The GP language:
past, present and future

GP>

The past

The GP interpreter was written by Dominique Bernardi and later maintained by Karim Belabas. It is a text-based parser/evaluator:

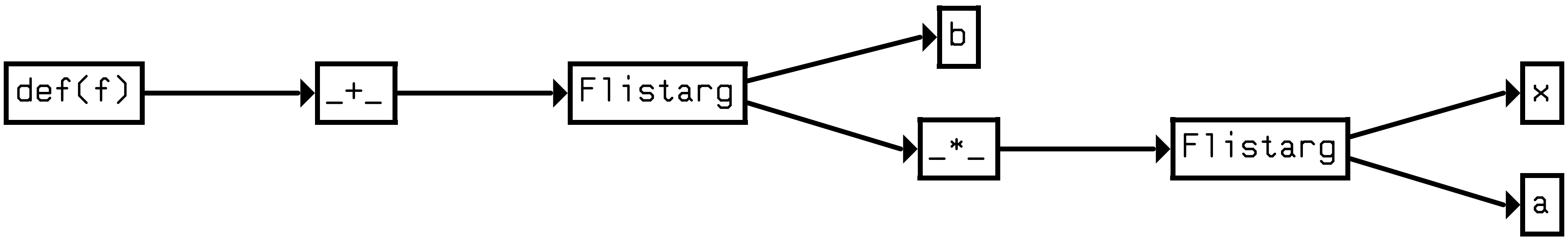
- It is slow since it needs to reparse the text code every time it is evaluated.
- It is difficult to maintain since everything is done at once.
- It cannot perform optimisation on the input.
- It is difficult to fix short-comings of the language.

The present

GP2C includes a new GP parser written in the standard YACC language. This parser was used to make an experimental GP interpreter available from

`<http://pari.math.u-bordeaux.fr/~bill/pari.bill.tar.gz>`

It uses the GP2C parser to build a syntactic tree and then evaluate directly the tree.



Syntactic tree for `f(x)=a*x+b`

The future

The ~~horror museum~~ wishlist 1/2

1. Separate name space for functions and variables.

2. Lexically scoped variables instead of dynamically scoped.

3. Passing functions as parameter of functions.

4. Blocks for functions and control statements.

The wishlist 2/2

5. Container types.

6. Objects referencing containers.

7. Support for `gp2c` types.

Separate namespace for functions and variables.

A function and a variable cannot have the same name. This leads to strange errors like

```
apply(f,x)=subst(f,variable(f),x);
f(x)=x
  ***   unused characters: f(x)=x
                             ^-----
```

Lexically scoped variables.

One of the obscure and confusing features of GP is the local() variables. The scope of a local() variable expands to all subroutine calls. For example the below code prints 1, not 2:

```
g()=print(z);
f(z)=g();
z=2;f(1);
```

GP2C does not implement this behaviour.

Passing functions as parameter of functions.

It is not possible to implement new control statement or iterators in GP. Allowing to pass functions as parameters would allow that.

```
sumprime(a,b,f()) =
{
   local(s=0);
   forprime(p=a,b,s+=f(p));
   s
}
```

Blocks for functions and control statements.

Instead of code looking like

```
f(a,b) = local(n=0,p); for(i=a,b, p=i^2+1; if(isprime(p), n+=p)); n
```

We could allow a syntax similar to

```
f(a,b)=
{ local(n=0);
  for (i = a, b)
  { local(p = i^2+1);
    if (isprime(p), n += p)
  }
  n
}
```

and allow variables to be local() to a block.

Container types.

Problems with complex data structures returned by functions like nfinit(), bnfinit(), ellinit(), galoisinit(), etc. stored in a t_VEC:

- No reliable way to identify them: GP functions and member functions have to make a guess.

- Difficult to add new types without breaking assumptions.

```
? bnfinit(vector(9,i,x))
%1 = [[;], [;], [;], [;], []~, 0, [x, x, x, x, x, x, x, x, x],
[[1, [], []], 1, 1, [2, -1], []], [[;], [], []], 0]
```

Objects referencing containers.

Some operations on objects like ideals or number field elements require data from another structure. A reference to that structure could be stored in that object so that

```
K =bnfinit(P);
i2=idealprimedec(K,2)[1];
i3=idealprimedec(K,3)[1];
idealstar(K,idealmul(K,i2,i3))
```

The last line could be shortened to idealstar(i2*i3)

Support for gp2c types

GP2C introduced a new syntax for "types".

```
f(v:vec,n:small)=
{
  sum(i=1,n,
        v[i]:int^2)/n;
}
```

GP should be able to cope with the new syntax, to check that the types match at runtime, and maybe to optimize the evaluation.