STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-135

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-179

MLISP

BY

DAVID CANFIELD SMITH

OCTOBER 1970

COMPUTER SCIENCE DEPARTMENT

STANFORD UNIVERSITY

STANFORD ARTIFICIAL INTELLIGENCE PROJECT
MEMO AIM-135

October, 1970

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS179

MLISP

by

David Canfield Smith

ABSTRACT: MLISP is a high level list-processing and symbol-manipulation language based on the programming language LISP. MLISP programs are translated into LISP programs and then executed or compiled. MLISP exists for two purposes: (1) to facilitate the writing and understanding of LISP programs; (2) to remedy certain important deficiencies in the list-processing ability of LISP.

This research was supported by Grant PHS MH 066-45-09 and by the Advanced Research Projects Agency of the Department of Defense under contract SD-183. This report supersedes and replaces AI Memo 84.

Reproduced in the USA. Available from the Clearinghouse for Federal Scientific and Technical Information, Springfield, Virginia 22151. Price: Full size copy $3.00; microfiche copy $ .65.

TABLE OF CONTENTS

PAGE

SECTION 2 -- SYNTAX

SECTION 4 -- USER OPERATION OF MLISP

SECTION 5 -- RUN-TIME FUNCTIONS

SECTION 6 -- SAMPLE MLISP PROGRAM

SECTION 7 -- THE MLISP SCANNER

INTRODUCTION - SECTION 1

Most programming languages are designed with the idea that the syntax should be structured to produce efficient code for the computer. Fortran and Algol are outstanding examples. Yet, it is apparent that HUMANS spend more time with any given Program than the COMPUTER. Therefore, it has been our intention to construct a language which is as transparently clear and understandable to a HUMAN BEING as possible. Considerable effort has been spent to make the syntax concise and uncluttered. It reduces the number of parentheses required by LISP, introduces a more mnemonic and natural notation, clarifies the flow of control and permits comments. some "meta-expressions" are added to improve the list-processing power of LISP. 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, permitting the programmer to choose the most natural way of writing routines from a variety of possibilities.

LISP is a list-processing and symbol-manipulation language created at MIT by John McCarthy and his students (McCarthy, 1965). The outstanding features of LISP are: (1) the simplest and most elegant syntax of any language in existence, (2) high-level symbol manipulation capabilities, (3) an efficient set of list-processing primitives, and (4) an easily-usable power of recursion. Furthermore, LISP automatically handles all internal storage management, freeing the user to concentrate on problem solving. This is the single most important improvement over the other major list-processing language, IPL-V. LISP has found applications in many important artificial intelligence investigations, including symbolic mathmatics, natural-language handling, theorem proving and logic.

Unfortunately, there are several important weaknesses in LISP. Anyone who has attempted to understand a LISP program written by another programmer (or even by himself a month earlier) quickly becomes aware of several difficulties:

A. The flow of control is very difficult to follow. In fact, it is about as difficult to follow as machine language or Fortran. This makes understanding the purpose of routines (i.e. what do they do?) difficult. Since comments are not usually permitted, the programmer is unable to provide written assistance.

B. An inordinate amount of time must be spent balancing parentheses, whether in writing a LISP program or trying to understand one. It is frequently difficult to determine where one expression ends and another begins. Formatting utility routines ("pretty-print") help, but every LISP programmer knows the dubious pleasure of laboriously matching left and right parentheses in a function, when all he knows is that one is missing somewhere!

C. The notation of LISP (prefix notation for functions, parentheses around all functions and arguments, etc.), while uniform from a logician's point of view, is far from the most natural or mnemonic for a language. This clumsy notation also *makes* it difficult to understand LISP programs. Since MLISP programs are translated into LISP s-expressions, all of the elegance of LISP is preserved at the translated level; but the unpleasant aspects at the surface level are eliminated.

D. There are important omissions in the list-processing capabilities of LISP. Those are somewhat remedied by the MLISP "meta-expressions", expressions which have no direct LISP correspondence but instead are translated into sequences of LISP instructions. The MLISP meta-expressions are the FOR expression, WHILE expression, UNTIL expression, index expression, assignment expression, and vector operations. The particular deficiency each of these attempts to overcome is discussed in the subsection of SECTION 3 describing the meta-expression in detail.

MLISP was written at Stanford University by Horace Enea for the IBM 360/67 (Enea, 1968). The present author has implemented MLISP on the PDP-10 time-shared computer. He has rewritten the translator, expanded and simplified the syntax, and improved the run-time routines. All of the changes and additions are intended either to make the language more readable and understandable or to make it more powerful.

MLISP programs are first translated into LISP programs, and then these are passed to the LISP interpreter or compiler. 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 programs. In addition to several powerful MLISP run-time routines, the purpose of having such a superstructure is to improve the readability and writeability of LISP, long (in)famous for its obscurity. Since LISP is one of the most elegant and powerful symbol-manipulation languages (but not one of the most readable), it seems appropriate to try to facilitate the use of it.

MLISP has been running for several years on the Stanford PDP-10 time-shared computer. It has been distributed to the DEC User Services Group (DECUS). The MLISP translator and run-time routines are themselves compiled LISP programs. The Stanford version runs under the Stanford LISP 1.6 system (Quam, 1969). Some effort has been made to keep the translator as machine independent as possible; in theory MLISP could be implemented on any machine with a working LISP system by making only minor changes. The one probable exception to this is the MLISP scanner; to enable scanning (where most of the time is spent) to be as efficient as possible, the translator uses machine language scanning routines. While these routines have greatly increased translation speed (MLISP now translates at a rate of 3000-5000 lines per minute.), their use means that someone wishing to implement MLISP on a system without LISP 1.6 will have to use an equivalent scanner package. For this reason, a whole section of this manual (SECTION 7) is devoted to presenting an equivalent scanner.

While LISP was created with the goal of being machine independent, it has turned out that most LISP systems have unique features. The situation is so difficult that Anthony Hearn has attempted to define "a uniform subset of LISP 1.5 capable of assembly under a wide range of existing compilers and interpreters," called STANDARD LISP (Hearn, 1969). MLISP helps to alleviate this situation by introducing another level of machine independence: to implement MLISP on a given LISP system, one changes the underlying translator rather than the surface syntax. Dr. Hearn has also constructed an MLISP-like language called REDUCE (HEARN, 1970).

SYNTAX - SECTION 2.1

The complete MLISP syntax is contained in the following section. Several sets of meta-symbols are used to simplify the presentation of the syntax:

(1) <> - ANGLED BRACKETS enclose non-terminal symbols.

(2) {} - BRACES enclose optional elements; i.e. the elements inside tray or may not be present.

(3) {}* - These special meta-symbols enclose optional elements which may be present 0 or more times (i.e. the enclosed elements need not be present, but there is no limit to the number of times they may occur).

(4) ⊂⊃ - "HORSE SHOES" enclose alternative elements, which are separated by commas. The user may select any one of the enclosed elements to form a legal syntactic expression.

(5) The BNF symbols ::= and | are used to define syntax elements. The left-hand side of the ::= symbol is the syntax element being defined; the right-hand side is its definition. The vertical bar (|) is used to indicate alternative definitions.

(6) All other symbols stand for themselves.

There are several features of MLISP that am not explicitly in the syntax:

(a) IGNORED CHARACTERS - All spaces, carriage returns, line feeds, form feeds, tabs, vertical tabs, and altmodes are ignored by the scanner.

(b) COMMENTS - Any sequence of characters enclosed between percent signs (%) is taken to be a comment. The scanner ignores comments, considering them to be completely non-existent; ABC%<anything>%DEF is the same as ABCDEF as far as the scanner is concerned. NOTE: the comment symbol (%) may not be used in any other capacity than to start or end a comment!! The MLISP-defined atom PERCENT (value is %) facilitates dealing with the percent sign in other capacities.

The user should note that there are no "statements" in MLISP; everything returns a value, even FOR-loops, WHILE-loops, etc. Therefore, all major syntactic entities are "expressions".

DISCLAIMER: For reasons of simplicity, the syntax presented below is slightly different from the one the translator actually uses. The only difference is that infix operators do not *all* have the same precedence. Instead they are organized into a precedence (hierarchy) system. Example:

```
A + B * C - D CONS L
```

is the same as

```
((A + (B * C)) - D)  CONS  L.
```

From this it may be seen that * takes precedence over + and -, and all three take precedence over CONS. The complete precedence system is explained in the section on infix operators (SECTION 3.3). Giving infix operators different precedences helps to cut down on the number of parentheses needed.

