Developer's Guide

to

the PARI library

(version 2.5.1)

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 . . . 5
1.1 The type t_CLOSURE . . . 5
1.1.1 Debugging information in closure . . . 6
1.2 The type t_LIST . . . 6
1.3 Otherwise undocumented global variables . . . 7
1.4 Finite fields and black-box groups . . . 7
1.5 Public functions useless outside of GP context . . . 8
1.5.1 Output . . . 9
1.5.2 Input . . . 9
1.5.3 Control flow statements . . . 9
1.5.4 Iterators . . . 9
1.5.5 Function related to the GP parser . . . 10
1.5.6 Miscellaneous . . . 10
Chapter 2: Regression tests, benches . . . 11
Index . . . 12

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.

• `z[2]` points to a `t_STR` which hold the opcodes.

• `z[3]` points to a `t_VECSMALL` which hold the operands of the opcodes.

• `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.

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

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.

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.

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 Otherwise undocumented global variables.

`PARI_SIGINT_block`: set this to a non-zero value if you want to block the `SIGINT` signal in a critical part of your code. We use it before calling `malloc`, `free` and such. (Because `SIGINT` is non-fatal for us, and we don't want to leave the system stack in an inconsistent state.)

`PARI_SIGINT_pending`: if this is non-zero, then a `SIGINT` was blocked. Take action as appropriate.

1.4 Finite fields and 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);
  GEN (*rand)(void*);
  int (*cmp)(GEN, GEN);
  int (*equal1)(GEN);
};
```

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

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

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

`cmp(x,y)` implements a total ordering on the group elements (return value -1, 0 or 1).

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

A group is thus described by a `const bb_struct` 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 $g^k = x$. 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.

`GEN gen_plog(GEN x, GEN g, GEN N, void *E, const struct bb_group, GEN easy(void*, GEN, GEN, GEN))` 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`.

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.

FIXME. More generally, one should compute the minimal polynomial of x and restrict to its field of definition.

`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, GEN easy(void *E, GEN, GEN, GEN))` Generic Pohlig-Hellman algorithm. Assuming that g has order N, compute an integer k such that $g^k = x$. This requires $O(p^{1/2+\epsilon})$ group operations, where p is the largest prime divisor of N, and uses an auxiliary table containing $O(\sqrt{p})$ group elements.

`easy` is as in `gen_plog`.

`GEN gen_eltorder(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.

1.5 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.

1.5.1 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, GEN g, long flag)` as `print0`, using output context `out`.

`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`.

1.5.2 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`.

1.5.3 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(user,)`

`void warning0(GEN g)`. Use `pari_warn(user,)`

1.5.4 Iterators. `GEN apply0(GEN f, GEN A)` gp wrapper calling `genapply`, where f is a `t_CLOSURE`, applied to A. Use `genapply` latter 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.

1.5.5 Function related to the GP parser.

The GP parser can generate an opcode saving the current lexical context (pairs made of a lexical variable name and its value) in a `GEN`, called `pack` in the sequel. These can be used from debuggers (e.g. gp's break loop) to track values of lexical variable. Indeed, lexical variables have disappeared from the compiled code, only their values in a given scope exist (on some value stack). Provided the parser generated the proper opcode, there remains a trace of lexical variable names and everything can still be unravelled.

`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.

`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).

1.5.6 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.

• Go to the relevant `Oxxx` directory, then

```
patch -p0 < 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.)

Index

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

A

apply0 9

B

bb_group 7
break0 9

C

closure 5

E

error0 9

F

f_PRETTYMAT 8
f_RAW 8
f_TEX 8

G

genapply 9
genselect 9
GENtoTeXstr 9
gen_eltorder 8
gen_PH_log 8
gen_plog 8
gen_Pollard_log 8
gen_Shanks_log 8
gen_Shanks_sqrtn 8
getenv 10
geval_gp 9
gp_read_file 9
gunclone 6
gunclone_deep 6

L

list 6
listcopy 7
list_data 6
list_nmax 6
localvars_find 10
localvars_read_str 9

M

mklistcopy 7

N

next0 9

O

os_getenv 10
os_signal 10
output 8
out_print0 8

P

pari_infile 9
pari_printf 8, 9
PARI_SIGINT_block 7
PARI_SIGINT_pending 7
pari_sprintf 9
print 8
print0 8
print1 8
printf0 9
printtex 8

R

return0 9

S

SA_NODEFER 10
select0 9
sigaction 10
signal 10
SIG_IGN 10
Strprintf 9
switchin 9

T

t_CLOSURE 5
t_LIST 6

W

warning0 9
write0 9
write1 9
writetex 9