An introduction to gp2c

By Bill Allombert and Ariel Pacetti

May 12, 2025

Contents

1 What is gp2c? 1
1.1 Installing gp2c 2

2 A gp2c tutorial 2
2.1 How can I compile and run my scripts? 2
2.2 How can I compile directly with gp2c? 4
2.3 Using gp2c to find errors in GP scripts 6
2.4 Using compiled functions in a new program 6
2.5 Hand-editing the C file generated by gp2c 7

3 Advanced use of gp2c 8
3.1 gp2c types 8
3.2 Type declaration 8
3.3 Effect of types declaration on default values 9
3.4 Type casting 9
3.5 Example of optimisation 10
3.6 Types and member functions 11

4 Common problems 11
4.1 Meta-commands. 11
4.2 Unsupported functions. 12
4.3 Dynamically-scoped local variables. 12
4.4 Memory handling and global variables. 12
4.5 Support for t_VECSMALL 13
4.6 GP lists 13
4.7 The *install* command 14
4.8 What about main() ? 14

5 Command-line options of gp2c 14

1 What is gp2c?

The gp2c compiler is a package for translating GP routines into the C programming language, so that they can be compiled and used with the PARI system or the GP calculator.

The main advantage of doing this is to speed up computations and to include your own routines within the preexisting GP ones. It may also find bugs in GP scripts.

This package (including the latest versions) can be obtained at the URL:
`http://pari.math.u-bordeaux.fr/download.html#gp2c`

1.1 Installing gp2c

After downloading the file `gp2c-`*x.y.z*`plt.tar.gz` (where *x,y,z* and *t* depend on the version), you first have to unzip the file with the command:
gunzip gp2c-*x.y.z*pl*t*.tar.gz

This will create the new file `gp2c-`*x.y.z*`plt.tar`. Next you have to extract the files with the **tar** program:
tar -xvf gp2c-*x.y.z*pl*t*.tar

Note: You can do both steps at once with GNU **tar** by using the command:
tar -zxvf gp2c-*x.y.z*pl*t*.tar.gz

This creates a directory `gp2c-`*x.y.z*`plt`, which contains the main `gp2c` files. Now you have to install the program.

You need the file `pari.cfg`. This file can be found in the PARI object directory and is installed in $prefix/lib/pari/.

Copy or link this file in the `gp2c` directory, be sure to call it `pari.cfg`.
ln -s .../lib/pari/pari.cfg pari.cfg

Run **./configure**, which will search for the PARI version and some other configuration tools of the system. To install the program, type **make**, and the program will be compiled. You can then run **make check** to verify that everything has gone fine (a bunch of OK's should show up). All of this is completely standard, and you are now ready to use `gp2c`.

You can use `gp2c` directly from this directory or you can install it by running **make install** as root. If you do not install it, you can run it from the `gp2c` directory by typing **./gp2c**

2 A gp2c tutorial

2.1 How can I compile and run my scripts?

The simplest way to use `gp2c` is to call `gp2c-run`. If you want to know what happens in detail, see next section.

To make the examples easier to follow, please move to the `gp2c` directory and link the root of your PARI source there:
ln -s .../pari .

As an example, we will take the file `pari/examples/squfof.gp`, which is a simple implementation of the well-known SQUFOF factoring method of D. Shanks.

We just run the command:
./gp2c-run pari/examples/squfof.gp

After a little processing we get a GP session. But this session is special, because it contains the compiled *squfof* function. Hence we can do the following:

```
parisize = 4000000, primelimit = 500000
? squfof(3097180303181)
```

```
[419]
i = 596
Qfb(133225, 1719841, -261451, 0.E-28)
%1 = 1691693
```

Let's try a bigger example:

```
? squfof(122294051504814979)
[20137]
  ***   the PARI stack overflows !
  current stack size: 4.0 Mbytes
  [hint] you can increase GP stack with allocatemem()
? allocatemem()
  ***   Warning: doubling stack size; new stack = 8.0 MBytes.
? squfof(122294051504814979)
[20137]
[20137, 3445]
i = 46474
Qfb(321233929, 131349818, -367273962, 0.E-28)
%2 = 73823023
```

