STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AI-84

MLISP USERS' MANUAL

BY

DAVID CANFIELD SMITH

JANUARY 1969

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AI-84

January, 1969

MLISP USERS' MANUAL

by

David Canfield Smith

Computer Science Department
School of Humanities and Sciences
Stanford University

ABSTRACT: MLISP is a LISP pre-processor designed to facilitate the writing, use, and understanding of LISP programs. This is accomplished through parentheses reduction, comments, introduction of a more visual flow of control with block structure and mnemonic key words, and language redundancy. In addition, some "meta-constructs" are introduced to increase the power of the language.

This research was supported by Grant PHS MH 066-45-07 and (in part) by the Advanced Research Projects Agency, Department of Defense (SD-183).

INDEX

I. INTRODUCTION

MLISP is a LISP (Ref. 2) pre-processor written at Stanford University by Horace Enea for the IBM 360/67. "Pre-processor" means the MLISP programs are translated directly into LISP s-expressions to form a valid LISP program, which is then passed to the LISP interpreter or compiler. The author has implemented MLISP on the PDP-6/10 and has added a few auxiliary features, among them compiled object programs, floating point numbers, a new FOR loop, improved error messages and recovery, and additional string manipulation facilities.

As its name implies, MLISP is a "meta-LISP" language; MLISP programs may be viewed as a superstructure over the underlying LISP processor. All of the underlying LISP functions are available to MLISP. The purpose of having such a superstructure is to improve the readability and writeability of LISP, long (in)famous for its obscurity. As LISP is one of the most powerful symbol manipulative and list processing languages, it seems appropriate to attempt to facilitate the use of it.

Anyone who has attempted to understand a large LISP program written by another programmer (or even by himself a month earlier) quickly becomes aware of several features:

(a) The flow of control is very difficult to follow. Since comments are not permitted, the programmer is completely unable to include written assistence.

(b) An inordinate amount of time must be spent balancing parentheses. It is frequently a non-trivial task just to determine which expressions belong with what other expressions.

(c) The notation of LISP is far from the most natural or mnemonic for a language, making the understanding of most routines overly difficult.

MLISP was designed with these difficulties in mind. The syntax reduces the number of parentheses, permits comments, and clarifies the flow of control. A more mnemonic and natural notation is introduced. Some "meta-constructs" are added to simplify the formation of complex expressions. Strings and string manipulation features, particularly useful for input/output, are included. In addition, a substantial amount of redundancy has been built into the language. This usually facilitates the writing of routines, permitting the user to choose the most natural from a variety of formulations.

MLISP is now running on the Stanford PDP-10. It currently translates at a speed of around 1000 lines per minute in a time-shared environment. The MLISP translator is itself a compiled LISP program. Some effort has been taken to keep the translator as machine independent as possible; in theory MLISP could be implemented on any machine with a working LISP system with only minor changes.

The rest of the introduction will be devoted to supplying the information necessary to get an MLISP program running on the Stanford PDP-10. Part II of this manual will present the syntax of MLISP; Part III, the semantics of the language; Part IV, some special constructs and data structures; and Part V, a complete MLISP program.

A. Loading Sequence

There are two versions of MLISP, both residing on the system area of the disk:

MLISP - a core image containing LISP and the compiled MLISP translator.

MLISPC - a core image containing the LISP compiler in addition to the above.

These programs may be loaded by typing:

R MLISP <core size>

or R MLISPC <core size> . Minimum required core for MLISP is _____, and for MLISPC _____.

B. Translator Commands

After his MLISP program has been translated, a user may specify that the LISP s-expressions are to be (1) executed, (2) compiled, or (3) output to a selected device in a format suitable for later compilation or execution. The supervisory function named (you guessed it) MLISP provides the means for accomplishing these alternatives. Once the user has loaded the system by typing one of the two commands above, he may begin translation of his MLISP program by typing one of the following alternatives:

(1) (MLISP)

(2) (MLISP <input filename>)

(3) (MLISP <input filename> <output filename> T)

(4) (MLISP <input filename> <output filename> NIL)

These will now be explained in detail.

(1) (MLISP) - No arguments. The translator assumes an input file has already been set up with the LISP "INPUT" function. The translator does an (INC T), translates, and then executes the translated program (a "compile-

and-go" procedure). (See Quam's STANFORD LISP manual, ref. 3, for explanation of the input/output functions, DEFPROP, etc.)

(2) (MLISP <input filename>) - Same as above except that an (INPUT DSK: <filename>) is automatically executed before the (INC T). The special case (MLISP TTY) suppresses the (INC T) and will accept a program from the teletype.

(3) (MLISP <input filename> <output filename> T) - Both an (INPUT DSK: <input filename>) and an (OUTPUT DSK: <output filename>) are automatically executed, and all output except error messages and function names goes to the disk. The third argument, T, specifies that the translated functions are to be compiled; the LAP code goes to the selected output file. Note: with this alternative all SPECIAL variables must be declared before they are used in a function. See Ref. 3 for more information on SPECIAL variables. The MLISPC version must be used for this alternative.

(4) (MLISP <input filename> <output filename> NIL) - (INPUT DSK: <input filename>) and (OUTPUT DSK : <output filename>) are executed to set up input and output files on the disk. With this alternative, however, instead of the functions being compiled as they are defined, they are GRINDEF'ed onto the output file. GRINDEF is a LISP function which prints function names and definitions onto the output file in a special form:

(DEFPROP <function name> <function definition> <indicator>)

which is suitable either for later compilation by the LISP compiler or for input into the LISP interpreter. This is frequently useful if the user desires a "clean" system; he can read his DEFPROP'ed function definitions into a fresh LISP system (i.e., without the MLISP translator et. al.) thus making maximum use of core storage. The MLISP (as opposed to MLISPC) core image may be used with this alternative.

C. Miscellaneous

It is sometimes desirable for the user to call the MLISP translator under program control. This is made possible by the special function MTRANS; calling MTRANS has the following effect:

(a) An MLISP program is read from whatever input device has been specified at the moment. The first character in the buffer should be the first character in the program. See Part II for the definition of a syntactic MLISP program.

(b) The resulting LISP program will be returned as the value of the function.

Note that the function MLISP should not be called under program control, since it has several side effects which are generally undesirable in a program. MTRANS has no side effects. It requires no arguments.

COMMENTS: Any sequence of characters enclosed by percent signs (%) is taken to be a comment and is ignored by the translator. The value of comments should be obvious: in general the clarity of a program is directly proportional to the number of comments.

