Upmail Technical Report no. 5
Revised December 8 1983

On Implementing Prolog In Functional Programming

Mats Carlsson

December 8 1983

Keywords: Prolog, Logic Programming, Unification, Lisp, Functional Programming, Continuations

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, Uppsala University
P.O. Box 2059
S-750 02 Uppsala Sweden
Domestic 018 155400 ext. 1860
International +46 18 111925
Telex 76024 univups s

Note: This paper is a slight revision of a paper presented at the 1984 International Symposium on Logic Programming, Atlantic City NJ USA.

The research reported herein was sponsored by the National Swedish Board for Technical Development (STU).

Upmail Technical Report no. 5
Revised December 8 1983

On Implementing Prolog In Functional Programming

Mats Carlsson

December 8 1983

Keywords: Prolog, Logic Programming, Unification, Lisp, Functional Programming, Continuations

UPMAIL
Uppsala Programming Methodology and Artificial Intelligence Laboratory
Department of Computing Science, Uppsala University
P.O. Box 2059
S-750 02 Uppsala Sweden
Domestic 018 155400 ext. 1860
International +46 18 111925
Telex 76024 univups s

Note: This paper is a slight revision of a paper presented at the 1984 International Symposium on Logic Programming, Atlantic City NJ USA.

The research reported herein was sponsored by the National Swedish Board for Technical Development (STU).

Table of Contents

1. Abstract

This report surveys techniques for implementing the programming language Prolog. It focuses on explaining the procedural semantics of the language in terms of functional programming constructs. The techniques *success continuations* and *proof streams* are introduced, and it is shown how Horn clause interpreters can be built upon them. Continuations are well known from denotational semantics theory, in this paper it is shown that they are viable constructs in actual programs.

Other issues include implementation of logical variables, structure sharing vs. structure copying, builtin predicates, and *cut*.

2. Achieving Backtracking

Several authors [4], [11], [7] have proposed abstract machinery to implement the backtracking behavior of Prolog. Typically the abstract machine includes a set of registers and various stacks carrying the state of the machine. Backtracking amounts to restoring parts of this state as it was at some previous time.

In this section, we will give a more abstract implementation of backtracking using concepts from functional programming. Continuations [9] are the workhorse for achieving the desired control structure. Recursion is used instead of manipulation of explicit stacks, and parameter passing is used instead of assignments to machine registers.

There are at least two fundamentally different techniques for achieving backtracking and we will call them *proof streams* and *success continuations*. In this paper, the word "continuation" denotes any function that is either passed as an argument or returned as a value.

2.1 Success Continuation Scheme

The idea here is that the theorem prover receives an extra argument, the *success continuation*. If the theorem prover succeeds in proving its goal, it calls this continuation. If it fails, it simply returns. Backtracking is achieved by the possibility that the theorem prover finds several proofs of its goal, in which case it calls the continuation for each proof found.

2.2 Proof Stream Scheme

Here, the theorem prover returns a *proof stream*, which conceptually is a lazily evaluated list of environments, corresponding to the possible proofs of the given query. In a concrete implementation, a proof stream could be either

()

for failure i.e. no more proofs, or a pair

(*environment* . *continuation*)

for success i.e. a proof was found. *Continuation* is a function that returns a new proof stream. *Environment* could be the set of variable substitutions involved in a particular proof. Backtracking is achieved by calling the continuation of successive proof streams until failure eventually

results. To our knowledge this idea was first conceived by Abelson [1]. It has been used later by Kornfeld [6].

2.3 Interpreters

We present here running implementations of the two techniques written in pure Lisp.

A continuation is implemented as a Lisp function consed to an incomplete argument list. The continuation is called with additional arguments that complete the argument list, which is then passed to the Lisp function.

Prolog datatypes used are atoms, pairs, and variables. Variables are implemented as symbols that begin with a "?".

Variable bindings are kept in an association list consisting of pairs

((*variable* . *index*1) . (*term* . *index*2))

where the indexes are used to distinguish between synonymous variables belonging to different uses of the same assertion. The index is increased once per resolution step. This is essentially the *structure sharing* technique of Boyer and Moore [2]. It is discussed in more detail in chapter 3.

The database is only indexed on predicate symbols. The *:assertions* property of a symbol contains a list of assertions.