We need a large stack because by default gp2c does not generate code to handle the stack (the so-called gerepile code). To instruct gp2c to add gerepile code automatically, we must use the **-g** option. So quit this GP session and launch a new one with -g. Oh well, before that type
ls pari/examples/squfof.gp*

```
pari/examples/squfof.gp    pari/examples/squfof.gp.run
pari/examples/squfof.gp.c  pari/examples/squfof.gp.so
pari/examples/squfof.gp.o
```

These are the files generated by gp2c-run:

- pari/examples/squfof.gp.c is the C file generated by gp2c.
- pari/examples/squfof.gp.o is the object file generated by the C compiler.
- pari/examples/squfof.gp.so is the shared library generated by the linker.
- pari/examples/squfof.gp.run is a file that contains the commands needed to load the compiled functions inside GP.

It is the shared library which is used by GP.

Now let's continue:
./gp2c-run -g pari/examples/squfof.gp

```
parisize = 4000000, primelimit = 500000
? squfof(122294051504814979)
[20137]
[20137, 3445]
i = 46474
Qfb(321233929, 131349818, -367273962, 0.E-28)
%1 = 73823023
```

This time it works with no difficulty using the default stack. We would like to know how much faster the compiled code runs, so we need to load the non compiled *squfof* file in GP:

```
? \r pari/examples/squfof.gp
  ***   unexpected character: squfof(n)=if(isprime(n),retur
                                       ^--------------------
```

Why?? Because *squfof* already exists as an installed function and GP refuses to overwrite it. To solve this problem, we will add a suffix to the name of the compiled function under GP. Quit the session and type:
./gp2c-run -g -s_c pari/examples/squfof.gp

Now the function squfof is named squfof_c instead, so we can do

```
parisize = 4000000, primelimit = 500000
? \r pari/examples/squfof.gp
? #
   timer = 1 (on)
? squfof(122294051504814979)
[20137]
[20137, 3445]
i = 46474
Qfb(321233929, 131349818, -367273962, 0.E-28)
time = 5,810 ms.
%1 = 73823023
? squfof_c(122294051504814979)
[20137]
[20137, 3445]
i = 46474
Qfb(321233929, 131349818, -367273962, 0.E-28)
time = 560 ms.
%2 = 73823023
```

So the compiled version is more than ten times faster than the noncompiled one. However for more complex examples, compiled code usually runs only three times faster on average.

An extra trick: once you have run `gp2c-run` on your script, it is compiled and you can use the compiled version outside `gp2c-run` in any GP session by loading the file with extension `.gp.run`. For example quit the `gp2c-run` session and start `gp` and do

```
parisize = 4000000, primelimit = 500000
? \r pari/examples/squfof.gp.run
```

Now you have access to the compiled function *squfof_c* as well.

2.2 How can I compile directly with `gp2c`?

Now we want to compile directly with `gp2c` to understand what happens. We should run the command

./gp2c pari/examples/squfof.gp > squfof.gp.c

This creates a file squfof.gp.c in the `gp2c` directory. Now read this file with your favorite editor.

The first line is highly system-dependent, but should be similar to:

```
/*-*- compile-command: "/usr/bin/gcc -c -o pari/examples/squfof.gp.o
  -O3 -Wall -I/usr/local/include pari/examples/squfof.gp.c
  && /usr/bin/gcc -o pari/examples/squfof.gp.so
  -shared   pari/examples/squfof.gp.o"; -*-*/
```

This is the command needed to compile this C file to an object file with the C compiler and then to make a shared library with the linker. If you use `emacs`, typing 'M-x compile' will know about this command, so you will just need to type `Return` to compile.

The second line is

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

This includes the PARI header files. It is important that the header files come from the same PARI version as GP, else it will create problems.

The next lines are

```
/*
GP;install("squfof","D0,G,p","squfof","./pari/examples/squfof.gp.so");
GP;install("init_squfof","v","init_squfof","./pari/.../squfof.gp.so");
*/
```

The characters "GP;" denote a command that should be read by GP at start-up. Here, the *install()* commands above must be given to GP to let it know about functions defined by the library. `gp2c-run` copy such commands to the file `./pari/examples/squfof.gp.run`.

Please read the entry about the *install()* command in the PARI manual.

