JOURNAL
de Théorie des Nombres
de BORDEAUX
anciennement Séminaire de Théorie des Nombres de Bordeaux

Bill ALLOMBERT

A new interpretor for PARI/GP

Tome 20, n° 3 (2008), p. 531-541.

<http://jtnb.cedram.org/item?id=JTNB_2008__20_3_531_0>

Journal de Théorie des Nombres
de Bordeaux **20** (2008), 531-541

A new interpretor for PARI/GP

par Bill ALLOMBERT

Dedicated to Henri Cohen on his 60th birthday

Résumé. Quand Henri Cohen et ses collaborateurs décidèrent d'écrire la bibliothèque PARI il y a vingt ans, la création d'un langage et d'un interpréteur pour la manipuler a été un effet secondaire, presque accidentel. Bien que le langage GP soit devenu l'interface de la bibliothèque PARI de très loin la plus utilisée, et permette la manipulation d'objets de très haut niveau, il est resté très primitif, de même que son interprèteur gp, venant directement des années 70.

Nous avons réécrit gp comme un compilateur/évaluateur, implantant plusieurs functionnalités de haut-niveau, qui devrait permettre à GP d'entrer dans les années 90.

Abstract. When Henri Cohen and his coworkers set out to write PARI twenty years ago, GP was an afterthought. While GP has become the most commonly used interface to the PARI library by a large margin, both the gp interpretor and the GP language are primitive in design. Paradoxically, while gp allows to handle very high-level objects, GP itself is a low-level language coming straight from the seventies.

We rewrote GP as a compiler/evaluator pair, implementing several high-level features (statically scoped variables, anonymous functions, closures as first class objects) that should move GP into the nineties.

Contents

Manuscrit reçu le 14 novembre 2008.

1. Introduction

The PARI/GP computer algebra system [4] 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 primitive and hard to change. As a result, the GP language is slow and limited in features.

The first GP incarnation was written 20 years ago (mainly by Dominique Bernardi[1]) as a desktop calculator that would use the then nascent PARI library to perform multi-precision arithmetic. When support for functions was added, it became the dominant paradigm for new features: for example in GP, not only do control statements look syntactically like function calls, but they are actually implemented as function calls. As a consequence, GP is a small language in term of features since all the high-level interfaces are actually defined by the PARI library.

When we started to work on the GP2C compiler [1], the deficiency of GP became obvious, and the lack of formal specification hindered the project. As part of our work on GP2C, we documented what the GP interpretor was doing and wrote a test-suite exercising that behavior.

This finally allowed us to write a new GP interpretor that preserved as much as advisable the behavior of the old interpretor. This new GP interpretor is used by PARI/GP since version 2.4.2. Some features presented in this paper are only implemented in version 2.4.3 and later.

2. Design of the old GP interpretor

The old GP interpretor was based on a recursive-descent parser that evaluated the code as soon as it was parsed. To perform basic syntax checking before evaluation, the whole parser was duplicated (minus the evaluating code) to form a dummy interpretor that only did syntax checking. Expressions where first syntax-checked and then evaluated.

This design leads to poor performance since the same expressions need to be parsed every time they are evaluated. In the example below, we observe that longer variables name take significantly more time to be evaluated.

[1]It was later cleaned up and enhanced by Karim Belabas and Ilya Zakharevich in the mid-nineties

```
? N = 10^7;
? for(i=1,N, Henri_Cohen)
time = 2,720 ms.
? for(i=1,N, ACourseInComputationalAlgebraicNumberTheory)
time = 4,220 ms.
```

Features implemented by the old GP interpretor were mostly limited to the ability to call built-in functions, to define new functions and call them, to use global variables (that did not need to be declared), to define dynamically-scoped local variables (in a function header only, not in arbitrary blocks), and to use them. While a large range of control statements and control flow operations were available, they were implemented as built-in PARI functions, not performed by the evaluator properly. However the interpretor was able to detect control-flow change, i.e. analogs of the C statements `break`, `continue` and `return`.

The main task performed by the interpretor was to convert built-in GP function calls to C function calls. For that purpose, built-in GP functions were associated to a C function pointer and a signature which was a character string indicating the return type (`void`, `long`, `int` or PARI object type) and the type of each argument in order.

Special codes denoted arguments of control statement, which should not be evaluated but passed as a character string for deferred execution, and optional arguments.

Calling user functions was performed by recursively calling the interpretor on the function text.

3. Design of the new GP interpretor

The new GP interpretor is made of three parts, that are called successively: a parser, a bytecode compiler and a bytecode evaluator, which are detailed below.

3.1. The parser.

3.1.1. ***Input filtering and preprocessing.*** This stage removes comments and white spaces from the input and deals with meta-commands and file inclusions.