Success Continuation Interpreter

This is a MacLisp implementation of the above ideas. It defines a Horn clause interpreter and includes predicates about appended and reversed lists.

```
(defun prove (env j goals i cont)
  ;;Proves "goals" seen through index "i" and calls
  ;; "cont". "J" is next available index.
  (cond ((null env) nil)
        ;;An impossible environment is empty.
        ((null goals) (invoke cont env j))
        (t (resolve (car goals) i (assertions (car goals)) j '(prove ,(cdr goals) ,i ,cont) env))))


(defun resolve (goal i assertions j cont env)
  ;;For each proof of "goal" with respect to "assertions",
  ;; "cont" is called.
  (cond ((null assertions) nil)
        ((prove (unify goal i (caar assertions) j env) (1+ j) (cdar assertions) j cont))
        (t (resolve goal i (cdr assertions) j cont env))))


(defun unify (x i y j e)
  ;;Returns a non-empty environment upon success.
  (unify1 (ult (cons x i) e) (ult (cons y j) e) e))
```

```
(defun unify1 (xi yj e)
  (cond ((null e) e)
        ((and (eq (car xi) (car yj)) (eq (cdr xi) (cdr yj)))
         e)
        ((and (consp (car xi)) (consp (car yj)))
         (unify (cdar xi) (cdr xi) (cdar yj) (cdr yj)
                (unify (caar xi) (cdr xi) (caar yj) (cdr yj)
                       e)))
        ((variable-symbol-p (car xi)) (cons (cons xi yj) e))
        ((variable-symbol-p (car yj)) (cons (cons yj xi) e))
        ((equal (car xi) (car yj)) e)))


(defun ult (x e)
  ;;Follows chain of linked variables.
  (let ((pair (assoc x e)))
    (cond ((null pair) x)
          ((eq x (cdr pair)) x)
          (t (ult (cdr pair) e)))))
```

A predicate to append two lists could for example be represented as

```
(defprop append
  (((append (?X . ?XS) ?Y (?X . ?ZS)) (append ?XS ?Y ?ZS))
   ((append () ?X ?X)))
  :assertions)
```

Auxiliary functions include invoke for invoking continuations, variable-symbol-p for recognizing variables, assertions for retrieving appropriate assertions from the database, and a toplevel function.

Proof Stream Interpreter

To arrive at an interpreter based on proof streams, the functions prove and resolve above should take proof streams instead of continuations as arguments, and should return proof streams as values.

The following code implements streams as ordinary lists. Under lazy evaluation this would result in the desired backtracking. The environment in the above discussion of proof streams is actually implemented as a pair of (*i*) the variable binding alist and (*ii*) next available index.

```
(defun prove (stream goals i)
  ;;Each goal "filters" the proof stream.
  (cond ((null stream) ())
        ((null goals) stream)
        (t (prove (resolve stream (car goals) i (assertions (car goals)))
                  (cdr goals)
                  i))))
```

```
(defun unify1 (xi yj e)
  (cond ((null e) e)
        ((and (eq (car xi) (car yj)) (eq (cdr xi) (cdr yj)))
         e)
        ((and (consp (car xi)) (consp (car yj)))
         (unify (cdar xi) (cdr xi) (cdar yj) (cdr yj)
                (unify (caar xi) (cdr xi) (caar yj) (cdr yj)
                       e)))
        ((variable-symbol-p (car xi)) (cons (cons xi yj) e))
        ((variable-symbol-p (car yj)) (cons (cons yj xi) e))
        ((equal (car xi) (car yj)) e)))


(defun ult (x e)
  ;;Follows chain of linked variables.
  (let ((pair (assoc x e)))
    (cond ((null pair) x)
          ((eq x (cdr pair)) x)
          (t (ult (cdr pair) e)))))
```

A predicate to append two lists could for example be represented as

```
(defprop append
  (((append (?X . ?XS) ?Y (?X . ?ZS)) (append ?XS ?Y ?ZS))
   ((append () ?X ?X)))
  :assertions)
```

Auxiliary functions include invoke for invoking continuations, variable-symbol-p for recognizing variables, assertions for retrieving appropriate assertions from the database, and a toplevel function.

Proof Stream Interpreter

To arrive at an interpreter based on proof streams, the functions prove and resolve above should take proof streams instead of continuations as arguments, and should return proof streams as values.

