PLL: Programming in Logic in Lisp

PLL: Programming in Logic in Lisp

PLL is a simple Prolog in Scheme, developed as a teaching aid. It is supposed to make it easy to understand the execution model of Prolog, including variables, cuts, and meta-predicates. There is also a simple FFI to Scheme.

When I say "simple", I really mean it. This implementation is NOT efficient at all, and it misses several features. The manual is also very basic, although it covers most of what really matters.

The code is documented, but not "densely documented" – this is mostly because the interpreter was developed as an auxiliary tool for a course, and the implementation details are explained in the course notes which were written in Portuguese. However, the code may be used as a companion to chapter 4 of Sterling and Shapiro's "The Art of Prolog", as it closely follows the execution model explained there.

LOADING PLL

There are two ways to use PLL. In implementations that support R7RS, you can import it. This may be useful if you want to rename identifiers. In implementations that do not support R7RS, there is a file that will load the interpreter.

```
(load "pll-standalone.scm")    ;; this works on all supported schemes.

(import (pll))                 ;; only on the systems marked as having
                               ;; support for R7RS modules
```

VARIABLES

Traditionally, Prolog systems treat symbols beginning with uppercase letters as variables: X, Y, Variable are variables, while x, y, constant are constants.

Lisp-based Prologs usually do this differently: the variables are the symbols that begin with a question mark: `?x`, `?y`, `?variable`, etc. We do the same in this implementation.

CALLING THE INTERPRETER

Our Prolog use the following syntax:

```
(pure-prolog PROGRAM GOAL)
```

The program is a list of assertions. The program

```
 p(x).
 p(Z) :- q, r(Z).
```

is written as a list of assertions. Each assertion is a list where the head represents the left side (the consequent), and the tail is a list of the goals to be met in order to prove the consequent. For example,

```
'(( (p x) ))
  ( (p ?z) q (r ?z)))
```

The GOAL is a list of goals:

```
p(1), q(X).
```

is written as

```
'((p 1) (q ?x))
```

EXAMPLE: Suppose we want to enter the following program:

```
f(0).
f(Z) :- g(Z).
h(3).
h(4).
p(Z,Y,S) :- f(Z),g(Y),h(S)
g(10).
```

And ask the question

```
p(10,D,A), q(A).
```

We can do this:

```
(define facts '(( (f 0) )
                ( (f ?z) (g ?z) )
                ( (h 3) )
                ( (h 4) )
                ( (q 4) )
                ( (p ?z ?y ?s) (f ?z) (g ?y) (h ?s) )
                ( (g 10) )))
```

```
(define goals '((p 10 ?d ?a) (q ?a)))
```

And call

```
(pure-prolog facts goals)
```

Or, directly enter the facts and goal (as we do in the examples that are included in `prolog-examples.scm`):

```
(pure-prolog '(( (f 0) )
               ( (f ?z) (g ?z))
               ( (h 3) )
               ( (h 4) )
               ( (q 4) )
               ( (p ?z ?y ?s) (f ?z) (g ?y) (h ?s))
               ( (g 10) ))
             '((p 10 ?d ?a) (q ?a)))
```

The result will be a list of substitutions that satisfy the goal:

```
((?a . 4) (?d . 10))
```

GETTING MULTIPLE ANSWERS

This can be done with the `amb+` operator.

```
(pure-prolog '(((f 1))
               ((f 2)))
             '((f ?x)))

((?x 1))
```

If we call `(amb+)`, then we get another solution:

```
((?x 2))
```

DIFFERENT INTERPRETERS

The different interpreters included are:

- `pure prolog` (prolog only)
- `prolog+built-ins` (with built-in predicates with side-effect)
- `prolog+scheme` (with an FFI to Scheme, but NO built-ins)
- `prolog+local` (with "IS" and local vars; on top of prolog+scheme)
- `prolog+meta` (with assert and retract; on top of prolog+local)
- `prolog+cut` (with cut (!); on top of prolog+meta)

So the features are added one on top of the other, except for the built-ins feature, which is not included in the later interpreters.

In the source code, there is one initial implementation (pure-prolog) with short comments explaining how it works. Each interpreter after that one has comments only on the modified parts.

Pure Prolog

This is the most basic of all. You can only statet facts as we described above, nothing else. Call it as

```
(pure-prolog facts goals)
```

For example, we define a graph by declaring its edges, then define a `reach/2` predicate.

```
edge(a,b).
edge(a,c).
edge(c,b).
edge(c,d).
edge(d,e).
reach(A,B) :- edge(A,B).
reach(A,B) :- edge(A,X), reach(X,B).
```

We could then as "`reach(b,e)`" and find that *b* doesn't reach *e*.

In PLL:

```
(define graph '( ((edge a b))
                 ((edge a c))
                 ((edge c b))
                 ((edge c d))
                 ((edge d e))
                 ((reach ?a ?b) (edge ?a ?b))
                 ((reach ?a ?b) (edge ?a ?x)
                                (reach ?x ?b))))

(pure-prolog graph '( (reach b e) ))
#f
```

Prolog with built-in "procedures"

This adds some predicates with side-effects.