. SYNTAX - SECTION 2.2

```
<program>                ::=  <expression> .


<expression>             ::=  <simple_expression>
                                 (<infix_operator> <simple_expression>)*


<infix_operator>         ::=  <regular_infix>
                          |   <vector_infix>


<regular_infix>          ::=  ⊂*, /, +, -, ↑, ↓, @, =, ≠, ≤, ≥, ∈, &, ∧, |, ∨⊃
                          |   <identifier>


<vector_infix>           ::=  <regular_infix> •


<prefix_operator>        ::=  <regular_prefix>
                          |   <vector_prefix>


<regular_prefix>         ::=  ⊂+, -, ¬⊃
                          |   <identifier>


<vector_prefix>          ::=  <regular_prefix> •


<simple_expression>      ::=  <block>
                          |   <function_definition>
                          |   <LAMBDA_expression>
                          |   <DEFINE_expression>
                          |   <IF_expression>
                          |   <FOR_expression>
                          |   <WHILE_expression>
                          |   <UNTIL_expression>
                          |   <assignment_expression>
                          |   <function_call>
                          |   <index_expression>
                          |   <list_expression>
                          |   <quoted_expression>
                          |   <atom>
                          |   <prefix_operator> <simple_expression>
                          |   ( <expression> )


<block>                  ::=  BEGIN
                                 (<declaration> ;)*
                                 (<expression> ;)*
```

```
                              (<expression>)
                           END


<declaration>            ::=  NEW <identifier_list>
                          |   SPECIAL <identifier_list>


<identifier_list>        ::=  <identifier> (, <identifier>)*
                          |   <empty>


<function_definition>    ::=  «EXPR, FEXPR, LEXPR, MACRO» <identifier>
                                ( <lambda_identifier_list> ); <expression


<LAMBDA_expression>      ::=  LAMBDA
                                ( <lambda_identifier_list> ); <expression


<lambda_identifier_list>::=  (SPECIAL) <identifier>
                                (, (SPECIAL) <identifier>)*
                          |   <empty>


<DEFINE_expression>      ::= DEFINE  <DEFINE_clause>  (, <DEFINE_clause>)*


<DEFINE_clause>          ::=  <DEFINE_symbol>  PREFIX
                          |   <DEFINE_symbol> (PREFIX) <alternate_name>
                          |   <DEFINE_symbol>  (PREFIX) (<alternate_name>)
                                <integer> <integer>


<DEFINE_symbol>          ::=  <identifier>
                          |   <any character except %>


<alternate_name>         ::=  <identifier>
                          |   <any  character except %, ; or ,>


<IF_expression>          ::=  IF <expression>
                                THEN  <expression> (ALSO <expression>)*
                               (ELSE <expression>  (ALSO  <expression>)* )


<FOR_expression>         ::=  <FOR_clause> (<FOR_clause>)*
                                «DO, COLLECT, ; <identifier>» <expressi
                                (UNTIL <expression>)
```

```
<FOR_clause>                 ::= FOR (NEW) <identifier> ⊂IN, ON⊃ <expression>
                              |   FOR (NEW) <identifier>    ←     <expression>
                                     TO <expression> (BY <expression>)

<WHILE_expression>           ::=  WHILE <expression> ⊂DO, COLLECT:, <expression>

<UNTIL_expression>           ::=  ⊂DO, COLLECT⊃ <expression> UNTIL <expression>

<assignment_expression>      ::=  <regular_assignment>
                              |   <array_assignment>
                              |   <index_assignment>
                              |   <decomposition>

<regular_assignment>         ::=  <identifier> ← <expression>

<array_assignment>           ::=  <identifier> (<argument_list>) ← <expression>

<index_assignment>           ::=  <identifier> [<argument_list>] ← <expression>

<decomposition>              ::=  <simple_expression> ←• <expression>

<function_call>              ::=  <identifier>            ( <argument_list> )
                              |   <LAMBDA_expression> ; ( <argument_list> )

<argument_list>              ::=  <expression> (, <expression>)*
                              |   <empty>

<index_expression>           ::=  <simple_expression> [ <argument_list> 3

<list_expression>            ::=  <  <argument_list>  >

<quoted_expression>          ::=  ' <s-expression>

<s-expression>               ::=  <atom>
                              |   0
                              |   ( <s-expression> . <s-expression> )
                              |   ( <s-expression> ((,) <s-expression>)* )
```

```
<atom>                       ::=  <identifier>
                              |   <number>
                              |   <string>

<identifier>                 ::=  <letter>  ( ⊂<letter>, <digit>⊃ )*

<letter>                     ::=  ⊂A, B, C, ..., Z, a, b, c, ..., z, _, !, !⊃
                              |   <literally_character> <any character except

<literally_character>        ::=  ?

<number>                     ::=  <integer>
                              |   OCTAL <octal_integer>
                              |   <real>

<integer>                    ::= <digit> (<digit>)*

<digit>                      ::=  ⊂0, 1, 2, 3, 4, 3, 6, 7, 8, 9⊃

<octal_integer>              ::= <octal_digit> (<octal_digit>)*

<octal_digit>                ::=  ⊂0, 1, 2, 3, 4, 5, 6, 7⊃

<real>                       ::= <integer>  <exponent>
                              |   <integer> . <integer> (<exponent>)

<exponent>                   ::=  E (⊂+, -⊃) <integer>

<string>                     ::=  " (<any character except " and %>)* "
```

. SYNTAX - SECTION 2,3

Reserved words for MLISP:

- BEGIN
- NEW
- SPECIAL
- END
- IF
- THEN
- ALSO
- ELSE
- FOR
- IN
- ON
- TO
- BY
- 00
- COLLECT
- UNTIL
- EXPR
- FEXPR
- LEXPR
- MACRO
- DEFINE
- LAMBDA
- OCTAL
- WHILE

Reserved symbols for MLISP:

```
(      )      [      ]      <
;      ,      .      '      ←
?      %      "
```

Symbols pre-defined in MLISP:

Symbol	MLISP Translation	
*	TIMES	
/	QUOTIENT	
+	PLUS	
-	DIFFERENCE	(MINUS If usedas a prefix)
↑	PRELIST	(see SECTION 5,2)
↓	SUFLIST	(see SECTION 5,2)
@	APPEND	
=	EQUAL	
≠	NEQUAL	(see SECTION 5,2)
≤	LEQUAL	(see SECTION 5,2)
≥	GEQUAL	(see SECTION 5,2)
ε	MEMBER	
&	ANO	
∧	AND	
|	OR	
∨	OR	
¬	NOT	

SYNTAX - SECTION 2.4

Atoms having MLISP-defined values:

Atom	Value	Ascii (octal)
TRUE	T	124
FALSE	NIL	none
F	NIL	none
CIRCLEX	•	26
COLON	:	72
COMMA	,	34
DASH	-	55
DBQUOTE	"	42
DOLLAR	$	44
EQSIGN	=	75
LARROW	←	137
LABR	< (left angled bracket)	74
LPAR	((left parenthesis)	50
LSBR	[(left square bracket)	133
PERCENT	%	43
PERIOD	.	56
PLUSS	+	53
QT	'	47
RABR	> (right angled bracket)	74
RPAR	) (right parenthesis)	31
RSBR	] (right square bracket)	133
SEMICOLON	;	73
SLASH	/	57
STAR	*	52
UNDERBAR	_	30
TAB	<tab>	11
LF	<line feed>	12
VT	<vertical tab>	13
FF	<form feed>	14
CR	<carriage return>	15
BLANK	<blank>	40
ALTMODE	<altmode>	175

. SYNTAX - SECTION 2,5

Precedence of Infix operators In MLISP (from highest to lowest), The following table Is included here purely for reference; If is explained fully In SECTION 3,3, Any function8 not present In the table below will hrvr the default precedence (precedence level 3) and default binding powers,

Symbol	Function	Precedence	Binding Power Left	Right
*	TIMES	1	700	750
TIMES	TIMES	1	700	750
*TIMES	*TIMES	1	700	750
/	QUOTIENT	1	700	750
QUOTIENT	QUOTIENT	1	700	750
*QUO	*QUO	1	700	750
+	PLUS	2	600	650
PLUS	PLUS	2	600	650
*PLUS	*PLUS	2	600	650
-	DIFFERENCE	2	600	650
DIFFERENCE	DIFFERENCE	2	600	650
*DIF	*DIF	2	600	650
<default>		3	500	550
@	APPEND	4	450	400
APPEND	APPEND	4	450	400
*APPEND	*APPEND	4	450	400
NCONC	NCONC	4	450	400
CONS	CONS	4	450	400
XCONS	XCONS	4	450	400
CAT	CAT	4	450	400
EQ	EQ	5	300	350
NEQ	NEQ	5	300	350
=	EQUAL	5	300	350
EQUAL	EQUAL	5	300	350
≠	NEQUAL	5	300	350
NEQUAL	NEQUAL	5	300	350
LESSP	LESSP	5	300	350
*LESS	*LESS	5	300	350
≤	LEQUAL	5	300	350
LEQUAL	LEQUAL	5	300	350
GREATERP	GREATERP	5	300	350
*GREAT	*GREAT	5	300	350

≥	GEQUAL	5	300	350
GEQUAL	GEQUAL	5	300	350
∈	MEMBER	5	300	350
MEMBER	MEMBER	5	300	350
MEMQ	MEMQ	5	300	350
&	AND	6	200	230
∧	AND	6	200	250
AND	AND	6	200	250
|	OR	7	100	150
∨	OR	7	100	150
OR	OR	7	100	150

. SEMANTICS - SECTION 3

This section presents the meaning of each of the elements in the syntax. First the syntactic parts about to be explained are listed. than their meaning is explained in detail. Finally, a series of examples illustrates them, and in many cases their actual LISP translations are given.

It is assumed that the user has a working knowledge of LISP. If not, Weissman's PRIMER (Weissman, 1967) provides a good tutorial. McCarthy's PROGRAMMER'S MANUAL (McCarthy, 1965) is the standard reference manual. In addition, the user should familiarize himself with the manual for his LISP system, since, as was pointed out, LISP systems may vary from computer to computer.

In this section the symbol "→" means "is translated into".

, SEMANTICS - SECTION 3,1

<program> ::= <expression> .

An MLISP program is an expression followed by a period, Usually the program is a series of expressions enclosed In a BEGIN-END pair; i.e. it is black, This permits more than one MLISP expression to be translated at the same time, The translation of thr program gets bound to the function RESTART after translation has been completed. Example: If the MLISP program Is

```
BEGIN
   NEW Xi
   X ← READ();
   PRINTSTR("I JUST READ " CAT X);
END,
```

then RESTART would be defined to be

```
(DEFPROP RESTART
 (LAMBDA NIL
  (PROG (X)
        (SETQ x (READ))
        (PRINTSTR (CAT (QUOTE "I JUST REAR ") X))))
EXPR)
```

Basically the RESTART function serves to give a name to thr main body of the program, so that the user can execute his program at any time by calling it, For example, typing (RESTART) to LISP Would cause tho above program to be executed,

'Any expression whose translation Is NIL (i.e. function definitions and DEFINE expressions) are not Included In the RESTART function; only executable (non-NIL) expressions are included, Example: If the MLISP program is

```
BEGIN
   NEW X,Y;
   EXPR MAX (X,Y);  If X≥ Y THEN  X  ELSE Y;
   EXPR TPRINT(X); TERPRI  PRINT  X;
   TPRINT MAX(X ← READ(), Y ← READ());
END .
```

then RESTART would be defined to be

```
(DEFPROP RESTART
 (LAMBDA NIL
  (PROG  (X  Y)
        (TPRINT (MAX (SETQ X (READ)) (SETQ  Y  (READ))))))
EXPR)
```

SEMANTICS - SECTION 3.2

```
<expression> ::= <simple_expression> (<infix_operator><simple_expression>)*
```

An expression is one or more simple expressions separated by infix operators:

```
<simple_expression>
<simple_expression> <infix_operator> <simple_expression>
<simple_expression> <infix_operator> <simple_expression>
        <infix_operator> <simple_expression>
...
```

From this description, it appears that all infix operators have the same precedence. However, several often-used LISP functions have been given different precedences from the others. This often enables one to eliminate parentheses that would be necessary to group the terms in an expression. The precedences have been chosen to be as natural and useful to the LISP programmer as possible. Example:

```
        A+B*C = D/E | X=Y & Z=W
```

is the same as

```
        ((A +(B*C)) = D) | ((X=Y) & (Z=W)),
```

but the former is far more readable than the latter. The precedence system used is explained in detail in the following section on infix operators (SECTION 3.3).

