A LIST EXPRESSION INTERPRETER
AS A TEACHING TOOL

Jeffrey A. Brumfield

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-84-23 July 1934

Abstract. Implementing an interpreter for a simple list manipulation language gives students experience with a variety of data structures and programming techniques. This paper describes a complete programming project and discusses its educational value.

1. INTRODUCTION

This paper describes a programming project used in a data structures course taken by second year undergraduate computer science majors. While the primary goal of the assignment is to give students experience with generalized lists, several important programming techniques and language concepts are used.

The assignment requires the students to implement an interpreter for a simple list manipulation language. Although the language is similar to Lisp, no knowledge of that language is required. In fact, students are often not told of the similarities until after the assignment has been completed.

An interesting feature of this assignment is that the software being written could be extended to a complete Lisp interpreter. The program is not a toy used only to illustrate a particular data structure or programming technique.

In the next section, we overview the project and describe how it is partitioned for presentation to students. We then discuss the important concepts that are illustrated by the assignment. Finally, we describe several variations on the project that increase its usefulness. The complete text of the assignment can be found in the appendix, along with test data and a sample solution to the first part of the project.

2. OVERVIEW OF THE PROJECT

A list expression is a generalized list (i.e., a list whose items may themselves be lists) that specifies a composition of functions operating on data. A list expression is written as a sequence of items enclosed in parentheses. The first item specifies a function; the remaining items are operands, which may also be list expressions. For example, the arithmetic expression 2*4 + 7

could be represented by the list expression (PLUS (TIMES 2 4) 7). The value of a list expression is the value of the function applied to the operands.

The goal of the project is to design and implement an interpreter that reads and evaluates list expressions. A set of primitive functions is built into the interpreter; one of these functions allows the user to define additional functions. Because the data on which functions operate has the same syntax as a list expression, a QUOTE function must be provided to prevent the evaluation of data.

To make the project manageable by students who have never written large programs, the assignment is divided into several parts. Each part builds on the previous parts. Thoroughly testing each part before beginning the next has proved essential.

In part 1, routines to read and write generalized lists are implemented. All input and output is performed by these routines. An internal representation of generalized lists that does not require the use of Pascal variant records was chosen. Details can be found in Appendix A.

In part 2, the core of the interpreter is implemented. The resulting program is capable of reading a list expression, evaluating it, and writing the result. Seven primitive functions and predicates are supported at this stage so the interpreter can be tested on simple list expressions.

To allow simple programs to be written as list expressions, two features are needed: a conditional function (analogous to an if-then-else statement in procedural languages) and the ability to define recursive functions. Both of these features are added to the interpreter in Part 3. A built-in function called DEFINE allows a list expression having at most two parameters to be saved and later referenced by name. The evaluation routine must be modified to distinguish among built-in functions, user-defined functions, and parameter names.

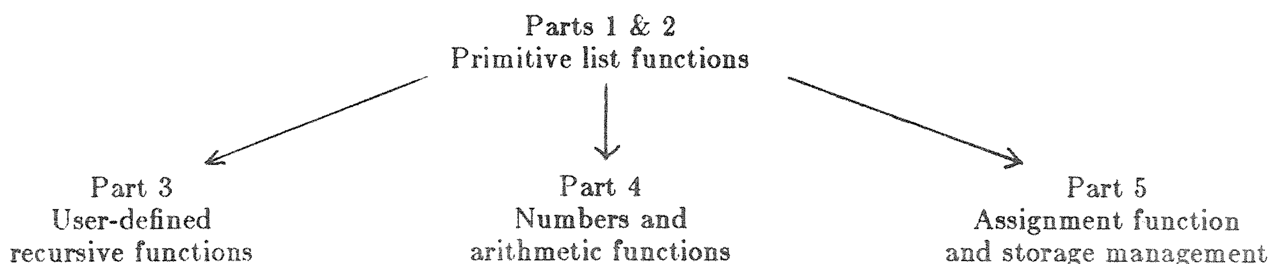


Figure 1. Possible extensions to the basic assignment.

components selected, a 10 day to 6 week assignment can be created.