There are several MLISP-defined atoms to which the user may refer:

ATOM	VALUE
F	NIL
BLANK	<Blank Space>
RPAR	)
LPAR	(
DOLLAR	$
COMMA	,
SLASH	/
EQSIGN	=
STAR	*

ATOM	VALUE
COLON	:
PLUSS	+
DASH	-
DBQUOTE	"
PERIOD	.
PERCENT	%
CR	<Carriage Return>
LF	<Line Feed>
FF	<Form Feed>
LARROW	←
ALTMODE	<Altmode>

These facilitate the handling of special characters.

II. SYNTAX

The complete MLISP syntax follows. The LISP equivalents (into which the syntatic parts are translated) are listed in Part III. The meta-symbols [ ] bracket optional elements which may be repeated zero or more times. The curly brackets { } enclose alternative elements.

<program> ::= <expression> .

<expression> ::= <simple expression> [<infix operator> <simple expression>]

<simple expression> ::= <block>
| <conditional expression>
| <FOR expression>
| <UNTIL expression>
| <WHILE expression>
| <list expression>
| <string>
| <prefix> <simple expression>
| (<expression>)
| <function call>
| <function definition>
| <assignment expression>
| '<s-expression>
| <number>
| <empty>

<block> ::= BEGIN [<declaration>;] [<expression>;] END

<declaration> ::= NEW <identifier list>
SPECIAL <identifier list>

<identifier list> ::= <identifier> [, <identifier>]

| <empty>

<conditional expression> ::= IF <expression> THEN <expression> [ELSE <expression>]

<FOR expression> ::= FOR <identifier> {IN / ON} <expression> {DO / COLLECT} <expression> [UNTIL <expression>]

|FOR <identifier> {:=} <expression> TO <expression> [BY <expression>] {DO / COLLECT} <expression> [UNTIL <expression>]

<UNTIL expression> ::= {DO / COLLECT} <expression> UNTIL <expression>

<WHILE expression> ::= WHILE <expression> {DO / COLLECT} <expression>

<list expression> ::= < <arglist> >

<arglist> ::= <argument> [, <argument>]

<argument> ::= <expression>

|# <identifier list> : <expression>

<string> ::= "<any sequence of characters - possibly none - except ">"

<function call> ::= <identifier> (<arglist>)

<function definition> ::= <identifier> {:=} {# / *} <identifier list> : <expression>

<assignment expression> ::= <identifier> {:=} <expression>

<infix operator> ::= *|**|-|/|+|&|@|=

| <identifier>

<prefix> := +|-|NOT|NULL|ATOM

<identifier> ::= <letter> [<any sequence of letters or digits>]

<letter> ::= A|B|C|...|X|Y|Z|!

<digit> ::= 1|2|3|4|5|6|7|8|9|ø

<number> ::= <digit> [<digit>]

| <digit> [<digit>] . [<digit>]

III. SEMANTICS

This section will serve two purposes:

(a) Present the LISP expressions into which the MLISP constructs are translated;

(b) Explain the "meta-constructs" and how they are evaluated.

In the following it will be assumed that the user has a working knowledge of LISP. McCarthy (Ref. 2) provides a reference manual for the standard LISP functions, while Weissman (Ref. 4) furnishes a good tutorial. In the cases below where the meaning of MLISP constructs is given by their LISP equivalents, no further explanation will be presented. Any non-standard LISP elements will be fully explained; Quam's users' manual for Stanford LISP (Ref. 3) may be consulted for additional information.

A. <program> ::= <expression>

In general MLISP programs are enclosed within a BEGIN-END pair; that is, a <program> is usually a <block> (cf. C below). This permits more than one MLISP expression to be translated at one time. (See Part V for a sample program.)

B. <expression> ::= <simple expression> [<infix operator> <simple expression>]

An expression is a simple expression (paragraphs C-N) optionally followed by any number of infix operator-simple expression pairs. There is no hierarchy of operators; association is to the right. For example, A + B - C will be translated as (PLUS A (DIFFERENCE B C)). Again, A CONS B CONS NIL will become (CONS A (CONS B NIL)).

Examples of expressions are (note that "⇒" means "is translated to"):

```
A
16
A := X ** 2 ⇒ (SETQ A (EXPT X 2))
A := B := C ⇒ (SETQ A (SETQ B C))
"THIS IS A STRING"
(A + B) - C ⇒ (DIFFERENCE (PLUS A B) C)
(((A + B)) - C)⇒ (DIFFERENCE (PLUS A B) C)
PLUS (A,B) - C ⇒ (DIFFERENCE (PLUS A B) C)
DIFFERENCE (A + B,C) ⇒ (DIFFERENCE (PLUS A B) C)
FOR I IN L COLLECT CAR(I)
WHILE NOT (A EQ B) DO A := READ( )
BEGIN A := B END CONS C ⇒ (CONS (PROG NIL (SETQ A B)) C)
```

Note that since an operator may be an arbitrary identifier, virtually any user-defined function of two arguments may be used as an infix operator. Parentheses may always be used to alter the association of arguments; for example:

```
(A CONS B) CONS C ⇔ (CONS (CONS A B) C).
```

The following paragraphs (C-N) explain simple expressions.

C. <block> ::= BEGIN [<declaration>;] [<expression>;] END
<declaration> ::= SPECIAL <identifier list>
| NEW <identifier list>

BEGIN-END blocks are translated into PROG's. Variables declared using NEW become the PROG variables; all PROG variables are initialized to NIL. The scope of variables so defined is the scope of the PROG, i.e., until the matching END. If there are no NEW declarations, NIL is used for the PROG variables. Expressions within the block are separated by semicolons. As with PROG's, a value may be returned for the block via the RETURN function. Again, lables may be transferred to via the

GO function; e.g., GO(L). Labels may be formed by following the label identifier immediately with a semicolon; e.g., L; . (However the iteration expressions described below (E-H) are to be much recommended over labels and GO transfers.) An example of a block and its PROG translation follows.

```
BEGIN                          (PROG
  NEW A,B,C; NEW X;              (A B C X)
  A := READ ( );                 (SETQ A (READ))
  X := (B + C) / A;              (SETQ X (QUOTIENT (PLUS B C) A))
  RETURN (X);                    (RETURN X)
END                            )
```

SPECIAL declarations may be mixed in with NEW declarations in any manner. All NEW and SPECIAL declarations, however, must appear at the start of a block, before any executable expression.

