Developer's Guide

to

the PARI library

(version 2.7.7)

The PARI Group

Institut de Mathématiques de Bordeaux, UMR 5251 du CNRS.
Université Bordeaux 1, 351 Cours de la Libération
F-33405 TALENCE Cedex, FRANCE
`e-mail: pari@math.u-bordeaux.fr`

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

Table of Contents

Chapter 1: Work in progress

This draft documents private internal functions and structures for hard-core PARI developers. Anything in here is liable to change on short notice. Don't use anything in the present document, unless you are implementing new features for the PARI library. Try to fix the interfaces before using them, or document them in a better way. If you find an undocumented hack somewhere, add it here.

Hopefully, this will eventually document everything that we buried in `paripriv.h` or even more private header files like `anal.h`. Possibly, even implementation choices! Way to go.

1.1 The type `t_CLOSURE`.

This type holds closures and functions in compiled form, so is deeply linked to the internals of the GP compiler and evaluator. The length of this type can be 6, 7 or 8 depending whether the object is an "inline closure", a "function" or a "true closure".

A function is a regular GP function. The GP input line is treated as a function of arity 0.

A true closure is a GP function defined in a non-empty lexical context.

An inline closure is a closure that appears in the code without the preceding `->` token. They are generally associated to the prototype code 'E' and 'I'. Inline closures can only exist as data of other closures, see below.

In the following example,

```
f(a=Euler)=x->sin(x+a);
g=f(Pi/2);
plot(x=0,2*Pi,g(x))
```

`f` is a function, `g` is a true closure and both `Euler` and `g(x)` are inline closures.

This type has a second codeword `z[1]`, which is the arity of the function or closure. This is zero for inline closures. To access it, use

`long closure_arity(GEN C)`

• `z[2]` points to a `t_STR` which holds the opcodes. To access it, use

`GEN closure_get_code(GEN C)`.

`const char * closure_codestr(GEN C)` returns as an array of `char` starting at 1.

• `z[3]` points to a `t_VECSMALL` which holds the operands of the opcodes. To access it, use

`GEN closure_get_oper(GEN C)`

• `z[4]` points to a `t_VEC` which hold the data referenced by the `pushgen` opcodes, which can be `t_CLOSURE`, and in particular inline closures. To access it, use

`GEN closure_get_data(GEN C)`

• `z[5]` points to a `t_VEC` which hold extra data needed for error-reporting and debugging. See Section 1.1.1 for details. To access it, use

`GEN closure_get_dbg(GEN C)`

Additionally, for functions and true closures,

• `z[6]` usually points to a `t_VEC` with two components which are `t_STR`. The first one displays the list of arguments of the closure without the enclosing parentheses, the second one the GP code of the function at the right of the `->` token. They are used to display the closure, either in implicit or explicit form. However for closures that were not generated from GP code, `z[6]` can point to a `t_STR` instead. To access it, use

`GEN closure_get_text(GEN C)`

Additionally, for true closure,

• `z[7]` points to a `t_VEC` which holds the values of all lexical variables defined in the scope the closure was defined. To access it, use

`GEN closure_get_frame(GEN C)`

1.1.1 Debugging information in closure.

Every `t_CLOSURE` object `z` has a component `dbg=z[5]` which which hold extra data needed for error-reporting and debugging. The object `dbg` is a `t_VEC` with 3 components:

`dbg[1]` is a `t_VECSMALL` of the same length than `z[3]`. For each opcode, it holds the position of the corresponding GP source code in the strings stored in `z[6]` for function or true closures, positive indices referring to the second strings, and negative indices referring to the first strings, the last element being indexed as -1. For inline closures, the string of the parent function or true closure is used instead.

`dbg[2]` is a `t_VECSMALL` that lists opcodes index where new lexical local variables are created. The value 0 denotes the position before the first offset and variables created by the prototype code 'V'.

`dbg[3]` is a `t_VEC` of `t_VECSMALL`s that give the list of `entree*` of the lexical local variables created at a given index in `dbg[2]`.