The *init_squfof* function is an initialization function that is created automatically by `gp2c` to hold codes that is outside any function. Since in our case there are none, this is a dummy function. In other cases, it is essential. The next lines are

```
GEN squfof(GEN n, long prec);
void init_squfof(void);
/*End of prototype*/
```

This is the C prototypes of your functions. The rest of the file is the C code proper.

For teaching purpose, let's run the command

./gp2c -s_c pari/examples/squfof.gp > squfof2.gp.c

and look at the difference between squfof.gp.c and squfof2.gp.c:

diff -u squfof.gp.c squfof2.gp.c

```
--- squfof.gp.c Tue Feb 26 13:44:42 2002
+++ squfof2.gp.c        Tue Feb 26 13:44:49 2002
@@ -1,8 +1,8 @@
 /*-*- compile-command: "/usr/bin/gcc -c -o pari/examples/squfof.gp.o
  -DMEMSTEP=1048576 -g -Wall -Wno-implicit  -I/usr/local/include
  pari/examples/squfof.gp.c && /usr/bin/ld -o pari/examples/squfof.gp.so
  -shared   pari/examples/squfof.gp.o"; -*-*/
```

```
 #include <pari/pari.h>
 /*
-GP;install("squfof","D0,G,p","squfof","./pari/examples/squfof.gp.so");
-GP;install("init_squfof","v","init_squfof","./pari/.../squfof.gp.so");
+GP;install("squfof","D0,G,p","squfof_c","./pari/...les/squfof.gp.so");
+GP;install("init_squfof","v","init_squfof_c","./par.../squfof.gp.so");
 */
 GEN squfof(GEN n, long prec);
 void init_squfof(void);
```

If you are not familiar with the `diff` utility, the above means that only the two lines starting with *GP;install* have changed. In fact *squfof* is still named *squfof* in the C file, but the install command tells GP to rename it *squfof_c* in the GP session.

2.3 Using gp2c to find errors in GP scripts

The `gp2c` compiler can also be used to find errors in GP programs. For that we should use the -W option like in

./gp2c -W pari/examples/squfof.gp > squfof.gp.c

```
Warning:pari/examples/squfof.gp:7:variable undeclared
p
Warning:pari/examples/squfof.gp:11:variable undeclared
dd
Warning:pari/examples/squfof.gp:11:variable undeclared
d
Warning:pari/examples/squfof.gp:11:variable undeclared
b
...
Warning:pari/examples/squfof.gp:45:variable undeclared
b1
```

This option lists variables that are used but not declared. It is important to declare all your variables with *my()*, or with *global()*. For `gp2c`, undeclared variables are taken to be "formal variables" for polynomials. For example if you write a function to build a second degree polynomial like

```
pol(a,b,c)=a*x^2+b*x+c
```

you must not declare 'x' here, since it stands for the formal variable x.

2.4 Using compiled functions in a new program

One you have successfully compiled and tested your functions you may want to reuse them in another GP program.

The best way is to copy the install commands of the functions you use at the start of the new program so that reading it will automatically load the compiled functions.

As an example, we write a simple program `fact.gp` that reads

```
install("squfof","D0,G,p","squfof","./pari/examples/squfof.gp.so");
fact_mersenne(p)=squfof(2^p-1)
```

and run GP:

```
parisize = 4000000, primelimit = 500000
? \rfact
? fact_mersenne(67)
i = 2418
Qfb(10825778209, 4021505768, -13258245519, 0.E-28)
%1 = 193707721
```

So all goes well. But what is even better is that gp2c understands the *install* command and will be able to compile this new program.

Also this particular example will fail because as stated above, PARI/GP already has a squfof function, and the linker will pick the wrong one, which is unfortunate.

So use the -p option to gp2c-run to change *squfof* to *my_squfof*.

./gp2c-run -pmy_ -g pari/examples/squfof.gp

This option prefixes my_ to every GP name in the program so as to avoid name clashes. Change `fact.gp` to

```
install("my_squfof","D0,G,p","squfof","./pari/examples/squfof.gp.so");
fact_mersenne(p)=squfof(2^p-1)
```

and run

./gp2c-run -g fact.gp

```
parisize = 4000000, primelimit = 500000
? fact_mersenne(67)
i = 2418
Qfb(10825778209, 4021505768, -13258245519, 0.E-28)
%1 = 193707721
```

Nice isn't it?