Part 4 could consist of adding numbers and arithmetic functions to the interpreter. A minimum set of functions should include relational predicates as well as addition, subtraction, multiplication, and division. While this extension requires only a few new ideas, it makes the list manipulation language more powerful and interesting. If interactive execution is possible, the interpreter can then be used as a desk calculator. Parts 3 and 4 together allow list expressions to be written for solving complicated programming problems.

Part 5 could introduce an assignment function that allows a value to be associated with a name. Like the DEFINE function, the assignment function requires that a name and the internal representation of a generalized list be saved after the evaluation of a list expression. A complementary function could disassociate the name and the value. To allow large amounts of input to be processed, the storage occupied by list structures that are no longer needed should be freed at the earliest possible time. Since several names can be associated with different parts of a list structure, reference counts must be used. Nodes in the internal representation of a generalized list can be disposed only when they are not associated with any name.

Error detection and recovery is an important part of any language processor. If students will be writing their own list expressions, the detection of some common errors will make their interpreter much easier to use. An error message can be issued if an unrecognized function or parameter name is referenced, or a built-in function is invoked with the wrong number or type of arguments. When one of these errors is detected, the evaluation of the current expression can be aborted; the processing of the remaining expressions should be unaffected. While extra closing parentheses in a list expression could be ignored, a missing closing parenthesis will cause all remaining input to be read as part of the current list expression and an unexpected end-of-file to be encountered. There is no way to prevent this without using a special list expression terminating character. If a data structure such as the function symbol table or the execution stack becomes full, the interpreter should write a descriptive message before halting.

5. EXPERIENCE AND CONCLUSIONS

Student feedback on this assignment has been encouraging. Students most frequently remark that they feel comfortable with recursion for the first time. (In fact, after completing the assignment students often produce recursive solutions to exam problems for which their instructor was expecting iterative solutions!) Many students express an interest in learning Lisp; others plan to extend their programs to support additional features.

As with any challenging project, not all students successfully complete this assignment. Part 1 seems to take the greatest toll. Depending on the amount of assistance students are given, 10 to 20 percent of all students are not able to implement the first two parts. This assignment has been a motivating factor for students withdrawing from the course.

The success of this project indicates the value of programming assignments that students find meaningful. The large number of variations allow the assignment to be used several semesters, stressing a different set of data structures and algorithms each time. Overall, the project has proved to be ideal for use in an undergraduate programming course.

Acknowledgments

The initial idea for this project came from a problem in *Fundamentals of Data Structures* by Horowitz and Sahni, Computer Science Press, 1976.

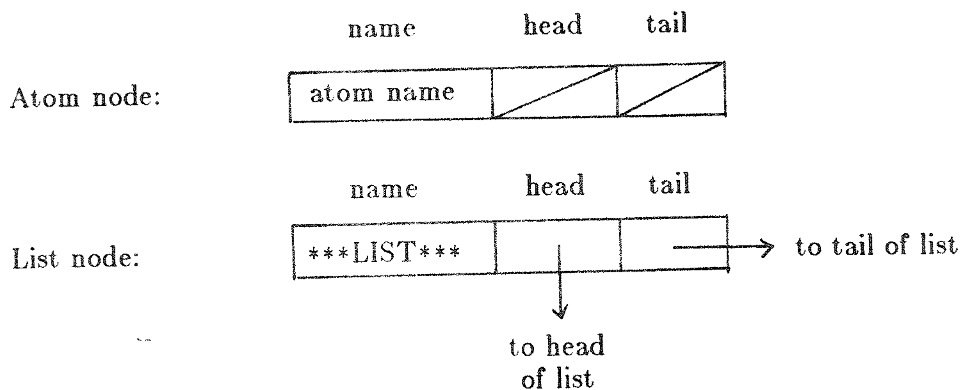
Appendix A - Text of the Assignment

External format of generalized lists. The format of a generalized list can be defined recursively as

