Le futur de l'interpréteur GP.

$\textcolor{red}{\mathrm{GP}}>$

PARI/GP

PARI/GP est un système de calcul formel orienté vers la théorie des nombres. Il est composé de :

- PARI, une bibliothèque de fonctions C.
- GP, un langage simplifié permettant d'accéder au fonctions de PARI.
- GP2C, un programme convertissant les scripts GP en programme C utilisant PARI.
- Distribué sous la license GNU GPL.
- Créé en 1987 par Henri Cohen à Bordeaux.
- GP a été écrit par Dominique Bernardi.
- Maintenu par Karim Belabas depuis 1995.
- Site Web : `http://pari.math.u-bordeaux.fr`

L'interpréteur GP traditionnel

Jusqu'à la version 2.3, GP utilise un évaluateur récursif qui évalue le code dès qu'il est interprété, ce qui oblige à réinterpréter le même code à chaque exécution.

```
? 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.
```

Ce système ne permet aucune optimisation et rend difficile l'introduction de fonctionnalités avancées dans le langage.

Le nouvel interpréteur GP 2.4

Pour corriger ces limitations, nous avons décider de réécrire l'évaluateur GP et d'ajouter de nouvelle fonctionnalités (version 2.4).

Le but de cet exposé est d'expliquer :

1. Comment le nouvel interpréteur GP fonctionne.
2. Quelles sont les nouvelles fonctionnalités et leur impact sur le langage GP.
3. Quelles sont les optimisations qui ont été rendues possibles.

Réécrire GP comme un compilateur/évaluateur combiné

Trois étapes :

1. Transformer l'entrée en un arbre syntaxique.
2. Transformer l'arbre syntaxique en une suite de codes.
3. Évaluer la suite de codes.

Grammaire

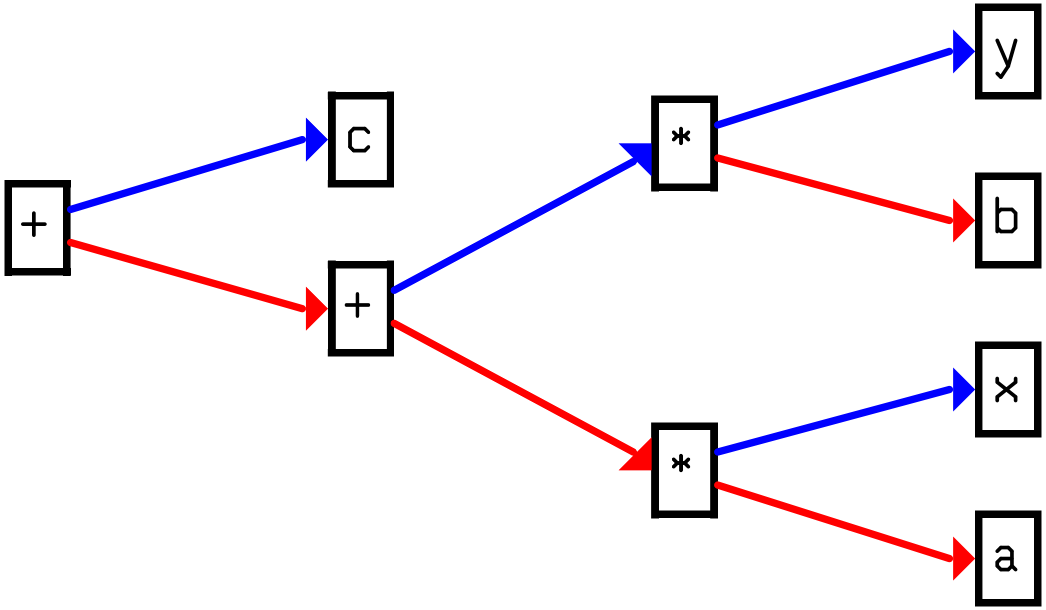
Nous réutilisons la grammaire écrite pour GP2C.

```
...
expr: INTEGER
    | REAL
    | VARIABLE
...
    | expr '*' expr
    | expr '+' expr
...
```