Variables declared SPECIAL have the following effect: (1) The indicator "SPECIAL" is immediately put on their property lists. (2) If the translated MLISP program is to be GRINDEF'ed (i.e., printed) onto an output file (cf. Part I), all SPECIAL variables are collected into a list and GRINDEF'ed onto the head of the file, before any function definitions. Thus all SPECIAL variables are guaranteed to be defined before they are used, no matter where they appear in the MLISP program. This enables the user to mark variables SPECIAL wherever in the program it is convenient to do so. Note that SPECIAL variables are <u>not</u> made PROG variables; that is done only with NEW declarations. However, NEW variables may also be declared SPECIAL.

See Ref. 3 for further discussion of SPECIAL variables.

D. <conditional expression> ::= IF <expression>
THEN <expression> [ELSE <expression>]

Without the optional ELSE term,

```
IF e1 THEN e2
```

is translated into

```
(COND (e1 e2))
```

where e1, e2 are any arbitrary expressions. If the ELSE term is present,

```
IF e1 THEN e2 ELSE e3
```

becomes

```
(COND (e1 e2) (T e3)).
```

Conditionals are automatically optimized in the sense that

```
IF e1 THEN e2 ELSE
IF e3 THEN e4 ELSE e5
```

becomes

```
(COND (e1 e2) (e3 e4) (T e5))
```

rather than

```
(COND (e1 e2) (T (COND (e3 e4) (T e5)))).
```

The nested form

```
IF e1 THEN
  IF e2 THEN e3 ELSE e4
ELSE e5
```

is permitted and becomes

```
(COND (e1 (COND (e2 e3) (T e4))) (T e5)).
```

Note that all of the predicates in a COND may evaluate to NIL, in which case NIL is returned as the value of the conditional. (cf. Ref. 3 for further details).

E. <FOR expression> ::= FOR <identifier> $\{^{IN}_{ON}\}$ <expression> $\{^{DO}_{COLLECT}\}$
<expression> [UNTIL <expression>]

This is the first of the MLISP "meta-constructs," all of which are iteration expressions closely resembling ALGOL statements. Its evaluation will be informally illustrated through an example:

For I IN e1 DO e2 UNTIL e3.

Evaluation:

```
        V ← NIL          (NOTE:  V,L,X are temporary variables)
        L ← value (e1)   (NOTE:  e1 should evaluate to a list)
A:      if NULL (L) then PROG2( I ← NIL, RETURN (V))
*       I ← CAR (L)
        L ← CDR (L)
**      V ← value (e2)
***     X ← value (e3)
****    if X ≠ NIL then RETURN (V)
        GO TO A
```

It may be seen that the control variable I "steps through" the list L which is the value of e1; i.e., I becomes successively CAR(L), CADR(L),... . If the UNTIL condition is omitted, then lines *** and **** are omitted. If ON rather than IN is specified, then line * is replaced by I ← L; i.e., I becomes successively L, CDR (L), CDDR (L), etc. If COLLECT rather than DO is specified, then line ** becomes V ← V APPEND value (e2).

Expression e1 is evaluated only once. Expressions e2 and e3 (if e3 is present) are evaluated each time through the loop. Both may be functions of the control variable (I in the above example).

The use of COLLECT is one of the most powerful features of MLISP. With it, lists of arbitrary complexity may be assembled with little effort (see F,G,H). For example, the value of

```
FOR I IN '((GOOD . 16) BAD (GOOD . 58)) COLLECT
    IF I EQ 'BAD THEN NIL ELSE <CDR (I)>
```

is (16 58). See WHILE-COLLECT (p.) for another example. Note that <CDR(I)> is used rather than CDR (I); <CDR (I)> is translated into (LIST (CDR I)), so that the APPEND'ed list will be a list of the CDR's, as desired.

It should be mentioned that there is never any danger of MLISP confusing its temporary variables with user-defined variables. The temporary variables specified above were just examples; in reality MLISP uses special character strings which are illegal identifiers in the MLISP language.

F. <FOR expression> ::= FOR <identifier> {:=, ←} <expression> TO <expression>
[BY <expression>]
{DO, COLLECT} <expression> [UNTIL <expression>]

This is similar to (D) above. Again by example:

FOR I := e1 to e2 BY e3 DO e4 UNTIL e5

Evaluation:

```
          V ← NIL                    (NOTE:  Again V, UPPER, INC, X are tempories)
          I ← value (e1)
          UPPER ← value (e2)
*         INC ← value (e3)
     A:   if I > UPPER then PROG2 (I ← NIL, RETURN (V))
**        V ← value (e4)
***       X ← value (e5)
****      if X ≠ NIL then RETURN (V)
          I ← I + INC
          go to A
```

Again, if the UNTIL condition is omitted, lines *** and **** are left out. If the BY specification of the increment is omitted, line * becomes INC ← 1 . If COLLECT rather than DO is specified, line ** becomes V ← V APPEND value (e4).

Example: FOR I := 1 TO 10 COLLECT <PRINT (I)> UNTIL (I/4) = 1

would print onto the output device

```
1
2
3
4
```

and return (1 2 3 4) as its value.

G. <UNTIL expression> ::= $\{^{\text{DO}}_{\text{COLLECT}}\}$ <expression> UNTIL <expression>

For the case DO e1 UNTIL e2, evaluation is as follows:

```
*    A:  V ← value (e1)
         X ← value (e2)
         if X ≠ NIL then RETURN (V)
         go to A
```

Example: DO M ← READ ( ) UNTIL M EQ 'STOP would read successive s-expressions from the input device until "STOP" was read. The value of the loop would then be "STOP" also, since M ← READ ( ) is translated into (SETQ M (READ)) which has as its value the last thing read.

If COLLECT rather than DO is specified, line * becomes V ← V APPEND value (e1).

H. <WHILE expression> ::= WHILE <expression> $\{^{\text{DO}}_{\text{COLLECT}}\}$ <expression>

For the case WHILE e1 DO e2, evaluation is as follows:

```
         V ← NIL
    A:   X ← value (e1)
         if NOT X then RETURN (V)
*        V ← value (e2)
         go to A
```

If COLLECT rather than DO is specified, line * becomes V ← V APPEND value (e2).

The only difference between the UNTIL expression and the WHILE expression is that in the WHILE expression the test for the terminating condition is made first, while in the UNTIL expression it is made second.

As an example of the power of the COLLECT variation of these expressions, consider:

WHILE NOT ((A ← READ ( )) EQ 'STOP) COLLECT <CADR(A)>

If the input file contains a sequence of lists in the form (DEFPROP <function name> <lambda definition> <indicator>), which is a standard form for function definitions, then the above WHILE expression would assemble a list of all the function names in the file until the atom "STOP" was encountered. Concise statements of complex expressions such as this is one of the primary purposes of MLISP.

