, GEN y, ulong p)`

`GEN Flx_rem(GEN x, GEN y, ulong p)`

`GEN Flx_deriv(GEN z, ulong p)`

`GEN Flx_gcd(GEN a, GEN b, ulong p)` returns a (not necessarily monic) greatest common divisor of x and y .

`GEN Flx_gcd_i(GEN a, GEN b, ulong p)`, same as `Flx_gcd` without collecting garbage.

`GEN Flx_extgcd(GEN a, GEN b, ulong p, GEN *ptu, GEN *ptv)`

`GEN Flx_pow(GEN x, long n, ulong p)`

5.6.8.2 Miscellaneous operations

`GEN Flx_normalize(GEN z, ulong p)`, as `FpX_normalize`.

`GEN Flx_Fl_mul(GEN y, ulong x, ulong p)`

`GEN Flx_recip(GEN x)`, returns the reciprocal polynomial

`ulong Flx_resultant(GEN a, GEN b, ulong p)`, returns the resultant of a and b

`ulong Flx_extresultant(GEN a, GEN b, ulong p, GEN *ptU, GEN *ptV)` returns the resultant and sets Bezout coefficients (if the resultant is 0, the latter are not set).

`GEN Flx_invmontgomery(GEN T, ulong p)`, returns the Montgomery inverse of T , i.e. `truncate(x / polrecip(T)+0(x^n)`. Assumes $T(0) \neq 0$.

GEN **Flx_rem_montgomery**(GEN x, GEN mg, GEN T, ulong p), returns x modulo T, where mg is the Montgomery inverse of T.

GEN **Flx_renormalize**(GEN x, long l), as **FpX_renormalize**, where $l = \lg(x)$, in place.

GEN **Flx_shift**(GEN T, long n), returns T multiplied by x^n .

long **Flx_valuation**(GEN x) returns the valuation of x, i.e. the multiplicity of the 0 root.

GEN **FlxYqQ_pow**(GEN x, GEN n, GEN S, GEN T, ulong p), as **FpXQYQ_pow**.

GEN **Flx_div_by_X_x**(GEN A, ulong a, ulong p, ulong *rem), returns the Euclidean quotient of the Flx A by $X - a$, and sets rem to the remainder A(a).

ulong **Flx_eval**(GEN x, ulong y, ulong p), as **FpX_eval**.

GEN **FlxV_Flc_mul**(GEN V, GEN W, ulong p), as **FpXV_FpC_mul**.

int **Flx_is_squarefree**(GEN z, ulong p)

long **Flx_nbfact**(GEN z, ulong p), as **FpX_nbfact**.

long **Flx_nbroots**(GEN f, ulong p), as **FpX_nbroots**.

GEN **Flv_polint**(GEN x, GEN y, ulong p, long sv) as **FpV_polint**, returning an Flx in variable v.

GEN **Flv_roots_to_pol**(GEN a, ulong p, long sv) as **FpV_roots_to_pol** returning an Flx in variable v.

5.6.9 Flxq. See FpXQ operations.

GEN **Flxq_mul**(GEN y, GEN x, GEN pol, ulong p)

GEN **Flxq_sqr**(GEN y, GEN pol, ulong p)

GEN **Flxq_inv**(GEN x, GEN T, ulong p)

GEN **Flxq_invsafe**(GEN x, GEN T, ulong p)

GEN **Flxq_pow**(GEN x, GEN n, GEN pol, ulong p)

GEN **Flxq_powers**(GEN x, long l, GEN T, ulong p)

GEN **FlxqV_roots_to_pol**(GEN V, GEN T, ulong p, long v) as **FqV_roots_to_pol** returning an FlxqX in variable v.

5.6.10 FlxX. See FpXX operations.

GEN **FlxX_add**(GEN P, GEN Q, ulong p)

GEN **FlxX_renormalize**(GEN x, long l), as **normalizepol**, where $l = \lg(x)$, in place.

GEN **FlxX_shift**(GEN a, long n)

5.6.11 FlxqX. See FpXQX operations.

```
GEN FlxqX_mul(GEN x, GEN y, GEN T, ulong p)
GEN FlxqX_Flxq_mul(GEN P, GEN U, GEN T, ulong p)
GEN FlxqX_normalize(GEN z, GEN T, ulong p)
GEN FlxqX_sqr(GEN x, GEN T, ulong p)
GEN FlxqX_divrem(GEN x, GEN y, GEN T, ulong p, GEN *pr)
GEN FlxqX_red(GEN z, GEN T, ulong p)
GEN FlxqXV_prod(GEN V, GEN T, ulong p)
GEN FlxqXQ_pow(GEN x, GEN n, GEN S, GEN T, ulong p)
```

5.6.12 Flv, Flm. See FpV, FpM operations.

```
GEN Flm_Flc_mul(GEN x, GEN y, ulong p)
GEN Flm_deplin(GEN x, ulong p)
GEN Flm_gauss(GEN a, GEN b, ulong p)
GEN Flm_indexrank(GEN x, ulong p)
GEN Flm_inv(GEN x, ulong p)
GEN Flm_ker(GEN x, ulong p)
GEN Flm_ker_sp(GEN x, ulong p, long deplin), as Flm_ker, in place (destroys x).
GEN Flm_mul(GEN x, GEN y, ulong p)
```

5.6.13 FlxqV, FlxqM. See FqV, FqM operations.

```
GEN FlxqM_ker(GEN x, GEN T, ulong p)
```

5.6.14 QX.

GEN **QXQ_inv**(GEN A, GEN B) returns the inverse of A modulo B where A and B are QXs.

5.6.15 RgX.

```
GEN RgX_add(GEN x, GEN y) adds x and y.
GEN RgX_sub(GEN x, GEN y) subtracts x and y.
GEN RgX_neg(GEN x, GEN p) returns -x.
```

The functions above are currently implemented through the generic routines, but it might change in the future.

GEN **RgX_mul**(GEN x, GEN y) multiplies the two **t_POL** (in the same variable) x and y. Uses Karatsuba algorithm.

GEN **RgX_mulspec**(GEN a, GEN b, long na, long nb). Internal routine: a and b are arrays of coefficients representing polynomials $\sum_{i=0}^{na} a[i]X^i$ and $\sum_{i=0}^{nb} b[i]X^i$. Returns their product (as a true GEN).

GEN **RgX_sqr**(GEN x) squares the t_POL x. Uses Karatsuba algorithm.

GEN **RgX_sqrsec**(GEN a, long na). Internal routine: a is an array of coefficients representing polynomial $\sum_{i=0}^{na} a[i]X^i$. Return its square (as a true GEN).

GEN **RgX_divrem**(GEN x, GEN y, GEN *r)

GEN **RgX_div**(GEN x, GEN y, GEN *r)

GEN **RgX_div_by_X_x**(GEN A, GEN a, GEN *r) returns the quotient of the RgX A by $(X - a)$, and sets r to the remainder A(a).