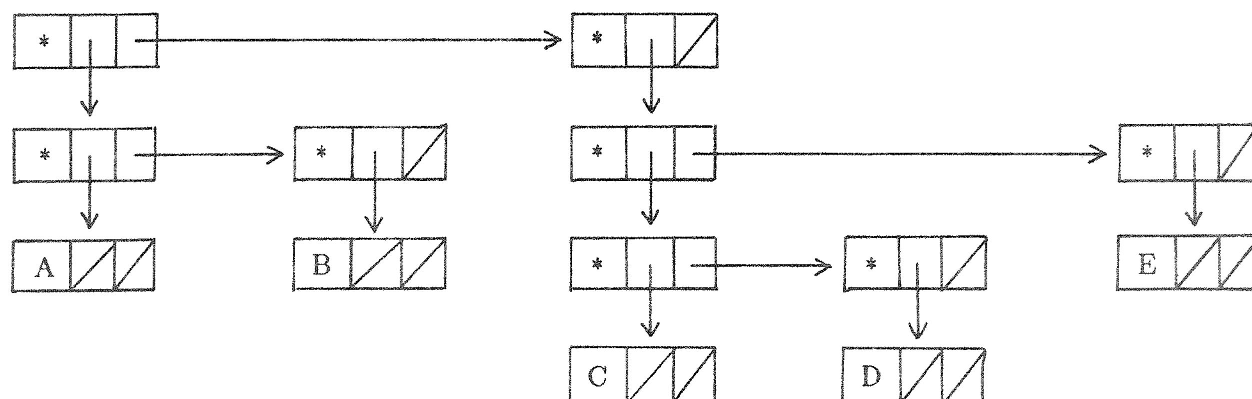
$$\textit{g-list} \quad = \quad \begin{cases} \textit{atom} \\ (\ \textit{g-list} \ \ldots \ \textit{g-list}\ ) \end{cases}$$

where *g-list* ... *g-list* is a sequence of zero or more generalized lists separated by at least one blank. An *atom* is an alphanumeric string of up to 10 characters. See the test data file for examples of generalized lists. In the test data each generalized list will begin on a new line, but may span several input lines.

Internal representation of generalized lists. In our representation of a generalized list, nodes will have three fields whose contents depend on whether the node represents an atom or a list.



The following structure shows the representation of the list ((A B) ((C D) E)). The list node flag is abbreviated '*'.



Part 1. Design, code, and test two recursive procedures named READLIST and WRITELIST. READLIST inputs a generalized list and creates its internal representation as described above. WRITELIST outputs a generalized list. As a list is being read, every character must be echoed to the output file.

Implementing READLIST and WRITELIST will be simpler if you first write a set of input/output routines. A function NEXTINPUT could return the next nonblank character from the input file without advancing the file pointer past it. Procedures READSTRING and WRITESTRING could read and write alphanumeric strings. NEXTINPUT and READSTRING could also echo the input characters to the output file. Try to eliminate all input/output statements from READLIST and WRITELIST.

Part 1 can be implemented in about 130 lines of Pascal, not including comments. Test your routines by reading and then writing each list in the first set of test data. If you have difficulty debugging your routines, test them on the sequence of lists A, (A), (A B), ((A) B), ((A) (B)), and desk check your code for the first input that causes your program to fail.

Part 2. Write procedures or functions to perform the following basic operations on the internal representations of generalized lists:

Function	Value
HEAD *l*	first item in *l*
TAIL *l*	list *l* with first item removed
CONS *s* *l*	a list with head *s* and tail *l*
QUOTE *s*	*s* itself
NULL *l*	true if *l* contains no items
ATOM *s*	true if *s* is an atom
EQUAL *s* 1 *s* 2	true if *s* 1 and *s* 2 are identical

An argument *l* may not be an atom; *s* , *s* 1, and *s* 2 may be atoms or lists. The parameters of these procedures should be pointers to generalized lists. Operations that have a value of true or false should return a pointer to a predefined atom *T* or *F*.