Examples of expressions:

A	→	A
(A)	→	A
¬A	→	(NOT A)
'A	→	(QUOTE A)
"A"	→	(QUOTE "A")
<A>	→	(LIST A)
16	→	16
123.45E+10	→	1.2345E12
OCTAL 100	→	64 (decimal)
"THIS IS A STRING."	→	(QUOTE "THIS IS A STRING.")
"ANOTHER " CAT "STRING"	→	(CAT (QUOTE "ANOTHER ") (QUOTE "STRING"))
'(A (B.C) D)	→	(QUOTE (A (B.C) D))
<'A,'(B.C),'D>	→	(LIST (QUOTE A) (QUOTE (B.C)) (QUOTE D))

```
A + 10                        →        (PLUS A 10)
A + 1                         →        (ADD1 A)
A - 10                        →        (DIFFERENCE A 10)
A - 1                         →        (SUB1 A)

A /B- C                       →        (DIFFERENCE (QUOTIENT A B) C)
((A / B) - C)                 →        (DIFFERENCE (QUOTIENT A B) C)
QUOTIENT(A,B)- C              →        (DIFFERENCE (QUOTIENT A 8) C)
DIFFERENCE(A / B,C)           →        (DIFFERENCE (QUOTIENT A B) C)

X ∈ L                         →        (MEMBER X L)
X = Y                         →        (EQUAL X Y)
L1 @ L2                       →        (APPEND L1 L2)
L1 @ L2 @ L3                  →        (APPEND L1 L2 L3)
A CONS B CONS NIL             →        (CONS A (CONS B NIL))
(A CONS B) CONS NIL           →        (CONS (CONS A B) NIL)

A + B GREATERP 10             →        (GREATERP (PLUS A B) 10)
A*B + C CONS L1@L2            →        (CONS (PLUS (TIMES A B) C) (APPEND L1 L2
A CONS B = C | ¬Y             →        (OR (EQUAL (CONS A B) C) (NOT Y))
X = 7 & Y = A/B               →        (AND (EQUAL X 7) (EQUAL Y (QUOTIENT A B)
X EQ 'A | X EQ '8             →        (OR (EQ X (QUOTE A)) (EQ X (QUOTE B)))
A∧B∧C ∨¬A A ¬B                →        (OR (ANDA B C) (AND (NOT A) (NOT B)))

A ← B ← C                     →        (SETQ A (SETQ B C))
A . B(I) ← C                  →        (SETQ A (STORE (B I) C))
A * B ← C                     →        (TIMES A (SETQ B C))
A ← B * c                     →        (SETQ A (TIMES B C))
A * B ← C * D                 →        (TIMES A (SETQ B (TIMES C D)))
A ← B * C ← D                 →        (SETQ A (TIMES B (SETQ C D)))

IF FOO(X,Y) THEN              →        (COND
BEGIN NEW N;                            ((FOO x Y)
   N ← X MAX Y;                          (PROG (N)
   X ← Y ← NIL;                                (SETQ N (MAX X Y))
   PRINTSTR "HO HO";                           (SETQ X (SETQ Y NIL))
   -PRINT N                                    (PRINTSTR (QUOTE "HO HO"))
END                                            (PRINT N W
ELSE PRINTSTR "HA HA"                   (T(PRINTSTR (QUOTE "HAHA"))))

DEFINE FOO PREFIX, AND &, OR | 100 150, SUFLIST + 490 491;

EXPR MAX (X,Y); IF X≥Y THEN X ELSE Y

FOR NEW I IN L COLLECT <CAR I> UNTIL I = '(STOP);

WHILE A NEQ 'STOP DO A← READO

DO A ← READ() UNTIL hEQ 'STOP
```

SEMANTICS - SECTION 3,3

```
<infix_operator>  ::=  <regular_infix>
                   |   <vector_infix>

<regular_infix>   ::=  c*, /, +, -, ↑, ↓, @, =, ≠, ≤, ≥, ∈, &, ∧, |, ∨>
                   |   <identifier>

<vector_infix>    ::=  <regular_infix> ●
```

An infix operator is either a regular infix or a vector infix, A regular infix is any of the symbols listed, or an identifier which is the name of a function taking two arguments, A vector infix is a regular infix followed by the vector indicator (●),

(A) Regular Infixes

The normal LISP way of writing function calls is the "prefix notation," the function name occuring first followed by its arguments:

```
        PLUS(A,B),
```

MLISP permits functions called with two arguments to be written In the "infix notation," the function name occuring between the arguments:

```
        A PLUS 8,
```

In addition, certain commonly-wad LISP and MLISP functions have been given abbreviations:

```
        A + B,
```

Below la a complete list of these abbreviations. The user can define abbreviations for his own functions, or change the MLISP-defined ones, by using the DEFINE expression(SECTION3,7).

Abbreviation	Function	
*	TIMES	
/	QUOTIENT	
+	PLUS	(May be used a@ a prefix.)
-	DIFFERENCE	(MINUS if used as a prefix)
↑	PRELIST	(see SECTION 5,2)
↓	SUFLIST	(see SECTION 5,2)
@	APPEND	
=	EQUAL	
≠	NEQUAL	(see SECTION 5,2)
≤	LEQUAL	(see SECTION 5,2)
≥	GEQUAL	(see SECTION 5,2)
∈	MEMBER	
&	AND	
∧	AND	
|	OR	
∨	OR	
¬	NOT	(This is a prefix, not an infix.)

Infix operators do not all have the same precedence; some take priority over others when expressions are parsed. Example:

```
        A + B * C - D / E
```

is parsed:

```
        (A + (B * C)) - (D / E),
```

A precedence system for infix operators has been set up (a) to help cut down on the number of parentheses needed; and (b) because most programming languages have a precedence system, and so having one is more natural to a Programmer than not having one.

Listed below is the complete precedence system for infix operators in MLISP, Any function which does not appear explicitly in the table below will be assigned the default precedence and binding powers (unless the user assigns different ones with the DEFINE expression). For reference, the table below is also listed in SECTION 2,5,

Symbol	Function	Precedence	Binding Power	
			Left	Right
*	TIMES	1	700	750
TIMES	TIMES	1	700	750
*TIMES	*TIMES	1	700	750
/	QUOTIENT	1	700	750
QUOTIENT	QUOTIENT	1	700	750
*QUO	*QUO	1	700	750
+	PLUS	2	600	650
PLUS	PLUS	2	600	650
*PLUS	*PLUS	2	600	650
-	DIFFERENCE	2	600	650
DIFFERENCE	DIFFERENCE	2	600	650
*DIF	*DIF	2	600	650
<default>		3	500	550
@	APPEND	4	450	400
APPEND	APPEND	4	450	400
*APPEND	*APPEND	4	450	400
NCONC	NCONC	4	450	400
CONS	CONS	4	450	400
XCONS	XCONS	4	450	400
CAT	CAT	4	450	400
EQ	EQ	5	300	350
NEQ	NEQ	5	300	350
=	EQUAL	5	300	330
EQUAL	EQUAL	5	300	350
≠	NEQUAL	5	300	350
NEQUAL	NEQUAL	5	300	350
LESSP	LESSP	5	300	350
*LESS	*LESS	5	300	350
≤	LEQUAL	5	300	350
LEQUAL	LEQUAL	5	300	350
GREATERP	GREATERP	5	300	330
*GREAT	*GREAT	5	300	350
≥	GEQUAL	5	300	350
GEQUAL	GEQUAL	5	300	350

ε	MEMBER	5	300	350
MEMBER	MEMBER	5	300	350
MEMQ	MEMQ	5	300	350
&	AND	6	200	230
A	AND	6	200	250
AND	AND	6	200	250
|	OR	7	100	150
v				
OR	OR OR	7 7	100 100	150 150

The reader has probably noticed that the last two columns in this table are labeled Binding Power - Left and Right. Basically, the "binding powers" of an infix operator are the strengths with which it "binds" or pulls on the elements to the left and right of it. The concept of binding powers is sufficient to completely specify any precedence system. For example, consider:

A + B * C.

Both + and * are trying to attach B as the second argument for their functions (PLUS and TIMES). But the left binding power of * (700) is greater than the right binding power of + (650), so this expression would be parsed:

A + (B * C).

As another example, suppose the user has defined a two-argument function MAX. Since MAX does not occur explicitly in the precedence system above, the default binding powers (500 and 550) are used. Then

A MAX B MAX C

is parsed:

(A MAX B) MAX C

since for default functions the right binding power is greater than the left binding power. This is also true for all other functions except those on precedence level 4, the s-expression building functions (APPEND, CONS, etc.). For a LISP user, it is not only more natural but more efficient to have the association of these functions go to the right:

4 CONS B CONS C CONS NIL

is parsed:

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

The user should study the precedence system above. Parentheses may be used at any time to alter the associations of the precedence system, but hopefully it has been constructed carefully enough so that the user will seldom have to do this.

All user-defined infix functions normally get assigned the default binding powers. Note that this means that user-defined functions normally take precedence over some LISP and MLISP functions (those on precedence levels 4 - 7). However; the user can assign different binding powers to his functions, or even to the functions above, by means of the DEFINE expression (SECTION 3.7). With the DEFINE expression, he can set up any Precedence system he choses.