GEN **RgX_rem**(GEN x, GEN y, GEN *r)

GEN **RgX_mulXn**(GEN x, long n) returns $x * t^n$. This may be a t_FRAC if $n < 0$ and the valuation of x is not large enough.

GEN **RgX_shift**(GEN x, long n) returns $x * t^n$ if $n \geq 0$, and $x \backslash t^{-n}$ otherwise.

GEN **RgX_shift_shallow**(GEN x, long n) as **RgX_shift**, but shallow (coefficients are not copied). This is not suitable for gerepile or gerepileupto.

GEN **RgX_extgcd**(GEN x, GEN y, GEN *u, GEN *v) returns $d = \text{GCD}(x, y)$, and sets *u, *v to the Bezout coefficients such that $*ux + *vy = d$.

GEN **RgXQ_mul**(GEN y, GEN x, GEN T)

GEN **RgXQ_norm**(GEN x, GEN T) returns the norm of Mod(x, T).

GEN **RgXQ_sqr**(GEN x, GEN T)

GEN **RgXQ_powers**(GEN x, long n, GEN T, GEN p) returns $[x^0, \dots, x^n]$ as a t_VEC of RgXQs.

GEN **RgXQC_red**(GEN z, GEN T) z a vector whose coefficients are RgXs (arbitrary GENs in fact), reduce them to RgXQs (applying grem coefficientwise) in a t_COL.

GEN **RgXQV_red**(GEN z, GEN T) z a t_POL whose coefficients are RgXs (arbitrary GENs in fact), reduce them to RgXQs (applying grem coefficientwise) in a t_VEC.

GEN **RgXQX_red**(GEN z, GEN T) z a t_POL whose coefficients are RgXs (arbitrary GENs in fact), reduce them to RgXQs (applying grem coefficientwise).

GEN **RgXQX_mul**(GEN x, GEN y, GEN T)

GEN **RgX_Rg_mul**(GEN y, GEN x) multiplies the RgX y by the scalar x.

GEN **RgX_Rg_div**(GEN y, GEN x) divides the RgX y by the scalar x.

GEN **RgXQX_RgXQ_mul**(GEN x, GEN y, GEN T) multiplies the RgXQX y by the scalar (RgXQ) x.

GEN **RgXQX_sqr**(GEN x, GEN T)

GEN **RgXQX_divrem**(GEN x, GEN y, GEN T, GEN *pr)

GEN **RgXQX_div**(GEN x, GEN y, GEN T, GEN *r)

GEN **RgXQX_rem**(GEN x, GEN y, GEN T, GEN *r)

GEN **RgX_rescale**(GEN P, GEN h) returns $h^{\deg(P)} P(x/h)$. P is an RgX and h is non-zero. (Leaves small objects on the stack. Suitable but inefficient for gerepileupto.)

GEN **RgX_unscale**(GEN P, GEN h) returns $P(hx)$. (Leaves small objects on the stack. Suitable but inefficient for `gerepileupto`.)

GEN **RgXV_unscale**(GEN v, GEN h)

5.6.16 Conversions involving single precision objects

5.6.16.1 To single precision

GEN **ZX_to_Flx**(GEN x, ulong p) reduce ZX x modulo p (yielding an Flx).

GEN **ZV_to_Flv**(GEN x, ulong p) reduce ZV x modulo p (yielding an Flv).

GEN **ZXV_to_FlxV**(GEN v, ulong p), as **ZX_to_Flx**, repeatedly called on the vector's coefficients.

GEN **ZXX_to_FlxX**(GEN B, ulong p, long v), as **ZX_to_Flx**, repeatedly called on the polynomial's coefficients.

GEN **ZXXV_to_FlxXV**(GEN V, ulong p, long v), as **ZXX_to_FlxX**, repeatedly called on the vector's coefficients.

GEN **ZM_to_Flm**(GEN x, ulong p) reduce ZM x modulo p (yielding an Flm).

GEN **ZV_to_zv**(GEN z), converts coefficients using `itos`

GEN **ZV_to_nv**(GEN z), converts coefficients using `itou`

GEN **ZM_to_zm**(GEN z), converts coefficients using `itos`

GEN **FqC_to_FlxC**(GEN x, GEN T, GEN p), converts coefficients in Fq to coefficient in Flx, result being a column vector.

GEN **FqV_to_FlxV**(GEN x, GEN T, GEN p), converts coefficients in Fq to coefficient in Flx, result being a line vector.

GEN **FqM_to_FlxM**(GEN x, GEN T, GEN p), converts coefficients in Fq to coefficient in Flx.

5.6.16.2 From single precision

GEN **Flx_to_ZX**(GEN z), converts to ZX (`t_POL` of non-negative `t_INTs` in this case)

GEN **Flx_to_ZX_inplace**(GEN z), same as **Flx_to_ZX**, in place (z is destroyed).

GEN **FlxX_to_ZXX**(GEN B), converts an FlxX to a polynomial with ZX or `t_INT` coefficients (repeated calls to **Flx_to_ZX**).

GEN **FlxC_to_ZXC**(GEN x), converts a vector of Flx to a column vector of polynomials with `t_INT` coefficients (repeated calls to **Flx_to_ZX**).

GEN **FlxM_to_ZXM**(GEN z), converts a matrix of Flx to a matrix of polynomials with `t_INT` coefficients (repeated calls to **Flx_to_ZX**).

GEN **zx_to_ZX**(GEN z), as **Flx_to_ZX**, without assuming coefficients are non-negative.

GEN **Flc_to_ZC**(GEN z), converts to ZC (`t_COL` of non-negative `t_INTs` in this case)

GEN **Flv_to_ZV**(GEN z), converts to ZV (`t_VEC` of non-negative `t_INTs` in this case)

GEN **Flm_to_ZM**(GEN z), converts to ZM (`t_MAT` with non-negative `t_INTs` coefficients in this case)

GEN **zc_to_ZC**(GEN z) as **Flc_to_ZC**, without assuming coefficients are non-negative.

GEN **zv_to_ZV**(GEN z) as **Flv_to_ZV**, without assuming coefficients are non-negative.

GEN **zm_to_ZM**(GEN z) as **Flm_to_ZM**, without assuming coefficients are non-negative.

5.6.16.3 Mixed precision linear algebra Assumes dimensions are compatible. Multiply a multiprecision object by a single-precision one.

GEN **RgM_zc_mul**(GEN *x*, GEN *y*)

GEN **RgM_zm_mul**(GEN *x*, GEN *y*)

GEN **RgV_zc_mul**(GEN *x*, GEN *y*)

GEN **RgV_zm_mul**(GEN *x*, GEN *y*)

GEN **ZM_zc_mul**(GEN *x*, GEN *y*)

GEN **ZM_zm_mul**(GEN *x*, GEN *y*)

5.6.16.4 Miscellaneous

GEN **zero_Flx**(long *sv*) returns a zero Flx in variable *v*.