Implement a recursive procedure EVAL that invokes the function specified by the head of a list using the values of the tail of the list as arguments. For example,

(QUOTE (A B C)) has the value (A B C)
(HEAD (QUOTE (A B C))) has the value A
(CONS (QUOTE A) (QUOTE (B C))) has the value (A B C)
(EQUAL (QUOTE (A B)) (QUOTE (A C))) has the value *F*

Write a driver program that calls upon the routines you have implemented to process an input file consisting of a sequence of lists. Each list should be read and evaluated; its value should then be written to the output file. The output could be formatted as follows:

```
LIST MANIPULATION LANGUAGE INTERPRETER

EXPRESSION = (QUOTE (A B C))
VALUE      = (A B C)

EXPRESSION = (HEAD (QUOTE (A B C)))
VALUE      = A

END OF INPUT
```

Run your program on the first test data file. The test data will contain no errors. At this time you are not required to dispose of nodes that are no longer needed.

Part 3. Add to your list processing language a function named COND whose value depends on whether a test is true or false. The syntax of COND is

(COND *test* *true-val* *false-val*)

where *test*, *true-val*, and *false-val* are list expressions. The following is an example of its use:

```
(COND (NULL (QUOTE ()))
      (QUOTE (LIST IS NULL))
      (QUOTE (LIST IS NOT NULL)) )  has value (LIST IS NULL)
```

The evaluation of COND causes the list *test* to be evaluated. If its value is true, then the list *true-val* is evaluated and its value becomes the value of the COND function. Otherwise, the list *false-val* is evaluated and its value becomes the value of COND.

Also, add to your language a function named DEFINE which allows the user to define a (possibly recursive) function having at most two parameters. The syntax of DEFINE is

(DEFINE *fctn-name* *param-list* *fctn-body*)

where *fctn-name* is an atom different than any system function name, and *param-list* will always be a list. The following is an example of its use:

```
(DEFINE SECOND (LIS)
               (HEAD (TAIL LIS)) )
```

The evaluation of the DEFINE function causes the following information to be saved in a table: (i) the name of the function, (ii) the names of its parameters, and (iii) a pointer to the

list representing the function body. We will arbitrarily let the value of DEFINE be the name of the newly defined function. DEFINE is evaluated for its effect, not its value.

When a function name is encountered in a list, your EVAL procedure must determine if it is a system function or a user-defined function. If the function is user-defined, its parameters must be evaluated and pointers to their values must be pushed onto an execution stack. (An element of this stack will consist of the name and value of each parameter.) Then the function body should be evaluated. Finally, the pointers to the parameters must be poped from the stack.

EVAL must also handle references to parameter names. Whenever EVAL is given an atom, it determines which parameter is being referenced. It then obtains the value of the parameter from the top of the execution stack. Note that there are no local or global variables; a function can reference only its parameters.

Run your program on the second set of test data. The test data contains no errors. After a list expression has been evaluated, you may wish to dispose of its nodes provided that the expression does not define a new function.

It may be helpful to add to your program a trace option that causes the values of the parameters to be printed each time a function is invoked. The functional value may also be printed when the evaluation is complete. The trace option may be implemented as a pair of system functions, TRACE and UNTRACE, that have no parameters. Use your WRITELIST routine for output.