I. <list expression> ::= < <arglist> >

This construct is the MLISP equivalent of the LISP LIST expression; <A,B,C> is translated into (LIST A B C), < > into (LIST) = NIL, etc. Note that LIST(A,B,C) is also translated into (LIST A B C); <A,B,C> is the shorter notation. A CONS B CONS C CONS NIL would also have the same effect. As mentioned in the introduction, it is a general property of MLISP that most constructs may be expressed in several ways. This redundancy is deliberate; it facilitates the writing of programs by relieving the user from finding the <u>one</u> correct expression.

Arguments inside the list brackets may be any arbitrary expression (cf. J below). This is also true of function calls (cf. L below).

Example:

<A CONS B, <MIN (3 + I, J) , NIL>, LAST (J)> goes to

```
(LIST  (CONS A B)
       (LIST (MIN (PLUS 3 I)J) NIL)
       (LAST J)
       )
```

J. <arglist> ::= <argument> [, <argument>]

<argument> ::= <expression>
| # <identifier list> <expression>

An argument list is one or more expressions or functional arguments separated by commas. Since an expression may be empty, an argument list may be empty.

A special case is made of the functional argument in order to permit arbitrary LAMBDA expressions to be included in an argument list. The sharp symbol (#) is translated into "LAMBDA", the identifier list into the LAMBDA-bound variables, and the expression into the LAMBDA expression. Example:

FOO (# A,B: IF A EQ B THEN T ELSE A) goes to
(FOO (LAMBDA (A B) (COND ((EQ A B) T) (T A)))).

Functional arguments which are passed by name may be treated like any other function call (see L below). Example: FUNCTION (CAR). Here CAR is a function which is being passed as a functional argument by name. See the LISP 1.5 Manual (Ref. 2) for further information on functional arguments.

K. <string> ::= "<any sequence of characters except " > "

Strings are a special MLISP data form (see Part IV). Strings are stored as a list of the individual characters in the string; for example, "THIS IS A STRING!" is stored as (T H I S /_ I S /_ A /_ S T R I N G !). (The symbols /_ represent the blank character.) With the MLISP-defined atom DBQUOTE -- defined to be "--and the MLISP string function CAT (see Part IV), double quotes may be inserted into strings; e.g., "THIS STRING CONTAINS" CAT DBQUOTE CAT" A DOUBLE QUOTE".

L. <function call> ::= <identifier> (<arglist>)

In MLISP the function names are brought outside the parentheses, and the arguments, if there are more than one, are separated by commas. Examples:

CAR (A)	is translated to	(CAR A)
PLUS (1,2,3)		(PLUS 1 2 3)
FUNCTION(CAR)		(FUNCTION CAR)
APPEND (A CONS B,C CONS NIL)		(APPEND(CONS A B) (CONS C NIL))

Functions having exactly two arguments may alternatively be used as infix operators. For example, CONS(A,B) and A CONS B are equivalent, both being translated to (CONS A B).

M. <function definition> ::= <identifier> $\{\overset{:=}{\leftarrow}\}$ $\{\overset{\#}{*}\}$<identifier list>:<expression>

This is the mechanism for defining functions. The identifier is the function name. If the sharp (#) is used, the function will be an EXPR; if the star (*) is used, it will be an FEXPR. In either case the translation is "LAMBDA." The identifier list specifies the LAMBDA-bound variables, and the expression is the LAMBDA expression. Examples:

```
MIN := # A,B:  IF A LESSP B THEN A ELSE B;
SMALLEST := *L:
     % FINDS THE SMALLEST NUMBER AMONG ITS ARGUMENTS %
     BEGIN NEW X,Y,I;
       IF NULL(L) THEN RETURN (NIL);
       X := EVAL (CAR (L)); L← CDR(L);
       FOR I IN L DO
           IF (Y ← EVAL(I)) LESSP X THEN X ← Y;
       RETURN (X);
     END;
```

The first function is an EXPR with two arguments that returns the smaller of the two. The second function is an FEXPR that takes any number of arguments and returns the samllest one. Note that in the second function, since FEXPR's do not evaluate their arguments, the arguments have to be evaluated explicitly by calls on EVAL.

The second function illustrates several features of MLISP:

(a) The phrase between percent signs (%) is a comment.

(b) The BEGIN-END block specifies it to be a PROG, with PROG variables X, Y and I.

(c) The RETURN expression is used to return a value for the function.

(d) The FOR loop steps through the list L comparing each element to the current smallest element and making the appropriate change.

(e) `X := Y;`

`X ← Y;`

`X SETQ Y;`

`SETQ (X,Y);`

These are all equivalent, and may be used interchangeably ; all are translated to (SETQ X Y).

When a function is defined, one of three things occurs. Either:

(1) The function definition is immediately added to the property list of the function name, under the appropriate indicator (EXPR or FEXPR).

(2) The function definition is GRINDEF'ed onto the selected output file (cf. Part I), in the correct form for input into the compiler.

(3) The function is immediately compiled onto the output device without further ado.

Alternative (1) occurs if the second and third arguments to the MLISP function (cf. Part I) are absent. Alternative (2) occurs if the third argument is NIL; alternative (3), if it is T. MLISPC must be used for alternative (3). In any case, after the function has been defined, its name is printed onto the teletype.

Function definitions are not considered to be executable expressions at runtime, and, after one of the above three alternatives has been executed, NIL is returned as their translation. After the entire MLISP program has been translated, all non-NIL expressions are collected into a PROG and a new function named RESTART is defined, with the PROG as its definition. For this function, also, one of the above three alternatives is executed. The RESTART function provides a convenient means for re-entering MLISP programs. Typing (RESTART) will begin execution of the program, starting with the first executable expression. An example may clarify this. Suppose the MLISP program is

```
BEGIN          NEW X;
     MIN := # A,B: IF A LESSP B THEN A ELSE B;
     X ← READ( ); PRINT(MIN(X, 0.5));
END
```

This creates the following property lists:

```
MIN            EXPR (LAMBDA (A B) (COND ((LESSP A B) A) (T B)))
RESTART        EXPR (LAMBDA NIL (PROG (X)
                                      (SETQ X (READ))
                                      (PRINT (MIN X 0.5))))
```

N. <assignment expression> ::= <identifier> $\{\overset{:=}{\leftarrow}\}$ <expression>

Either the colon-equal or left arrow symbol may be used in assignment expressions. Both translate into SETQ. The following all translate into (SETQ A (PLUS B C)):

```
A := B + C
A ← B + C
A SETQ B + C
SETQ (A, B + C).
```

The author prefers the left arrow as being the clearest.