GEN **zero_zx**(long *sv*) as **zero_Flx**

GEN **polx_Flx**(long *sv*) returns the variable *v* as degree 1 Flx.

GEN **polx_zx**(long *sv*) as **polx_Flx**

GEN **Fl_to_Flx**(ulong *x*, long *sv*) converts a unsigned long to a scalar Flx in shifted variable *sv*.

GEN **Z_to_Flx**(GEN *x*, ulong *p*, long *v*) converts a t_INT to a scalar polynomial in variable *v*.

GEN **Flx_to_Flv**(GEN *x*, long *n*) converts from Flx to Flv with *n* components (assumed larger than the number of coefficients of *x*).

GEN **zx_to_zv**(GEN *x*, long *n*) as **Flx_to_Flv**.

GEN **Flv_to_Flx**(GEN *x*, long *sv*) converts from vector (coefficient array) to (normalized) polynomial in variable *v*.

GEN **zv_to_zx**(GEN *x*, long *n*) as **Flv_to_Flx**.

GEN **matid_Flm**(long *n*) returns an Flm which is an $n \times n$ identity matrix.

GEN **Flm_to_FlxV**(GEN *x*, long *sv*) converts the columns of Flm *x* to an array of Flx (repeated calls to **Flv_to_Flx**).

GEN **zm_to_zxV**(GEN *x*, long *n*) as **Flm_to_FlxV**.

GEN **Flm_to_FlxX**(GEN *x*, long *sv*, long *w*) converts the columns of Flm *x* to the coefficient of an FlxX, and normalize the result.

GEN **FlxV_to_Flm**(GEN *v*, long *n*) reverse **Flm_to_FlxV**, to obtain an Flm with *n* rows (repeated calls to **Flx_to_Flv**).

GEN **FlxX_to_Flm**(GEN *v*, long *n*) reverse **Flm_to_FlxX**, to obtain an Flm with *n* rows (repeated calls to **Flx_to_Flv**).

5.7 Operations on general PARI objects.

5.7.1 Assignment

`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`.

5.7.2 Conversions

5.7.2.1 Scalars

`double rtodbl(GEN x)` applied to a `t_REAL` `x`, converts `x` into a `double` if possible.

`GEN dbltor(double x)` converts the `double x` into a `t_REAL`.

`double gtodouble(GEN x)` if `x` is a real number (not necessarily a `t_REAL`), converts `x` into a `double` if possible.

`long gtolong(GEN x)` if `x` is an integer (not necessarily a `t_INT`), converts `x` into a `long` if possible.

`GEN fractor(GEN x, long l)` applied to a `t_FRAC` `x`, converts `x` into a `t_REAL` of length `prec`.

`GEN quadtoc(GEN x, long l)` applied to a `t_QUAD` `x`, converts `x` into a `t_REAL` or `t_COMPLEX` depending on the sign of the discriminant of `x`, to precision `l` BIL-bit words.
line brk at hyphen here
[GN]

`GEN ctofp(GEN x, long prec)` converts the `t_COMPLEX` `x` to a complex whose real and imaginary parts are `t_REAL` of length `prec`, using `gtofp`;

`GEN gtofp(GEN x, long prec)` converts the complex number `x` (`t_INT`, `t_REAL`, `t_FRAC`, `t_QUAD` or `t_COMPLEX`) to either a `t_REAL` or `t_COMPLEX` whose components are `t_REAL` of length `prec`.

`GEN gcvtop(GEN x, GEN p, long l)` converts `x` into a `t_PADIC` `p`-adic number of precision `l`.

`GEN gprec(GEN x, long l)` returns a copy of `x` whose precision is changed to `l` digits. The precision change is done recursively on all components of `x`. Digits means *decimal*, *p*-adic and *X*-adic digits for `t_REAL`, `t_SER`, `t_PADIC` components, respectively.

`GEN gprec_w(GEN x, long l)` returns a shallow copy of `x` whose `t_REAL` components have their precision changed to `l words`. This is often more useful than `gprec`. Shallow copy means that unaffected components are not copied; in particular, this function is not suitable for `gerepileupto`.

`GEN gprec_wtrunc(GEN x, long l)` returns a shallow copy of `x` whose `t_REAL` components have their precision *truncated* to `l words`. Contrary to `gprec_w`, this function may never increase the precision of `x`. Shallow copy means that unaffected components are not copied; in particular, this function is not suitable for `gerepileupto`.

5.7.2.2 Modular objects

`GEN gmodulo(GEN x, GEN y)` creates the object `Mod(x,y)` on the PARI stack, where `x` and `y` are either both `t_INTs`, and the result is a `t_INTMOD`, or `x` is a scalar or a `t_POL` and `y` a `t_POL`, and the result is a `t_POLMOD`.

`GEN gmodulgs(GEN x, long y)` same as `gmodulo` except `y` is a `long`.

`GEN gmodulss(long x, long y)` same as `gmodulo` except both `x` and `y` are `longs`.

5.7.2.3 Between polynomials and coefficient arrays

GEN **gtopoly**(GEN x , long v) converts or truncates the object x into a t_POL with main variable number v . A common application would be the conversion of coefficient vectors (coefficients are given by decreasing degree). E.g. $[2,3]$ goes to $2*v + 3$

GEN **gtopolyrev**(GEN x , long v) converts or truncates the object x into a t_POL with main variable number v , but vectors are converted in reverse order compared to **gtopoly** (coefficients are given by increasing degree). E.g. $[2,3]$ goes to $3*v + 2$. In other words the vector represents a polynomial in the basis $(1, v, v^2, v^3, \dots)$.

GEN **normalizepol**(GEN x) applied to an unnormalized t_POL x (with all coefficients correctly set except that **leading_term**(x) might be zero), normalizes x correctly in place and returns x . For internal use.

The following routines do *not* copy coefficients on the stack (they only move pointers around), hence are very fast but not suitable for **gerepile** calls. Recall that an **RgV** (resp. an **RgX**, resp. an **RgM**) is a t_VEC or t_COL (resp. a t_POL , resp. a t_MAT) with arbitrary components. Similarly, an **RgXV** is a t_VEC or t_COL with **RgX** components, etc.

GEN **RgV_to_RgX**(GEN x , long v) converts the **RgV** x to a (normalized) polynomial in variable v (as **gtopolyrev**, without copy).

GEN **RgX_to_RgV**(GEN x , long N) converts the t_POL x to a t_COL v with N components. Other types than t_POL are allowed for x , which is then considered as a constant polynomial. Coefficients of x are listed by increasing degree, so that $y[i]$ is the coefficient of the term of degree $i - 1$ in x .

