Moving GP out of the Dark Ages

GP>

(The title is a reference to Chapter 8 of the first book of Henri, "Factoring in the Dark Ages")

Terminology

As the name imply, the PARI/GP computer algebra system is made of two components:

- PARI, a C library of number-theory oriented functions
- GP, a simple scripting language giving access to PARI.

Since the origin, PARI has been the main focus of PARI/GP development. As a consequence, the design of the GP interpretor is very primitive and hard to change. This resulted in the GP language to be very limited in feature.

Good Ol' GP limitation

The GP interpretor executes the code as soon as it parses it. There is no room for optimisation and syntax check based on the knowledge of the whole program:

```
? for(i=1,10^7,HC)
? ##
  ***   last result computed in 3,556 ms.
? for(i=1,10^7,Henri_Cohen)
? ##
  ***   last result computed in 4,148 ms.
? for(i=1,10^7,A_Course_in_Computational_Algebraic_Number_Theory)
? ##
  ***   last result computed in 8,044 ms.
```

This also makes very hard to add new higher-level feature.

Starting from scratch

To address those issues, we decided to rewrite the GP evaluator mostly from scratch, and then to add new features.

The aim of this talk is to explain

1. How the new GP evaluator works.

2. What are the new features implemented and their impact on the language.

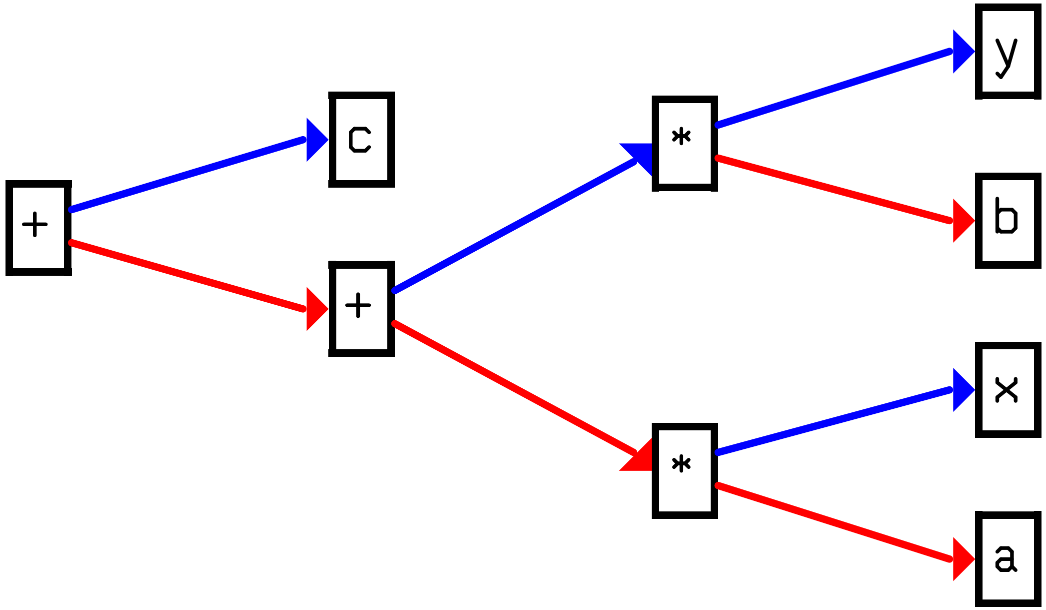
Rewriting GP as a combined compiler/evaluator

Three steps

1. translate the input to a syntactic tree
2. translate the syntactic tree to a sequence of codes
3. evaluate the sequence of codes

For the first stage we actually reuse the GP2C parser, which is written in GNU bison.

Consider the expression $a{*}x{+}b{*}y{+}c$. The parser will return the following syntactic tree:



We do a depth-first traversal of the tree to transform it to Reverse Polish Notation (RPN): $a\ x\ *\ b\ y\ *\ +\ c\ +$.

This notation is to be understood as follow: When you see an operand you put it on a stack. When you see an operation you apply it to the operands at the top of the stack, replacing them by the result. Evaluating $a\ x\ *\ b\ y\ *\ +\ c\ +$ lead to write successively:

a

$a \quad x$

$a * x$

$a * x \quad b$

$a * x \quad b \quad y$

$a * x \quad b * y$

$a * x + b * y$

$a * x + b * y \quad c$

$a * x + b * y + c$

Using a RPN-like technique, we encode the tree in GP assembly:

```
push a
push x
call *
push b
push y
call *
call +
push c
call +
```

The left part are called opcode, the right part are operands. The GP assembly is very simple: each opcodes take exactly one operand. Actually opcode and operands are stored as numerical values.

Actually GP-new includes a disassembler so you can check what I say:

```
? disassemble(closure(a*x+b*y+c))
00001   pushdyn         a
00002   pushdyn         x
00003   callgen2        _*_
00004   pushdyn         b
00005   pushdyn         y
00006   callgen2        _*_
00007   callgen2        _+_
00008   pushdyn         c
00009   callgen2        _+_
```

The evaluation is then very simple: We maintain a stack and perform the operation as requested.