Multiple assignments are permitted, since assignments are themselves expressions:

```
X ← Y ← Z ← ∅
```

is translated to

```
(SETQ X (SETQ Y (SETQ Z ∅))).
```

The value of an assignment expression is, as in LISP, the value of the expression on the right; this should be clear from the LISP translation.

0. <infix operator> ::= *|**|-|/|+|&|@|=|<identifier>

The above symbols all have pre-defined meanings in MLISP, to provide a concise short-hand for certain commonly-used LISP functions. The meanings are:

=	EQUAL	
*	TIMES	
**	EXPT	(only integer exponents are allowed)
-	DIFFERENCE	(MINUS if used as a unary (prefix) operator)
/	QUOTIENT	
+	PLUS	
&	AND	
@	APPEND	

Thus A TIMES B and A*B are equivalent, for example. Also any other function of two arguments may be used as an infix operator. Thus the MIN function defined in (M) above may be called either by MIN (A,B) or A MIN B. This helps to reduce further the number of parentheses and frequently improves the readability of a program.

There are two other MLISP-defined operators: NEQ and NEQUAL. A NEQ B is the same as NOT (A EQ B), and A NEQUAL B is equivalent to NOT (A=B).

P. <prefix> ::= +|-|NOT|NULL|ATOM

These are unary operators having the following translations:

+X	⇒	X
-X	⇒	(MINUS X)
NOT X	⇒	(NOT X)
NULL X	⇒	(NULL X)
ATOM X	⇒	(ATOM X)

The last three may also be called with parentheses around their arguments, like any other function: NOT(X), NULL(X), ATOM(X). Using them as a prefix operator, however, helps to reduce the number of parentheses in the program.

Q. <identifier> ::= <letter> [<any sequence of letters or digits>]

All identifiers must begin with a letter. Letters are considered to be any of the 26 letters in the alphabet or the exclamation point (!) symbol. The exclamation point is included to enable the user to form unusual identifiers; e.g., !SYSTEMVAR1. Examples of identifiers are A, AVERYLONGSTRING , M32BJE1, HELP!, GØØØ1. Limits as to the length of printnames, etc. are those of the underlying LISP processor (see Ref. 3).

R. <number> ::= <digit> [<digit>]
| <digit> [<digit>] . [<digit>]

Two types of numbers are permitted in MLISP, real and integer. Exponents may be specified via the ** operator; e.g., 1Ø **2. The base of all numbers in MLISP is 1Ø. Examples of numbers are

1

1Ø

1Ø.ØØ5

-9876.54321**6

This last get translated into (MINUS (EXPT 9876.54321 6)).
Recall that only integer exponents are allowed, unless the user wants to define his own exponent function.

Finally, s-expressions may be quoted using the single quote ('). Whereas the LISP user writes (QUOTE <s-expression>), the MLISP user writes '<s-expression>. Examples:

'A	⇒	(QUOTE A)
'(A B C)	⇒	(QUOTE (A B C))
'(A . B)	⇒	(QUOTE(A. B))
'(A (B . C) (E F) .G)	⇒	(QUOTE (A (B . C) (E F) . G))

IV SPECIAL CONSTRUCTS - DATA STRUCTURES

Several of the special MLISP constructs - the iteration expressions - have already been described. These were designed to simplify the writing of complex expressions, in addition to eliminating GO transfers and further clarifying the flow of control. It should be evident by now that none of the recursive power of LISP has been sacrificed; rather, MLISP extends this power, if we define the "power" of a language to be the ease with which complex routines may be written.

The data structures of MLISP are, of course, those of LISP: symbolic "atoms", numbers,and list structures. In addition, MLISP defines strings--sequences of characters enclosed by double quotes (")-- as an input/output oriented data form. Strings are stored internally as lists of the characters in the strings. Any LISP-defined character except double quote may appear in a string; double quote may be added with the MLISP-defined atom DBQUOTE. Certain string manipulative functions are defined in MLISP. These functions are CAT, PRINSTR, STR, STRING, and SUBSTR.

CAT - Concatenation, a function of two arguments which may be either strings or atoms. If they are both strings, the value is their concatenation. If one or both are atoms, the printname of the atom is first made into a string and then concatenated.

PRINSTR - A function of one argument, a string. It prints the argument exactly as it appears onto the current output device, ending the line with a carriage return. Value is NIL.

STR - A function of one argument, an atom. Value is a string of the atom's print name.

STRING - A function of one argument, a list. Value is a string of all the characters in the list, including parentheses and blanks.

SUBSTR - A function of three arguments. The first argument is a string. Value is a substring of the string. The second argument is the starting position of the substring counting from one. The third argument is the number of characters to be extracted.

These functions will be illustrated by examples:

```
"THIS  IS  A  "  CAT  "STRING" = "THIS IS A STRING"
"THIS  IS  A  " CAT  'STRING = "THIS IS A STRING"
STR ('STRING) = "STRING"
STRING ('(A (B C) D)) = "(A (B C) D)"
SUBSTR ("THIS IS A STRING", 6, 4) = "IS A"
PRINTSTR ("STRING") will print STRING
PRINT ("STRING") will print (S T R I N G)
PRINTSTR ("STRING" CAT DBQUOTE) will print STRING"
```

These functions provide the mechanism necessary to create, add to, fragment, and print strings. This mechanism allows the user complete control of his output statements.

There are two other MLISP-defined functions which are string related but which apply to arbitrary lists as well: PRELIST and SUFLIST. Both take two arguments. The first is a list or string; the second is an integer. Their values will be illustrated by example and by definition.

```
PRELIST ('(A B C D E), 3) = (A B C)
PRELIST ("THIS IS A STRING", 9) = "THIS IS A"
SUFLIST ('(A B C D E), 3) = (D E)
SUFLIST ("THIS IS A STRING", 9) = " STRING"
```

For all lists L and for all integers N,

PRELIST (L,N) @ SUFLIST (L,N) = L.