GEN **RgM_to_RgXV**(GEN x , long v) converts the **RgM** x to a t_VEC of **RgX**, by repeated calls to **RgV_to_RgX**.

GEN **RgXV_to_RgM**(GEN v , long N) converts the vector of **RgX** v to a t_MAT with N rows, by repeated calls to **RgX_to_RgV**.

GEN **RgM_to_RgXX**(GEN x , long v , long w) converts the **RgM** x into a t_POL in variable v , whose coefficients are t_POL s in variable w . This is a shortcut for

RgV_to_RgX(**RgM_to_RgXV**(x , w), v);

There are no consistency checks with respect to variable priorities: the above is an invalid object if **varncmp**(v, w) ≥ 0 .

GEN **RgXX_to_RgM**(GEN x , long N) converts the t_POL x with **RgX** (or constant) coefficients to a matrix with N rows.

GEN **RgXY_swap**(GEN P , long n , long w) converts the bivariate polynomial $P(u, v)$ (a t_POL with t_POL coefficients) to $P(\text{pol_x}[w], u)$, assuming n is an upper bound for $\deg_v(P)$.

GEN **greffe**(GEN x , long l , int **use_stack**) applied to a t_POL x , creates a t_SER of length l starting with x , but without actually copying the coefficients, just the pointers. If **use_stack** is 0, this is created through **malloc**, and must be freed after use. Intended for internal use only.

GEN **gtoser**(GEN x , long v) converts the object x into a t_SER with main variable number v .

GEN **gtocol**(GEN x) converts the object x into a t_COL

GEN **gtomat**(GEN x) converts the object x into a t_MAT .

GEN **gtovec**(GEN x) converts the object x into a t_VEC .

GEN **gtovecsmall**(GEN *x*) converts the object *x* into a *t_VECSMALL*.

GEN **normalize**(GEN *x*) applied to an unnormalized *t_SER* *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.

5.7.3 Clean Constructors

GEN **zeropadic**(GEN *p*, long *n*) creates a 0 *t_PADIC* equal to $O(p^n)$.

GEN **zeroser**(long *v*, long *n*) creates a 0 *t_SER* in variable *v* equal to $O(X^n)$.

GEN **scalarser**(GEN *x*, long *v*, long *prec*) creates a constant *t_SER* in variable *v* and precision *prec*, whose constant coefficient is (a copy of) *x*, in other words $x + O(v^{\text{prec}})$. Assumes that *x* is non-zero.

GEN **zeropol**(long *v*) creates a 0 *t_POL* in variable *v*.

GEN **scalarpol**(GEN *x*, long *v*) creates a constant *t_POL* in variable *v*, whose constant coefficient is (a copy of) *x*.

GEN **zerocol**(long *n*) creates a *t_COL* with *n* components set to *gen_0*.

GEN **zerovec**(long *n*) creates a *t_VEC* with *n* components set to *gen_0*.

GEN **col_ei**(long *n*, long *i*) creates a *t_COL* with *n* components set to *gen_0*, but the *i*-th one which is set to *gen_1* (*i*-th vector in the canonical basis).

GEN **vec_ei**(long *n*, long *i*) creates a *t_VEC* with *n* components set to *gen_0*, but the *i*-th one which is set to *gen_1* (*i*-th vector in the canonical basis).

GEN **zeromat**(long *m*, long *n*) creates a *t_MAT* with *m* x *n* components set to *gen_0*. Note that the result allocates a *single* column, so modifying an entry in one column modifies it in all columns. To fully allocate a matrix initialized with zero entries, use **zeromatcopy**.

GEN **zeromatcopy**(long *m*, long *n*) creates a *t_MAT* with *m* x *n* components set to *gen_0*. Note that

See also next section for analogs of the following functions:

GEN **mkcolcopy**(GEN *x*) creates a 1-dimensional *t_COL* containing *x*.

GEN **mkmatcopy**(GEN *x*) creates a 1-by-1 *t_MAT* containing *x*.

GEN **mkveccopy**(GEN *x*) creates a 1-dimensional *t_VEC* containing *x*.

GEN **mkvec2copy**(GEN *x*, GEN *y*) creates a 2-dimensional *t_VEC* equal to [*x*,*y*].

GEN **mkvecs**(long *x*) creates a 1-dimensional *t_VEC* containing *stoi(x)*.

GEN **mkvec2s**(long *x*, long *y*) creates a 2-dimensional *t_VEC* containing [*stoi(x)*, *stoi(y)*].

GEN **mkvec3s**(long *x*, long *y*, long *z*) creates a 3-dimensional *t_VEC* containing [*stoi(x)*, *stoi(y)*, *stoi(z)*].

GEN **mkvecsmall**(long *x*) creates a 1-dimensional *t_VECSMALL* containing *x*.

GEN **mkvecsmall2**(long *x*, long *y*) creates a 2-dimensional *t_VECSMALL* containing [*x*, *y*].

GEN **mkvecsmall3**(long *x*, long *y*, long *z*) creates a 3-dimensional *t_VECSMALL* containing [*x*, *y*, *z*].

5.7.4 Unclean Constructors

Contrary to the policy of general PARI functions, the functions in this subsection do *not* copy their arguments, nor do they produce an object a priori suitable for `gerepileupto`. In particular, they are faster than their clean equivalent (which may not exist). *If* you restrict their arguments to universal objects (e.g `gen_0`), then the above warning does not apply.

GEN `mkcomplex`(GEN `x`, GEN `y`) creates the `t_COMPLEX` $x + iy$.

GEN `mkfrac`(GEN `x`, GEN `y`) creates the `t_FRAC` x/y . Assumes that $y > 1$ and $(x, y) = 1$.

GEN `mkrfac`(GEN `x`, GEN `y`) creates the `t_RFRAC` x/y . Assumes that y is a `t_POL`, x a compatible type whose variable has lower or same priority, with $(x, y) = 1$.

GEN `mkcol`(GEN `x`) creates a 1-dimensional `t_COL` containing `x`.

GEN `mkintmod`(GEN `x`, GEN `y`) creates the `t_INTMOD` $\text{Mod}(x, y)$. The input must be `t_INTs` satisfying $0 \leq x < y$.

GEN `mkpolmod`(GEN `x`, GEN `y`) creates the `t_POLMOD` $\text{Mod}(x, y)$. The input must satisfy $\deg x < \deg y$ with respect to the main variable of the `t_POL` y . x may be a scalar.

GEN `mkmat`(GEN `x`) creates a 1-by-1 `t_MAT` containing `x`.

GEN `mkvec`(GEN `x`) creates a 1-dimensional `t_VEC` containing `x`.