But it gets even better: instead of writing the *install* command directly in your script you can just load the `squfof.gp.run` using `\r`: just change `fact.gp` to

```
\r ./pari/examples/squfof.gp.run
fact_mersenne(p)=squfof(2^p-1)
```

2.5 Hand-editing the C file generated by gp2c

If you have some experience in PARI programming, you may want to manually edit the C file generated by gp2c, for example to improve memory handling. Here some tips:

- If you preserve the *install()* at the start of the file, you can use the command **gp2c-run *file*.c** to recompile your file and start a new GP session with your functions added, just as you use **gp2c-run** with GP scripts.
- More generally, **gp2c-run** automatically passes any line in the C file starting with 'GP;' to GP at start-up.
- As explained in Section 2.2, under **emacs** you can type 'M-x compile' to recompile the shared library.

3 Advanced use of gp2c

3.1 gp2c types

Internally gp2c assign types to objects. The most common types are given below:

name	description
void	like in C
bool	boolean, true (1) or false (0)
negbool	antiboolean, true (0) or false (1)
small	C integer long
int	multiprecision integer
real	multiprecision floating point
mp	multiprecision number
var	variable
pol	polynomial
vecsmall	vector of C long (t_VECSMALL)
vec	vector and matrices (excluding *vecsmall*)
list	GP lists
str	characters string as a char *
genstr	characters string as a GEN (t_STR)
gen	generic PARI object (GEN)
lg	length of object (returned by *length*)
typ	type of object (returned by *type*)

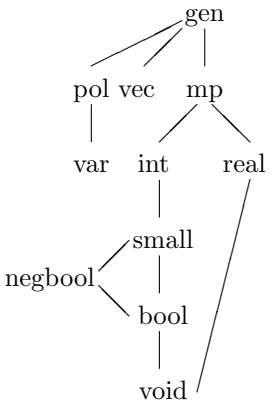

Table 1: Types preorder

Types are preordered as in Table 1. The complete preorder known by gp2c can be accessed by running **gp2c -t**.

Variables are typed. A variable can only take values having a type equal or lower than its type. By default, variables are of type *gen*.

3.2 Type declaration

To declare a variable as belonging to type *type*, use:

function(x:*type*,y:*type*=2)
my(x:*type*, y:*type*=2)

global(x*:type*, y*:type*=2)
for(i*:type*=...

To declare several variables of the same type *type* at once, use:

my(x, y=2)*:type*
global(x, y=2)*:type*

You can even mix the two ways:

my(x, y:*type2*=2)*:type1*

will declare x to be of type *type1* and y of type *type2*.

3.3 Effect of types declaration on default values

Under GP, all GP variables start with a default value, which is *0* for a local variable and *'v* for a global variable *v*.

The `gp2c` compiler follows this rule for variables declared without a type. However, when a variable is declared with a type, `gp2c` will not assign it a default value. This means that the declaration *my(g)* is equivalent to *my(g:gen=0)*, but not to *my(g:gen)*, *my(g)* is equivalent to *my(g:gen='g)*, but not to *my(g:gen)*, and *f(g)=...* is equivalent to *f(g:gen=0)=...*, but not to *f(g:gen)=...*.

This rule was chosen for several reasons:

- The default value (*0* or *'v*) might not be an object suitable for the type in question. For example, *my(v:vec)* declares *v* as being of type *vec*. It would make no sense to initialize *v* to *0* since *0* does not belong to type *vec*. Similarly *global(N:int)* declares *N* as being of type *int*. It would make no sense to initialize *N* to *'N* since *'N* does not belong to type *int*.
- This allows defining GP functions with mandatory arguments. This way, GP will issue an error if a mandatory argument is missing. Without this rule, there is no way to tell apart *0* from a missing argument.
- This allows telling `gp2c` not to generate useless default values.

3.4 Type casting

Sometimes, we know a more precise type than the one the transtyping algorithm can derive. For example if x is a real number, its logarithm might be complex. However, if we are sure x is positive, the logarithm will be real.