Define a list of Scheme functions that you want to be accessible from Prolog:

```
(define write-it
  (lambda (args sub)
    (cond ((not (null? args))
           (if (matching-symbol? (car args))
               (display (assoc (car args) sub))
               (display (car args)))
           (write-it (cdr args) sub))
          (else #t))))
```

Here “`sub`” is the current substitution, where Prolog will find the values of variables.

```
(define built-ins `((write . ,write-it)))
```

So `(write a b c ...)` will be translated to

```
(write-it (a b c ...) sub)
```

Then, whenever one of these predicates show up in the goal, Prolog will execute them:

```
father(john,johnny).
father(old-john,john).
grandpa(A,B) :- father(A,X), father(X,B).
```

Our goal is

```
grandpa(old-john,Who), write("grandson is: "), write(Who).
```

In our Prolog, this is expressed as follows:

```
(prolog+built-ins '( ((father john johnny))
                     ((father old-john john))
                     ((grandpa ?a ?b) (father ?a ?x) (father ?x ?b)) )
                  '( (grandpa old-john ?who)  (write "Grandson is: " ?who) ))

Grandson is: johnny
(((?who . johnny)) ())
```

The first line was the effect of having a `(write ...)` in the goal. The second is the answer.

Prolog with any scheme function

This interpreter has no sandbox – it will recognize and execute any Scheme function when it sees one.

Suppose we want to run this Prolog program, and use a Scheme function “`print`” in the goal:

```
a(5).
b(3).
b(5).
```

Goal:

```
a(X), b(Y), X=Y, print(X, " " Y).
```

In our Prolog, we have:

```
(prolog+scheme '( ((a 5))
                  ((b 3))
                  ((b 5)))
               '( (a ?x) (b ?y) ((= ?x ?y)) ((print ?x " " ?y))))

5 5
((?y . 5) (?x . 5))
```

The first line is the effect of the print. The second line is Prolog's answer.

With local variables

```
(prolog+local '( ((f ?x ?y)
                  (is ?y (* 2 ?x))
                  ((print 'OK: ?y))))
              '( (f 3 ?a) ))

OK:6
((?a . 6))
```

With assert and retract

This adds the meta-predicates `assert` and `retract`.

```
(prolog+meta '(( (f ?x) (asserta (h 1)))
               ( (g ?x) (h ?x)))
             '((f ?w) (g ?z)))
```

The result will be `((?z . 1))`

`retract` has the opposite effect. The Prolog program we show now is

```
f(2).
f(3).
g(1) :- retract(f(2)).
```

And the goal is

```
g(1), f(X).
```

In PPL's Schemish-Prolog, this is

```
(define prolog-retract '( ((f 2))
                          ((f 3))
                          ((g 1) (retract ((f 2))))
                          ((g 1) (f 3))))
```

```
(prolog+meta prolog-retract '((g 1) (f ?x)))
```

`(g 1)` causes retract to be evaluated, so the goal succeeds with *X=3*, and not *X=2*:

```
((?x 3))
```

With cut

This adds the cut operator. For example, in Prolog we could write

```
a(X) :- b(X), !, c(X).
b(1).
b(2).
c(2).
```

The goal `a(X)` fails, because after the interpreter chooses `b(1)`, it cannot backtrack, and `c(1)` is not true.

In our Prolog, we write:

```
(prolog+cut '(( (a ?x) (b ?x) ! (c ?x) )
              ( (b 1) )
              ( (b 2) )
              ( (c 2) ))
            '((a ?x)))

#f
```

If we remove the cut, then this goal will succeed with *X=2*.

The cut can be used also in the goal.

BUGS AND MISSING FEATURES

- There is no way to save and load the database
- There is no simple way to get all possible answers (substitutions) for a query in a list.
- This manual is too short.

A BIBLIOGRAPHY

The following is a list of some books related to Prolog programming and implementations of Prolog.

1. ***William Clocksin, Chris Mellish***. *"Programming in Prolog"*. Springer, 2003. [a very basic text]

2. ***Michael Covington, Donald Nute, André Vellino***. *"Prolog Programming in Depth"*. Scott, Foresman and Company, 1988. [quick introduction to Prolog, followed by some advanced programming techniques and application]

3. ***Leon Sterling, Ehud Shapiro***. *"The Art of Prolog"*. MIT Press, 1994. [solid introduction to Prolog, including the execution model – strongly recommended for those who want to understand the internals of the interpreter]

4. ***Richard O'Keefe***, *"The Craft of Prolog"*. MIT Press, 1990. [advanced programming techniques]

5. ***Harold Abelson, Gerald Jay Sussman***. *"Structure and Interpretation of Computer Programs"*. Addison-Wesley, 1996. [explains and implements in Scheme the AMB operator and a small Prolog interpreter]

6. ***Peter Norvig***. *"Paradigms of Artificial Intelligence Programming"*. Morgan Kaufmann, 1992. [part of the book explains unification and contains another Prolog implementation, in Common Lisp]

7. ***Jacques Chazarain***, *"Programmer Avec Scheme"*. International Thomson Publishing France, 1996 (in French). [a very good book on Scheme. chapters 15-19 focus on Prolog]

8. ***J. A. Campbell***. *"Implementations of PROLOG"*. Ellis Horwood, 1984. [a study of implementation techniques. quite advanced.]