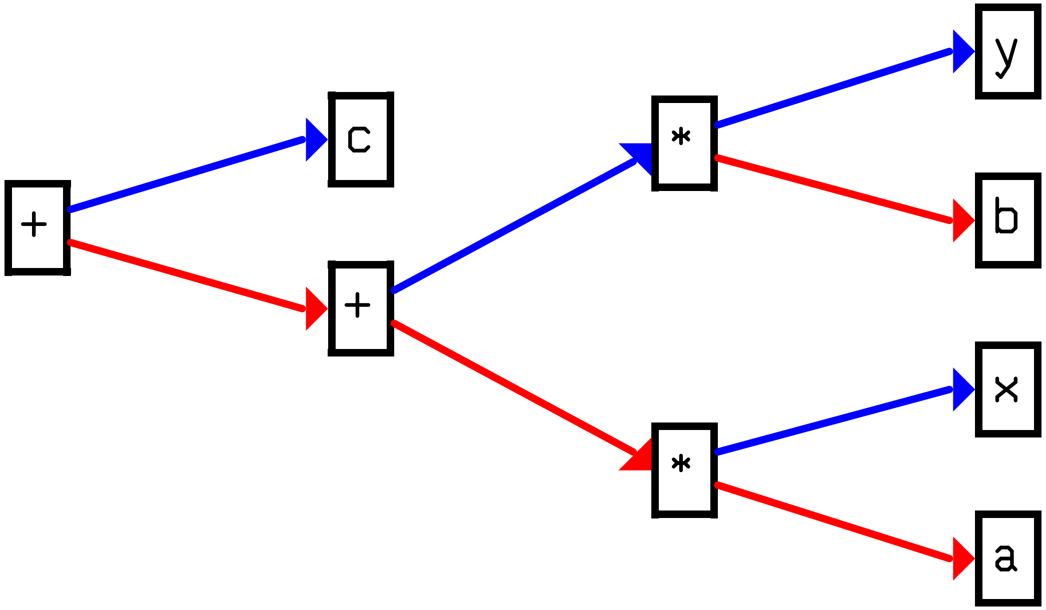
Le programme `bison` est utilisé pour générer un automate à pile qui reconnaît cette grammaire.

Arbre syntaxique

L'automate construit l'arbre syntaxique de l'expression en entrée.
Exemple : pour l'expression $a * x + b * y + c$, l'automate retourne



Compilation



Nous effectuons un parcours en profondeur de l'arbre pour obtenir la forme polonaise inverse (RPN) de l'expression : $a\ x * b\ y * + c +$.

Cette forme se lit de la manière suivante : On parcours la liste de gauche à droite. Si l'on voit une opérande, on la met sur la pile. Quand on voit une opération on l'applique au(x) élément(s) au sommet de la pile et que l'on remplace par le résultat. Évaluer $a\ x\ *\ b\ y\ *\ +\ c\ +$ donne successivement :

a

$a \quad x$

$a*x$

$a*x \quad b$

$a*x \quad b \quad y$

$a*x \quad b*y$

$a*x+b*y$

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

$a*x+b*y+c$

À partir de la forme polonaise inverse nous obtenons directement l'expression en assembleur GP :

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

La colonne de gauche contient les 'opcodes' et la droite les 'opérandes'. Chaque opcode prend exactement une opérande.

Le nouvel interpréteur GP inclus un désassembleur :

```
? disassemble(()->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        _+_
```

L'évaluation est très simple : Il suffit d'effectuer les opérations dans l'ordre et de mettre à jour la pile.

Il y a actuellement 51 opcodes définis :

```
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,
newframe, saveframe,
pushdyn, storedyn, newptrdyn, simpleptrdyn,
pushlex, storelex, newptrlex, simpleptrlex,
getarg, defaultarg, localvar, localvar0
checkargs, checkargs0, defaultitos
```

Variables locales à portée lexicale

Dans les notations mathématiques, la portée des variables est limitée à leurs blocs de définition. Dans l'expression suivante

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

nous avons $S \neq 0$. Cependant GP-2.3 donne

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

Quel est le problème ?