To force an expression to belong to type *type*, use the syntax:
expr:type
`gp2c` will check types consistency and output warnings if necessary. For example
f(x:int)=my(r:real); r=log(x^2+1)
`gp2c` will complain that the logarithm might not be real. Since *x^2+1* is always positive, we can write:
f(x:int)=my(r:real); r=log(x^2+1):real

3.5 Example of optimisation

Declaring the types of variables allow gp2c to perform some optimisations. For example, the following piece of GP code

```
rho(n)=
{
  my(x,y);

  x=2; y=5;
  while(gcd(y-x,n)==1,
    x=(x^2+1)%n;
    y=(y^2+1)%n; y=(y^2+1)%n
   );
  gcd(n,y-x)
}
```

generates the following output:

```
GEN
rho(GEN n)
{
  GEN x = gen_0, y = gen_0;
  x = gen_2;
  y = stoi(5);
  while (gequal1(ggcd(gsub(y, x), n)))
  {
    x = gmod(gaddgs(gsqr(x), 1), n);
    y = gmod(gaddgs(gsqr(y), 1), n);
    y = gmod(gaddgs(gsqr(y), 1), n);
  }
  return ggcd(n, gsub(y, x));
}
```

The functions gsqr, gaddgs, gmod, ggcd are generic PARI functions that handle *gen* objects. Since we only want to factor integers with this method, we can declare n, x y of type *int*:

```
rho(n:int)=
{
  my(x:int,y:int);

  x=2; y=5;
  while(gcd(y-x,n)==1,
    x=(x^2+1)%n;
    y=(y^2+1)%n; y=(y^2+1)%n
   );
  gcd(n,y-x)
}
```

The new C code output by gp2c is:

```
GEN
rho(GEN n)        /* int */
{
  GEN x, y;       /* int */
  if (typ(n) != t_INT)
    pari_err(typeer, "rho");
  x = gen_2;
  y = stoi(5);
  while (gequal1(gcdii(subii(y, x), n)))
  {
    x = modii(addis(sqri(x), 1), n);
    y = modii(addis(sqri(y), 1), n);
    y = modii(addis(sqri(y), 1), n);
  }
  return gcdii(n, subii(y, x));
}
```

Now, the code now uses the more specific functions sqri, addis, modii and gcdii.

The most efficient way to use typing is to declare some variables of type *small*. This way, these variables will be implemented by C long variables, which are faster than PARI integers and do not require garbage collecting. However, you will not be protected from integer overflow. For that reason, gp2c will automatically declare some loop indices of type *small* when the range cannot cause overflow. Sometimes gp2c can be too conservative but you can force a loop index to be *small* with the syntax *for(i:small=a,b,...)*.

3.6 Types and member functions

For use with members functions, gp2c provides the following types:

nf for ordinary number fields, i.e., a result given by the GP function *nfinit*.

bnf for big number fields, i.e., a result given by the GP function *bnfinit* which includes class and unit group data.

bnr for ray class groups, i.e., a result given by the GP function *bnrinit*.

ell for elliptic curves, i.e., a result given by the GP function *ellinit*.

gal for galois extensions, i.e., a result given by the GP function *galoisinit*.

prid for prime ideals, i.e., a component of the result given by the GP function *idealprimedec*.

Members functions on typed objects are much more efficient.

4 Common problems

4.1 Meta-commands.

Meta-commands (commands starting with a \) other than \r are currently ignored by gp2c, though a warning will be issued, because it is not clear what

they should do in a compiled program. Instead you probably want to run the meta-command in the GP session itself.

The meta-command `\r`*include* is replaced with the content of the file *include* (or *include*.gp) when `gp2c` reads the file. If you would prefer `gp2c` to link *include*.so to the program instead, see Section 2.4.

4.2 Unsupported functions.

Some GP functions are not yet available to C programs, so the compiler cannot handle them. If you use them you will get the error "unhandled letter in prototype". These are currently *forfactored* and *forsquarefree*.

The functions *forell*, *forsubgroup* and *forqfvec* are currently not implemented as an iterator but as a procedure with callbacks, which limits what you can do inside the loop.