Appendix B - Test Data

1. First Set

```
(QUOTE (MAKE SURE YOU ARE ECHOING THE INPUT EXACTLY))
(QUOTE (THIS
            LIST
                 SPANS
                       FIVE
                            LINES))
(QUOTE (ATOMS MAY HAVE TEN CHARACTERS))
(QUOTE ())
(QUOTE (A))
(QUOTE ((A B) C ((D E) (F)) () (G)))
(HEAD (QUOTE (A)))
(HEAD (QUOTE (A B C)))
(HEAD (QUOTE ((A (B)) ((C) D) ((E F)))))
(TAIL (QUOTE (A)))
(TAIL (QUOTE (A B C)))
(TAIL (QUOTE ((A (B)) ((C) D) ((E F)))))
(CONS (QUOTE A) (QUOTE (B)))
(CONS (QUOTE (A)) (QUOTE (B)))
(CONS (QUOTE ()) (QUOTE (B)))
(CONS (QUOTE A) (QUOTE ()))
(CONS (QUOTE (A (B C))) (QUOTE ((D E) F)))
(ATOM (QUOTE THISISTRUE))
(ATOM (QUOTE (THIS IS FALSE)))
(NULL (QUOTE ()))
(NULL (QUOTE A))
(NULL (QUOTE (A B C)))
(NULL (QUOTE (())))
(EQUAL (QUOTE A) (QUOTE A))
(EQUAL (QUOTE A) (QUOTE B))
(EQUAL (QUOTE A) (QUOTE (A)))
(EQUAL (QUOTE (A B C)) (QUOTE (A B C)))
(EQUAL (QUOTE (A B C)) (QUOTE (A B C D)))
(EQUAL (QUOTE ()) (QUOTE ()))
(EQUAL (QUOTE ((A B (C D)) ((E) ( ) F)))
       (QUOTE ((A B (C D)) ((E) ( ) F))) )
(EQUAL (QUOTE ((A B (C D)) ((E) ( ) F)))
       (QUOTE ((A B (C D)) ((E) F))) )
(HEAD (TAIL (TAIL (TAIL (QUOTE (A B C D E F))))))
(EQUAL (HEAD (TAIL (QUOTE (A B C D))))
```

```
      (HEAD (TAIL (TAIL (QUOTE (D C B A))))) )
(CONS (HEAD (QUOTE ((A (B)) ((C) D (E)))))
      (TAIL (QUOTE ((A (B)) ((C) D (E))))) )
(ATOM (NULL (QUOTE (A))))
```

2. Second Set

```
(COND  (NULL (QUOTE ()))
       (QUOTE (LIST IS NULL))
       (QUOTE (LIST IS NOT NULL)) )
(COND  (NULL (QUOTE (A B C)))
       (QUOTE (LIST IS NULL))
       (QUOTE (LIST IS NOT NULL)) )
(COND  (EQUAL (QUOTE (A B)) (QUOTE (B A)))
       (HEAD (TAIL (QUOTE (A))))
       (HEAD (TAIL (QUOTE (A B)))) )

(DEFINE CONST ()
       (QUOTE (THIS FUNCTION HAS A CONSTANT VALUE)) )
(CONST)

(DEFINE SECOND (LIS)
       (HEAD (TAIL (LIS)) )
(SECOND (QUOTE (A B C)))
(SECOND (CONST))

(DEFINE FIRSTIN (ARG1 ARG2)
      (COND  (ATOM ARG1)
             (CONS ARG1 ARG2)
             (COND  (NULL ARG1)
                    ARG2
                    (CONS (HEAD ARG1) ARG2) )))
(FIRSTIN (QUOTE A) (QUOTE (B C D)))
(FIRSTIN (QUOTE ()) (QUOTE (B C D)))
(FIRSTIN (QUOTE (A)) (QUOTE (B C D)))

(DEFINE APPEND (LIS1 LIS2)
      (COND  (NULL LIS1)
             LIS2
             (CONS (HEAD LIS1) (APPEND (TAIL LIS1) LIS2)) ) )
(APPEND (QUOTE ()) (QUOTE ()))
(APPEND (QUOTE ()) (QUOTE (A B C)))
```

```
(APPEND (QUOTE (A)) (QUOTE (B C)))
(APPEND (QUOTE (A B)) (QUOTE (C)))

(DEFINE REVERSE (LIS)
      (COND  (NULL LIS)
             (QUOTE ())
             (APPEND  (REVERSE (TAIL LIS))
                      (CONS (HEAD LIS) (QUOTE ())) )))
(REVERSE (QUOTE ()))
(REVERSE (QUOTE (A B C)))
(REVERSE (QUOTE ((A B) C (D E F) (G) H)))
```