1.2 The type `t_LIST`.

This type needs to go through various hoops to support GP's inconvenient memory model. Don't use `t_LIST`s in pure library mode, reimplement ordinary lists! This dynamic type is implemented by a `GEN` of length 3: two codewords and a vector containing the actual entries. In a normal setup (a finished list, ready to be used),

• the vector is malloc'ed, so that it can be realloc'ated without moving the parent `GEN`.

• all the entries are clones, possibly with cloned subcomponents; they must be deleted with `gunclone_deep`, not `gunclone`.

The following macros are proper lvalues and access the components

`long list_nmax(GEN L)`: current maximal number of elements. This grows as needed.

GEN list_data(GEN L): the elements. If v = list_data(L), then either v is NULL (empty list) or l = lg(v) is defined, and the elements are v[1], ..., v[l-1].

In most gerepile scenarios, the list components are not inspected and a shallow copy of the malloc'ed vector is made. The functions gclone, copy_bin_canon are exceptions, and make a full copy of the list.

The main problem with lists is to avoid memory leaks; in the above setup, a statement like a = List(1) would already leak memory, since List(1) allocates memory, which is cloned (second allocation) when assigned to a; and the original list is lost. The solution we implemented is

• to create anonymous lists (from List, gtolist, concat or vecsort) entirely on the stack, *not* as described above, and to set list_nmax to 0. Such a list is not yet proper and trying to append elements to it fails:

```
? listput(List(),1)
  ***   variable name expected: listput(List(),1)
  ***                                   ^----------------
```

If we had been malloc'ing memory for the List([1,2,3]), it would have leaked already.

• as soon as a list is assigned to a variable (or a component thereof) by the GP evaluator, the assigned list is converted to the proper format (with list_nmax set) previously described.

GEN listcopy(GEN L) return a full copy of the t_LIST L, allocated on the *stack* (hence list_nmax is 0). Shortcut for gcopy.

GEN mklistcopy(GEN x) returns a list with a single element x, allocated on the stack. Used to implement most cases of gtolist (except vectors and lists).

A typical low-level construct:

```
long l;
/* assume L is a t_LIST */
L = list_data(L); /* discard t_LIST wrapper */
l = L? lg(L): 1;
for (i = 1; i < l; i++) output( gel(L, i) );
for (i = 1; i < l; i++) gel(L, i) = gclone( ... );
```

1.3 Protection of non-interruptible code.

GP allows the user to interrupt a computation by issuing SIGINT (usually by entering control-C) or SIGALRM (usually using alarm()). To avoid such interruption to occurs in section of code which are not reentrant (in particular malloc and free) the following mechanism is provided:

BLOCK_SIGINT_START() Start a non-interruptible block code. Block both SIGINT and SIGARLM.

BLOCK_SIGALRM_START() Start a non-interruptible block code. Block only SIGARLM. This is used in the SIGINT handler itself to delay an eventual pending alarm.

BLOCK_SIGINT_END() End a non-interruptible block code

The above macros make use of the following global variables:

PARI_SIGINT_block: set to 1 (resp. 2) by BLOCK_SIGINT_START (resp. BLOCK_SIGALRM_START).

`PARI_SIGINT_pending`: Either 0 (no signal was blocked), `SIGINT` (`SIGINT` was blocked) or `SIGALRM` (`SIGALRM` was blocked). This need to be set by the signal handler.

Inside a block, a auto variable `int block` is defined which holds the value of `PARI_SIGINT_block` when entering the block.

1.3.1 Multithread interruptions.

To support multithread, `BLOCK_SIGINT_START` and `BLOCK_SIGALRM_START` calls `MT_SIGINT_BLOCK(block);`, and `BLOCK_SIGINT_END` calls `MT_SIGINT_UNBLOCK(block);`.

`MT_SIGINT_BLOCK` and `MT_SIGINT_UNBLOCK` are defined by the multithread engine. They can calls the following public functions defined by the multithread engine.

`void mt_sigint_block(void)`

`void mt_sigint_unblock(void)`

In practice this mechanism is used by the POSIX thread engine to protect against asychronous cancellation.