The following code implements streams as ordinary lists. Under lazy evaluation this would result in the desired backtracking. The environment in the above discussion of proof streams is actually implemented as a pair of (*i*) the variable binding alist and (*ii*) next available index.

```
(defun prove (stream goals i)
  ;;Each goal "filters" the proof stream.
  (cond ((null stream) ())
        ((null goals) stream)
        (t (prove (resolve stream (car goals) i (assertions (car goals)))
                  (cdr goals)
                  i))))
```

```
(defun resolve (stream goal i assertions)
  ;;Each stream element is replaced by a new stream
  ;; incorporating the proofs of "goal".
  (cond ((null stream) ())
        (t (append (resolve1 goal i assertions (cdar stream) (caar stream))
                   (resolve (cdr stream) goal i assertions)))))


(defun resolve1 (goal i assertions j env)
  (cond ((null assertions) ())
        ((null env) ())
        (t (let ((env1 (unify goal i (caar assertions) j env)))
             (append (prove '((,env1 . ,(1+ j))) (cdar assertions) j)
                     (resolve1 goal i (cdr assertions) j env))))))
```

A concrete implementation along the lines in Section 2.2 is given in Appendix I.

2.4 Comparison

It is fairly obvious that using success continuations recurses deeper and so consumes more stack space, whereas using proof streams constructs more delayed objects and so consumes more cons space. However, by introducing in the code special cases for e.g. last conjunct or last disjunct, the behavior of both schemes improves significantly.

Moreover both interpreters contain plenty of direct and indirect tail recursion, which of course is transformed to iterative form in "production" versions of the algorithms.

3. Structure Sharing vs. Structure Copying

A major source of inefficiency in the above interpreters is the implementation of the binding environment. It is implemented as an association list without any indexing. To get the bound value of a variable one may have to search the whole list. Worse, one may have to do repeated searches in case there are variable-to-variable bindings. This means that execution times become at least $O(n^2)$, where n is the number of nodes of the and-tree of a proof.

In *structure sharing* implementations such as Warren's [11] every use of an assertion has its own activation-record like binding environment. There is then no need to search the environment for a binding. Warren lets a base register point to the activation record and assigns offsets to variables. He is then able to get a variable binding in just one machine instruction. Structure sharing is typically implemented to let unify destructively update the binding environments.

Another method is *structure copying* in which one uses *copies* of assertions. A new copy is constructed in each resolution step. The copies contain value cells, and unify is allowed to destructively update these. Whether it is worth while to recycle the copies is an open question.

A more sophisticated variant is unification driven structure copying, where "pure code" is copied only when it is unified with a variable. This variant is used in the systems described by Mellish [8] and by Carlsson and Kahn [3].

With destructive changes, the need to undo these arises. Structure sharing and structure copying implementations typically use a *reset list* or *trail* to record all variable bindings, in order to know what to undo upon backtracking.

The cost of constructing copies of assertions should be weighted against the relative complexity of structure sharing implementations where terms always must be "seen" through an index (a pointer to a binding environment). This means that twice as many arguments have to be passed around in the inner loop of the interpreter.

On computers with indirect addressing support in Lisp one can even make pointers to value cells totally transparent. This is a very attractive feature since it drastically reduces the cost of interfacing Prolog to Lisp, which one typically wants in a Lisp-based Prolog system. An extensive comparison of structure sharing vs. structure copying has been done by Mellish [8].

We will now further refine the success continuation technique to use structure copying. It is left as an exercise to the reader to implement indexed structure sharing.

3.1 Structure Copying Interpreter

Value cells are represented here as pairs

(var. *value*)

The *value* field of an unbound value cell points to the cell itself.

The following is the central parts of a success continuation interpreter that uses "naive" structure copying. Note here that an assertion is represented as *code* to construct a copy of the assertion in question.

```
(defun prove (goals cont)
  ;;Proves "goals" and calls "cont".
  (cond (goals (resolve (car goals) `(prove ,(cdr goals) ,cont)))
        (t (invoke cont))))


(defun resolve (goal cont)
  ;;For all proofs of "goal", "cont" is called.
  (try-assertions goal (assertions goal) *trail* cont))


(defun try-assertions (goal assertions mark cont)
  (cond (assertions
         (cond ((try-assertion goal (car assertions) cont))
               (t (reset mark)
                  ;;Reset trail back to "mark".
                  (try-assertions goal (cdr assertions) mark cont))))))


(defun try-assertion (goal assertion cont)
  (cond ((unify goal (funcall (car assertion)))
         (prove (funcall (cdr assertion)) cont))))
```

```
(defun unify (x y)
  (cond ((eq x y))
        ((and (consp x) (consp y))
         (and (unify (dereference (car x)) (dereference (car y)))
              (unify (dereference (cdr x)) (dereference (cdr y)))))
        ((variable-p x) (unify-variable x y))
        ((variable-p y) (unify-variable y x))
        ((equal x y))))

(defun unify-variable (x y)
  ;;Unifies a variable with a term.
  (progn (push x *trail*) (rplacd x y)))

(defun dereference (x)
  ;;Follows chain of linked variables.
  (cond ((variable-p x)
         (cond ((eq x (cdr x)) x)
               (t (dereference (cdr x)))))
        (t x)))
```

The global variable *trail* holds the reset stack. The function cell creates a value cell. The function reset restores value cells to the unbound state. Variable-p is a predicate for recognizing value cells.