Bien que a soit une variable locale dans la somme `sum`, cette variable est à portée dynamique et est donc vue par la fonction f ce qui cache la variable globale a.

Avec GP-2.4, a est à portée lexicale, et est seulement visible dans S :

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

Declaration des variables locales

GP-2.4 permet de déclarer des variables locales soit à portée lexicale, soit à portée dynamique :

- `my(a)` : déclare a à portée lexicale.
- `local(b)` déclare b à portée dynamique.

De plus il est maintenant possible de déclarer des variables locales n'importe où dans le code.

```
? sum(i=1,100,my(s=sin(i));s*exp(s))
%1 = 56.700493549770327090320315999006449691
? s
%2 = s
```

Incompatibilités entre GP 2.3 et 2.4

Sous GP 2.4, les variables locales déclarées implicitement (les paramètres de fonctions et les indices de boucles) sont toujours à portée lexicale. C'est une incompatibilité avec GP 2.3 qui les considérait à portée dynamique, ce qui souvent cachait des 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]
```

Dans `ellsquare` la variable e n'est pas utilisée.

Fonctions anonymes et fermetures

Soit l'expression mathématique suivante :

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

Soit α un rationnel et $f = M(\alpha)$: f est une fonction qui "se souvient" de la valeur de α : en effet $f(1) = \alpha$.

Essayons de définir M sous GP 2.3 :

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

Ça ne marche pas.

GP 2.4 permet de définir des fonctions anonymes ainsi :

$$(x_1,\ldots,x_n)\texttt{->expr}\qquad.$$

Si $n = 1$, le raccourci x_1`->expr` est accepté. Nous pouvons donc définir M comme une fonction retournant une fonction :

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

De plus les fonctions sont fermées par rapport aux variables globales lexicales : elles "se souviennent" des valeurs des variables lexicales qui étaient visibles quand elles furent définies. On obtient alors :

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

Fonctions anonymes

Nous avons ajouté un nouveau type d'objet PARI ('t_CLOSURE') pour définir des fonctions anonymes. Ce type est défini ainsi :

– f[1] : nombre d'arguments de la fonctions.
– f[2] : liste des opcodes.
– f[3] : liste de opérandes.
– f[4] : liste des données.
– f[5] : code source GP de la fonction.
– f[6] : contexte, listes des valeurs de variables prédéfinies.

Dans notre exemple, f[6] contient 'alpha=5'.

Avantages

- `f(`x_1`,...,`x_n`)=expr` est simplement un alias pour `f=(`x_1`,...,`x_n`)->expr`.
- Les fonctions GP peuvent retourner des fonctions.
- Les fonctions GP peuvent prendre des fonctions comme paramètres.
- Les fonctions GP peuvent être stockées dans des variables locales.
- Les fonctions GP peuvent être stockées dans des vecteurs ou des matrices.

Fonctions de niveau supérieur

- `apply(f,v)`
  ```
  ? apply(x->x^2,[1,2,3,4])
  %2 = [1, 4, 9, 16]
  ```
- `select(v,f)`
  ```
  ? select([1,2,3,4],isprime)
  %1 = [2, 3]
  ```
- f' : dérivation numérique de f.
  ```
  ? f=log'; f(2)
  %1 = 0.5000000000000000000000000000000000000000
  ```
- `vecsort(v,cmp)`
  ```
  ? vecsort([-4,1,2,5],(x,y)->abs(x)-abs(y))
  %1 = [1, 2, -4, 5]
  ```

Optimisations

Le compilateur permet un certain nombre d'optimisations :
- Éviter la multi-évaluation des constantes.
- Accélérer l'accès au variables.
- Accélérer les appels de fonctions.
- Éviter les copies lors des passages d'arguments.