1.4 Black box groups.

A black box group is defined by a `bb_group` struct, describing methods available to handle group elements:

```
struct bb_group
{
  GEN (*mul)(void*, GEN, GEN);
  GEN (*pow)(void*, GEN, GEN);
  ulong (*hash)(GEN);
  GEN (*rand)(void*);
  int (*equal)(GEN, GEN);
  int (*equal1)(GEN);
  GEN (*easylog)(void *E, GEN, GEN, GEN);
};
```

`mul(E,x,y)` returns the product xy.

`pow(E,x,n)` returns x^n (n integer, possibly negative or zero).

`hash(x)` returns a hash value for x (`hash_GEN` is suitable for this field).

`rand(E)` returns a random element in the group.

`equal(x,y)` returns one if $x = y$ and zero otherwise.

`equal1(x)` returns one if x is the neutral element in the group, and zero otherwise.

`easylog(E,a,g,o)` (optional) returns either NULL or the discrete logarithm n such that $g^n = a$, the element g being of order o. This provides a short-cut in situation where a better algorithm than the generic one is known.

A group is thus described by a `struct bb_group` as above and auxiliary data typecast to `void*`. The following functions operate on black box groups:

`GEN gen_Shanks_log(GEN x, GEN g, GEN N, void *E, const struct bb_group *grp)`
Generic baby-step/giant-step algorithm (Shanks's method). Assuming that g has order N, compute an integer k such that $g^k = x$. Return `cgetg(1, t_VEC)` if there are no solutions. This requires $O(\sqrt{N})$ group operations and uses an auxiliary table containing $O(\sqrt{N})$ group elements.

`GEN gen_Pollard_log(GEN x, GEN g, GEN N, void *E, const struct bb_group *grp)`
Generic Pollard rho algorithm. Assuming that g has order N, compute an integer k such that $g^k = x$. This requires $O(\sqrt{N})$ group operations in average and $O(1)$ storage. Will enter an infinite loop if there are no solutions.

`GEN gen_plog(GEN x, GEN g, GEN N, void *E, const struct bb_group)` Assuming that g has prime order N, compute an integer k such that $g^k = x$, using either `gen_Shanks_log` or `gen_Pollard_log`. Return `cgetg(1, t_VEC)` if there are no solutions.

If `easy` is not `NULL`, call `easy(E,a,g,N)` first and if the return value is not `NULL`, return it. For instance this is used over $\mathbf{F}_q^*$ to compute the discrete log of elements belonging to the prime field.

`GEN gen_Shanks_sqrtn(GEN a, GEN n, GEN N, GEN *zetan, void *E, const struct bb_group *grp)` returns one solution of $x^n = a$ in a black box cyclic group of order N. Return `NULL` if no solution exists. If `zetan` is not `NULL` it is set to an element of exact order n.

This function uses `gen_plog` for all prime divisors of $\gcd(n, N)$.

`GEN gen_PH_log(GEN a, GEN g, GEN N, void *E, const struct bb_group *grp)` Generic Pohlig-Hellman algorithm. Assuming that g has order N, compute an integer k such that $g^k = x$. Return `cgetg(1, t_VEC)` if there are no solutions. This calls `gen_plog` repeatedly for all prime divisors p of N.

`easy` is as in `gen_plog`.

`GEN gen_order(GEN x, GEN N, void *E, const struct bb_group *grp)` computes the order of x. If N is not `NULL` it is a multiple of the order, as a `t_INT` or a factorization matrix.

`GEN gen_factored_order(GEN x, GEN N, void *E, const struct bb_group *grp)` returns $[o, F]$, where o is the order of x and F is the factorization of o. If N is not `NULL` it is a multiple of the order, as a `t_INT` or a factorization matrix.

`GEN gen_select_order(GEN v, GEN N, void *E, const struct bb_group *grp)` v being a vector of possible order of the group, try to find the true order by checking orders of random points. This will not terminate if there is an ambiguity.

`GEN gen_gener(GEN o, void *E, const struct bb_group *grp)` returns a random generator of the group, assuming it is of order exactly o (which can be given by a factorization matrix).