The *forstep* function is supported when the step is a number. If it is a vector, you must add a tag *:vec* to make GP know about it like in

```
f(x)=
{
  my(v);
  v=[2,4,6,6,6,6,6,2,4,6,6]
  forstep(y=7,x,v:vec,print(y))
}
```

This is not needed if the step is a vector or a variable of type vec, but is needed if the step is only an expression which evaluates to a vector.

Some functions are passed to GP by `gp2c-run` at start-up (using the `GP`; syntax) instead of being translated in C: *install* and *addhelp*. In practice, they can be considered as supported.

Also the functions *read, eval, kill* may compile fine but have a surprising behaviour in some case, because they may modify the state of the GP interpreter, not of the compiled program. Please see Section 4.4 for details. For example `f(n)=eval("n^2")` is very different from `f(n)=n^2`. To read files at compile-time use `\r` instead of *read*.

4.3 Dynamically-scoped local variables.

Currenlty `gp2c` does not support dynamically-scoped local variables declared with *local()*. Instead *local()* is treated as an alias for *my()* which declares statically-scoped local variables.

Supporting dynamically-scoped local variables is cumbersome to do in C.

4.4 Memory handling and global variables.

While a lot of work has been done to ensure that `gp2c` handles global variables properly, the use of global variables is still a lot of trouble, so try to avoid them if you do not understand the implications on memory handling.

First, there is a common practice to use undeclared variables as formal variables, for example we assume *x='x* and write *a*x+b* instead of *a*'x+b*. So `gp2c` will not raise an undeclared variable to the rank of global variable unless you

declare it with the *global()* command, or you use it at toplevel (i.e. outside any function). See also Section 2.3

Second, global variables seen by a compiled function are C variables, not GP variables. There is no connection between the two. You may well have two variables with the same name and a different content. Currently GP knows only how to install functions, not variables, so you need to write compiled functions in order to access global variables under GP.

Basically, global variables are allocated in the main stack which is destroyed each time GP prints a new prompt. This means you must put all your commands on the same line. Also global variables must be initialized using the *init_*`<filename>` function before being used, and are only supported with the -g flag.

So you end up doing **gp2c-run -g global.gp**

```
parisize = 4000000, primelimit = 500000
? init_global();myfunction(args);
```

Note that nothing prevents you from calling *init_global* in the GP program. In that case, you can omit the parentheses (i.e, write *init_global*, not *init_global()*) so that you can still run your noncompiled program.

Another way to handle global variables is to use the *clone* function which copies a PARI object to the heap, hence avoids its destruction when GP prints a new prompt. You can use *unclone* to free a clone. Please read the PARI/GP manual for more information about *clone*.

A good use of *clone* is for initializing constant variables: for example in `test/gp/initfunc.gp`, the vector T is initialized by

```
T=clone([4,3,2,2,1,1,1,1,0,0,0,0,0,0,0,0])
```

You must still run the *init_*`<filename>` after starting GP, but after that you can use T safely.

GP itself currently does not know about *clone* and *unclone*, but you can use dummy functions

```
clone(x)=x
unclone(x)=while(0,)
```

when running uncompiled.

4.5 Support for t_VECSMALL

When accessing the component of a t_VECSMALL, it is necessary that that the object has been declared of type *vecsmall*.

For example *my(v); v = vecsort(V,,1); print(v[1])* does not work, but *my(v:vecsmall); v = vecsort(V,,1); print(v[1])* or *my(v:vecsmall); v = vecsort(V,,1); print(v:vecsmall[1])* works.

4.6 GP lists

GP lists and maps are not fully supported by `gp2c`. A partial support is available with the *list* type. You must tell `gp2c` that a variable will contain a list or a map by using *L:list* inside a declaration, where L is the name of the variable as explained in Section 3.

Currently, assigning to a list element (*L[2]=x*) will not work and lists and maps will not be freed unless you explicitly use *listkill*.

Note: The PARI user's manual states that lists are useless in library mode.

4.7 The *install* command

The *install* command is interpreted as a `gp2c` directive. This allows using installed function in compiled programs, see Section 2.4.

However this has some side-effects:

- If present, the *lib* argument must be a string, not an expression that evaluate to a string.
- The *install* command is not compiled, instead it is added to the list of functions to install.