Each assertion in the database is represented by a pair of functions, where the first element constructs the head of an assertion and the second element constructs the body. This is exemplified by the database entry for append:

```
(defprop append
  ((app-1-1 . app-1-2) (app-2-1 . app-2-2))
  :assertions)
(defun app-1-1 ()
  (setq ?X (cell)) '(append () ,?X ,?X))
(defun app-1-2 () '())
(defun app-2-1 ()
  (setq ?X (cell) ?XS (cell) ?Y (cell) ?ZS (cell))
  '(append (,?X . ,?XS) ,?Y (,?X . ,?ZS)))
(defun app-2-2 () '((append ,?XS ,?Y ,?ZS)))
```

A Note on Determinacy

It should be noted that the stack space consumption of the above interpreter can be much reduced if prove can test whether a goal is determinate, viz.

```
(defun prove (goals cont)
  (cond ((null goals) (invoke cont))
        ((determinate (car goals))
         (and (resolve (car goals) '(true)) (prove (cdr goals) cont)))
        (t (resolve (car goals) '(prove ,(cdr goals) ,cont)))))
```

4. Adding Builtin Predicates

In addition to the pure Horn clause theorem proving capabilities, any Prolog implementation needs builtin predicates. This can be done e.g. in the following way: Let the *:assertions* property of the predicate symbol be the pair

(:builtin . *function*)

where *function* accepts two arguments: a goal and a continuation. Resolve needs to take care of this case. As an example, we show here how to implement bagof with this technique.

```
(defun resolve (goal cont)
  (let ((assertions (assertions goal)))
    (cond ((eq ':builtin (car assertions))
           (funcall (cdr assertions) goal cont))
          (t (try-assertions goal assertions *trail* cont)))))

;;(bagof ?t ?p ?b): ?b is the bag of all ?t such that ?p holds

(defprop bagof (:builtin . bagof-prover) :assertions)

(defun bagof-prover (goal cont)
  (let ((mark *trail*)
        (reslist (list ())) ((?t ?p ?b) (cdr goal)))
        ;;Reslist collects the result.
    (resolve ?p `(bagof-aux ,?t ,reslist))
    (reset mark)
    (cond ((unify (dereference ?b)
                  (nreverse (car reslist)))
           (invoke cont)))))

(defun bagof-aux (term reslist)
  (push (instantiate (dereference term)) (car reslist))
  nil)
```

where instantiate is a function that copies its argument, removing bound value cells and replacing unbound value cells by fresh ones. This is necessary since different proofs of ?p may assign different values to value cells in ?t.

Note that bagof-aux always returns nil. This is to force the theorem prover to really find all proofs of ?p. In try-assertions, the value returned from the non-tail-recursive call to prove is tested, and if non-nil, no more assertions are tried. This is used by the toplevel function prove so that the user can stop the search at a particular proof. It also prepares for the issue coming up in the next section.

5. Adding Cut

The cut control primitive needs extra machinery. The test in try-assertions mentioned above offers the control alternatives "find all proofs" vs. "find first proof" for a given goal. Cut, however, is more complex because it is lexically scoped, and so at run time one needs some device that can mimic lexical scoping. One such device is to keep track of the ancestor depth of the current and-tree node. Instead of returning nil for failure, the theorem prover and