PRELIST and SUFLIST are designed to facilitate the manipulation of lists. Their MLISP definitions are:

```
PRELIST := #L,N:
   IF (N=∅) OR NULL(L) THEN NIL
   ELSE CAR(L) CONS PRELIST(CDR(L), N-1);
SUFLIST := #L,N:
   IF (N=∅) OR NULL (L) THEN L
   ELSE SUFLIST (CDR(L), N-1);
```

To further simplify list handling, nine "field" functions are defined in MLISP: F1 through F9. F1 returns the first element in the list which is its argument, F2 the second element, etc. Examples:

```
F1('(A B C D E)) = A
F2('(A B C D E)) = B
F5('(A B C D E)) = E
F6('(A B C D E)) = undefined.
```

These functions were added on the theory that in manipulating the, say, sixth element in a list, it is clearer to write F6(I) rather than CADR(CDDDDR(L)).

```
BEGIN  NEW EXPRLIST,FEXPRLIST,LEN;      % GLOBAL DECLARATIONS %
       SPECIAL EXPRLIST,FEXPRLIST,LEN,L;

%      THERE ARE MANY WAYS TO WRITE THE FUNCTION 'REVERSE,'
 WHICH REVERSES THE TOP LEVEL OF A LIST, IN MLISP.  THE FOLLOWING
 PROGRAM PRESENTS SOME OF THOSE WAYS AND THEN EVALUATES THEM ON TEST LISTS,
 IT PRINTS OUT THE TIME REQUIRED BY EACH FUNCTION, SO THAT THEY
 MAY BE RANK-ORDERED IN EFFICIENCY. %

%##########################################################################%
%##########          DEFINE ALL THE REVERSE FUNCTIONS          ##########%
%##########################################################################%

REVERSE1 := #L:   REVERSE1A(L,NIL);
   % THIS IS AN EXPR,  USES IF-THEN-ELSE. %

REVERSE1A := #L,LL:
   % THE REVERSE OF 'L' IS BUILT UP IN THE SECOND ARGUMENT 'LL'. %
   IF NULL L THEN LL ELSE REVERSE1A(CDR(L),CAR(L) CONS LL);


REVERSE2 := #L:
   % THIS IS AN EXPR,  USES IF-THEN-ELSE AND A RECURSIVE CALL ON ITSELF. %
   % IN THIS VERSION, THE REVERSE OF THE REST OF THE LIST 'L' IS
      APPENDED (@) TO A LIST CONTAINING THE FIRST ELEMENT. %
   IF NULL L THEN NIL ELSE REVERSE2(CDR(L)) @ <CAR(L)>;


REVERSE3 := #L:
   % THIS IS AN FEXPR; L IS AN UNEVALUATED LIST OF THE ACTUAL ARGUMENTS
      PASSED TO REVERSE3.  USES A FOR LOOP. %
   % THE FOR-LOOP STEPS THROUGH THE LIST 'L,' SUCCESSIVELY ASSIGNING
      EACH ELEMENT IN IT TO 'I'. %
   BEGIN  NEW I,V;  SPECIAL I,V;
      % RECALL THAT ALL 'NEW,' I.E. PROG, VARIABLES ARE AUTOMATICALLY
         INITIALIZED TO 'NIL'. %
      FOR I IN L DO V ← I CONS V;
      RETURN(V);
   END;


REVERSE4 := #L:
   % THIS IS AN EXPR,  USES A WHILE LOOP. %
   % WHILE THERE IS STILL SOMETHING LEFT IN 'L,' THE PROG2 IS EXECUTED. %
   BEGIN  NEW V;  SPECIAL V;
      WHILE L DO PROG2(V ← CAR(L) CONS V,L ← CDR(L));
      RETURN(V);
   END;


REVERSE5 := #L:
   % THIS IS AN EXPR,  USES A DO-UNTIL LOOP. %
   % THE PROG2 IS EXECUTED UNTIL 'L' IS NIL. %
   IF NULL L THEN NIL ELSE


   BEGIN  NEW V;  SPECIAL V;
      DO PROG2(V ← CAR(L) CONS V,L ← CDR(L)) UNTIL NULL L;
      RETURN(V);
   END;

REVERSE6 := #L:
   % THIS IS ANOTHER FEXPR;  THE ARGUMENTS ARE NOT EVALUATED. %
   % THIS IS LESS GENERAL THAN THE OTHER FUNCTIONS;  IT WILL NOT WORK
      FOR LISTS OF LENGTH > 100. %
   % HOWEVER, IT IS INCLUDED AS AN EXAMPLE OF A FOR-LOOP USING A
      NUMERICAL INCREMENT. %
   IF NULL L THEN NIL ELSE
   BEGIN  NEW I,V;  SPECIAL I,V;
      FOR I←1 TO 100 DO PROG2(V ← CAR(L) CONS V,L ← CDR(L))
      UNTIL NULL L;      % THIS 'UNTIL' CONDITION STOPS THE FOR-LOOP IF
                           'L' CONTAINED FEWER THAN 100 ELEMENTS. %
      RETURN(V);
   END;

% IT IS IMPORTANT TO NOTE THAT IF ANY FOR-LOOP IS HALTED BY THE 'UNTIL'
   CONDITION BECOMING TRUE, THE CONTROL VARIABLE IS NOT RESET TO 'NIL'.
   THE CONTROL VARIABLE IS RESET IF AND ONLY IF THE FOR-LOOP HALTS
   NORMALLY (I.E. NOT DUE TO THE 'UNTIL' CONDITION).  IN THE ABOVE CASE,
   THIS MEANS THAT WE OBTAIN THE LENGTH OF THE LIST 'L' -- IN THE
   CURRENT VALUE OF 'I' -- AS A SIDE EFFECT OF REVERSING IT.
   MLISP IS SPECIFICALLY DESIGNED TO FACILITATE SUCH 'SIMULTANEOUS'
   PERFORMANCE OF OPERATIONS. %
```

```
REVERSE7 := *L;   REVERSE7A(L,NIL);
   % ANOTHER WAY OF WRITING REVERSE6, %
   % THIS DOES NOT REQUIRE ANY BEGIN-END, NEW DECLARATIONS, OR RETURN
      STATEMENT, %
   % IT ALSO PRESERVES THE LENGTH OF 'L' IN THE GLOBAL VARIABLE 'LEN', %

REVERSE7A := #L,V;
   IF NULL L THEN NIL ELSE
   FOR LEN←1 TO 100 DO DO2(V ← CAR(L) CONS V,L ← CDR(L)) UNTIL NULL L;

DO2 := #A,B;  A;
   % 'DO2' IS LIKE PROG2, EXCEPT THAT DO2 RETURNS THE VALUE OF ITS FIRST
      ARGUMENT, %


%##################################################################%
%################         EXECUTION BEGINS HERE          ##########%
%##################################################################%

   EXPRLIST ← '(REVERSE1 REVERSE2 REVERSE4 REVERSE5);
   FEXPRLIST ← '(REVERSE3 REVERSE6 REVERSE7);

   % READ IN TEST LISTS UNTIL THE ATOM 'STOP' IS ENCOUNTERED %

   PRINTSTR("TYPE A LIST TO BE REVERSED,");
   WHILE (L ← READ()) NEQ 'STOP DO


   BEGIN  NEW FUNCLIST,FUNCNAME,FUNC;  SPECIAL FUNCLIST,FUNCNAME,FUNC;
      TERPRI();                        % SKIP A LINE IN THE OUTPUT %
      PRINTSTR("THE TEST LIST IS " CAT STRING(L));