GEN `mkvec2`(GEN `x`, GEN `y`) creates a 2-dimensional `t_VEC` equal to $[x, y]$.

GEN `mkvec3`(GEN `x`, GEN `y`, GEN `z`) creates a 3-dimensional `t_VEC` equal to $[x, y, z]$.

GEN `mkvec4`(GEN `x`, GEN `y`, GEN `z`, GEN `t`) creates a 4-dimensional `t_VEC` equal to $[x, y, z, t]$.

GEN `mkintn`(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 `sizeof(long)` is irrelevant, the behaviour is also as above on 64-bit machines).

```
mkintn(3, a2, a1, a0);
```

returns $a_2 2^{64} + a_1 2^{32} + a_0$.

GEN `mkpoln`(long `n`, ...) Returns the `t_POL` whose n coefficients (GEN) follow, in order of decreasing degree.

```
mkpoln(3, gen_1, gen_2, gen_0);
```

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, hence *one more* than the degree.

GEN `mkvecn`(long `n`, ...) returns the `t_VEC` whose n coefficients (GEN) follow.

GEN `mkcoln`(long `n`, ...) returns the `t_COL` whose n coefficients (GEN) follow.

5.7.5 Integer parts

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 towards $+\infty$ the components of *x* to the nearest integers.

GEN **grndtoi**(GEN *x*, long **e*) same as **ground**, but in addition sets **e* to the binary exponent of $x - \text{ground}(x)$. If this is positive, all significant bits are 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 a real number (i.e. the unique integer closest to *x* among those between 0 and *x*). If *x* is a **t_SER**, it is truncated to a **t_POL**; if *x* is a **t_RFRAC**, this takes the polynomial part.

GEN **gcvttoi**(GEN *x*, long **e*) same as **grndtoi** except that rounding is replaced by truncation.

5.7.6 Valuation and shift

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 **t_INT** 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 a **t_INT**: **gshift** truncates, while **gmul2n** creates a fraction if necessary.

long **ggval**(GEN *x*, GEN *p*) returns the greatest exponent *e* such that 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 **t_POL** *x*.

long **polvaluation**(GEN *P*, GEN **z*) returns the valuation *v* of the **t_POL** *P* with respect to its main variable *X*. Check whether coefficients are 0 using **gcmp0**. If *z* is non-NULL, set it to P/X^v .

long **polvaluation_inexact**(GEN *P*, GEN **z*) as **polvaluation** but use **isexactzero** instead of **gcmp0**.

long **ZX_valuation**(GEN *P*, GEN **z*) as **polvaluation**, but assumes *P* has **t_INT** coefficients.

5.7.7 Comparison operators

int **isexactzero**(GEN *x*) returns 1 (true) if *x* is exactly equal to 0, 0 (false) otherwise. Note that many PARI functions return a pointer to **gen_0** 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 **isinexact**(GEN *x*) returns 0 (false) if *x* has an inexact component, and 1 (true) otherwise.

int **isint**(GEN *x*, GEN **n*) returns 0 (false) if *x* does not round to an integer. Otherwise, returns 1 (true) and set *n* to the rounded value.

int **issmall**(GEN *x*, long **n*) returns 0 (false) if *x* does not round to a small integer (suitable for **itos**). Otherwise, returns 1 (true) and set *n* to the rounded value.

`int gcmp0(GEN x)` returns 1 (true) if `x` is equal to 0, 0 (false) otherwise.

`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 gequal(GEN x, GEN y)` returns 1 (true) if `x` is equal to `y`, 0 otherwise. A priori, this makes sense only if `x` and `y` have the same type. When the types are different, a `true` result means that $x - y$ was successfully computed and found equal to 0 (by `gcmp0`). In particular

```
gequal(cgetg(1, t_VEC), gen_0)
```

is true, and the relation is not transitive. E.g. an empty `t_COL` and an empty `t_VEC` are not equal but are both equal to `gen_0`.

`long gequalsg(long s, GEN x)` returns 1 (true) if the `long s` is equal to `x`, 0 otherwise.

`long gequalgs(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.

5.7.8 Generic 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`.

5.7.9 Divisibility, Euclidean division

`GEN gdivexact(GEN x, GEN y)` returns the quotient x/y , assuming `y` divides `x`.

`int gdvd(GEN x, GEN y)` returns 1 (true) if `y` divides `x`, 0 otherwise.

`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 gdivent[z](GEN x, GEN y[, GEN z])` yields the true Euclidean quotient of `x` and the `t_INT` or `t_POL` `y`.

`GEN gdiventsg[z](long s, GEN y[, GEN z])`, as `gdivent` except that `x` is a `long`.

`GEN gdiventgs[z](GEN x, long s[, GEN z])`, as `gdivent` except that `y` is a `long`.

GEN **gmod**[z](GEN x, GEN y[, GEN z]) yields the true remainder of x modulo the t_INT or t_POL y. A t_REAL or t_FRAC y is also allowed, in which case the remainder is the unique real r such that $0 \leq r < |y|$ and $y = qx + r$ for some (in fact unique) integer q .

GEN **gmodsg**[z](long s, GEN y[, GEN z]) as **gmod**, except x is a long.

GEN **gmodgs**[z](GEN x, long s[, GEN z]) as **gmod**, except y is a long.

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 **poldivrem**(GEN x, GEN y, GEN *r) same as **gdivmod** but specifically for t_POLs x and y, not necessarily in the same variable. Either of x and y may also be scalars (treated as polynomials of degree 0)

GEN **gdeuc**(GEN x, GEN y) creates the Euclidean quotient of the t_POLs x and y. Either of x and y may also be scalars (treated as polynomials of degree 0)

GEN **grem**(GEN x, GEN y) creates the Euclidean remainder of the t_POL x divided by the t_POL y.

GEN **gdivround**(GEN x, GEN y) if x and y are t_INT, as **diviiround**. Operate componentwise if x is a t_COL, t_VEC or t_MAT. Otherwise as **gdivent**.

GEN **centermod.i**(GEN x, GEN y, GEN y2), as **centermodii**, componentwise.

GEN **centermod**(GEN x, GEN y), as **centermod.i**, except that y2 is computed (and left on the stack for efficiency).

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 a t_POL).

5.7.10 GCD, content and primitive part

GEN **subres**(GEN x, GEN y) creates the resultant of the t_POLs x and y computed using the sub-resultant algorithm. Either of x and y may also be scalars (treated as polynomials of degree 0)

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 **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 $ux + vy = \gcd(x, y)$ into *u and *v.

GEN **bezoutpol**(GEN a, GEN b, GEN *u, GEN *v), returns the GCD d of t_INTs a and b and sets u, v to the Bezout coefficients such that $au + bv = d$.

GEN **content**(GEN x) creates the GCD of all the components of x.

GEN **primitive_part**(GEN x, GEN *c), sets c to **content**(x) and returns the primitive part x / c .

