The world's shortest Prolog interpreter?

M. Nilsson, Uppsala University

The aim of this article is to describe a small structure-sharing Prolog interpreter which is implemented in LISP. It is easy to type in, fairly readable, and efficient enough to execute programs which are small and simple, yet not totally trivial.

IMPLEMENTATION

Admittedly, the title of this paper is a bit provocative. In fact, the interpreter can be made even smaller by replacing the selector and constructor functions (which are there only for readability) with their respective CARs, CDRs, and CONSes, etc.

The selectors and contructors are:

```
(LVL X)              = (CAR X)
(XPR X)              = (CADR X)
(MOLEC X Y)          = (LIST X Y)
(BIND X Y E)         = (CONS (CONS X Y) E)
(BOND X E)           = (CDR (OR (ASSOC X E)  '(NIL)))
(BUT-FIRST-GOAL X)   = (CONS (CDAR X) (CDR X))
```

To improve on the time/memory efficiency, the best suggestion is to optimise the variable binding scheme. In the version presented, linear list search is used for finding bindings of variables. The interpreter can be sped up by a factor of about 100 if instead some indexed data structure like an array is used (Nilsson, 1983). Unfortunately, in LISP such structures are often very implementation-dependent, so they have been avoided here.

The interpreter is written in MACLISP dialect (Moon, 1974). Only the most basic functions are used, with the exceptions of LET and LOOP. LET is a convenient way of writing LAMBDAs:

```
(LET ((⟨var1⟩ ⟨expr1⟩) . . . (⟨varN⟩ ⟨exprN⟩)) ⟨body⟩)
```

is exactly the same as

```
((LAMBDA (⟨var1⟩ . . . ⟨varN⟩) ⟨body⟩) ⟨expr1⟩ . . . ⟨exprN⟩)
```

```
; ---------- Prolog interpreter

(defun prove fexpr (goals)
   (seek (list goals) '(0) '((bottom-of-env)) 1))

(defun seek (to-prove nlist env n)
   (cond ((null to-prove) (apply *toplevel* (list env)))
         ((null (car to-prove)) (seek (cdr to-prove) (cdr nlist) env n))
         ((and (atom (car to-prove)) (null (var (car to-prove))))
          (apply (car to-prove) (list (cdr to-prove) nlist env n)))
         ((loop with goalmolec = (molec (car nlist) (caar to-prove)) and
           rest = (but-first-goal to-prove) and env2 = nil and tmp = nil
           for (head . tail) in *database* do
           (and (setq env2 (unify goalmolec (molec n head) env))
                (setq tmp (seek (cons tail rest) (cons n nlist) env2 (add1 n)))
                (return (and (null (equal tmp n)) tmp)))))))

(defun unify (x y env)
   (cond ((equal (setq x (lookup x env)) (setq y (lookup y env))) env)
         ((var (xpr x)) (bind x y env))
         ((var (xpr y)) (bind y x env))
         ((or (atom (xpr x)) (atom (xpr y)))
          (and (equal (xpr x) (xpr y)) env))
         ((setq env (unify (molec (lvl x) (car (xpr x)))
                           (molec (lvl y) (car (xpr y))) env))
          (unify (molec (lvl x) (cdr (xpr x)))
                 (molec (lvl y) (cdr (xpr y))) env))))

(defun lookup (p env)
   (for (and p (var (xpr p)) (lookup (bond p env) env) env)) p))

(defun var (x) (member x '(?a ?b ?c ?x ?y ?z ?u ?v ?w)))
```

The LOOP construct can easily be rewritten as a PROG, or a MACLISP DO, or an INTERLISP CLISP expression (Teitelman, 1974), or whatever iteration facility is available. The form

```
(LOOP WITH ⟨var1⟩ = ⟨expr1⟩ AND ⟨var2⟩ = ⟨expr2⟩
FOR (⟨var3⟩ . ⟨var4⟩) IN ⟨list⟩ DO ⟨expr3⟩)
```

can be paraphrased as: Set the local variables ⟨var1⟩ and ⟨var2⟩ to ⟨expr1⟩ and ⟨expr2⟩, respectively. Then for all elements in ⟨list⟩, set ⟨var3⟩ to the head of the element, and ⟨var4⟩ to the tail, and evaluate ⟨expr3⟩. Repeat evaluating ⟨expr3⟩ until ⟨list⟩ is exhausted, and return NIL when finished. However, if RETURN is called with an argument during evaluation, return that value immediately.