1.4.1 Black box groups with pairing.

Theses functions handle groups of rank at most 2 equipped with a family of bilinear pairings which behave like the Weil pairing on elliptic curves over finite field.

The function `pairorder(E, P, Q, m, F)` must return the order of of the m-pairing of P and Q, both of order dividing m, where F is the factorisation matrix of a multiple of m.

`GEN gen_ellgroup(GEN o, GEN d, GEN *pt_m, void *E, const struct bb_group *grp, GEN pairorder(void *E, GEN P, GEN Q, GEN m, GEN F))`

returns the elementary divisors $[d_1, d_2]$ of the group, assuming it is of order exactly $o > 1$ (which can be given by a factorization matrix), and that d_2 divides d. If $d_2 = 1$ then $[o]$ is returned, otherwise `m=*pt_m` is set to the order of the pairing required to verify a generating set which is to be used with `gen_ellgens`.

`GEN gen_ellgens(GEN d1, GEN d2, GEN m, void *E, const struct bb_group *grp, GEN pairorder(void *E, GEN P, GEN Q, GEN m, GEN F))` the parameters d_1, d_2, m being as returned by `gen_ellgroup`, returns a pair of generators $[P, Q]$ such that P is of order d_1 and the m-pairing of P and Q is of order m. (Note: Q needs not be of order d_2).

1.4.2 Functions returning black box groups.

`const struct bb_group * get_FpXQ_star(void **E, GEN T, GEN p)` returns a pointer to the black box group $(\mathbf{F}_p[x]/(T))^*$.

`const struct bb_group * get_FpE_group(void **pt_E, GEN a4, GEN a6, GEN p)` returns a pointer to a black box group and set `*pt_E` to the necessary data for computing in the group $E(\mathbf{F}_p)$ where E is the elliptic curve $E : y^2 = x^3 + a_4x + a_6$, with a_4 and a_6 in $\mathbf{F}_p$.

`const struct bb_group * get_FpXQE_group(void **pt_E, GEN a4, GEN a6, GEN T, GEN p)` returns a pointer to a black box group and set `*pt_E` to the necessary data for computing in the group $E(\mathbf{F}_p[X]/(T))$ where E is the elliptic curve $E : y^2 = x^3 + a_4x + a_6$, with a_4 and a_6 in $\mathbf{F}_p[X]/(T)$.

`const struct bb_group * get_FlxqE_group(void **pt_E, GEN a4, GEN a6, GEN T, ulong p)` idem for small p.

`const struct bb_group * get_F2xqE_group(void **pt_E, GEN a2, GEN a6, GEN T)` idem for $p = 2$.

1.5 Black box finite fields.

A black box finite field is defined by a `bb_field` struct, describing methods available to handle field elements:

```
struct bb_field
{
  GEN (*red)(void *E ,GEN);
  GEN (*add)(void *E ,GEN, GEN);
  GEN (*mul)(void *E ,GEN, GEN);
  GEN (*neg)(void *E ,GEN);
  GEN (*inv)(void *E ,GEN);
  int (*equal0)(GEN);
  GEN (*s)(void *E, long);
};
```

Note that, in contrast of black box group, elements can have non canonical forms, and only `red` is required to return a canonical form.

`red(E,x)` returns the canonical form of x.

`add(E,x,y)` returns the sum $x + y$.

`mul(E,x,y)` returns the product xy.

`neg(E,x)` returns $-x$.

`inv(E,x)` returns the inverse of x.

`equal0(x)` x being in canonical form, returns one if $x = 0$ and zero otherwise.

`s(n)` n being a small signed integer, returns n times the unit element.

