PURE LISP IN PURE PROLOG

Luís Moniz Pereira
António Porto
Departamento de Informática
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal

An evaluator for pure Lisp in pure Prolog is presented below:

1) eval(E, U, R) *takes an* S-expression *and evaluates it to* R, *in the context of association list* U *comprising two element lists pairing atoms to their associated values*
2) *It features the Prolog system predicates:*

X=Y	X unifies with Y
atom(A)	A is an atom
integer(I)	I is an integer
atomic(A)	A is an atom or integer

3) *Prolog syntax is used for lists. As usual in Lisp 'false' is represented by the empty list, in this case '[]'*
4) *Examples of calls are:*

```
?- eval( [ff,X], [ [X,[[1,2],3]] ], R).        gives R=1
?- lisp.
[alt,'[1,2,3,4,5]].                            gives [1,3,5]
```

5) *'equal' is made primitive rather than the implementation oriented concept 'eq'*
6) *numeric functions and predicates are left out*
7) *space may be recovered by garbage collecting each cycle:*

```
lisp :- repeat, solve( (read(E), eval(E,[],R), write(R), nl,nl) ), fail.
solve(G) :- G, !.
```

where 'repeat' is a system predicate that always solves again

8) *'assert' is used as an optional convenience for storing functions interactively*

```
?- op(10,fx,').
lisp :- read(E), eval( E, [], R), write(R), nl, nl, lisp.
eval( A, U, R) :- atomic(A),
        ( ( integer(A) ; A=[] ; A=true ), R=A ;
          assoc( A, U, [_,R]) ;
          error ).
eval( [quote,X], _, X).
eval( 'X, _, X).
eval( [cond,[T,B]|L], U, R) :- eval( T, U, ET),
                            ( ET=true, eval( B, U, R) ;
                              eval( [cond|L], U, R) ).
eval( [cond], _, []).
eval( [list,[X|L]], U, [EX|EL]) :- eval( X, U, EX), eval( [list,L], U, EL).
eval( [list], _, []).
eval( [car,X], U, Y) :- eval( X, U, EX) ( EX=[Y|_] ; error ).
eval( [cdr,X], U, Y) :- eval( X, U, EX) ( EX=[_|Y] ; error ).
eval( [cons,X,Y], U, [EX|EY]) :- eval( X, U, EX), eval( Y, U, EY).
eval( [atom,X], U, R) :- eval( X, U, EX), ( atomic(EX), R=true ; R=[] ).
eval( [equal,X,Y], U, R) :- X=Y ;
                            eval( X, U, EX),
                            eval( Y, U, EY),
                            ( EX=EY, R=true ; R=[] ).
eval( [F|L], U, R) :- assoc( F, U, P),
                      ( P=[_,EF], eval( [EF|L], U, R) ; error ).
eval( [[lambda,V,E]|A], U, R) :- evalist( A, U, EA),
                                 pair( V, EA, P),
                                 append( P, U, W),
                                 eval( E, W, R).
eval( [not,X], U, R) :- eval( X, U, EX), ( EX=true, R=[] ; R=true ).
eval( [and, X,Y], U, R) :- eval( X, U, EX), ( EX=[], R=[] ; eval( Y, U, R) ).
eval( [or, X,Y], U, R) :- eval( X, U, EX), ( EX=[], eval( Y, U, R) ; R=EX ).
eval( [defun,N,A,E], _, N) :- assert( definition( N, [lambda,A,E]) ).
eval( [eval,X], U, R) :- eval( X, U, EX), eval( EX, U, R).

/* extra notation */
eval( [null,X], U, R) :- eval( [equal,X,[]], U, R).
eval( [if,C,A,B], U, R) :- eval( [cond,[C,A],[true,B]], U, R).

/* association list */
assoc( X, _, [_,R]) :- definition( X, R).
assoc X, [[Y,VY]|U], R) :- X=Y, R=[Y,VY] ; assoc( X, U, R).

/* examples of defined functions*/
definition( ff, [lambda,[x],[if,
                              [atom,x],
                              x,
                              [ff,[car,x]]]] ).
definition( alt, [lambda,[u],[if,
                               [null,u],
                               [],
                               [if,
                                  [null,[cdr,u]],
                                  u,
                                  [cons,[car,u],[alt,[cdr,[cdr,u]]]]]]]
          ).

/* utilities */
error :- write(error), tab(2), abort.
evalist([H|T],U,[EH|ET]) :- eval( H, U, EH), evalist(T,U,ET).
evalist(  [] ,_,   []  ).
pair([X|Y],[U|V],[[X,U]|P]) :- pair(Y,V,P).
pair(  [] ,  [] ,    []   ).
append([H|T],L,[H|R]) :- append(T,L,R).
append(  [] ,L,   L  ).
```

REFERENCE: JOHN McCARTHY and CAROLYN TALCOTT: "Lisp Programming and Proving" (draft), Stanford University 1981.