Note that the versions of AND and OR used here return the value of their last evaluated argument. Note also that we use the MACLISP ASSOC which searches an A-list for an element whose CAR is EQUAL to a given expression. If found, the element is returned, otherwise NIL is returned. The equivalent INTERLISP function is SASSOC, not ASSOC, which uses EQ instead of EQUAL. The function NTH, called by (NTH ⟨n⟩ ⟨list⟩), returns the ⟨n+1⟩th element of ⟨list⟩.

If the interpreter is compiled, it might be necessary to declare the global variables *TOPLEVEL*, *DATABASE*, *BAGOF-TERM*, and *BAGOF-LIST*.

FUNCTION

The interpreter is called by

(PROVE ⟨goal1⟩ . . . ⟨goalN⟩)

where ⟨goal⟩ is a list (⟨pred-symbol⟩ ⟨arg1⟩ . . . ⟨argN⟩), and where ⟨pred-symbol⟩ is a predicate name. The ⟨arg⟩s are S-expressions which may contain Prolog variables, which are LISP symbols for which the LISP predicate VAR returns a non-NIL value. The list of clauses is pointed to by the global LISP variable *DATABASE*. A clause is a list of goals (⟨goal1⟩ . . . ⟨goalN⟩).

The function PROVE calls the function SEEK. SEEK looks up assertions in the database, and uses UNIFY to match them with the goals to be proved. SEEK and UNIFY are together more or less the complete interpreter.

```
1_ (setq *databases*
       '(((true))
         ((= ?x ?x))
         ((append nil ?x ?x))
         ((append (?x . ?y) ?z (?x . ?u)) (append ?y ?z ?u))
         ((father medium-ben small-ben))
         ((father big-ben medium-ben))
         ((grandfather ?x ?y) (father ?x ?z) (father ?z ?y))
         ((call ?x) ?x))
       *toplevel*
       '(lambda (env) env))
(LAMBDA (ENV) ENV)
2_ (prove (grandfather ?x ?y))
(((1 ?Y) 3 SMALL-BEN)              ((1 ?Z) 2 MEDIUM-BEN)
  ((1 ?X) 2 BIG-BEN)               ((0 ?Y) 1 ?Y)
  ((0 ?X) 1 ?X)                    (BOTTOM-OF-ENV))
3_
```

Some examples of predicate definitions. User type-in is in lower case. The grandfather is ?X on level 0, which is bound to BIG-BEN in the returned environment.

The argument TO-PROVE for the function SEEK is a list of goals left to prove.

The argument N of SEEK is the first unused level in the proof (execution) tree. These numbers distinguish variables with the same name but from different environments.

The argument NLIST is a list of levels, each corresponding to an element of TO-PROVE.

The argument ENV is an A-list holding Prolog variable bindings. The format of a binding is

```
(((variable-level) (variable-name)) . ((expr-level) (expr)))
```

which means that the variable ⟨variable-name⟩ is bound to ⟨expr⟩.

The first COND-clause in SEEK checks if TO-PROVE is empty. If so, everything has been proved and the toplevel function which is bound to *TOPLEVEL* is called with the environment as an argument. The simplest kind of *TOPLEVEL* just returns the environment, which says what variable instantiations satisfy the goals. A better *TOPLEVEL* allows the user to decide whether to stop or to backtrack. If he answers a 'MORE?'-question with NIL, the value T is returned, and it will eventually drop down through the succesive calls of SEEK to PROVE.

The value returned by SEEK is important. A returned value of NIL means failure, and the system tries to backtrack. A returned non-NIL value falls through SEEK, unless the value is a number which equals the current level of the proof tree. This feature is used by the evaluable predicate CUT.

The second COND-clause checks that if everything has been proved on the current level of the proof tree, SEEK should go on trying to prove the goals on the next level.

The third clause implements calls to evaluable predicates.

UNIFY is called with three arguments:

```
(UNIFY ((level-x) (x)) ((level-y) (y)) (env))
```

If ⟨x⟩ and ⟨y⟩ don't match, UNIFY returns NIL. Otherwise, it returns a new environment, which is the old one with possible new bindings pushed on top. For instance,

```
(UNIFY '(17 (A ?X)) '(4711 (?Y B)) '(((3 ?Z) . (2 C))
                                      (BOTTOM-OF-ENV))) =
= (((4711 ?Y) . (17 A))
   ((17 ?X) . (4711 B))
   ((3 ?Z) . (2 C))
   (BOTTOM-OF-ENV))
```