GEN **primpart**(GEN x) as **primitive_part** but the content is lost. (For efficiency, the content remains on the stack.)

5.7.11 Generic binary operators. Let “*op*” be a binary operation among

op=**add**: addition ($x + y$).
op=**sub**: subtraction ($x - y$).
op=**mul**: multiplication ($x * y$).
op=**div**: division (x / y).
op=**max**: maximum ($\max(x, y)$)
op=**min**: minimum ($\min(x, y)$)

The names and prototypes of the functions corresponding to *op* are as follows:

GEN **gop**[**z**](GEN *x*, GEN *y*[, GEN *z*])

GEN **gopgs**[**z**](GEN *x*, long *s*[, GEN *z*])

GEN **gopsg**[**z**](long *s*, GEN *y*[, GEN *z*])

GEN **gpow**(GEN *x*, GEN *y*, long *l*) creates x^y . If *y* is a `t_INT`, return **powgi**(*x*,*y*) (the precision *l* is not taken into account). Otherwise, the result is $\exp(y * \log(x))$ computed to precision *l*.

GEN **gpowgs**(GEN *x*, long *n*) creates x^n using binary powering.

GEN **powgi**(GEN *x*, GEN *y*) creates x^y , where *y* is a `t_INT`, using left-shift binary powering.

GEN **gsubst**(GEN *x*, long *v*, GEN *y*) substitutes the object *y* into *x* for the variable number *v*.

5.7.12 Miscellaneous functions

`const char* type_name(long t)` given a type number *t* this routine returns a string containing its symbolic name. E.g `type_name(t_INT)` returns “`t_INT`”. The return value is read-only.

5.8 Further type specific functions.

5.8.1 Vectors and Matrices See Section 5.7.3 and Section 5.7.4 for various useful constructors. Coefficients are accessed and set using **gel**, **gcoeff**, see Section 5.2.6. There are many internal functions to extract or manipulate subvectors or submatrices but, like the accessors above, none of them are suitable for **gerepileupto**. Worse, there are no type verification, nor bound checking, so use at your own risk.

Note. In the function names below, i stands for *interval* and p for *permutation*.

GEN shallowcopy(GEN x) returns a t_GEN whose components are the components of x (no copy is made). The result may now be used to compute in place without destroying x . This is essentially equivalent to

```
GEN y = cgetg(lg(x), typ(x));
for (i = 1; i < lg(x); i++) y[i] = x[i];
return y;
```

except that t_POLMOD (resp. t_MAT) are treated specially since a dummy copy of the representative (resp. all columns) is also made.

GEN shallowtrans(GEN x) returns the transpose of x , *without* copying its components, i. e., it returns a GEN whose components are (physically) the components of x . This is the internal function underlying **gtrans**.

GEN shallowconcat(GEN x , GEN y) concatenate x and y , *without* copying components, i. e., it returns a GEN whose components are (physically) the components of x and y .

GEN vconcat(GEN A , GEN B) concatenate vertically the two t_MAT A and B of compatible dimensions. A NULL pointer is accepted for an empty matrix. See **shallowconcat**.

GEN row(GEN A , long i) return $A[i,]$, the i -th row of the t_MAT A .

GEN row_i(GEN A , long i , long j_1 , long j_2) return part of the i -th row of t_MAT A : $A[i, j_1]$, $A[i, j_1 + 1] \dots, A[i, j_2]$. Assume $j_1 \leq j_2$.

GEN rowslice(GEN A , long i_1 , long i_2) return the t_MAT formed by the i_1 -th through i_2 -th rows of t_MAT A . Assume $i_1 \leq i_2$.

GEN rowpermute(GEN A , GEN p), p being a $t_VECSMALL$ representing a list $[p_1, \dots, p_n]$ of rows of t_MAT A , returns the matrix whose rows are $A[p_1,], \dots, A[p_n,]$.

GEN rowslicepermute(GEN A , GEN p , long x_1 , long x_2), short for

```
rowslice(rowpermute(A,p), x1, x2)
```

(more efficient).

GEN vecslice(GEN A , long j_1 , long j_2), return $A[j_1], \dots, A[j_2]$. If A is a t_MAT , these correspond to *columns* of A . The object returned has the same type as A (t_VEC , t_COL or t_MAT). Assume $j_1 \leq j_2$.

GEN vecpermute(GEN A , GEN p) p is a $t_VECSMALL$ representing a list $[p_1, \dots, p_n]$ of indices. Returns a GEN which has the same type as A (t_VEC , t_COL or t_MAT), and whose components are $A[p_1], \dots, A[p_n]$. If A is a t_MAT , these are the *columns* of A .

GEN vecslicepermute(GEN A , GEN p , long y_1 , long y_2) short for

```
vecslice(vecpermute(A,p), y1, y2)
```

(more efficient).

5.8.2 Low-level vectors and columns functions

These functions handle t_VEC as an abstract container type of GENs. No specific meaning is attached to the content.

They accept both t_VEC and t_COL as input, but **col** functions always return t_COL and **vec** functions always return t_VEC .

Note. All the functions below are shallow.

GEN **const_col**(long n, long c) returns a **t_COL** of n components equal to c.

GEN **const_vec**(long n, long c) returns a **t_VEC** of n components equal to c.

int **vec_isconst**(GEN v) Returns 1 if all the components of v are equal, else returns 0.

int **vec_is1to1**(GEN v) Returns 1 if the components of v are pair-wise distinct, i.e. if $i \mapsto v[i]$ is a 1-to-1 mapping, else returns 0.

GEN **vec_shorten**(GEN v, long n) shortens the vector v to n components.

GEN **vec_lengthen**(GEN v, long n) lengthens the vector v to n components. The extra components are not initialised.

5.8.3 Function to handle **t_VECSMALL**

Theses functions handle **t_VECSMALL** as an abstract container type of small signed integers. No specific meaning is attached to the content.

GEN **const_vecsmall**(long n, long c) returns a **t_VECSMALL** of n components equal to c.

GEN **vec_to_vecsmall**(GEN z) identical to **ZV_to_zv**(z).

GEN **vecsmall_to_vec**(GEN z) identical to **zv_to_ZV**(z).

GEN **vecsmall_to_col**(GEN z) identical to **zv_to_ZC**(z).

GEN **vecsmall_copy**(GEN x) makes a copy of x on the stack.

GEN **vecsmall_shorten**(GEN v, long n) shortens the **t_VECSMALL** v to n components.

GEN **vecsmall_lengthen**(GEN v, long n) lengthens the **t_VECSMALL** v to n components. The extra components are not initialised.

GEN **vecsmall_indexsort**(GEN x) performs an indirect sort of the components of the **t_VECSMALL** x and return a permutation stored in a **t_VECSMALL**.

void **vecsmall_sort**(GEN v) sorts the **t_VECSMALL** v in place.