The rationale for the precedence system:

1	*, TIMES, *TIMES /, QUOTIENT, *QUO	First come the arithmetic functions, which operate only on numbers and which yield only numerical values.
2	+, PLUS, *PLUS -, DIFFERENCE, *DIF	As is natural, multiplication and division take precedence over addition and subtraction.
3	all others	Then come all user-defined functions, and all LISP and MLISP functions not listed here explicitly.
4	@, APPEND, *APPEND, NCONC CONS, XCONS CAT	These are followed by functions which operate on s-expressions to build new s-expressions.
5	EQ =, EQUAL ≠, NEQUAL ≤, LEQUAL, LESSP, *LESS ≥, GEQUAL, GREATERP, *GREAT ∈, MEMBER, MEMQ	Of lower precedence are functions which operate on s-expressions, but which yield only boolean values.
6	&, ∧, AND	Of lowest precedence are functions which operate only on boolean values, and which yield only boolean values.
7	|, ∨, OR	As is natural, OR has a lower precedence than AND; In fact OR has the lowest precedence of any function.

In addition to infix operators, prefix operators and the assignment operator (←) have also been implemented using binding powers. The binding powers for prefixes are -1 and 1000; those for the assignment operator are $001 and 0. These may be changed by the DEFINE expression. They are listed only for reference; the use of prefixes and assignment expressions is explained better by the syntax.

(B) Vector Infixes

Vector infixes are a very powerful MLISP concept, They provide a concise means of mapping functions onto one or two lists, a facility not readily available in LISP, They developed from the observation that lists may be regarded as any-dimensional vectors, The LISP system then becomes an infinite-dimensional vector space, Scalars in this vector space are atoms, Vector infixes (and vector prefixes) are an attempt to define some primitive operations over this vector space, Basically vector infixes are functions which are mapped onto their vector (list) arguments to yield a vector (list) of results, much like a two-argument MAPCAR,

Suppose V = (v1, v2, ... vm) and W = (w1,w2, ... wn) are two vectors (i,e, lists), Addition of two vectors is accomplished by:

```
V +• W  =  (v1+w1, v2+w2, ... vk+wk),     where k =min(m,n).
```

Multiplication by a scalar:

```
10 *• v =  (10*v1, 10*v2, ... 10*vm)
v *• 10 =  (v1*10, v2*10, ... vm*10)
```

Multiplication of two scalars:

```
10 *• 20 = 10 * 20 = 200
```

To illustrate these vector primitives, we will use them to write the Euclidean inner product:

$$V \cdot W = \sum_{i=1}^{k}(vi + wi)$$

First observe that if we CONS the function PLUS onto a list of numbers, we get an executable expression:

```
'PLUS CONS '(1 2 3 4 5)  =  (PLUS 1 2 3 4 5),
```

Then:

```
EXPR INNERPROD (V,W); EVAL ('PLUS CONS V +• W)
```

is the desired inner product function using vector operations, It is worthwhile noting that we could also write:

```
EXPR INNERPROD (V,W);
   BEGIN NEW SUM;
      SUM ← 0;
      FOR NEW v IN V FOR NEW w IN W DO SUM ← SUM + (v+w);
      RETURN SUM
   END
```

or equivalently:

```
        EXPR INNERPROD(V,W); FOR NEW v IN V FOR NEW w IN W; PLUS v*w
```

Vector operations, however, provide the most concise means of writing the function.

The next logical step in the development of vector operations is to permit virtually any two-argument LISP, MLISP or user-defined function to be used as a vector operator:

```
        V *. W          =  (v1 * w1, v2 * w2, ... vk * wk)
        V CONS. W       =  (v1 CONS w1, v2 CONS w2, ..., vk CONS wk)
        V FOO. W        =  (v1 FOO w1, v2 FOO w2, ..., vk FOO wk)
```

where in each case k=min(length V, length W).

Note:

(a) The result of vector operations is a vector (i.e. list), unless both arguments are scalars (atoms).

(b) The length of the result vector is the shorter of the lengths of the two vector arguments, or the length of the vector argument if the other argument is a scalar.

Following an infix operator by the vector indicator (.) does not change its precedence. In determining the parsing of an expression, the presence of absence of . is ignored:

```
        A +. B *. C CONS. L
```

is parsed exactly the same as

```
        A + B * C CONS L,
```

namely

```
        (A +. (B *. C)) CONS. L.
```

In addition to two-argument vector infixes, one-argument vector prefixes are also permitted. These are discussed in the following section on prefix operators (SECTION 3.4). Example:

```
        ATOM. '(A B (C) D) = (T T NIL T).
```

Examples of infix operators:

```
A + 10                     →       (PLUS A 10)
A + 1                      →       (ADD1 A)
A - 10                     →       (DIFFERENCE A 10)
A - 1                      →       (SUB1 A)

A / B - C                  →       (DIFFERENCE (QUOTIENT A B) C)
((A / B) - C)              →       (DIFFERENCE (QUOTIENT A B) C)
QUOTIENT(A,B) - C          →       (DIFFERENCE (QUOTIENT A B) C)
DIFFERENCE(A / B,C)        →       (DIFFERENCE (QUOTIENT A B) C)

X ε L                      →       (MEMBER X L)
X = Y                      →       (EQUAL X y)
L1 @ L2                    →       (APPEND L1 L2)
L1 @ L2 @ L3               →       (APPEND L1 L2 L3)
<A,B,C> @ FOO(X,Y)         →       (APPEND (LIST A B C) (FOO X y))
A CONS B CONS NIL          →       (CONS A (CONS B NIL))
(A CONS B) CONS NIL        →       (CONS (CONS A B) NIL)
<A CONS B>                 →       (LIST (CONS A B))
A CONS L↓3 @ X             →       (CONS A (APPEND (SUFLIST L 3) X))

A + B GREATERP 10          →       (GREATERP (PLUS A B) 10)
A*B + C CONS L             →       (CONS (PLUS (TIMES A 6) C) L)
A CONS B = C | ¬Y          →       (OR (EQUAL (CONS A B) C) (NOT Y))
X = 7 & Y = A*B            →       (AND (EQUAL X 7) (EQUAL Y (TIMES A B)))
X EQ 'A | X EQ 'B          →       (OR (EQ X (QUOTE A)) (EQ X (QUOTE B)))
```

Vector Infixes:

```
'(1 2 3) +• '(4 5 6)              ≡       (5 7 9)
'(1 2 3) *• '(4 5 6 7)            a       (4 10 18)

2 *• '(1 2 3)                     ≡       (2 4 6)
2 *• '(1 2 3) +• '(4 5 6)         ≡       (6 9 12)
2 *•('(1 2 3) +• '(4 5 6))        ≡       (10 14 18)

'(1 2 3) CONS@ '(A B C D)         ≡       ((1 . A)(2 . B) (3 . C))
'((1 2) (3 4)) @• '((A B) (C D))  ≡       ((1 2 A B) (3 4 C D))

'((A 8 C) (D E F) (G H I)) ↑• 1   ≡       ((A) (D) (G))
'((A B C) (D E F) (G H I)) ↓• 1   ≡       ((B C) (E F) (H I))

'("JOHN " "MARY ") CAT. '("DOE" "SMITH")
                                  ≡       ("JOHN DOE" "MARY SMITH")

"JOHN," CAT@ '("DOE" "SMITH")     ≡       ("JOHN-DOE" "JOHN_SMITH")

AT. ("JOHN," CAT. '("DOE" "SMITH"))
                                  ≡       (JOHN_DOE JOHN_SMITH)
```

, SEMANTICS - SECTION 3,4

```
<prefix_operator>   ::=  <regular_prefix>
                     |   <vector_prefix>

<regular_prefix>    ;a+   c+, -, ¬ɔ
                     |   <identifier>

<vector_prefix>     ::=  <regular_prefix> •
```

4 prefixoperatoris ● lthot aregular prefix or a vector prefix, A regular prefix is any of the symbols +, - o or ¬, o or an identifier representing any one-argument functionwhich theMLISP translator know8 about, A vector prefix is a regular prefix followed by the vector indicator (•),

Regular prefixes

The main purposes of prefixes are toclarify expressions and to ● Jlmkat@ parentheses, NOT X isbetterthan NOT(X), though both are legal, and ¬X is even better, The translator knows about all one-argument LISP andMLISP functions, In addition, the translator notes all one-argument EXPR's translated, Later on in theprogram (i,e, after the function definition), that function may be used like any other prefix, Example: if the function definition

EXPR FOO(X); TERPRI PRINT X

occurred In a program, then In the rest of the program following this definition It would be legal to treat FOO a8 a prefix.

This is one way that the translator can be made aware Of user-defined prefixes, Another way is to use the DEFINE expression (SECTION 3,7):

DEFINE FOO PREFIX

has the effect of stating Co the translator that the function FOO, regardless of its definition (if any), will only have one argument In the rest of the Drogram and so shouldbe treated as a prefix,

Vector prefixes

Vector prefixes are avery interesting and very powerful extension of prefix operators, The concept of vector operations was explained in the preceding section, The basic idea is that vector prefixes operate on not Just one, but on a whole list of arguments, and they return a whole list of values, The prefix operator is mapped onto the list, with the operator applied to each element in it, This enables many complexexpressions to be written concisely, Vector prefixes may also Operate on atoms (scalars) instead of lists, Examples of prefix operators!

```
+X                          →      X
-X                          →      (MINUS X)
¬X                          →      (NOT X)
NOT X                       →      (NOT X)
NOT(X)                      ≡      (NOT X)

ATOM FOO(X,Y,Z + 10)        →      (ATOM (FOO X Y (PLUS Z 10)))
NULL CDR L                  →      (NULL (CDR L))
TERPRI PRINT CAR L          →      (TERPRI (PRINT (CAR L)))
LENGTH L + 10               →      (PLUS (LENGTH L) 10)
-4 ∧ ¬B ∧ ¬C                →      (AND (NOT A) (NOT 8) (NOT C))
NUMBERP X ∨ ¬ATOM X         →      (OR (NUMBERP X) (NOT (ATOM X)))

NOT ATOM X ← READ()         →      (NOT (ATOM (SETQ X (READ))))
NOT ATOM X & READ()         →      (AND (NOT (ATOM X)) (READ))
NOT ATOM X CONS READ()      →      (CONS (NOT (ATOM X)) (READ))
```

Vector prefixes!