A finite field is thus described by a `struct bb_field` as above and auxiliary data typecast to `void*`. The following functions operate on black box fields:

`GEN gen_Gauss(GEN a, GEN b, void *E, const struct bb_field *ff)`

`GEN gen_Gauss_pivot(GEN x, long *rr, void *E, const struct bb_field *ff)`

`GEN gen_det(GEN a, void *E, const struct bb_field *ff)`

`GEN gen_ker(GEN x, long deplin, void *E, const struct bb_field *ff)`

`GEN gen_matcolmul(GEN a, GEN b, void *E, const struct bb_field *ff)`

`GEN gen_matid(long n, void *E, const struct bb_field *ff)`

`GEN gen_matmul(GEN a, GEN b, void *E, const struct bb_field *ff)`

1.5.1 Functions returning black box fields.

`const struct bb_field * get_Fp_field(void **pt_E, GEN p)`

`const struct bb_field * get_Fq_field(void **pt_E, GEN T, GEN p)`

`const struct bb_field * get_Flxq_field(void **pt_E, GEN T, ulong p)`

`const struct bb_field * get_F2xq_field(void **pt_E, GEN T)`

1.6 Black box algebra.

A black box algebra is defined by a `bb_algebra` struct, describing methods available to handle algebra elements:

```
struct bb_algebra
{
  GEN (*red)(void *E, GEN x);
  GEN (*add)(void *E, GEN x, GEN y);
  GEN (*mul)(void *E, GEN x, GEN y);
  GEN (*sqr)(void *E, GEN x);
  GEN (*one)(void *E);
  GEN (*zero)(void *E);
};
```

Note that, in contrast with black box groups, elements can have non canonical forms, but only `add` is allowed to return a non canonical form.

`red(E,x)` returns the canonical form of x.

`add(E,x,y)` returns the sum $x + y$.

`mul(E,x,y)` returns the product xy.

`sqr(E,x)` returns the square x^2.

`one(E)` returns the unit element.

`zero(E)` returns the zero element.

An algebra is thus described by a `struct bb_algebra` as above and auxiliary data typecast to `void*`. The following functions operate on black box algebra:

`GEN gen_bkeval(GEN P, long d, GEN x, int use_sqr, void *E, const struct bb_algebra *ff, GEN cmul(void *E, GEN P, long a, GEN x))` x being an element of the black box algebra, and P some black box polynomial of degree d over the base field, returns $P(x)$. The function `cmul(E,P,a,y)` must return the coefficient of degree a of P multiplied by y. `cmul` is allowed to return a non canonical form.

The flag `use_sqr` has the same meaning as for `gen_powers`. This implements an algorithm of Brent and Kung (1978).

`GEN gen_bkeval_powers(GEN P, long d, GEN V, void *E, const struct bb_algebra *ff, GEN cmul(void *E, GEN P, long a, GEN x))` as `gen_RgX_bkeval` assuming V was output by `gen_powers`(x, l, E, ff) for some $l \geq 1$. For optimal performance, l should be computed by `brent_kung_optpow`.

`long brent_kung_optpow(long d, long n, long m)` returns the optimal parameter l for the evaluation of n/m polynomials of degree d. Fractional values can be used if the evaluations are done with different accuracies, and thus have different weights.

1.7 Black box free $\mathbf{Z}_p$-modules.

(Very experimental)

`GEN gen_ZpX_Dixon(GEN F, GEN V, GEN q, GEN p, long N, void *E, GEN lin(void *E, GEN F, GEN z, GEN q), GEN invl(void *E, GEN z))`

Let F be a `ZpXT` representing the coefficients of some abstract linear mapping f over $\mathbf{Z}_p[X]$ seen as a free $\mathbf{Z}_p$-module, let V be an element of $\mathbf{Z}_p[X]$ and let $q = p^N$. Return $y \in \mathbf{Z}_p[X]$ such that $f(y) = V \pmod{p^N}$ assuming the following holds for $n \leq N$:

- `lin`$(E, \texttt{FpX\_red}(F, p^n), z, p^n) \equiv f(z) \pmod{p^n}$
- $f(\texttt{invl}(E, z)) \equiv z \pmod{p}$

The rationale for the argument F being that it allows `gen_ZpX_Dixon` to reduce it to the required p-adic precision.