Appendix C - Solution to Part 1

```
PROGRAM RWLIST (INPUT,OUTPUT);

(*---------------------------------------------------------------*)
(*   THIS PROGRAM READS AND WRITES GENERALIZED LISTS             *)
(*---------------------------------------------------------------*)

CONST

   (* PROGRAM LIMITS *)

   MAXSTRING = 10;                (* MAX STRING LENGTH *)

   (* SPECIAL CHARACTERS AND FLAGS *)

   LPAREN     = _(_;
   RPAREN     = _)_;
   BLANK      = _ _;
   LISTFLAG   = _***LIST***_;     (* LIST NODE FLAG *)

TYPE
   STRING = PACKED ARRAY [1..MAXSTRING] OF CHAR;
   NODEPOINTER = ↑NODE;  (* POINTER TO NODE IN LIST *)
   LIST = NODEPOINTER;   (* POINTER TO FIRST NODE IN LIST *)

   NODE = RECORD             (* NODE IN LIST REPRESENTATION *)
             NAME : STRING;         (* ATOM NAME OR LIST FLAG *)
             HEAD : NODEPOINTER;    (* NIL IF ATOM NODE *)
             TAIL : NODEPOINTER;    (* NIL IF ATOM NODE *)
          END;  (* NODE *)

VAR
   LISTEXPR   : LIST;               (* LIST INPUT BY MAIN *)

   (* GLOBAL TO READSTRING AND WRITESTRING *)

   STRINGCHAR : SET OF _A_.._<_;  (* SET OF VALID STRING CHARS *)

VALUE
   STRINGCHAR = [_A_.._Z_,_0_.._9_];     (* VALID STRING CHARS *)

(*---------------------------------------------------------------*)
(*   PROCEDURES NEXTINPUT, SKIPIT, ENDFILE, READSTRING, AND      *)
(*   WRITESTRING PERFORM ALL INPUT AND SOME OUTPUT FOR THE       *)
(*   PROGRAM.                                                    *)
(*---------------------------------------------------------------*)
```

```
(*   NEXTINPUT RETURNS THE NEXT NONBLANK CHARACTER IN THE      *)
(*   INPUT FILE.  THE FILE POINTER IS LEFT POINTING TO THE     *)
(*   NONBLANK CHARACTER.                                       *)

FUNCTION NEXTINPUT : CHAR;
BEGIN
   WHILE (INPUT↑ = BLANK) AND (NOT EOF(INPUT)) DO BEGIN
      WRITE (INPUT↑);
      IF (EOLN(INPUT)) THEN BEGIN
         WRITELN;
         WRITE (_ _,_             _);     (* INDENT *)
         END;
      GET (INPUT);
      END;  (* WHILE *)
   NEXTINPUT := INPUT↑;
   END;  (* NEXTINPUT *)

(*   SKIPIT IS CALLED AFTER NEXTINPUT TO ECHO THE NONBLANK     *)
(*   CHARACTER AND ADVANCE THE FILE POINTER.                   *)

PROCEDURE SKIPIT;
BEGIN
   WRITE (INPUT↑);
   GET (INPUT);
   END;  (* SKIPIT *)

(*   ENDFILE IS TRUE IF THERE ARE NO REMAINING NONBLANK        *)
(*   CHARACTERS BEFORE THE END OF FILE MARKER.                 *)

FUNCTION ENDFILE : BOOLEAN;
BEGIN
   ENDFILE := (NEXTINPUT = BLANK) AND EOF(INPUT);
   END;  (* ENDFILE *)

(*   READSTRING READS AND ECHOS A STRING, PADDING OR           *)
(*   TRUNCATING IT TO MAXSTRING CHARACTERS.                    *)

PROCEDURE READSTRING (VAR S:STRING);
VAR
   I : INTEGER;     (* STRING INDEX *)
BEGIN
   I := 1;
   WHILE (INPUT↑ IN STRINGCHAR) AND (I <= MAXSTRING) DO BEGIN
      WRITE (INPUT↑);
      S[I] := INPUT↑;
      GET (INPUT);
      I := I + 1;
      END;
   WHILE I <= MAXSTRING DO BEGIN          (* PAD WITH BLANKS *)
      S[I] := BLANK;
      I := I + 1;
      END;
   WHILE INPUT↑ IN STRINGCHAR DO BEGIN   (* TRUNCATE IF NEEDED *)