      % COLLECT TOGETHER THE FUNCTION NAMES APPLIED TO THE TEST LIST %
      FUNCLIST ← FOR FUNCNAME IN EXPRLIST COLLECT <<FUNCNAME,<'QUOTE,L>>>;
      FUNCLIST ← FUNCLIST @ FOR FUNCNAME IN FEXPRLIST COLLECT <FUNCNAME CONS L>;
      LEN ← 0;                         % RESET THE LIST LENGTH, %

      % EVALUATE ALL OF THE FUNCTIONS WITH THE TEST ARGUMENT %
      FOR FUNC IN FUNCLIST DO
      BEGIN  NEW START;
         PRINT(CAR(FUNC));             % PRINT OUT THE FUNCTION NAME %
         START ← TIME();               % 'TIME()' YIELDS THE CURRENT TIME
                                       IN MILLISECONDS %
         PRINT(EVAL(FUNC));            % EVALUATE THE FUNCTION %
         PRINT((TIME()-START)/1000.0); % PRINT THE TIME REQUIRED %
         PRINTSTR("SECS");
         TERPRI();                     % SKIP A LINE IN THE OUTPUT %
      END;

      % THE LENGTH 'LEN' IS OBTAINED FROM REVERSE7, %
      PRINTSTR("THE LENGTH OF THE TEST LIST WAS " CAT LEN);
      TERPRI();  TERPRI();
      PRINTSTR("TYPE ANOTHER LIST TO BE REVERSED, OR 'STOP'");
   END;

% THE LISP PROGRAM, INTO WHICH THIS MLISP PROGRAM IS TRANSLATED, IS
   LISTED ON THE FOLLOWING PAGES.  IT HAS BEEN PRINTED USING A
   STANFORD-WRITTEN FUNCTION CALLED 'GRINDEF,' AN S-EXPRESSION UN-CRUNCHER,
   ON THE PAGES FOLLOWING THAT IS THE COMPLETE TELETYPE LISTING OF A PROGRAM
   RUN. %

END.


(DEFPROP EXPRLIST T SPECIAL)
(DEFPROP FEXPRLIST T SPECIAL)
(DEFPROP LEN T SPECIAL)
(DEFPROP L T SPECIAL)
(DEFPROP I T SPECIAL)
(DEFPROP V T SPECIAL)
(DEFPROP FUNCLIST T SPECIAL)
(DEFPROP FUNCNAME T SPECIAL)
(DEFPROP FUNC T SPECIAL)

(DEFPROP REVERSE1
 (LAMBDA (L) (REVERSE1A L NIL))
EXPR)

(DEFPROP REVERSE1A
 (LAMBDA (L LL) (COND ((NULL L) LL) (T (REVERSE1A (CDR L) (CONS (CAR L) LL)))))
EXPR)

(DEFPROP REVERSE2
 (LAMBDA (L) (COND ((NULL L) NIL) (T (APPEND (REVERSE2 (CDR L)) (LIST (CAR L)))))
EXPR)
```

```
(DEFPROP REVERSE3
 (LAMBDA (L) (PROG (I V) (:FORDO (QUOTE I) T L (QUOTE (SETQ V (CONS I V))) NIL) (RETURN V)))
FEXPR)

(DEFPROP REVERSE4
 (LAMBDA(L)
  (PROG (V) (:WHILEDO (QUOTE L) (QUOTE (PROG2 (SETQ V (CONS (CAR L) V)) (SETQ L (CDR L))))) (RETURN V)))
EXPR)

(DEFPROP REVERSE5
 (LAMBDA(L)
  (COND ((NULL L) NIL)
        (T
         (PROG (V)
               (:DOUNTIL (QUOTE (PROG2 (SETQ V (CONS (CAR L) V)) (SETQ L (CDR L)))) (QUOTE (NULL L)))
               (RETURN V))))))
EXPR)

(DEFPROP REVERSE6
 (LAMBDA(L)
  (COND ((NULL L) NIL)
        (T
         (PROG (I V)
               (:FORLOOP (QUOTE I)
                         (QUOTE (1, 100, 1,))
                         T
                         (QUOTE (PROG2 (SETQ V (CONS (CAR L) V)) (SETQ L (CDR L))))
                         (QUOTE (NULL L)))
               (RETURN V))))))
FEXPR)


(DEFPROP REVERSE7
 (LAMBDA (L) (REVERSE7A L NIL))
FEXPR)

(DEFPROP REVERSE7A
 (LAMBDA(L V)
  (COND ((NULL L) NIL)
        (T
         (:FORLOOP (QUOTE LEN)
                   (QUOTE (1, 100, 1,))
                   T
                   (QUOTE (DO2 (SETQ V (CONS (CAR L) V)) (SETQ L (CDR L))))
                   (QUOTE (NULL L)))))))
EXPR)

(DEFPROP DO2
 (LAMBDA (A B) A)
EXPR)

(DEFPROP RESTART
 (LAMBDA NIL
  (PROG (EXPRLIST FEXPRLIST LEN)
        (SETQ EXPRLIST (QUOTE (REVERSE1 REVERSE2 REVERSE4 REVERSE5)))
        (SETQ FEXPRLIST (QUOTE (REVERSE3 REVERSE6 REVERSE7)))
        (PRINTSTR (QUOTE (T Y P E /  A /  L I S T /  T O /  B E /  R E V E R S E D /,)))
        (:WHILEDO (QUOTE (NEQ (SETQ L (READ)) (QUOTE STOP)))
                  (QUOTE
                   (PROG (FUNCLIST FUNCNAME FUNC)
                         (TERPRI)
                         (PRINTSTR (CAT (QUOTE (T H E /  T E S T /  L I S T /  I S / )) (STRING L)))
                         (SETQ FUNCLIST
                               (:FORLIST (QUOTE FUNCNAME)
                                         T
                                         EXPRLIST
                                         (QUOTE (LIST (LIST FUNCNAME (LIST (QUOTE QUOTE) L))))
                                         NIL))
                         (SETQ FUNCLIST
                               (APPEND FUNCLIST
                                       (:FORLIST (QUOTE FUNCNAME)
                                                 T
                                                 FEXPRLIST
                                                 (QUOTE (LIST (CONS FUNCNAME L)))
                                                 NIL)))
                         (SETQ LEN 2,)
                         (:FORDO (QUOTE FUNC)
                                 T
                                 FUNCLIST
                                 (QUOTE
                                  (PROG (START)
                                        (PRINT (CAR FUNC))
                                        (SETQ START (TIME))
                                        (PRINT (EVAL FUNC))
                                        (PRINT (QUOTIENT (DIFFERENCE (TIME) START) 1000,0))
                                        (PRINTSTR (QUOTE (S E C S)))
```

```
                (TERPRI)))
        NIL)