`GEN gen_ZpX_Newton(GEN x, GEN p, long n, void *E, GEN eval(void *E, GEN a, GEN q), GEN invd(void *E, GEN b, GEN v, GEN q, long N))`

Let x be an element of $\mathbf{Z}_p[X]$ seen as a free $\mathbf{Z}_p$-module, and f some differentiable function over $\mathbf{Z}_p[X]$ such that $f(x) \equiv 0 \pmod{p}$. Return y such that $f(y) \equiv 0 \pmod{p^n}$, assuming the following holds for all $a, b \in \mathbf{Z}_p[X]$ and $M \leq N$:

- $v = \texttt{eval}(E, a, p^N)$ is a vector of elements of $\mathbf{Z}_p[X]$,
- $w = \texttt{invd}(E, b, v, p^M, M)$ is an element in $\mathbf{Z}_p[X]$,
- $v[1] \equiv f(a) \pmod{p^N \mathbf{Z}_p[X]}$,
- $df_a(w) \equiv b \pmod{p^M \mathbf{Z}_p[X]}$

and df_a denotes the differential of f at a. Motivation: eval allows to evaluate f and invd allows to invert its differential. Frequently, data useful to compute the differential appear as a subproduct of computing the function. The vector v allows eval to provide these to invd. The implementation of invd will generally involves the use of the function gen_ZpX_Dixon.

1.8 Public functions useless outside of GP context.

These functions implement GP functionality for which the C language or other libpari routines provide a better equivalent; or which are so tied to the gp interpreter as to be virtually useless in libpari. Some may be generated by gp2c. We document them here for completeness.

1.8.1 Conversions.

GEN toser_i(GEN x) internal shallow function, used to implement automatic conversions to power series in GP (as in cos(x)). Converts a t_POL or a t_RFRAC to a t_SER in the same variable and precision precdl (the global variable corresponding to seriesprecision). Returns x itself for a t_SER, and NULL for other argument types. The fact that it uses a global variable makes it awkward whenever you're not implementing a new transcendental function in GP. Use RgX_to_ser or rfrac_to_ser for a fast clean alternative to gtoser.

1.8.2 Output.

void print0(GEN g, long flag) internal function underlying the print GP function. Prints the entries of the t_VEC g, one by one, without any separator; entries of type t_STR are printed without enclosing quotes. *flag*is one of f_RAW, f_PRETTYMAT or f_TEX, using the current default output context.

void out_print0(PariOUT *out, const char *sep, GEN g, long flag) as print0, using output context out and separator sep between successive entries (no separator if NULL).

void printsep(const char *s, GEN g, long flag) out_print0 on pariOut followed by a newline.

void printsep1(const char *s, GEN g, long flag) out_print0 on pariOut.

char* pari_sprint0(const char *s, GEN g, long flag) displays s, then print0(g, flag).

void print(GEN g) equivalent to print0(g, f_RAW), followed by a \n then an fflush.

void print1(GEN g) as above, without the \n. Use pari_printf or output instead.

void printtex(GEN g) equivalent to print0(g, t_TEX), followed by a \n then an fflush. Use GENtoTeXstr and pari_printf instead.

void write0(const char *s, GEN g)

void write1(const char *s, GEN g) use fprintf

void writetex(const char *s, GEN g) use GENtoTeXstr and fprintf.

void printf0(GEN fmt, GEN args) use pari_printf.

GEN Strprintf(GEN fmt, GEN args) use pari_sprintf.

1.8.3 Input.

gp's input is read from the stream pari_infile, which is changed using

FILE* switchin(const char *name)

Note that this function is quite complicated, maintaining stacks of files to allow smooth error recovery and gp interaction. You will be better off using gp_read_file.

1.8.4 Control flow statements.

GEN break0(long n). Use the C control statement break. Since break(2) is invalid in C, either rework your code or use goto.

GEN next0(long n). Use the C control statement continue. Since continue(2) is invalid in C, either rework your code or use goto.

GEN return0(GEN x). Use return!

void error0(GEN g). Use pari_err(e_USER,)

void warning0(GEN g). Use pari_warn(e_USER,)

1.8.5 Accessors.

GEN vecslice0(GEN A, long y1, long y2) used to implement $A[y_1..y_2]$.

GEN matslice0(GEN A, long x1, long x2, long y1, long y2) used to implement $A[x_1..x_2, y_1..y_2]$.