      WRITE (INPUT↑);
```

```
      GET (INPUT);

      END;
   END;  (* READSTRING *)

(*   WRITESTRING WRITES A STRING UP TO THE FIRST BLANK              *)
(*   CHARACTER.                                                     *)

PROCEDURE WRITESTRING (S:STRING);
VAR
   I    : INTEGER;      (* STRING INDEX *)
   DONE : BOOLEAN;      (* FINISHED FLAG *)
BEGIN
   I := 1;
   DONE := FALSE;
   WHILE (NOT DONE) AND (I <= MAXSTRING) DO
      IF S[I] <> BLANK THEN BEGIN
         WRITE (S[I]);
         I := I + 1;
         END
      ELSE DONE := TRUE;
   END;  (* WRITESTRING *)

(*----------------------------------------------------------------*)
(*   PROCEDURE READLIST READS A LIST FROM THE INPUT FILE           *)
(*   AND CREATES ITS INTERNAL REPRESENTATION.  UPON RETURN,        *)
(*   L POINTS TO THE FIRST NODE.                                   *)
(*----------------------------------------------------------------*)

PROCEDURE READLIST (VAR L:LIST);
VAR
   PREV,TEMP : NODEPOINTER;     (* TEMPORARY LIST POINTERS *)
BEGIN
   NEW (L);                (* ALLOCATE A NEW NODE *)
   L↑.HEAD := NIL;         (* AND INITIALIZE IT *)
   L↑.TAIL := NIL;
   IF NEXTINPUT IN STRINGCHAR THEN          (* ATOM *)
      READSTRING (L↑.NAME)
   ELSE IF NEXTINPUT = LPAREN THEN BEGIN    (* LIST *)
      SKIPIT;
      L↑.NAME := LISTFLAG;
      IF NEXTINPUT <> RPAREN THEN READLIST (L↑.HEAD);
      PREV := L;
      WHILE NEXTINPUT <> RPAREN DO BEGIN
         NEW (TEMP);
         TEMP↑.NAME := LISTFLAG;
         PREV↑.TAIL := TEMP;
         READLIST (TEMP↑.HEAD);
         PREV := TEMP;
         END;  (* WHILE *)
      SKIPIT;
      PREV↑.TAIL := NIL;
      END;
   END;  (* READLIST *)
```

```
(*-------------------------------------------------------------*)
(*   PROCEDURE WRITELIST FORMATS AND OUTPUTS THE LIST          *)
(*   WHOSE FIRST NODE IS POINTED TO BY L.                      *)
(*-------------------------------------------------------------*)

PROCEDURE WRITELIST (L:LIST);
VAR
   TEMP : NODEPOINTER;      (* TEMPORARY LIST POINTER *)
BEGIN
   IF L↑.NAME <> LISTFLAG THEN      (* ATOM *)
      WRITESTRING (L↑.NAME)
   ELSE BEGIN                       (* LIST *)
      WRITE (LPAREN);
      IF L↑.HEAD <> NIL THEN WRITELIST (L↑.HEAD);
      TEMP := L↑.TAIL;
      WHILE TEMP <> NIL DO BEGIN
         WRITE (BLANK);
         WRITELIST (TEMP↑.HEAD);
         TEMP := TEMP↑.TAIL;
         END;
      WRITE (RPAREN);
      END;
   END;  (* WRITELIST *)


(*-------------------------------------------------------------*)
(*   MAIN PROCEDURE                                            *)
(*-------------------------------------------------------------*)

BEGIN  (* MAIN *)
   WRITELN (_1_, _TEST OF READLIST AND WRITELIST_);
   WRITELN;

   (* READ AND WRITE LISTS UNTIL END OF FILE *)

   WHILE NOT ENDFILE DO BEGIN

      WRITE (_ _, _READLIST  => _);
      READLIST (LISTEXPR);
      WRITELN;

      WRITE (_ _, _WRITELIST => _);
      WRITELIST (LISTEXPR);
      WRITELN;

      WRITELN;
      READLN;
      END;  (* WHILE *)

   WRITELN (_ _, _*** END OF INPUT ***_);
   END.
```