builtin functions can return an integer specifying an ancestor depth to return to. These ideas are implemented as follows.

```
(defun prove (goals cont d)
  ;;Proves "goals" at depth "d" and calls "cont".
  (cond (goals (resolve (car goals) '(prove ,(cdr goals) ,cont ,d) d))
        (t (invoke cont))))
(defun resolve (goal cont d)
  (let ((assertions (assertions goal)))
    (cond ((eq ':builtin (car assertions))
           (funcall (cdr assertions) goal cont d))
          (t (try-assertions goal assertions *trail* cont d)))))
(defun try-assertions (goal assertions mark cont d)
  (cond (assertions
         (let ((msg (try-assertion goal (car assertions) cont d)))
           ;;This code returns "msg" to the right level.
           (cond ((> msg d)
                  (reset mark)
                  (try-assertions goal (cdr assertions) mark cont d))
                 ((= msg d) *failure*)
                 (t msg))))
        (t *failure*)))
(defun try-assertion (goal assertion cont d)
  (cond ((unify goal (funcall (car assertion)))
  ;;The depth is increased in each resolution step.
         (prove (funcall (cdr assertion)) cont (1+ d)))
        (t *failure*)))
(defprop cut (:builtin . cut-prover) :assertions)
(defun cut-prover (ignore cont d)
  ;;Upon backtracking, cut fails the parent goal.
  (invoke cont) (1- d))
```

The constant *failure* contains a large integer.

6. Prolog

To extend the toy interpreters of this paper into full-fledged Prolog systems is straight forward and has been done. The resulting system is comparable in speed with interpreted DECsystem-10 Prolog and is listed in Appendix II.

7. Conclusions

We have surveyed techniques for implementing Prolog interpreters in Lisp. We have accounted for the procedural semantics of the language in terms of functional programming constructs which can be considered a very high level abstract machine.

Related work includes Komorowski [5]. He gives a denotational semantics for Prolog involving configurations and stacks to hold the state of his abstract machine. He made no use of continuations. This is surprising considering how well they suggest themselves for defining the operational semantics of Prolog as we have tried to show in this paper.

8. Appendix I: Proof Streams With Continuations

The following is the central part of a proof stream interpreter as was discussed in section 2.2. All of these functions return proof streams which are either

()

for failure i.e. no more proofs, or a pair

(*environment* . *continuation*)

for success i.e. a proof was found. *Continuation* is a function that returns a new proof stream. *Environment* is a pair of (*i*) the variable substitutions involved in a particular proof and (*ii*) next available index. Backtracking is achieved by calling the continuation of successive proof streams until failure eventually results. All of these functions return proof streams.

```
(defun prove (env newi goals oldi)
  ;;Returns the proofs of "goals" in "env".
  (cond ((null env) ())
        ((null goals) `((,env . ,newi) . (false)))
        (t (invoke-in-each
             (resolve (car goals) oldi (assertions (car goals)) newi env)
             `(prove ,(cdr goals) ,oldi)))))

(defun resolve (goal oldi assertions newi env)
  ;;Returns the proofs of "goal" in "env".
  (cond ((null assertions) ())
        (t (invoke-after
             (prove (unify goal oldi (caar assertions) newi env)
                    (1+ newi)
                    (cdar assertions)
                    newi)
             `(resolve ,goal ,oldi ,(cdr assertions) ,newi ,env)))))

(defun invoke-after (stream1 cont)
  ;;Appends "cont" onto the stream "stream1".
  (cond ((null stream1) (invoke cont))
        (t (cons (car stream1) `(invoke-after* ,(cdr stream1) ,cont)))))

(defun invoke-in-each (stream1 cont)
  ;;Invokes "cont" in each element of a proof stream
  (cond ((null stream1) ())
        (t (invoke-after (invoke cont (caar stream1) (cdar stream1))
                         `(invoke-in-each* ,(cdr stream1) ,cont)))))
```

```
(defun invoke-after* (delayed continuation)
  (invoke-after (invoke delayed) continuation))
(defun invoke-in-each* (delayed continuation)
  (invoke-in-each (invoke delayed) continuation))
```