Suppose L ≡ (A B (C D) NIL E).

```
ATOM• L                     ≡      (T T NIL T T)
NOT• ATOM. L                ≡      (NIL NIL T NIL NIL)
¬• L                        ≡      (NIL NIL NIL T NIL)
LENGTH• L                   ≡      (0 0 2 0 0)

+• '(1 2 3 4)               ≡      (1 2 3 4)
-• '(1 2 3 4)               ≡      (-1 -2 -3 -4)
NUMBERP• '(1 2 3 4)         ≡      (T T T T)
ADD1• '(1 2 3 4)            ≡      (2 3 4 5)
-• SUB1• '(1 2 3 4)         ≡      (0 -1 -2 -3)

AT• '("THIS" "IS" "A" "LIST" "OF" "STRINGS")
                            ≡      (THIS IS 4 LIST OF STRINGS)
AT• "STRINGS"               ≡      STRINGS
AT "STRINGS"                ≡      STRINGS
STR• '(MORE STRINGS)        ≡      ("MORE" "STRINGS")
STR '(MORE STRINGS)         ≡      "WORE STRINGS)"

ATOM• 10                    ≡      T
ATOM 10                     a      T
NUMBERP• 10                 ≡      T
NULL• 10                    ≡      NIL

CAR. '((A 1) (B 2) (C 3))                  a      (A B C)
CDR. '((A 1) (B 2) (C 3))                  ≡      ((1) (2) (3))
CADR• '((A 1) (6 2) (C 3))                 3      (1 2 3)
NUMBERP• CADR• '((A 1) (B 2) (C 3))        ≡      (T T T)
```

SEMANTICS - SECTION 3.5

Each of the remaining sub-sections in SECTION 3 explains an example of a simple expression.

```
<block>             ::=  BEGIN
                            (<declaration> ;)*
                            (<expression> ;)*
                            (<expression>)
                         END

<declaration>       ::=  NEW <identifier_list>
                      |  SPECIAL <identifier_list>

<identifier_list>   ::=  <identifier> (, <identifier>)*
                      |  <empty>
```

A block is the reserved word BEGIN, followed by any number of declarations separated by semicolons (;), followed by any number of expressions separated by semicolons, followed by the reserved word END. The last expression need not have a semicolon after it. A declaration is either of the reserved words NEW or SPECIAL, followed by an identifier list. An identifier list is any number of identifiers (possibly none) separated by commas.

A block is translated into a PROG. Any variables (identifiers) declared using the NEW declaration become the PROG variables. Check your LISP manual to see whether or not PROG variables are automatically initialized to NIL in your version of LISP. The scope of NEW variables is the scope of the PROG, i.e. until the matching END. NEW variables may also be declared SPECIAL. Each expression following the declarations until the END becomes a statement in the PROG. There should be a semicolon after each expression, with the exception that the last semicolon is optional. END closes off the PROG.

SPECIAL declarations are somewhat unique in that they have no translation; instead they have an effect on the translator. A flag for the LISP 1.6 compiler is put on the property list of each variable declared SPECIAL. This flag enables the compiler to compile free variables and global variables correctly.

SPECIAL declarations have the effect of declaring their variables SPECIAL throughout the entire program, regardless of the physical location of the declaration in the program. This enables the user to mark variables SPECIAL wherever it is convenient to do so, and simultaneously prevents the compiler (and user) from getting confused when variables are sometimes SPECIAL and sometimes not, It is a good idea to make SPECIAL variable names distinct from other variable names as a way of keeping track of them, For example, an exclamation mark (!) could be included in each SPECIAL variable name; SPECIAL !A, !B, !C, In general, the fewer variables that have to be declared SPECIAL, the better; the code for SPECIAL variables runs somewhat slower than that for non-SPECIAL ones,

For the user's reference, the following section is reproduced from Quam's LISP 1,6 manual (Quam, 1969),

> In compiled functions, any variable which is bound in a LAMBDA or PROG and has a free occurrence elsewhere must be declared SPECIAL, [A variable is said to have a free occurrence if it not bound in any LAMBDA or PROG containing the occurrence,] [Also,] variables which are used in a functional context must be declared SPECIAL or else the compiler will mistake them for undefined EXPR's,

Similar restrictions hold for many other LISP compilers, It is UP to the user to make sure he understands fully the conventions fat compiling in his LISP system, For the MLISP user, there is one further restriction: variables in the left-hand side of a decomposition assignment expression (SECTION 3,111 must be declared SPECIAL if the expression is to work correctly,

AS with PROG's, a value may be returned for a block by using the RETURN function, Labels may be transferred to using the GO function; labels are declared by following the label immediately with a semicolon (e,g, L;), not with a colon, However, the iteration "meta-expressions" described in following sections are to be much recommended over labels and GO transfers, Examples of blocks:

```
BEGIN                          →      (PROG NIL)
END
```

```
BEGIN                          →      (PROG NIL
L; X ← READ();                         L    (SETQ X (READ))
   IF X EQ Y THEN RETURN TRUE               (COND ((EQ X Y) (RETURN TRUE))
   ELSE PRINT <X,Y>;                              (T (PRINT (LIST X Y))))
   GO L;                                    (GO L))
END
```

```
BEGIN NEW X1,X2,X3;            →      (PROG (X1 X 2 X3)
   SPECIAL X3,Y,Z;                          (SETQ Z NIL)
   Z ← NIL;                                 (SETQ X 1(READ))
   X1 ← READ();                             (SETQ X 2 (ADD1 (TIMES 10 X1)))
   X2 ← 10*X1 + 1;                          (COND
   IF FOO(X1,Y,Z) & X3=1 THEN                ((AND (FOO X 1 Y Z) (EQUAL X 3 1))
      PRINTSTR("ANSWER=" CAT Y)               (PRINTSTR
   ELSE X3 ← X2 + X1;                           (CAT (QUOTE "ANSWER=") Y)))
   RETURN X3                                (T (SETQ X3 (PLUS X2 X1))))
END                                         (RETURN X3))
```

```
BEGIN                          →      (PROG NIL
   EXPR MAX (X,Y);                          (DEFPROP MAX
      IF X≥Y THEN X ELSE Y;                  (LAMBDA (X Y)
                                              (COND ((GEQUAL X Y) X) (T Y)))
                                            EXPR)

   EXPR MAX-LIST (L,M);                     (DEFPROP MAX_LIST
      IF NULL L THEN M ELSE                  (LAMBDA (L M) (COND
      MAX_LIST(CDR L,MAX(M,CAR L));           ((NULL L) M)
                                              (T (MAX_LIST (CDR L)
                                                        (MAX M (CAR L))))))
                                            EXPR)

   PRINT MAX_LIST(READ(),0);                (PRINT (MAX_LIST (READ) 0)))
END
```

SEMANTICS - SECTION 3,6

```
<function_definition>       ::=  cEXPR, FEXPR, LEXPR, MACRO:, <identifier>
                                   ( <lambda_identifier_list> ); <expression>

<LAMBDA_expression>         ::=  LAMBDA
                                   ( <lambda_identifier_list> );<expression>

<lambda_identifier_list>    ::=  (SPECIAL) <identifier>
                                   {, (SPECIAL) <identifier>}*
                            |    <empty>
```

A function definition is one of the function types: EXPR, FEXPR, LEXPR, MACRO, followed by an identifier (the name of the function), followed by a LAMBDA variable list and LAMBDA body, A LAMBDA expression is essentially the same thing, being the reserved Word LAMBDA followed by a LAMBDA variable list and LAMBDA body, A LAMBDA identifier list is any numbar of identifiers (possibly none) separated by commas, Each identifier may be preceeded by the word SPECIAL, This and the SPECIAL declaration in blocks are the two ways the user may declare variables to be SPECIAL, (SECTION 3,5 discusses SPECIAL variables,)

When the MLISP translator encounters a function definition, the following three steps occur:

(1) The complete function definition is translated,

(2) The function is then immediately defined (i,e, the definition is carried out), without waiting for the rest of the program to be translated,

(3) NIL is returned as the translation for the expression,

Note that since step (2) is carried out in the middle of the translation of the program, the user might accidentally redefine some LISP or MLISP function that would cause the rest of his program to be translated incorrectly, To guard against this, each function name is first checked to see if it already has a function definition of any type; If it does, a warning message is printed, If this happens, change the name of the function and recompile the program,

Usually a program consists of a BEGIN-END pair enclosing a series of function definitions and other expressions, Function definitions are not executable at run time; their effect occurs at translation time, in step (2) above, As step (3) states, NIL will be returned as the translation for function definitions, All exectuable expressions will have non-NIL translations, In translating a program, all NIL translations are thrown out and only non-NIL ones retained, Examples of function definitions:

```
EXPR NOTHING(); PRINTSTR "THIS ISN'T MUCH OF A FUNCTION";

EXPR REV(L);    IF NULL L THEN NIL ELSE REV(CDR L)@<CAR L>;

FEXPR OPEN(X);  EVAL <'INPUT,'DSK:,CAR X>;

MACRO NOT_MEMBER(X);   <'NOT, <'MEMBER,X[2],X[3]>>;

EXPR FOOBAZ(X, SPECIAL Y);
   BEGIN
      NEW A;
      IF X = REV(X) THEN A ← Y
      ELSE BEGIN
         OPEN(FOO);
         NOTHING();
         CLOSE(FOO);
      END;
      RETURN <A, REV(A)>;
   END;

EXPR INNER_PRODUCT (V, W); EVAL ('PLUS CONS V +*W);
   % This takes the inner product of two vectors (lists). %

EXPR INNER_PRODUCT(V,W); FOR NEW V IN V FOR NEW W IN W; PLUS V+W;
 % So does this. %
```

Examples of LAMBDA expressions:

Assume that "F00" represents a function which has bean defined to be the same as the LAMBDA expression In each of the following examples.