4.8 What about main() ?

There are two different issues with main(): first this is reserved function name in C, so using it with `gp2c` will cause a name clash. To avoid this, either rename it or use the flag -p.

Secondy `gp2c` has no support for generating stand-alone GP programs. However adding manually a main() C function is not difficult in general.

5 Command-line options of gp2c

Here is a brief description of the main options of `gp2c`, which can be seen with `./gp2c -h`.

In Section 2.1 we saw how to use the `-g` option.

- -g tells `gp2c` to generate gerepile calls to clean up the PARI stack and reduce memory usage. You will probably need this option, although the C code will be easier to read or hand-edit without it.
- -o*file* tells `gp2c` to write the generated C file to the file *file* instead of the standard output.
- -iN allows you to change the indentation of the generated C file. So if you want 4 spaces, just use the `-i4` option with `gp2c`. The default is 2.
- -C adds code to perform range checking for GP constructs like *x[a]* and *x[a,b]*. This also checks whether *x* has the correct type. By default `gp2c` does not perform such check, which can lead to a runtime crash with invalid code. This option causes a small runtime penalty and a large C code readability penalty.
- -L adds compiler directives to the code so that warning and error found by the compiler are prefixed with the line number in the original GP file instead of the C file.
- -W is useful for debugging the .gp file, in the sense that it detects if some local variables are undeclared. For example, if the file `algorithm.gp` has a routine like

```
radical(x)=F=factor(x)[,1];prod(i=1,length(F),F[i])
```

The variable 'F' is undeclared in this routine, so when running gp2c with the -W option it prints

Warning:algorithm.gp:1:variable undeclared F

At present, an undeclared variable is taken to be a "formal variable" for polynomials by gp2c, so do not declare it if that is what you intend. For example in `pol(a,b,c)=a*x^2+b*x+c` you must not declare x since it stands for the formal variable *'x*.

- -p*prefix* A problem with C is that it is subject to name clashes, i.e., if a GP variable in your routine has the same name as a C symbol in the pari library, the compiler will report strange errors. So this option changes ALL user variables and user routine names by adding a prefix *prefix* to them. For example the GP routine *add(x,y)* with **-pmy_** will become the C function my_add(x,y).

  Try this option each time the compiler fails to compile gp2c output to see if there is a name conflict. If this is the case, change the name in your GP script. It may be difficult to find conflicting names if your compiler is not verbose enough and if you are not familiar with the PARI code and C in general.

  Example of conflicting names are top,bot,prec,un, but there are thousands of others and they may be system-dependent.

- -s*suffix*: Add *suffix* to the names of the installed functions under GP. This is to avoid clashes with the original GP script. For example, if you want to compare timings you may want to use the option **-s_c** This does not affect the C code, only the *install* commands.

- -S: Assume strict prototypes for functions. This is related to the 'strictargs' GP default. This makes all arguments of functions defined in the file mandatory unless the function supplies an explicit default value. This does not affect the C code, only the prototype code in the *install* commands. In this example, the prototype code changes from "D0,G,DG" to "GDG".

  ```
  test(data,flag=0)={CODE}
  ```

- -h gives the help.

- -v gives the gp2c version.

- -l prints a list of all the GP functions known by the compiler. So if a routine contains a GP routine not on this list, gp2c will show an error when trying to translate it.

  Reasons why a GP function may not be known by the compiler are:

  – The function is not part of the PARI library. See Section 4.2

  - You use the old PARI 1.39 function names instead of the new ones. `gp2c` currently does not know about the 'compat' default. Use *whatnow* under GP to get the current name. For example, *mod()* is now *Mod()*.
  - You use a GP function that does not exists in the GP version `gp2c` was compiled against. Please recompile `gp2c` against this GP version.

    Normally no functions are added between two stable releases of GP with the same minor version number (say 2.1.1 and 2.1.2) so there is no need to recompile `gp2c` when you upgrade. But if you use the developement versions, you need to recompile. Also some new developement versions may break old versions of `gp2c`, so upgrade gp2c at the same time.

    However, if you want to compile scripts which do not use the new functions, you do not need to recompile. Note that you may use the GP environment variables to tell `gp2c-run` which GP to use.

- -t Output the table of types known to the compiler, see Section 3.