9. Appendix II: A Complete Interpreter

We give here the source listing of a complete structure copying Prolog interpreter.

```
; -*- Mode: Lisp; Base: 10.; -*-
; A structure copying Prolog written in MacLisp. 1200-1500 LIPS unless GC.
; Written by Mats Carlsson, UPMAIL, Uppsala University, 1983.

;This is an interpreter for "naked" Prolog. It has very few builtin predicates.

;It's free to use by anyone but if you make money out of it I would like some.

;A lot of direct and indirect tail recursion to be transformed to iteration
;by someone who has a good source to source optimizer.

;Syntax:        Terms and predications are lists (<functor> . <arguments>),
;               assertions are lists (<consequent> . <antecedents>),
;               variables are symbols beginning with "?".
;
;Toplevel functions:
;(yaq)                          Read-prove-print top loop.
;(define <name> . <assertions>) Defines predicates.
;
;Builtin predicates:
;(cut)                          Infamous control primitive.
;(call ?goal)                   Tries to prove ?goal.
;(bagof ?t ?goal ?b)            ?B is a bag of instances of ?T that satisfy ?goal.
;(lisp-predicate ?form)         ?Form is lisp-evaluated and must return non-NIL.
;(lisp-command ?form)           ?Form is lisp-evaluated, may return whatever.
;(lisp-value ?var ?form)        ?Var is the lisp-value of ?form.
;(cl ?clause)                   ?Clause is an assertion in the knowledge base. Backtracks.
;(addcl ?clause)                ?Clause is added to the knowledge base.
;(addcl ?clause ?n)             ?Clause is added after the ?n-th clause of its
;                               procedure.
;(delcl ?clause)                Clauses matching ?clause are deleted. Backtracks.
;(delcl ?predicate ?n)          Deletes ?n-th clause of ?predicate.

(declare (fixsw t) (special *cells* *variables*))
(defvar *trail* ())
(defvar *inferences* 0)
(defvar *failure* 32767)

(defmacro cell () '(let ((x (cons '|var| nil))) (rplacd x x)))

(defmacro consp (x) '(not (atom ,x)))

(defmacro variable-p (x) '(and (consp ,x) (eq '|var| (car ,x))))

(defmacro unify-variable (x y) '(progn (push ,x *trail*) (rplacd ,x ,y)))

(defmacro assertions (goal) '(get (car ,goal) ':assertions))

(defmacro head (assertion) '(funcall (car ,assertion)))

(defmacro body (assertion) '(funcall (cdr ,assertion)))

(defmacro invoke (x) '(apply (car ,x) (cdr ,x)))

(defmacro defbuiltin (name &rest body)
  (let ((g (gensym)))
    '(progn 'compile
            (defun ,g (goal continuation depth) ,@body)
            (defprop ,name (:builtin . ,g) :assertions))))

(defun instantiate (x generator)
  ;;create a copy of x where unbound variables are handled by generator
  (let (*cells*) (instantiate-1 x generator)))

(defun instantiate-1 (x generator)
  (cond ((variable-p x)
         (cond ((assq x *cells*) (cdr (assq x *cells*)))
               (t (let ((c (funcall generator x)))
                    (push (cons x c) *cells*) c))))
        ((consp x) (cons (instantiate-1 (dereference (car x)) generator)
                         (instantiate-1 (dereference (cdr x)) generator)))
        (t x)))

(defun copycell (ignore)
  ;;this generator just makes a new value cell
  (cell))

(defun ?cell (ignore)
  ;;this generator creates a symbol ?0 ?1...
  (maknam '(#/? ,(+ #/0 (length *cells*)))))

;;------------------------- KERNEL OF INTERPRETER --------------------------;;
```

```
                    (head-name (gensym))
                    (body-name (gensym)))
                (push (cons head-name body-name) toplist)
                `((defun ,head-name ()
                    ,@(mapcar '(lambda(v) `(setq ,v (cell))) headvars)
                    ,headcode)
                  (defun ,body-name ()
                    ,@(mapcar '(lambda(v) `(setq ,v (cell)))
                              (diffq bodyvars headvars))
                    ,bodycode))))
            assertions)
          (defprop ,name ,(nreverse toplist) :assertions))))

(defun yaq ()
  (do ((mark *trail*)) (())
    (format t "~%YAQ>")
    (let* ((sexpr (read))
           ((sexprvars sexprcode) (variables-and-constructor sexpr))
           (cells
            (mapcar '(lambda(x) (set x (cell))) sexprvars)))
      (gc)
      (unwind-protect
       (prove
        `(,(eval sexprcode)
          (lisp-predicate-?vars
           (display ',sexprvars ',cells ',*inferences* ',(runtime))))
        '(+) 0)
       (reset mark)))))

(defmacro gprin1 (x) `(funcall (or prin1 'prin1) ,x))

(defun display (names cells inf time)
  (let ((dt (- (runtime) time)))
    (format t "~%It took ~S microseconds, ~S LIPS." dt
            (// (* 1000000. (- *inferences* inf)) dt))
    (mapc #'(lambda (n c) (format t "~%~S = " n) (gprin1 c)) names cells)
    (progn (format t "~%Ok?") (y-or-n-p))))

(sstatus linmode nil)
(setq base 10. ibase 10. *nopoint t)
(alloc '(list (65536. 131072. 0.25)))

;;--------------------------- BUILTIN PREDICATES ---------------------------;;

(define lisp-command
  ((lisp-command ?X) (lisp-predicate (progn ?X t))))

(define lisp-value
  ((lisp-value ?X ?Y) (lisp-predicate (unify '?X ?Y))))

(define cl
  ((cl ?cl) (|cl-delcl| ?cl nil)))

(define addcl
  ((addcl ?cl) (|addcl| ?cl))
  ((addcl ?cl ?n) (|addcl| ?cl ?n))) ;;after nth, 1-indexed

(define delcl
  ((delcl ?cl) (|cl-delcl| ?cl t))
  ((delcl ?pred ?n) (|delcl-1| ?pred ?n))) ;;delete nth, 1-indexed
```

```
        (t *failure*)))

(defbuiltin |delc1-1|
  (let* ((name (dereference (cadr goal)))
         (n (dereference (caddr goal)))
         (assertions (get name ':assertions)))
    (putprop name (delq (nth (1- n) assertions) assertions) ':assertions)
    (invoke continuation)))

(defbuiltin |addc1|
  (let* ((clause (instantiate (dereference (cadr goal)) '?cell))