EXAMPLE 1,

```
mlisp:  MAPCAR(FUNCTION(LAMBDA (X); X CONS NIL), '(A B C))

lisp:   (MAPCAR (FUNCTION (LAMBDA (X) (CONS X NIL))) (QUOTE (A B C)))

equivalently:  MAPCAR(FUNCTION(FOO), '(A B C),
```

EXAMPLE 2,

```
mlisp:  LAMBDA (X,Y);
            IF X EQ Y THEN PRINTSTR "THEY ARE THE SAME" ELSE
            IF NOT ATOM X THEN PRINTSTR "FIRST IS NOT AN ATOM"
            ELSE PRINTSTR("X =" CAT X);
        (READ(), READ())

lisp:   ((LAMBDA (X Y)
          (COND
           ((EQ X y) (PRINTSTR (QUOTE "THEY ARE THE SAME")))
           ((NOT (ATOM X)) (PRINTSTR (QUOTE "FIRST IS NOT AN ATOM")))
           (T (PRINTSTR (CAT (QUOTE "X =") X)))))
         (READ)
         (READ) )

equivalently:  FOO(READ(), READ())
```

EXAMPLE 3,

```
mlisp:  LAMBDA (X,Y,SPECIAL Q);
            LAMBDA (Z);
                IF FOO(X) THEN PRINT Z ELSE PRINT Q;
            (<X,Y>);
        (A,B+1, NIL)

lisp:   ((LAMBDA (X Y Q)
          ((LAMBDA (Z) (COND ((FOO X) (PRINT Z)) (T (PRINT Q)))) (LIST X Y)))
         A
         (ADD1 B)
         NIL)

equivalently:  FOO(A, B+1, NIL)
```

. SEMANTICS - SECTION 3.7

```
<DEFINE_expression>  ::=  DEFINE <DEFINE_clause> {, <DEFINE_clause>}*

<DEFINE_clause>      ::=  <DEFINE_symbol> PREFIX
                      |   <DEFINE_symbol> {PREFIX} <alternate_name>
                      |   <DEFINE_symbol> {PREFIX} {<alternate_name>}
                              <integer> <integer>

<DEFINE_symbol>      ::=  <identifier>
                      |   <any character except " or %>

<alternate_name>     ::=  <identifier>
                      |   <any character except ", %, ; or ,>
```

A DEFINE expression is one or more DEFINE clauses separated by commas. A DEFINE clause is an identifier or any character except " or % (the DEFINE symbol), followed by any or all of (1) the word PREFIX, (2) an alternate name (abbreviation) for the DEFINE symbol, and (3) two integers representing left and right binding powers for the DEFINE symbol. An alternate name is an identifier or any character except ", %, semicolon (;) or comma (,).

The DEFINE expression provides a versatile means of communicating with the MLISP translator. As with function definitions, the translation of DEFINE expressions is NIL. Instead of a translation, the DEFINE expression has an effect on the translator. The effect is to assign certain properties to the DEFINE symbol which the translator will make use of in the rest of the program. The DEFINE expression will be explained by examples.

Examples of DEFINE expressions:

(1) DEFINE FOO PREFIX

This informs the translator that hereafter in the program the function FOO is to be treated like a prefix (SECTION 3.4). This means that FOO may be used without parentheses around its argument, and it may be used as a vector operator. Only identifiers which are the names of one-argument functions should be defined to be prefixes.

(2) DEFINE UNION ∪

This informs the translator that the symbol ∪ is to be considered as an abbreviation for the function UNION. After this DEFINE expression has been translated, whenever the scanner encounters ∪, it will immediately convert it to UNION, The effect of writing ∪ will be exactly the same as if UNION had been written, The alternate name may be an identifier:

DEFINE CAR a

would convert every subsequent occurrence of a into CAR, Also, the DEFINE symbol itself may be a special character:

DEFINE ; ,

would translate all commas in the rest of the program into semicolons,

DEFINE ; STOP

would translate every subsequent occurrence of the word STOP into a semicolon, to illustrate this, consider the following example:

```
BEGIN                          →      (PROG NIL
   DEFINE  CAR a, CDR d, NULL n,
       IF if, THEN →, ELSE else;
                                             (DEFPROP rev
   EXPR rev (l);                              (LAMBDA (l)
       if n l → nil                            (COND((NULLl) NIL)
       else rev(d l) @  <a l>;                      (T (APPEND (rev (CDR l))
END                                                           (LIST (CAR l))))))
                                             EXPR))
```

(3) DEFINE UNION 360 370

This specifies that the left and right binding powers for the function UNION are to be 360 and 370 respectively, Binding powers are explained in the section on infix operators (SECTION 3,3), The values above would give UNION a precedence of between 4 and 5 in the precedence system for infix operators (c,f, SECTION 3,3), Only identifiers representing one- and two-argument functions (prefixes and infixes) should be given binding powers,

(4) DEFINE UNION ∪ 360 370

This defines ∪ to be an abbreviation for UNION and simultaneously sets up left and right binding powers for UNION,

(5) DEFINE FOO PREFIX α

This specifies that the function FOO is to be treated as a prefix, and that the symbol α is to be considered an abbreviation for it.

(6) DEFINE NOT PREFIX ¬ -1 1000

This specifies that the function NOT is to be treated as a prefix, that the symbol ¬ is to be considered an abbreviation for it, and that its left and right binding powers are to be -1 and 1000 respectively. The equivalent of this expression has already been executed for all one-argument LISP and MLISP functions.

(7) DEFINE UNION ∪ 360 370, INTERSECTION ∩ 380 390, RETURN PREFIX -1 0;

After this DEFINE expression has been translated,

```
RETURN A @ B ∪ C @ D ∩ E @ F
```

would be translated

```
(RETURN (UNION (APPEND A B) (INTERSECTION (APPEND C D)(APPEND E F)))),
```

exactly as if it had been written

```
RETURN ((A @ B) ∪ ((C @ D) ∩ (E @ F))).
```

UNION is given lower binding powers than INTERSECTION, and both of them have lower binding powers than the 400 and 450 binding powers of (@) APPEND (SECTION 2.5). Seting the right binding power of RETURN to 0 insures that an entire expression (In this case: A @ B ∪ C ∩ D @ E) will be translated as its argument, rather than just a simple expression as would normally be the case (since RETURN is a prefix). This is because all binding powers in MLISP are larger than 0; therefore, all infix operators will bind up their arguments before RETURN does. In fact, anything with a right binding power of 0 will gobble up everything until the next expression-stopper (reserved word, ")", ";", etc.).

, SEMANTICS - SECTION 3.8

```
<IF_expression>  ::=  IF <expression>
                        THEN <expression> (ALSO <expression>)*
                       (ELSE<expression>  (ALSO  <expression>)* )
```

An IF(or conditional) expression is the reserved word IF, followed by any expression, followed by the reserved word THEN end another expression, optionally followed by any number of ALSO clauses. This is optionally followed by the reserved word ELSE and a third expression, again optionally followed by any number of ALSO clauses. An ALSO clause is the reserved word ALSO followed by any expression.

Conditional expressions get translated into LISP COND's. In LISP 1.6 there may be more than one expression after the predicate;example:

```
(COND (P1 E1) (P2 E2 E3) (P3 E4 E5 E6))
```

is a legal LISP 1.6 conditional expression. Where there is more than one expression, the expressions are evaluated from left to right; the value of the last one becomes the value of the COND.

In the following, E1, E2, E3 ... represent any expressions.

```
IF E1 THEN E2               →       (COND (E1 E2) (T NIL))

IF E1 THEN E2 ELSE E3       →       (COND (E1 E2) (T E3))

IF E1 THEN E2 ALSO E3       →       (COND (E1 E2 E3) (T E4 E5 E6))
ELSE E4 ALSO E5 ALSO E6

IF E1 THEN E2 ELSE          →       (COND (E1 E2) (E3 E4) (T E5))
IF E3 THEN E4 ELSE E5
```

Nesting of conditionals is permitted to any degree of complexity. Each ELSE term is matched up with the nearest preceeding THEN, unless parentheses are used to group the terms differently.

```
IF E1 THEN                  →       (COND (E1 (COND (E2 E3) (T E4)))
   IF E2 THEN E3 ELSE E4                  (T NIL))

IF E1 THEN                  →       (COND (E1 (COND (E2 E3) (T NIL)))
   (IF E2 THEN E3)                        (T E4))
ELSE E4

IF E1 THEN                  →       (COND (E1 (COND (E2 E3) (T E4)))
   IF E2 THEN E3 ELSE E4                  (T E5))
ELSE E5
```

Examples of IF expressions:

```
IF X= 10 THEN PRINT Y .              (COND((EQUAL X 10)(PRINT Y))(T NIL))


IF X ≠10 THEN PRINT y .              (CONO((NEQUAL X 10)(PRINT Y))
ELSE PRINT Z                               (T (PRINT Z)))


                                     (COND
IF A & B & C & D    THEN  →           ((AND  A  B C D)
   IF X∈L THEN PRINT 'MATCH            (COND ((MEMBER XL) (PRINT (QUOTE MATCH))
   ELSE PRINT 'NO_MATCH                       (T(PRINT (QUOTE NO_MATCH)))))
ELSE IF FOO(A,B)&¬C THEN              ((AND(FOO A B)(NOT C))
   IF <X>∈L THEN PRINT T               (COND((MEMBER (LIST X)L)(PRINT T))
   ELSE PRINT NIL                             (T (PRINT NIL))))
ELSE PRINT 'OH_WELL                   (T (PRINT (QUOTE OH_WELL))))


IF X ≤ 100 THEN         →            (COND
   Y ← 2*X ALSO CO L                  ((LEQUAL X100)(SETQ Y (TIMES 2 X))(GO L
ELSE Y←X+1 ALSO RETURN <X,Y>          (T(SETO Y (ADD1 X))(RETURN (LIST X Y))))
```

SEMANTICS - SECTION 3.9

```
<FOR_expression>  ::=  <FOR_clause> (<FOR_clause>)*
                         ⊂DO, COLLECT, ; <identifier>⊃ <expression>
                         (UNTIL <expression>)

<FOR_clause>        ::=  FOR (NEW) <identifier> ⊂IN, ON⊃ <expression>
                      |  FOR (NEW) <identifier>  ←   <expression>
                         TO <expression> (BY <expression>)
```

A FOR expression is any number of FOR clauses, followed by the reserved word DO, the reserved word COLLECT, or a semicolon (;) together with an identifier which is a two-argument function name. This is followed by an expression (the "body" of the FOR-loop), which is optionally followed by the reserved word UNTIL and another expression. A FOR clause is the reserved word FOR, optionally followed by the reserved word NEW, followed by an identifier (the control variable), and followed by either: (a) the reserved word IN or the reserved word ON, and an expression which evalutes to a list (possibly NIL), or (b) a left arrow (←), followed by an expression which evaluates to a number (the lower limit), followed by the reserved word TO and another expression which evaluates to a number (the upper limit). This is optionally followed by the reserved Word BY and a third expression which evaluates to a number (the increment).