1.8.6 Iterators.

GEN apply0(GEN f, GEN A) gp wrapper calling genapply, where f is a t_CLOSURE, applied to A. Use genapply or a standard C loop.

GEN select0(GEN f, GEN A) gp wrapper calling genselect, where f is a t_CLOSURE selecting from A. Use genselect or a standard C loop.

GEN vecapply(void *E, GEN (*f)(void* E, GEN x), GEN x) used to implement [a(x)|x<-b].

GEN vecselect(void *E, long (*f)(void* E, GEN x), GEN A) used to implement [x<-b,c(x)].

GEN vecselapply(void *Epred, long (*pred)(void* E, GEN x), void *Efun, GEN (*fun)(void* E, GEN x), GEN A) used to implement [a(x)|x<-b,c(x)].

1.8.7 Function related to the GP parser.

The prototype code C instructs the GP compiler to save the current lexical context (pairs made of a lexical variable name and its value) in a GEN, called pack in the sequel. This pack can be used to evaluate expressions in the corresponding lexical context, providing it is current.

GEN localvars_read_str(const char *s, GEN pack) evaluate the string s in the lexical context given by pack. Used by geval_gp in GP to implement the behaviour below:

```
? my(z=3);eval("z=z^2");z
%1 = 9
```

long localvars_find(GEN pack, entree *ep) does pack contain a pair whose variable corresponds to ep? If so, where is the corresponding value? (returns an offset on the value stack).

1.8.8 Miscellaneous.

`char* os_getenv(const char *s)` either calls `getenv`, or directly return `NULL` if the `libc` does not provide it. Use `getenv`.

`sighandler_t os_signal(int sig, pari_sighandler_t fun)` after a

```
typedef void (*pari_sighandler_t)(int);
```

(private type, not exported). Installs signal handler `fun` for signal `sig`, using `sigaction` with flag `SA_NODEFER`. If `sigaction` is not available use `signal`. If even the latter is not available, just return `SIG_IGN`. Use `sigaction`.

Chapter 2: Regression tests, benches

This chapter documents how to write an automated test module, say `fun`, so that `make test-fun` executes the statements in the `fun` module and times them, compares the output to a template, and prints an error message if they do not match.

• Pick a *new* name for your test, say `fun`, and write down a GP script named `fun`. Make sure it produces some useful output and tests adequately a set of routines.

• The script should not be too long: one minute runs should be enough. Try to break your script into independent easily reproducible tests, this way regressions are easier to debug; e.g. include `setrand(1)` statement before a randomized computation. The expected output may be different on 32-bit and 64-bit machines but should otherwise be platform-independent. If possible, the output shouldn't even depend on `sizeof(long)`; using a `realprecision` that exists on both 32-bit and 64-bit architectures, e.g. `\p 38` is a good first step.

• Dump your script into `src/test/in/` and run `Configure`.

• `make test-fun` now runs the new test, producing a `[BUG]` error message and a `.dif` file in the relevant object directory `Oxxx`. In fact, we compared the output to a non-existing template, so this must fail.

• Now

```
patch -p1 < Oxxx/fun.dif
```

generates a template output in the right place `src/test/32/fun`, for instance on a 32-bit machine.

• If different output is expected on 32-bit and 64-bit machines, run the test on a 64-bit machine and patch again, thereby producing `src/test/64/fun`. If, on the contrary, the output must be the same, make sure the output template land in the `src/test/32/` directory (which provides a default template when the 64-bit output file is missing); in particular move the file from `src/test/64/` to `src/test/32/` if the test was run on a 64-bit machine.

• You can now re-run the test to check for regressions: no `[BUG]` is expected this time! Of course you can at any time add some checks, and iterate the test / patch phases. In particular, each time a bug in the `fun` module is fixed, it is a good idea to add a minimal test case to the test suite.

• By default, your new test is now included in `make test-all`. If it is particularly annoying, e.g. opens tons of graphical windows as `make test-ploth` or just much longer than the recommended minute, you may edit `config/get_tests` and add the `fun` test to the list of excluded tests, in the `test_extra_out` variable.