         (n (dereference (caddr goal)))
         (name (caar clause))
         (save (get name ':assertions)))
    (eval `(define ,name ,clause))
    (cond ((and (numberp n) (nthcdr n save))
           (let ((x (nthcdr n save)))
             (rplacd x (cons (car x) (cdr x)))
             (rplaca x (car (get name ':assertions)))
             (putprop name save ':assertions)))
          (t (putprop name (nconc save (get name ':assertions)) ':assertions)))
    (invoke continuation)))

(defbuiltin bagof
  (let ((mark *trail*) (reslist (list ())) ((?t ?p ?b) (cdr goal)))
    (resolve (dereference ?p) `(bagof-aux ,?t ,reslist) depth)
    (reset mark)
    (cond ((unify (dereference ?b) (nreverse (car reslist)))
           (invoke continuation))
          (t *failure*))))

(defun bagof-aux (term reslist)
  (push (instantiate (dereference term) 'copycell) (car reslist))
  *failure*)

(defbuiltin call (resolve (dereference (cadr goal)) continuation depth))

(defbuiltin cut (invoke continuation) (1- depth))

(defbuiltin lisp-predicate
  (cond ((eval (instantiate (cadr goal) 'prog1)) (invoke continuation))
        (t *failure*)))

(defbuiltin lisp-predicate-?vars
  (cond ((eval (instantiate (cadr goal) '?cell)) (invoke continuation))
        (t *failure*)))

;;------------------------- DEFINE MACRO & TOP LEVEL -------------------------;;

(eval-when (compile eval load)
 (defmacro syntactic-variable-p (x)
   ;;an ugly way of saying (eq '? (getchar x 1))
   `(and (symbolp ,x) (= #/? (lsh (car (pnget ,x 7)) -29.)))))

 (defun variables-and-constructor (x)
   ;;returns variables occurring in x and a form that will create x
   (let* ((*variables*) (code (constructor x)))
     (list (nreverse *variables*) code)))

 (defun constructor (x)
   ;;returns code that will create x
   (cond ((eq x '?) '(cell))
         ((syntactic-variable-p x)
          (or (memq x *variables*) (push x *variables*))
          x)
         ((ground x) `',x)
         (t `(cons ,(constructor (car x)) ,(constructor (cdr x))))))

 (defun ground (x)
   (cond ((syntactic-variable-p x) nil)
         ((atom x) t)
         ((ground (car x)) (ground (cdr x)))))

 (defun diffq (x y)
   (mapcan '(lambda (x1) (and (not (memq x1 y)) (list x1))) x)))

(defmacro define (name &rest assertions)
 (let ((toplist))
  `(progn 'compile
          ,@(mapcan
             '(lambda(a)
               (let (((headvars headcode) (variables-and-constructor (cdar a)))
                     ((bodyvars bodycode) (variables-and-constructor (cdr a)))
```

```
(defun dereference (x)
  ;;follows a chain of linked variables
  (cond ((variable-p x)
         (cond ((eq x (cdr x)) x) (t (dereference (cdr x)))))
        (t x)))

(defun unify (x y)
  (cond ((variable-p x) (unify-variable x y))
        ((variable-p y) (unify-variable y x))
        ((consp x)
         (and (consp y)
              (unify (dereference (car x)) (dereference (car y)))
              (unify (dereference (cdr x)) (dereference (cdr y)))))
        ((equal x y))))

(defun reset (mark)
  (cond ((not (eq mark *trail*))
         (let ((cell (pop *trail*))) (rplacd cell cell) (reset mark)))))

(defun prove (goals continuation depth)
  (cond ((cdr goals)
         (resolve
          (car goals) `(prove ,(cdr goals) ,continuation ,depth) depth))
        (goals
         (resolve (car goals) continuation depth))
        (t (invoke continuation))))

(defun resolve (goal continuation depth)
  (setq *inferences* (1+ *inferences*))
  (let ((assertions (assertions goal)))
    (cond ((eq ':builtin (car assertions))
           (funcall (cdr assertions) goal continuation depth))
          (t (try-assertions goal assertions *trail* continuation depth)))))

(defun try-assertions (goal assertions mark continuation depth)
  (cond ((cdr assertions)
         (let ((msg (try-assertion goal (car assertions) continuation depth)))
           (cond ((> msg depth)
                  (reset mark)
                  (try-assertions
                   goal (cdr assertions) mark continuation depth))
                 ((= msg depth) *failure*)
                 (t msg))))
        (assertions (try-assertion goal (car assertions) continuation depth))
        (t *failure*)))

(defun try-assertion (goal assertion continuation depth)
  (cond ((unify (cdr goal) (funcall (car assertion)))
         (prove (funcall (cdr assertion)) continuation (1+ depth)))
        (t *failure*)))

;;---------------------------- BUILTIN PRIMITIVES ----------------------------;;

(defbuiltin |cl-delcl|
  (let* ((clause (dereference (cadr goal)))
         (delcl-p (dereference (caddr goal)))
         (name (caar clause)))
    (|cl-delcl-clauses|
     clause delcl-p name *trail* (assertions (car clause)) continuation depth)))

(defun |cl-delcl-clauses|
  (clause delcl-p name mark assertions continuation depth)
  (cond ((cdr assertions)
         (let ((msg (|cl-delcl-clause|
                     clause delcl-p name (car assertions) continuation)))
           (cond ((> msg depth)
                  (reset mark)
                  (|cl-delcl-clauses| clause delcl-p name mark (cdr assertions)
                                      continuation depth))
                 ((= msg depth) *failure*)
                 (t msg))))
        (assertions
         (|cl-delcl-clause| clause delcl-p name (car assertions) continuation))
        (t *failure*)))

(defun |cl-delcl-clause| (clause delcl-p name assertion continuation)
  (cond ((and (unify (cdar clause) (head assertion))
              (unify (cdr clause) (body assertion)))
         (cond (delcl-p
                (putprop name
                         (delq assertion (get name ':assertions))
                         ':assertions)))
         (invoke continuation))
```