(PRINTSTR
 (CAT (QUOTE (T H E / L E N G T H / O F / T H E / T E S T / L I S T / W A S / ))
      LEN))
(TERPRI)
(TERPRI)
(PRINTSTR
 (QUOTE
  (T Y
     P
     E
     /
     A
     N
     O
     T
     H
     E
     R
     /
     L
     I
     S
     T
     /
     T
     O
     /
     B
     E
     /
     R
     E
     V
     E
     R
     S
     E
     D
     /,
     /
     O
     R
     /
     ,
     S
     T
     O
     P
     ')))))))))
EXPR)
```

```
.RUN DSK:MLISP

*(MLISP MTEST)


*
REVERSE1
REVERSE1A
REVERSE2
REVERSE3
REVERSE4
REVERSE5
REVERSE6
REVERSE7
REVERSE7A
DO2
RESTART
*
6.6000000   SECONDS COMPILATION TIME

1281.8181   LINES PER MINUTE COMPILATION SPEED

NO   ERRORS WERE DETECTED

NO   FUNCTIONS WERE REDEFINED

TYPE A LIST TO BE REVERSED.
*(A B C D E)

THE TEST LIST IS (A B C D E)

REVERSE1
(E D C B A)
0.32999999E-1 SECS


REVERSE2
(E D C B A)
0.32999999E-1 SECS


REVERSE4
(E D C B A)
0.32999999E-1 SECS


REVERSE5
(E D C B A)
0.33999999E-1 SECS


REVERSE3
(E D C B A)
0.32999999E-1 SECS
```

```
REVERSE6
(E D C B A)
0.33999999E-1 SECS


REVERSE7
(E D C B A)
0.32999999E-1 SECS

THE LENGTH OF THE TEST LIST WAS 5


TYPE ANOTHER LIST TO BE REVERSED, OR 'STOP'
*(A (B.C) D (E F G) H I J FOO)

THE TEST LIST IS (A (B . C) D (E F G) H I J FOO)

REVERSE1
(FOO J I H (E F G) D (B . C) A)
0.67000000E-1 SECS


REVERSE2
(FOO J I H (E F G) D (B . C) A)
0.5E-1 SECS


REVERSE4
(FOO J I H (E F G) D (B . C) A)
0.67000000E-1 SECS


REVERSE5
(FOO J I H (E F G) D (B . C) A)
0.5E-1 SECS


REVERSE3
(FOO J I H (E F G) D (B . C) A)
0.5E-1 SECS


REVERSE6
(FOO J I H (E F G) D (B . C) A)
0.67000000E-1 SECS


REVERSE7
(FOO J I H (E F G) D (B . C) A)
0.67000000E-1 SECS

THE LENGTH OF THE TEST LIST WAS 8


TYPE ANOTHER LIST TO BE REVERSED, OR 'STOP'
*NIL
```

```
THE TEST LIST IS NIL

REVERSE1
NIL
0.16999999E-1 SECS


REVERSE2
NIL
0.0 SECS


REVERSE4
NIL
0.0 SECS


REVERSE5
NIL
0.16999999E-1 SECS


REVERSE3
NIL
0.0 SECS


REVERSE6
NIL
0.16999999E-1 SECS


REVERSE7
NIL
0.16999999E-1 SECS

THE LENGTH OF THE TEST LIST WAS 0


TYPE ANOTHER LIST TO BE REVERSED, OR 'STOP'
*(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)

THE TEST LIST IS (A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
)

REVERSE1
(Z Y X W V U T S R Q P O N M L K J I H G F E D C B A)
0.15000000 SECS


REVERSE2
(Z Y X W V U T S R Q P O N M L K J I H G F E D C B A)
0.13299999 SECS
```

```
REVERSE4
(Z Y X W V U T S R Q P O N M L K J I H G F E D C B A)
0.11699999 SECS


REVERSE5
(Z Y X W V U T S R Q P O N M L K J I H G F E D C B A)
0.13299999 SECS


REVERSE3
(Z Y X W V U T S R Q P O N M L K J I H G F E D C B A)
0.11699999 SECS


REVERSE6
(Z Y X W V U T S R Q P O N M L K J I H G F E D C B A)
0.15000000 SECS


REVERSE7
(Z Y X W V U T S R Q P O N M L K J I H G F E D C B A)
0.15000000 SECS

THE LENGTH OF THE TEST LIST WAS 26


TYPE ANOTHER LIST TO BE REVERSED, OR 'STOP'
*(1 2 3 4 5 6 7 8 9 10
*11 12 13 14 15 16 17 18 19 20
*21 22 23 24 25 26 27 28 29 30
*31 32 33 34 35 36 37 38 39 40
*41 42 43 44 45 46 47 48 49 50
*51 52 53 54 55 56 57 58 59 60
*61 62 63 64 65 66 67 68 69 70
*71 72 73 74 75 76 77 78 79 80
*81 82 83 84 85 86 87 88 89 90
*91 92 93 94 95 96 97 98 99 100)

THE TEST LIST IS (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 4
4 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99 100)

REVERSE1
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 7
8 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
0.70000000 SECS
```

```
REVERSE2
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 7
8 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
1.0170000 SECS


REVERSE4
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 7
8 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
0.61599999 SECS


REVERSE5
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 7
8 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
0.54999999 SECS


REVERSE3
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 7
8 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
0.53400000 SECS


REVERSE6
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 7
8 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
0.63300000 SECS


REVERSE7
(100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80 79 7
8 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56 55
 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8
7 6 5 4 3 2 1)
0.60000000 SECS
```

```
THE LENGTH OF THE TEST LIST WAS 100

TYPE ANOTHER LIST TO BE REVERSED, OR 'STOP'
*STOP

END OF RUN

NIL
*
```

VI BIBLIOGRAPHY

1. Enea, Horace, MLISP, Technical Report No. CS92, Computer Science Department, Stanford University, 1968.

2. McCarthy, J., Abrahams, P., Edwards, D., Hart, T., Levin, M., LISP 1.5 Programmer's Manual, The Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, MIT Press, 1965.

3. Quam, Lynn, Stanford LISP 1.6 Manual, Stanford Artificial Intelligence Laboratory Operating Note 28.2, Stanford University, 1968.

4. Weissman, Clark, LISP 1.5 Primer, Dickenson Publishing Company, Belmont, California, 1967.