Currently there are 50 opcodes defined:

```
pushlong, pushgen, pushreal, pushstoi, pushvar,
pop,
stoi, itos, tostr, varn, copy, copyifclone,
precreal, precdl,
vec, mat, col,
stackgen,
compo1, compo2, compoC, compoL,
pushptr, endptr,
compo1ptr, compo2ptr, compoCptr, compoLptr,
calllong, callgen, callgen2, callint, callvoid, calluser,
derivgen, derivuser,
newframe, saveframe,
pushdyn, storedyn, newptrdyn, simpleptrdyn,
pushlex, storelex, newptrlex, simpleptrlex,
getarg, defaultarg, localvar, localvar0
```

Lexically-scoped local variables

In Mathematic notation, variables are lexically scoped: If you define

$$a = Pi/4, \quad f : \left| \begin{array}{ccc} \mathbb{R} & \longrightarrow & \mathbb{R} \\ x & \longmapsto & sin(x-a) \end{array} \right., \quad S = \sum_{a=1}^{100} f(a) \ ,$$

you do not expect to get $S = 0$. However with GP:

```
? a=Pi/4;
? f(x)=sin(x-a);
? sum(a=1,100,f(a))
%2 = 0.E-38
```

What happened ?

While a is a local variable in the `sum`, it is dynamically-scoped, so it is seen by the function f hidding the global variable a.

With the new GP evaluator, a is lexically-scoped, and so is only visible inside f:

```
? a=Pi/4;
? f(x)=sin(x-a);
? sum(a=1,100,f(a))
%3 = 0.28646139829840941455658208923051808773
```

Declaring local variables

The new GP evaluator allows to declare both lexically-scoped and dynamically-scoped local variables using the keywords:

- `my(...)` for lexically-scoped local variables.
- `local(...)` for dynamically-scoped local variables.

Furthermore these keywords can appear anywhere, not just at the start of the function. The scope last until the end of the block.

```
? sum(i=1,100,my(s=sin(i));s*exp(s))
%1 = 56.70049354977032709032031599900644969 1
? s
%2 = s
```

Backward incompatibility

With the new GP evaluator, local variables implicitly declared (function parameters and loop indices) are always lexically-scoped. This is an incompatibility with the old GP evaluator. However sometimes this was hidding bugs:

```
ellsquare(e,P)=elladd(E,P,P);

ellquad(E,P)=ellsquare(E,ellsquare(E,P));

ellquad(ellinit([0,0,0,-4,0]),[2,1])

%1 = [5452/1681, -324319/68921]
```

Closures and anonymous functions

Consider the following mathematical bits: let us define M as

$$M \left| \begin{array}{ccccc} \mathbb{Q} & \to & \mathbb{Q} & \to & \mathbb{Q} \\ \alpha & \mapsto & x & \mapsto & \alpha x \end{array} \right.$$

Let α be an element of $\mathbb{Q}$ and $f = M(\alpha)$: f is a function which "remember" the value of α: indeed $f(1) = \alpha$. Now compare with the equivalent GP code:

```
? M(alpha)=g(x)=alpha*x
? M(5)
? g(2)
%1 = 2*alpha
```

We see two problems: first there is no way for f to return a function and we do not get the correct result.

The new GP interpretor allow to define anonymous functions: the syntax is $(x_1, \ldots, x_n)$`->expr`

If there is a single variable, the shortcut `x->expr` is also accepted. We can define M as function returning a function:

```
? M(alpha)=x->alpha*x
```

furthermore in GP-new, functions acts as closure with respect to lexically-scoped local variables: they remember the value of lexical variables that were visible when they were first defined. We get

```
? f=M(5)
? f(2)
%2 = 10
```

Functions as first-class object

GP-new adds a new GEN type 't_CLOSURE' to store functions. This type is defined as follow:

- f[1] is the arity of the function.
- f[2], f[3] are the list of opcodes and operands.
- f[4] is the list of data (which might include t_CLOSUREs so it is a recursive type)
- f[5] is the GP code for the function (used to print it).
- f[6] if present is the closure context, i.e. the value of variables that are predefined.

In our example f[6] would contains alpha=5.

Advantages

- `f(`x_1`,...,`x_n`)=expr` is now just an alias for `f=(`x_1`,...,`x_n`)->expr`.
- GP functions can return functions.
- GP functions can take functions as parameter:

```
? apply(func,vec)=vector(#vec,i,func(vec[i]));
? apply(x->x^2,[1,2,3,4])
%2 = [1, 4, 9, 16]
```

- GP functions can be stored in local variables.
- GP functions can be stored in vectors, matrices, etc.

Lambda calculus

Probably the most asked-for pari feature: we can do lambda calculus in GP and define Church numerals

```
one = f->f
two = f->x->f(f(x))
three=f->x->f(f(f(x)))
inc = n->f->x->f((n(f))(x))
add = (n,m)->f->x->(n(f))((m(f))(x))
mul = (n,m)->f->n(m(f))
pow = (n,m)->m(n)
dec = n->f->x-> ((n (g->h->h (g(f)))) (u-> x)) (u-> u)
sub = (n,m)->(m(dec))(n)
```

When all these new features will be available ?

Today actually! Just get the experimental archive at

`http://pari.math.u-bordeaux.fr/~bill/pari.2.4.2.bill1.tar.gz`

Other planned features

- a debugger for GP scripts
- *fill in the blank with your features*