GEN **vecsmall_uniq**(GEN v) given a sorted **t_VECSMALL** v, return the vector of unique occurrences.

int **vecsmall_lexcmp**(GEN x, GEN y) compares two **t_VECSMALL** lexically

int **vecsmall_prefixcmp**(GEN x, GEN y) truncate the longest **t_VECSMALL** to the length of the shortest and compares them lexicographically.

GEN **vecsmall_prepend**(GEN V, long s) prepend s to the **t_VECSMALL** V.

GEN **vecsmall_append**(GEN V, long s) append s to the **t_VECSMALL** V.

GEN **vecsmall_concat**(GEN u, GEN v) concat the **t_VECSMALL** u and v.

long **vecsmall_coincidence**(GEN u, GEN v) returns the numbers of indices where u and v agree.

long **vecsmall_pack**(GEN v, long base, long mod) handles the **t_VECSMALL** v as the digit of a number in base base and return this number modulo mod. This can be used as an hash function.

5.8.4 Functions to handle bits-vectors These functions manipulate vectors of bits (stored in `t_VECSMALL`). Bits are numbered from 0.

`GEN bitvec_alloc(long n)` allocates a bits-vector suitable for `n` bits.

`GEN bitvec_shorten(GEN bitvec, long n)` shortens a bits-vector `bitvec` to `n` bits.

`long bitvec_test(GEN bitvec, long b)` returns the bit of index `b` of `bitvec`.

`void bitvec_set(GEN bitvec, long b)` (in place) sets the bit of index `b` of `bitvec`.

`void bitvec_clear(GEN bitvec, long b)` (in place) clears the bit of index `b` of `bitvec`.

5.8.5 Functions to handle vectors of `t_VECSMALL` These functions manipulate vectors of `t_VECSMALL` (`vecvecsmall`).

`GEN vecvecsmall_sort(GEN x)` sorts lexicographically the components of the vector `x`.

`GEN vecvecsmall_indexsort(GEN x)` performs an indirect lexicographic sorting of the components of the vector `x` and return a permutation stored in a `t_VECSMALL`.

`long vecvecsmall_search(GEN x, GEN y, long flag)` `x` being a sorted `vecvecsmall` and `y` a `t_VECSMALL`, search `y` inside `x`. `flag` has the same meaning as for `setsearch`.

Appendix A:

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 `extgcd.c`, then a sample Makefile might look as follows. Note that the actual file `examples/Makefile` is more elaborate and you should have a look at it if you intend to use `install()` on custom made functions, see Section ??.

```
CC = cc
INCDIR = /usr/pkg/include
LIBDIR = /usr/pkg/lib
CFLAGS = -O -I$(INCDIR) -L$(LIBDIR)

all: extgcd

extgcd: extgcd.c
        $(CC) $(CFLAGS) -o extgcd extgcd.c -lpari -lm
```

We then give the listing of the program `examples/extgcd.c` seen in detail in Section 4.8.

```
#include <pari/pari.h>
/*
GP;install("extgcd", "GG&&", "gcdex", "./libextgcd.so");
*/
/* return d = gcd(a,b), sets u, v such that au + bv = gcd(a,b) */
GEN
extgcd(GEN A, GEN B, GEN *U, GEN *V)
{
    pari_sp av = avma;
    GEN ux = gen_1, vx = gen_0, a = A, b = B;
    if (typ(a) != t_INT || typ(b) != t_INT) pari_err(typeer, "extgcd");
    if (signe(a) < 0) { a = negi(a); ux = negi(ux); }
    while (!gcmp0(b))
    {
        GEN r, q = dvmdii(a, b, &r), v = vx;
        vx = subii(ux, mulii(q, vx));
        ux = v;
        a = b; b = r;
    }
    *U = ux;
    *V = diviiexact( subii(a, mulii(A,ux)), B );
    gerepileall(av, 3, &a, U, V); return a;
}

int
```

```

main()
{
    GEN x, y, d, u, v;
    pari_init(1000000,2);
    printf("x = "); x = gp_read_stream(stdin);
    printf("y = "); y = gp_read_stream(stdin);
    d = extgcd(x, y, &u, &v);
    pariprintf("gcd = %Z\nu = %Z\nv = %Z\n", d,u,v);
    return 0;
}