For this stage we reuse the old gp filtering code. At the end, the output is a single string.

3.1.2. ***Lexical analysis.*** This stage transforms the input string into a string of tokens. Each token has an associated syntactic value which is its offset in the input string and its length.

For this stage, we have written a simple custom lexical analyzer (function `pari_lex`) which returns the type of the next token in the string along with its length and position in the string. No semantic values are affected

to tokens at this stage. Semantic values are computed at a later stage from the offset and length by the compiler. Thus, the lexical analyzer does not need to allocate memory to store values, and spend time copying them.

3.1.3. ***Syntactic analysis.*** This stage builds a binary tree from the string of tokens. To each node we associate the location of the original expression in the input string.

For this stage, we reuse the grammar written for GP2C. The formal description of the grammar (in `bison` [2] format) is in the source file `src/language/parse.y` of the PARI distribution, that we excerpt below.

```
seq: /*empty*/ | expr | seq ';' | seq ';' expr

expr: KINTEGER
    | KREAL
    | KSTRING
    | '\'' KENTRY
    | expr '(' listarg ')'
    | KENTRY '(' listarg ')'
    | funcid
    | lvalue
    | matrix
    | definition
    | lvalue '=' expr
    | lvalue "++"
    ...
    | lvalue "+=" expr
    | '!' expr
    | '#' expr
    | expr "||"  expr
    | expr "&&"  expr
    | expr "=="  expr
    ...
    | expr '<'   expr
    | expr '+'   expr
    ...
    | expr '*'   expr
    | '+' expr
    | '-' expr
    | expr '^' expr
    | expr '~'
    | expr '\''
    | expr '!'
    | expr matrix_index
```

```
    | '(' expr ')'

matrix_index: '[' expr ',' expr ']'
    | '[' expr ']'
    | '[' expr ',' ']'
    | '[' ',' expr ']'

lvalue: KENTRY | lvalue matrix_index

matrixelts: /*empty*/ | matrixelts ',' expr

matrixlines: matrixelts  ';' matrixelts
   | matrixlines ';' matrixelts

matrix: '[' ']'
      | '[' ';' ']'
      | '[' matrixelts ']'
      | '[' matrixlines ']'

listarg: seq | listarg ',' seq

definition: KENTRY '(' listarg ')' '=' seq
          | lvalue "->" seq
          | '(' listarg ")->" seq
```

The `bison` program is used to generate a stack automaton that matches the grammar (function `pari_parse`) and constructs a syntactic tree. This tree is stored as an indexed array of nodes (defined in the header file `src/language/tree.h`). It is a binary tree, each node having a label and optionally a left child and a right child. Each node carry the following information:

- function: A label identifying the function of the node (whether this is a constant, a function call, a list of arguments, etc.)
- left/right child: a left or right child (optional)
- text start, text length: length and pointer to the start of the corresponding text of the full expression represented by the node and its children.

We use the location tracking feature of `bison` to automatically compute the location of a node from the location of the components tokens.

3.2. The bytecode compiler.

3.2.1. ***The `t_CLOSURE` PARI object.*** The purpose of the bytecode compiler is to convert the syntactic tree to a `t_CLOSURE` PARI object, hold closures and functions in compiled form. This type has a number of components, some of them optional:

(1) arity of the object: a small integer.
(2) bytecode, given as pairs of opcodes and operands; each opcode takes exactly one operand.
(3) program data: a vector of PARI objects that are referenced by the operands. They can be integers, character strings, or `t_CLOSURE` for deferred evaluation.
(4) debugging data, for source-level debugging: code position corresponding to each opcode, original names of lexical local variables (as input by the user).
(5) (optional) source code: a string containing the GP function code as input by the user. This is used to display the object and for source-level debugging.
(6) (optional) the closure context.

3.2.2. ***The bytecode.*** We designed a simple bytecode targeting an evaluator that operates on a last-in first-out stack. Each instruction is composed of a one-byte opcode and an operand, coded as a C long integer.

Generally, executing an instruction involves reading a number of input data at the top of the stack, removing them, and pushing a number of output data on the stack which will serve as input to subsequent opcodes.

The major opcode families are

- *push*: push an object on the stack.
- *call*: evaluate a function on arguments taken from the top of the stack.
- *typecast*: convert the object at the top of the stack to another type, e.g. convert a C integer to a PARI integer.
- *store*: store the object at the top of the stack in a variable.
- *component*: extract a component of an object specified by the value at the top of the stack.

There are no opcodes for control flow, branching or jumping: this task is deferred to special functions that take closures as input.