• The `get_tests` script also defines the recipe for `make bench` timings, via the variable `test_basic`. A test is included as `fun` or `fun_`n, where n is an integer ≤ 1000; the latter means that the timing is weighted by a factor $n/1000$. (This was introduced a long time ago, when the `nfields` bench was so much slower than the others that it hid slowdowns elsewhere.)

2.1 Functions for GP2C.

2.1.1 Functions for safe access to components.

Theses function returns the adress of the requested component after checking it is actually valid. This is used by GP2C -C.

`GEN* safegel(GEN x, long l)`, safe version of `gel(x,l)` for `t_VEC`, `t_COL` and `t_MAT`.

`long* safeel(GEN x, long l)`, safe version of `x[l]` for `t_VECSMALL`.

`GEN* safelistel(GEN x, long l)` safe access to `t_LIST` component.

`GEN* safegcoeff(GEN x, long a, long b)` safe version of `gcoeff(x,a, b)` for `t_MAT`.

Chapter 3: Parallelism

3.1 The PARI MT interface.

PARI provides an abstraction for doing parallel computations.

`void mt_queue_start(struct pari\_mt *pt, GEN worker)` Let `worker` be a `t_CLOSURE` object of arity 1. Initialize the structure `pt` to evaluate `worker` in parallel.

`void mt_queue_submit(struct pari\_mt *pt, long taskid, GEN task)` Submit `task` to be evaluated by `worker`, or `NULL` if no further task is left to be submitted. The value `taskid` is user-specified and allows to later match up results and submitted tasks.

`GEN mt_queue_get(struct pari\_mt *pt, long *taskid, long *pending)` Return the result of the evaluation by `worker` of one of the previously submitted tasks. Set `pending` to the number of remaining pending tasks. Set `taskid` to the value associate to this task by `mt_queue_submit`. Returns `NULL` if more tasks need to be submitted.

`void mt_queue_end(struct pari\_mt *pt)` End the parallel execution.

Calls to `mt_queue_submit` and `mt_queue_get` must alternate: each call to `mt_queue_submit` must be followed by a call to `mt_queue_get` before any other call to `mt_queue_submit`, and conversely.

The first call to `mt_queue_get` will return `NULL` until a sufficient number of tasks have been submitted. If no more tasks are left to be submitted, use

```
mt_queue_submit(handle, id, NULL)
```

to allow further calls to `mt_queue_get`. If `mt_queue_get` sets `pending` to 0, then no more tasks are pending and it is safe to call `mt_queue_end`.

The parameter `taskid` can be chosen arbitrarily. It is associated to a task but is not available to `worker`. It provides an efficient way to match a tasks and results. It is ignored when the parameter `task` is `NULL`.

3.1.1 Miscellaneous.

`void mt_broadcast(GEN code)`: do nothing unless the MPI threading engine is in use. In that case, it evaluates the closure `code` on all secondary nodes. This can be sued to change the states of the MPI child nodes. This is used by `install`.

3.2 Initialization.

This section is technical.

`void pari_mt_init(void)` When using MPI, it is sometimes necessary to run initialization code on the child nodes after PARI is initialized. This can be done as follow:

• call `pari_init_opts` with the flag `INIT_noIMTm`. This initializes PARI, but not the MT engine.

• call the required initialization code.

• call `pari_mt_init` to initialize the MT engine. Note that under MPI, this function only returns on the master node. On the child nodes, it enters slave mode. Thus it is no longer possible to run initialization code on the child nodes.

See the file `examples/pari-mt.c` for an example.

`void pari_mt_close(void)` When using MPI, calling `pari_close` will terminate the MPI execution environment. If this is undesirable, you should call `pari_close_opts` with the flag `INIT_noIMTm`. This closes PARI without terminating the MPI execution environment It is allowed to call `pari_mt_close` later to terminate it. Note that the once MPI is terminated it cannot be restarted, and that it is considered an error for a program to end without having terminated the MPI execution environment.

Index

SomeWord refers to PARI-GP concepts.
`SomeWord` is a PARI-GP keyword.
SomeWord is a generic index entry.

N

O

P

R

S

T

V

W