10. References

[1] Abelson H., Sussman, G.J., *Course notes of MIT EECS 6.001 Structure and Interpretation of Computer Programs,* Problem Set 9, December 1981

[2] Boyer, R.S., Moore, J.S., *The sharing of structure in theorem proving programs,* Machine Intelligence 7, (ed. Meltzer and Mitchie), Edinburgh UP, 1972

[3] Carlsson, M., Kahn, K.M., *LM-Prolog User Manual,* UPMAIL Technical Report No. 24, Computing Science Department University of Uppsala, Sweden, 1983

[4] van Emden, R.H., An Interpreting Algorithm For Logic Programs, Proc. 1st. International Logic Programming Conference, Marseille, 1982

[5] Komorowski, H.J., *A Specification Of An Abstract Prolog Machine And Its Application To Partial Evaluation,* Ph.D. Thesis, Linköping University, Linköping, Sweden, 1981

[6] Kornfeld, W.A., *Equality for Prolog,* Proc. 8th IJCAI, Karlsruhe, Germany, 1983

[7] McCabe, F., *Abstract PROLOG machine - a specification,* Department of Computing, Imperial College, London, 1983

[8] Mellish, C.S., *An Alternative To Structure Sharing In The Implementation Of A Prolog Interpreter,* in [10].

[9] Strachey, C., Wadsworth, C.P. *Continuations - A Mathematical Semantics For Handling Full Jumps,,* Programming Research Group, Oxford University, 1974

[10] Tärnlund, S.-Å., Clark, K.L. (eds.), *Logic Programming,* Academic Press, London, 1982

[11] Warren, D.H.D., *Implementing Prolog - compiling predicate logic programs,* Department of Artificial Intelligence, University of Edinburgh, 1977