The PARI library includes a function `closure_disassemble` that allows to disassemble a `t_CLOSURE` object. The following is a commented example of disassembly:

```
? install(closure_disassemble, vG)
? f(x) = 2*x+1
? closure_disassemble(f)
```

```
00001   getargs         1       \\ take one argument
00002   pushlong        gen_2   \\ push 2
00003   pushlex         -1      \\ push argument
00004   callgen2        _*_     \\ call multiply
00005   pushlong        gen_1   \\ push 1
00006   callgen2        _+_     \\ call add
```

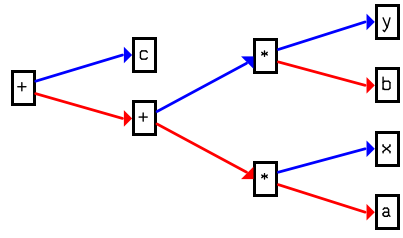
Others bytecodes designed to operate on a stack include the JAVA bytecode targetting the Java virtual machine ([3]), and the internal encoding of user input in several Hewlett-Packard pocket calculators.

3.2.3. ***Copy optimisation.*** This stage is performed before actual compilation. It annotates the syntactic tree, marking nodes associated to subexpressions that do not alter GP variables. This allows the compiler to know whether it is safe to reuse the pointer to an object instead of duplicating the object. Since GP objects are numerous and possibly very large, removing spurious copying is crucial.

3.2.4. ***Bytecode compilation.*** This stage generates a `t_CLOSURE` PARI object from a syntactic tree.

The compiler performs a depth-first traversal of the tree, generating codes for each nodes in turn, in a similar way to the conversion of a syntactic tree to reverse Polish notation.

As a simple example, consider the expression $a*x+b*y+c$. The parser returns the following syntactic tree:



Depth-first traversal of the tree leads to its simplified Reverse Polish Notation (RPN): $a\ x\ *\ b\ y\ *\ +\ c\ +$. The compiler actually generates the following series of pairs of opcodes and operands:

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

The compiler uses the function signatures to compile built-in function calls, by generating opcodes to convert arguments to the desired type, and by generating `t_CLOSURE` PARI objects when the function requires deferred execution, i.e. it is a control statement.

User functions can unfortunately be redefined at run-time, after compilation, and have to use a different calling convention. They take an integer n followed by n data. The evaluator reports a run-time error if n is larger than the function arity; otherwise it completes the number of data to n with zeros, and calls the function with these arguments. The latter is responsible for later removing the data from the stack.

The compiler performs other tasks as well:

- it computes the value of immediate data from their text location and stores them in the *program data* component of the closure.
- it generates the full text of the closure and stores it in the *source code* component of the closure.
- it records, for each opcode, the text location of the corresponding node and stores them in the *debugging data* component of the closure.
- it recursively generates closures for objects that require deferred execution: function definition, arguments of control statement function, etc. They are also stored in *program data* component of the closure.
- it generates copy opcodes when necessary, as instructed by the copy optimizer.
- it generates pairs of instructions that performs garbage collection at run-time.

The compiler is defined in the file `src/language/compile.c`.

3.3. Bytecode evaluation. The bytecode evaluator takes a `t_CLOSURE` PARI object and evaluates each opcode in turn with respect to a global evaluator stack, then returns, leaving results at the top of the stack. The evaluator does not perform any branching or jumping by itself. However, the evaluation of an opcode can lead to the evaluation of another closure, causing the evaluator to call itself recursively.

A function called by the evaluator can request that the evaluator restore the stack and return immediately. This mechanism allows to implement control-flow change functions (`break, next` and `return`).

The evaluator maintains a call-trace stack that is used by the debugger. Each time it is called, the evaluator creates a call-trace at the top of a stack and records pointers to the `t_CLOSURE` PARI object it is evaluating, and to the variable holding the index of the current opcode. When the evaluator returns, the call-trace is removed from the stack.

The evaluator handles garbage collection, global variables, dynamically-scoped and lexically-scoped local variables.

The evaluator is defined in the file `src/language/eval.c`.

4. New features

4.1. Lexically-scoped local variables. The old GP interpretor only allowed to define dynamically-scoped local variables, using the `local` keyword, and only at the start of functions. Dynamically-scoped local variables are inadequate in a large number of situation, because they propagate through function calls.

We added support for *lexically-scoped* local variables through the `my` keyword. Furthermore, all implicitly declared variables (functions arguments and loop indices) are now statically scoped as well. While this breaks backward compatibility, this was what users generally intended, and prevents common mistakes. For instance, in the following pair of functions

```
ellsquare(e/*mistake*/, P)= elladd(E, P,P);
ellquad(E, P) = ellsquare(E, ellsquare(E, P));
```

the name of the first argument of `ellsquare` was mispelled, yet if `E` is dynamically-scoped to `ellquad`, then the function `ellquad` will work correctly. Later on, users will wonder why the function `ellsquare` does not work outside of `ellquad`.