The FOR expression (FOR-loop) is the most powerful meta-expression in MLISP. It is designed to facilitate dealing with individual elements in lists. The MLISP FOR expression carries the development of LISP's MAPLIST and MAPCAR functions to their logical conclusion. Extensive work has gone into the design and implementation of FOR expressions. Used thoughtfully, they can greatly simplify manipulating lists. The FOR expression is not just one, but many expressions; there is an unbounded number of possible expressions that may be built up from its syntax.

FOR expressions provide the ability to:

(A) Step through a list, dealing with each element in it individually (use IN).

(B) Step through a list, dealing with the whole list, the CDR of it, the CDDR of it, the CDDDR of it, etc. (use ON).

(C) Step through a numerical range (e.g. from 1 to 10) using any numerical increment (use ←). There are no restrictions on the numbers involved (lower limit, upper limit, increment).

(D) Step through any number of lists and/or numerical ranges in parallel (use more than one FOR clause).

(E) Make the control variables local to the body of the FOR-loop (use NEW or to preserve their values when the FOR-loop exits. Control variables should be specified to be NEW whenever possible, because the LISP code for NEW variables is more efficient. Unless you are interested in the value of a control variable after the FOR-loop exits, declare it NEW.

(F) Control the value returned by the FOR expression. The value using DO is the value of the FOR-loop body the last time through the loop; the value with COLLECT is a list formed by APPEND'ing together the values of the FOR-loop body each time through the loop. Alternatively, any two-argument function may be used to generate a FOR-expression value: the first time through the loop, the value of the FOR-loop body becomes the value of the loop; each succeeding time through, the two-argument function is applied to the previous value of the loop and to the current value of the FOR-loop body to yield a new value for the loop.

(G) Terminate execution of the loop at any time (use UNTIL).

Example:

```
FOR NEW I IN X@Y FOR N←1 TO 10 BY 2 DO PRINT <N,I> UNTIL I EQ 'STOP
```

In this example, "FOR NEW I IN X@Y" and "FOR N←1 TO 10 BY 2" are "FOR clauses". I and N are "control variables". I is "local" to the body of the FOR-loop by virtue of its being declared NEW; N is not local. The expression "X@Y" (APPEND X Y) should evaluate to a list; I will step through that list, being set to the CAR of it, the CADR of it, the CADDR of it, etc. The control variable N steps through a numerical range (1-10) in increments of 2. The expression "PRINT <I,N>" is the "body" of the FOR-loop. The UNTIL condition is "I EQ 'STOP". Since DO is used, the value of the FOR-loop is the value of "PRINT <I,N>" the last time it was executed.

The execution of FOR expressions proceeds as follows:

(1) The list or numerical range for each FOR clause is checked. If any list is NIL, or if any numerical range is exhausted, then the FOR-loop exits returning its current value (initially NIL). Before exiting, each clause is examined. If any clause has a control variable which was declared NEW, that control variable is reset to the value it had when the loop was entered. If any clause has a control variable which was not declared NEW, and if the list or range for that clause is exhausted, then that control variable is set to NIL. Otherwise, the control variable is left set to the value it had the last time through the loop; this may be useful for determining which lists or ranges were exhausted, and how many times the loop was executed.

A numerical range is said to be "exhausted" if (a) the increment is positive and the lower limit > the upper limit, or (b) the increment is negative and the lower limit < the upper limit. An increment of 0 is, of course, illegal.

(2) Next, each control variable is assigned a value. This value is: (a) the CAR of its list if IN is used, (b) the entire list if ON is used, or (c) the lower limit if a numerical range is used.

(3) Then the body of the FOR-loop is executed, and a value is computed for the loop as explained in (F) above.

(4) Finally, the UNTIL expression (if any) is evaluated. If its value is true, the FOR-loop exits immediately. No control variables are reset except the ones declared NEW, which are set to the values they had when the loop was entered. Thus all non-NEW control variables will remain set to the values they had when the UNTIL condition became true. This is sometimes useful for testing how many times the loop was executed, and for determining the cause of termination. Example: In

```
    FOR I IN L DO PRINT I UNTIL I EQ 'STOP,
```

when the loop exits I will be set to NIL if it got all the way through the list L without encountering the atom STOP; otherwise it will be set to STOP.

(5) If the UNTIL expression was false (or non-existent), the lists and numerical ranges are advanced as follows: (a) each list is set to the CDR of itself; (b) in each numerical range, the increment is added to the lower limit to yield a new lower limit.

(6) Then step (1) is executed again.

Continuing the example above, suppose X = '(A B C) and Y = '(D); then executing:

```
FOR NEW I IN X@Y FOR N←1 TO 10 BY 2 DO PRINT <N,I> UNTIL I EQ 'STOP
```

would

```
(a) print       (1 A)
                (3 B)
                (5 C)
                (7 D)
```

(b) return a value of (7 D), and

(c) leave N set to 7.

The FOR expression

`FOR NEW I;I IN X@Y FOR N←1 TO 10 BY 2 DO PRINT <N,I> UNTIL I EQ 'STOP`

is equivalent to the following block:

```
     BEGIN    NEW    V,L1,L2,I;
        L1 ← X @ Y;
        L2 ← 1;

  LOOP; IF NULL L1 | L2 GREATERP 10 THEN GO EXIT;
        I ← CAR L1;
        N ← L2;
        V ← PRINT <N,I>;
        IF I EQ 'STOP THEN RETURN V;
        LI ← CDR L1;
        L2 ← L2 + 2;
        GO LOOP;

  EXIT; IF NULL L2 THEN N ← NIL;
        RETURN V;
     END;
```

Examples of FOR expressions: Suppose 1. = '(A (B,C) D),

FOR NEW I IN L DO PRINT I	would print A (B,C) D and return D
FOR IINL DO PRINTI	would print A (B,C) D set I to NIL and return D
FOR NEW I ON L DO PRINTI	would print (A (B,C) D) ((B,C) D) (D) and return (D)
FOR NEW I I N L COLLECT P R I N T <I>	would print (A) ((B,C)) (D) and return (A (B,C) D)
FOR I IN L COLLECT PRINT <<I>>	would print ((A)) (((B,C)) ((D)) setI to NIL and return ((A) ((B,C)) (D))
FOR NW I ON L COLLECT PRINT I	would print (A (B,C) D) ((B,C) D) (D) and return (A (B,C) D (B,C) D D)
FOR NEW I ON L ; APPEND PRINT I	would have exactly the same effect as the preceeding FOR expression.

```
FOR I IN L DO PRINT I UNTIL NOT ATOM I would print     A
                                                       (B,C)
                                       set I to (B,C)
                                       and return (B,C)


FOR I ON L DO PRINT I UNTIL NOT ATOM CAR I
                                       would print     (A (B,C) D)
                                                       ((B,C) D)
                                       set I to ((B,C) D)
                                       and return ((B,C) D)


POR NEW I IN L COLLECT PRINT <I> UNTIL NOT ATOM I
                                       would print     (A)
                                                       ((B,C))
                                       and return (A (B,C))


FOR I ON L COLLECT PRINT I UNTIL NOT ATOM CAR I
                                       would print     (A (B,C) D)
                                                       ((B,C) D)
                                       set I to ((B,C) D)
                                       and return (A (B,C) D (B,C) D)


FOR NEW I←1 TO 4 DO PRINT I            would print     1
                                                       2
                                                       3
                                                       4
                                       and return 4


FOR NEW I←1 TO 100 BY 30 DO PRINT I    would print     1
                                                       31
                                                       61
                                                       91
                                       and return 91


FOR NEW I←10 TO -10 BY -5 DO PRINT I   would print     10
                                                       5
                                                       0
                                                       -5
                                                       -10
                                       and return -10
```

```
FOR I←3,14 TO 8,69 BY 0,002 DO PRINT I UNTIL I ≥ 3,2
                                   would print     3,14
                                                   3,16
                                                   3,18
                                                   3,2
                                   set I to 3,2
                                   and return  3.2
```

```
FOR NEW I IN I. FOR NEW J←1 TO 10 COLLECT PRINT <I,J>
                                   would print     (A 1)
                                                   ((B,C) 2)
                                                   (D 3)
                                   and return  (A 1 (B,C) 2 D 3)
```

```
FORNEWIINL FOR J←1 TO 10 COLLECT PRINT <<I,J>>
                                   would print     ((A 1))
                                                   (((B,C) 2))
                                                   ((D 3))
                                   set J to 3
                                   and return  ((A 1) ((B,C) 2) (D 3))
```

```
FOR J←1 TO 10 COLLECT              would print     (A 1)
   FOR I IN L COLLECT PRINT <I,J>                  ((B,C) 1)
   UNTIL NOT ATOM I                                (A 2)
UNTIL J=3                                          ((B,C) 2)
                                                   (A 3)
                                                   ((B,C) 3)
                                   set I to (B.C)
                                   set J to 3
                       and return  (A 1 (B,C) 1 A 2 (B,C) 2 A 3 (B,C) 3)
```

```
DECK ←
   FOR NEW SUIT IN '(SPADE HEART DIAMOND CLUB) COLLECT
   FOR NEW N←1 TO 13 COLLECT
      <<SUIT,N>>;

would ret DECK =
   ((SPADE 1) (SPADE 2) ,,,. (SPADE 13) (HEART 1) (HEART 2) ,,, )
```

As was stated in (D) above, more than one list or numerical range may be stepped through in parallel. Below are some examples of parallel FOR's (*):

```
EXPR PAIR-UP (VECTOR1, VECTOR2);
   FOR NEW X1 IN VECTOR1 FOR NEW X2 IN VECTOR2 COLLECT <X1 CONS X2>;

'(A B C)        PAIR_UP '(1 3 5 7 9,       =       ((A.1) (B.3) (C.5))
'("JOHN" CDR) PAIRUP '("SMITH" (X))        =       (("JOHN"."SMITH") (CDR X))
```

Vector operations also provide an interesting way to accomplish this:

```
'(A B C)        CONS* '(1 3 5 7 9)         =       ((A.1) (B.3) (C.5))
'("JOHN" CDR)  CONS* '("SMITH" (X))        =       (("JOHN"."SMITH") (CDR X))
```

```
EXPR STRIP (ITEMS, VECTOR);
   BEGIN NEW V;
      FOR V ON VECTOR FOR NEW I IN ITEMS 00 NIL UNTIL I NEQ CAR V;
      RETURN V
   END;

'(a b x) STRIP '(a b c d e)                =       (c d e)
'(x b c) STRIP '(a b c d e)                =       (a b c d e)
'(a b c d e f) STRIP '(a b c d e)          a       NIL
```

```
EXPR WHERE_IN (X, VECTOR);
   BEGIN NEW V,N;
      FOR V IN VECTOR FOR N←1 TO 1000 DO NIL UNTIL V = X;
      RETURN IF NULL V THEN 0 ELSE N
   END;

'a WHERE_IN '(b c a d)                     =       3
'z WHERE_IN '(b o a d)                     =       0
```

(*) I am indebted to Larry Tesler for suggesting these examples.