```

Appendix B: 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: gen_0, gen_1, gen_m1, gen_2
t_FRAC: ghalf
t_COMPLEX: gi
t_POL: pol_1[..], pol_x[..]
```

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. $0 \leq i \leq n-1$) 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 ??).

`gpi`. This is the number π . It is initialized by the first call to `mppi` (see Section ??).

The use of both `geuler` and `gpi` is deprecated since it is 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 π) 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(size_t 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 < 2BIL - 2048`, 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) pari_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 `pari_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 (malloced) 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, are present in the table, even if you set `maxprime` to zero. in the `pari_init` call.

Index

SomeWord refers to PARI-GP concepts.

SomeWord is a PARI-GP keyword.

SomeWord is a generic index entry.

A

absi_cmp	53
absi_equal	53
absr_cmp	53
addhelp	36
addii	9
addir	9
addis	9
addll	47
addllx	47
addmul	48
addri	9
addr	9
addmului	57
affii	49
affir	49
affiz	49
affrr	50
affsi	49
affsr	49
affsz	49
affui	50
affur	50
assignment	18
avma	10, 19

B

bern	87
bezout	32, 57
bezoutpol	79
bfffo	47
BIGDEFAULTPREC	9
BIGINT	24, 26
BIL	39
BITS_IN_LONG	9, 10, 39
bitvec_alloc	82
bitvec_clear	83
bitvec_set	82
bitvec_shorten	82
bitvec_test	82
bit_accuracy	10, 41
bit_accuracy_mul(x,y)	41
brute	29, 31

C

cbezout	57
ceilr	51
ceil_safe	51
centermod	79
centermodii	54
centermod_i	79
cgcd	57
cgetc	17, 44, 49
cgetg	16, 17, 44
cgeti	17, 44, 49
cgetp	44
cgetr	17, 44, 49
cgiv	11, 45
character string	26
clone	8, 19
cmpii	52
cmpir	52
cmpis	52
cmpri	52
cmprr	53
cmprrs	53
cmpsi	52
cmpsr	52
column vector	25
col_ei	75
complex number	23
compo	43
constant_term	24, 44
const_col	81
const_vec	82
const_vecsmall	82
content	79
conversions	20
copy	19
copyifstack	47
creation	16
ctofp	73

D

dblton	20, 73
debug	31
debugging	31
DEBUGLEVEL	31, 32
DEBUGMEM	31
debugmem	31
DEFAULTPREC	9
definite binary quadratic form	25

degpol	24, 42	fetch_named_var	27
degree	24	fetch_user_var	27
delete_var	28	fetch_var	27
destruction	11	Flc_to_ZC	71
diffptr	8, 87	Flm_deplin	69
diviexact	55	Flm_Flc_mul	69
diviiround	51	Flm_gauss	69
divis_rem	56	Flm_indexrank	69
diviuexact	55	Flm_inv	69
diviu_rem	56	Flm_ker	69
divll	48	Flm_ker_sp	69
divsi_rem	56	Flm_mul	69
divss_rem	56	Flm_to_FlxV	72
dvdii	55	Flm_to_FlxX	72
dvdiiiz	55	Flm_to_ZM	71
dvdisz	55	floorr	51
dvdiuz	55	Flv_polint	68
dvmdii	55	Flv_roots_to_pol	68
dvmdiiiz	55	Flv_to_Flx	72
dvmdis	55	Flv_to_ZV	71
dvmdsi	55	FlxC_to_ZXC	71
dvmdss	55	FlxM_to_ZXM	71
E		FlxqM_ker	69
effective length	21	FlxqV_roots_to_pol	68
entree	35	FlxqXQ_pow	69
equalii	53	FlxqXV_prod	68
equalis	53	FlxqX_divrem	68
equaliu	53	FlxqX_Flxq_mul	68
equalsi	53	FlxqX_mul	68
equalui	53	FlxqX_normalize	68
errfile	30	FlxqX_red	68
error	30	FlxqX_sqr	68
evalexp	42	Flxq_inv	68
evallg	42	Flxq_invsafe	68
evallgefint	42	Flxq_mul	68
evallgeflist	43	Flxq_pow	68
evalprecp	42	Flxq_powers	68
evalsigne	42	Flxq_sqr	68
evaltyp	42	FlxV_Flc_mul	68
evalvalp	42	FlxV_to_Flm	72
evalvarn	42	FlxX_add	68
expi	41	FlxX_renormalize	68
expo	22, 25, 41	FlxX_shift	68
F		FlxX_to_Flm	72
factoru	57	FlxX_to_ZXX	71
factoru_pow	57	FlxYqQ_pow	68
		Flx_add	67
		Flx_deriv	67
		Flx_div	67

Flx_divrem	67	FpM_indexrank	66
Flx_div_by_X_x	68	FpM_intersect	66
Flx_eval	68	FpM_inv	66
Flx_extgcd	67	FpM_invimage	66
Flx_extresultant	67	FpM_ker	66
Flx_Fl_mul	67	FpM_mul	66
Flx_gcd	67	FpM_rank	66
Flx_gcd_i	67	FpM_red	65
Flx_invmontgomery	67	FpM_suppl	66
Flx_is_squarefree	68	FpM_to_mod	65
Flx_mul	67	fprintferr	31, 36
Flx_nbfact	68	FpV_FpC_mul	66
Flx_nbroots	68	FpV_polint	62
Flx_neg	67	FpV_red	65
Flx_neg_inplace	67	FpV_roots_to_pol	62
Flx_normalize	67	FpV_to_mod	65
Flx_pow	67	FpXQXV_prod	64
Flx_recip	67	FpXQX_divrem	64
Flx_red	67	FpXQX_extgcd	64
Flx_rem	67	FpXQX_gcd	64
Flx_rem_montgomery	67	FpXQX_mul	64
Flx_renormalize	67	FpXQX_red	64
Flx_resultant	67	FpXQX_sqr	64
Flx_shift	67	FpXQYQ_pow	64
Flx_sqr	67	FpXQ_charpoly	64
Flx_sub	67	FpXQ_div	63
Flx_to_Flv	72	FpXQ_ffisom_inv	65
Flx_to_ZX	71	FpXQ_inv	63
Flx_to_ZX_inplace	71	FpXQ_invsafe	63
Flx_valuation	67	FpXQ_minpoly	64
Fl_add	48	FpXQ_mul	63
Fl_center	48	FpXQ_pow	63
Fl_div	48	FpXQ_powers	64
Fl_inv	48	FpXQ_sqr	63
Fl_mul	48	FpXV_FpC_mul	62
Fl_neg	48	FpXV_prod	62
Fl_pow	48	FpXV_red	61
Fl_sqrt	48	FpXX_add	64
Fl_sub	48	FpXX_red	64
Fl_to_Flx	72	FpXX_renormalize	64
format	29, 31	FpX_add	61
FpC_FpV_mul	66	FpX_center	61
FpC_Fp_mul	66	FpX_chinese_coprime	62
FpC_red	65	FpX_degfact	62
FpC_to_mod	65	FpX_div	61
FpM_deplin	66	FpX_divrem	61
FpM_FpC_mul	66	FpX_div_by_X_x	61
FpM_gauss	66	FpX_eval	62
FpM_image	66	FpX_extgcd	61

I		L	
icopy	50	lcmii	57
icopyifstack	47	leading_term	24, 44
indefinite binary quadratic form	25	Legendre symbol	48, 57
infile	30	lexcmp	78
init_Fq	65	lg	20, 41
input	28	lgefint	21, 41
install	29, 31, 36	lgeflist	25
int2n	49	library mode	7
int2u	49	Linux	36
integer	21	list	25
int_LSW	21	LONG_IS_64BIT	9
int_MSW	21		
int_nextW	22	M	
int_normalize	22	matbrute	29
int_precW	21	matid_Flm	72
int_W	21	matrix	25
invmod	56	maxprime	7, 87
isclone	20	maxss	57
iscomplex	78	MAXVARN	7, 27
isexactzero	77	MEDDEFAULTPREC	9
isinexact	77	minss	58
isint	77	mkcol	76
ismonome	78	mkcolcopy	75
isonstack	46	mkcoln	18, 76
isprime	52	mkcomplex	76
issmall	77	mkfrac	76
is_const_t	43	mkintmod	76
is_extscalar_t	43	mkintn	17, 18, 50, 76
is_intreal_t	43	mkmat	76
is_matvec_t	43	mkmatcopy	75
is_rational_t	43	mkpolmod	76
is_recursive_t	43	mkpoln	18, 76
is_scalar_t	43	mkrffrac	76
is_vec_t	43	mkvec	76
itor	50	mkvec2	76
itos	20, 50	mkvec2copy	75
itos_or_0	50	mkvec2s	75
itou	50	mkvec3	76
itou_or_0	50	mkvec3s	75
		mkvec4	76
K		mkveccopy	75
krois	57	mkvecn	18, 76
Kronecker symbol	48, 57	mkvecs	75
kronecker	57	mkvecsnull	75
krosi	57	mkvecsnull2	75
kross	57	mkvecsnull3	75
krouu	48	mod2	22

<code>_evalexpo</code>	42
<code>_evallg</code>	42
<code>_evalvalp</code>	42