A further rationale for this change is that the GP2C compiler does not support dynamically-scoped variable and treats all local variables as lexically scoped.

Finally, lexically-scoped variables lead to better run-time performances and less copying: if a lexically-scoped local variable is passed as argument of a user function, this function does not see this variable, so cannot change it; thus an implicit call-by-parameter is used instead of a call-by-value, which would require copying the variable. Such optimizations cannot be performed for dynamically-scoped local variables, because user functions might be redefined at run-time and thus the compiler cannot know whether a function will modify the variable or not.

The following is an example showing the advantage of lexically-scoped local variable:

```
? last(v) = v[#v];
? V = vector(10^3,i,i); N = 10^5;
? my(v=V); for(i=1,N, last(v))
? ##
  ***   last result computed in 28 ms.
? local(v=V); for(i=1,N, last(v))
  ***   Warning: compiler generates copy for 'v'.
? ##
  ***   last result computed in 2,940 ms.
```

As the example shows it is now possible to define new local variables, either lexically or dynamically scoped, anywhere in the code; their scope lasts until the end of the current `t_CLOSURE` object.

4.2. Closures and anonymous functions. The introduction of the new `t_CLOSURE` object allows PARI to handle functions as first-level objects, and to support anonymous functions and closures.

For this purpose we extended the GP language to allow the definition of anonymous functions using the following syntax:

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

The traditional form

$$f(x_1, \ldots, x_n) = expr$$

is now just an alias for

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

Evaluating a bare built-in function name (i.e. not followed by parentheses and arguments, as in `f = sin`) returns a `t_CLOSURE` PARI object that evaluates to the function, for compatibility with user-defined functions.

This new feature provide a large number of benefits, in particular functions can take functions as parameters, return functions, be stored in local variables, in vectors and matrices. A small number of such higher-level functions have been added (`apply`, `select`, `vecsort`, numerical derivation). For instance, the following one-liner user function returns representatives for the elements of $(\mathbb{Z}/N\mathbb{Z})^*$:

```
invertibles(N) = select(x -> gcd(x,N)==1, vector(N,i,i))
```

Furthermore, functions now act as closures with respect to lexically-scoped variables. The closure context is stored in the last component of the closure, and is strictly read-only. More precisely, change to variables in the context are only visible inside the closure. Lifting this restriction would require more fancy garbage collecting than PARI provides.

The following example defines a function `pol2func` that returns the polynomial function associated to a formal polynomial as a closure:

```
? pol2func(P, v='x) = x -> subst(P,v,x);
? P = x^4+1; f = pol2func(P);
? f(2)
%3 = 17
```

4.3. The GP debugger. When an error or a user interrupt occurs, the debugging facility uses the call-trace stack generated by the evaluator and the *debugging data* section of `t_CLOSURE` objects to display a source-level call trace. Furthermore, the compiler is preseeded with the state of variables

at the interrupt point, so expressions can be evaluated in that context. In case of an user interrupt, it is possible to resume the computation.

The following is an example of session where we diagnose an error:

```
? f(p) = sum(i=1, p, 1/Mod(i^2+1,p)); \\requires p%4==3
? f(17)
  ***   at top-level: f(17)
  ***                 ^-----
  ***   in function f: sum(i=1,p-1,1/Mod(i^2+1,p))
  ***                                ^--------------
  *** _/_: impossible inverse modulo: Mod(0, 17).
  ***   Break loop: type <Return> three times, or Control-d,
  ***   to go back to GP)
break> i
4
break> Mod(i^2+1, p)
Mod(0, 17)
```

Acknowledgement

The author would like to thanks Karim Belabas for his advices while preparing this paper.

References

[1] B. Allombert, *GP2C, the GP to C translator* , `http://pari.math.u-bordeaux.fr/pub/pari/GP2C/`, version `0.0.5pl6`, 2008.

[2] R. Corbett, R. Stallman, *BISON, the GNU parser generator*, `http://www.gnu.org/software/bison/`, version `2.3`, 2006.

[3] T. Lindholm, F. Yellin, *The Java Virtual Machine Specification , Second Edition*, `http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html`, 1999.

[4] the PARI Group, *PARI/GP*, `http://pari.math.u-bordeaux.fr/` version `2.3.4`, 2008.

Bill Allombert
Université Montpellier 2
CNRS I3M/LIRMM
Place Eugène Bataillon
F-34095 Montpellier cedex, France
E-mail: `Bill.Allombert@univ-montp2.fr`