The unification algorithm is a version of the algorithm described in detail in (Robinson and Sibert, 1982).

```
; ---------- Evaluable Predicates

(defun inst (p env)
   (cond ((atom (xpr (setq p (lookup p env)))) (xpr p))
         ((cons (inst (molec (lvl p) (car (xpr p))) env)
                (inst (molec (lvl p) (cdr (xpr p))) env)))))

(defun escape-to-lisp (to-prove nlist env n)
   (and (setq env (unify (molec (car nlist) '?y)
                         (molec n (eval (inst (molec (car nlist) '?x) env)))
                        env))
        (seek to-prove (cdr nlist) env (add1 n))))

(defun cut (to-prove nlist env n)
   (or (seek to-prove (cdr nlist) env n) (cadr nlist)))

(defun bagof-topfun (env)
   (setq *bagof-list* (cons (inst (molec *bagof-env* '?x) env) *bagof-list*))
   nil)

(defun bagof (to-prove nlist env n)
   (let ((*bagof-list* nil) (*bagof-env* (car nlist)))
      (let ((*toplevel* 'bagof-topfun)) (seek '(((call ?y))) nlist env n))
      (and (setq env (unify (molec (car nlist) '?z) (molec n *bagof-list*) env))
           (seek to-prove (cdr nlist) env (add1 n)))))
```

A FEW EVALUABLE PREDICATES

The evaluable predicates described here are non-logical extensions to Prolog, but they are often available in Prolog systems.

For each evaluable predicate ⟨pred⟩ there is a special clause in the database

(((pred⟩ . ⟨args⟩) . ⟨pred-lisp-function⟩)

This kind of clause is recognised by the ATOM-test in SEEK, and the LISP function ⟨pred-lisp-function⟩ is called.

If the ESCAPE-TO-LISP predicate is called, (ESCAPE-TO-LISP ?X ?Y), ?X will be evaluated by the LISP interpreter and the result will be unified with ?Y.

(CUT) is the usual CUT-primitive. When it is entered, the level to which it should cut is saved, and if backtracking through CUT is attempted, that number is returned.

(BAGOF ?X ?Y ?Z) unifies ?Z with the list of all instances of ?X such that ?Y can be proved. BAGOF is like DEC-10 Prolog's bagof, but quantifies existentially over all free variables in ?Y, and does not fail if an empty list is generated.

```
3_ (setq *database*
       (append *database*
               '(((escape-to-lisp ?x ?y) . escape-to-lisp)
                 ((cut) . cut)
                 ((bagof ?x ?y ?z) . bagof)
                 ((writeln ?x) (escape-to-lisp (progn (prin1 '?x) (terpri)) ?y))
                 ((cannot-prove ?x) (call ?x) (cut) (fail))
                 ((cannot-prove ?x) (true))
                 ((list-of-sons ?x) (bagof ?y (father ?z ?y) ?x))))
       *toplevel*
       '(lambda (env) (princ "MORE? ") (null (read))))
(LAMBDA (ENV) (PRINC "MORE? ") (NULL (READ))))
4_ (prove (append ?x (a b c) ?z) (writeln (?x on (a b c) is ?z)))
(NIL ON (A B C) IS (A B C))
MORE? t
((?X) ON (A B C) IS (?X A B C))
MORE? t
((?X ?X) ON (A B C) IS (?X ?X A B C))
MORE? nil
T
5_ (prove (list-of-sons ?x) (writeln (some sons are . ?x)) (fail))
(SOME SONS ARE MEDIUM-BEN SMALL-BEN)
NIL
6_
```

Some evaluable-predicate examples.

ACKNOWLEDGEMENTS

The inspiration to write this Prolog came from Alan Robinson. I also want to thank John Campbell who encouraged me to write this article.

REFERENCES

Moon, D. A., (1974), MACLISP reference manual, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts.

Nilsson, M., (1983), FOOLOG – A Small and Efficient Prolog Interpreter, Technical Report No. 20, UPMAIL, Computing Science Department, Uppsala, Sweden.

Pereira, L. M., *et al.*, (1978), *User's Guide to the DECsystem-10 Prolog*, Divisão de Informática, Laboratorio Nacional de Engenharia Civil, Lisbon.

Robinson, J. A. and Sibert, E. E., (1982), LOGLISP: Motivation, Design and Implementation, in Clark, K. L. and Tärnlund, S.-Å. (ed.), *Logic Programming*, pp. 299–313, Academic Press, London.

Teitelman, W., (1974), *INTERLISP reference manual*, Xerox Palo Alto Research Center, Palo Alto, California.