. SEMANTICS - SECTION 3.10

<WHILE_expression> ::= WHILE <expression> ⊂DO, COLLECT⊃ <expression>

<UNTIL_expression> ::= ⊂DO, COLLECT⊃ <expression> UNTIL <expression>

A WHILE expression is the reserved word WHILE, followed by any expression, followed by either of the reserved words DO or COLLECT and another expression (the "body" of the WHILE-loop). An UNTIL expression is either of the reserved words DO or COLLECT, followed by any expression (the "body" of the UNTIL-loop), followed by the reserved word UNTIL and another expression.

WHILE and UNTIL expressions are two more of the MLISP "meta-expressions". They have no direct counterparts in LISP. They are translated into LISP PROG's. Their execution involves iteration; it does not involve recursion. Therefore, these loops may be executed any number of times with no danger of overflowing the pushdown list.

The execution of WHILE expressions is carried out as follows.

(1) The expression after the WHILE is evaluated. If its value is NIL, then the loop exits returning its current value (initially NIL).

(2) Then the body of the WHILE-loop is evaluated and a new value for the loop is computed. As with FOR expressions, DO and COLLECT control how the value of the WHILE-loop is built up. With DO, each time the body of the loop is executed, the value that results becomes the value of the WHILE-loop; with COLLECT these values are APPEND'ed together. Then step (a) is carried out again.

UNTIL expressions are very similar to WHILE expressions. The only difference is that in WHILE-loops the test for the terminating condition is made first and then the body is executed; whereas in UNTIL-loops it is made second, after the body has been executed. This means that in UNTIL-loops, the body of the loop is sure to get executed at least once; but in WHILE-loops it may not be executed at all. To get a description of UNTIL-loops, just interchange steps (1) and (2) above in the description of WHILE-loops.

As an example of the power of using COLLECT with WHILE-loops and UNTIL-loops, suppose an input file contains a sequence of lists in the form:

```
(DEFPROP <function_name> <lambda_body> <function_type>),
```

which is a standard form for LISP 1.6 function definitions. Suppose it is desired to assemble all the function names in the file into a list, printing out each function name as it is read. Each of the following two expressions does this. Concise statements of complex expressions such as this is one of the primary purposes of MLISP.

```
L ← WHILE NOT ATOM X←READ() COLLECT <PRINT X[2]>;

X ← READ();   L ← COLLECT <PRINT X[2]> UNTIL ATOM X←READ();
```

Examples of WHILE expressions:

```
WHILE A=B DO A←FOO(A,B)

WHILE ATOM X←READ() 00 PRINT X

WHILE X≠10 COLLECT PROG2(X ← X+1,<FOO(X,Y)>)

WHILE ¬(X ∈ L) DO X←READ()

WHILE ¬STOP 00
BEGIN
   NEW !X,Y;   SPECIAL !X;
   !X ← READ();
   Y  . FOO(!X,READ(),READ());
   IF IX EQ 'STOP THEN STOP←T ELSE PRINT Y
END
```

Examples of UNTIL expressions:

```
DO A←FOO(A,B) UNTIL A#B

00 PRINT X UNTIL NOT ATOM X←READ()

COLLECT PROG2(X ← X+1,<FOO(X,Y)>) UNTIL X=10

DO X←READ() UNTIL X ∈ L

DO BEGIN
   NEW !X,Y;   SPECIAL !X;
   !X ← READ();
   Y  ← FOO(!X,READ(),READ());
   IF !X EQ 'STOP THEN STOP←T ELSE PRINT Y
END
UNTIL STOP
```

. SEMANTICS - SECTION 3.11

```
<assignment_expression> ::=  <regular_assignment>
                          |  <array_assignment>
                          |  <index_assignment>
                          |  <decomposition>


<regular_assignment>    ::=  <identifier> ← <expression>


<array_assignment>      ::= <identifier> (<argument_list>) ← <expression>


<index_assignment>      ::= <identifier> [<argument_list>] ← <expression>


<decomposition>         ::=  <simple_expression> ←← <expression>
```

The assignment expression is one of the most powerful expressions In MLISP, With it, one can change the value of a variable, store into an array, change a single element In a list leaving the other elements untouched, or decompose a list according to a "template". In all cases, the value of an assignment expression Is the value of the expression on the right-hand side.

Making an assignment expression a <simple_expression> has an interesting property: It removes the assignment operator (←) from the normal realm of infix operators, In particular, when

```
      ATOM X ← READ()
```

is encountered, it is reduced as follows:

```
    ATOM                X           ←     READ()
     ↓
<prefix_operator> <identifier> ← <expression>
     ↓
<prefix_operator> <assignment_expression>
     ↓
<prefix_operator> <simple_expression>
     ↓
<simple_expression>
     ↓
<expression>
```

and so the prefix will modify the entire assignment expression. However, for infix operators, when

```
      ATOM X & READ()
```

is encountered, it is reduced as:

```
      ATOM              X                            &          READ()
        ↓
<prefix_operator> <identifier>          <infix_operator> <expression>
        ↓
<prefix_operator> <simple_expression> <infix_operator> <expression>
        ↓
<simple_expression>                     <infix_operator> <expression>
        ↓
<expression>
```

and so the prefix will modify only the identifier. The assignment operator acts like it has an extremely high left binding power (binding powers are discussed in SECTION 3.3), and an extremely low right binding power, which it does: the left binding power is 1001 and the right is 0. In other words, the left binding power of ← is stronger than any infix or prefix, while the right binding power of ← is weaker than any infix or prefix. Therefore,

```
        ATOM X ← READ()          →        (ATOM (SETQ X (READ)))
whereas
        ATOM X & READ()          →        (AND (ATOM X) (READ))
        ATOM X CONS READ()       →        (CONS (ATOM X)(READ))
```

Regular assignment

The regular assignment is the simplest of the options, It Just translates into SETQ, The left-hand side must be an identifier; the right-hand side may be any expression, Example:

```
X ← Y + 1                    →        (SETQ X (ADD1 Y))
```

Array assignment

The array assignment is the means for storing values into array cells, LISP 1,6 permits 1-5 dimensional arrays as a data structure, The assignment operator is here translated into STORE, The left-hand side must be a call on an array; the right-hand side may be any expression, Example:

```
A(I,J) ← Y + 1               →        (STORE (A I J) (ADD1 Y))
```

Index assignment

The index assignment provides a means for changing a single element in a list, leaving the other elements untouched, This facility is not readily available in LISP, The left-hand side must be an identifier whose value is a list, followed by an index list as in index expressions (SECTION 3,131; the right-hand side may be any expression, The index list is used to reference the location in the list which is to be changed, Into this location is placed the value of the right-hand side, Example: If

```
L = (A  B (C D) E F),
```

then

```
L[3,1] ← 1
```

would change the value of L to

```
L = (A B (1 D) E F),
```

It is permissible to place values into cells which did not exist in the original list; in this case, NIL is placed into any locations that had to be created, Example:

```
L[3,5] ← 1
```

would change the value of L to

```
L = (A B (C D NIL NIL 1) E F),
```

Decomposition assignment

The decomposition assignment is the most powerful in MLISP. It provides a means of decomposing a list according to a "template". The left-hand side is a simple expression which should evaluate to an s-expression (the template); the right-hand side may be any expression. The template is an s-expression composed of variables, each of which is to be set to the element in the corresponding location of the right-hand side, hereafter called the "RHS".

One word of caution: if the decomposition assignment expression is used in a compiled program, all the variables in the template must be declared SPECIAL. Otherwise, the variables will not be set correctly.

Example:

```
'(X Y Z) ←• '(A B (C D) E F)
```

would set

```
X to A
Y to B
Z to (C D).
```

Regular assignment expressions are a subset of decomposition assignment expressions. Any regular assignment, such as:

```
X ← Y + 1
```

may be written as a trivial decomposition assignment:

```
'X ←• Y + 1
```

provided X is declared SPECIAL.

The decomposition assignment expression raises the interesting possibility that some variables may fail to get set because the template structure in which they occur does not correspond to the structure of the right-hand side (RHS). Any such variable which cannot be set to an RHS value is set to NIL. A template variable will always be set to an RHS value if the template position in which it occurs is "compatible" with the corresponding RHS position. The only "incompatible" case is when the template position is a non-atomic s-expression, and the corresponding RHS position is an atom. In this case, all variables occupying the incompatible template position will be set to NIL. Example:

```
'((X Y) Z) ← '(A B (C D) E F)
```

would set

X,Y to NIL	Because the first template position is a list, (X Y), whereas the first RHS position is an atom, A. Thus the first template position is "incompatible" with the first RHS position, and the variables in it are set to NIL.
Z to B	Because the second template position is "compatible" with the second RHS position.

The CDR of the RHS may be obtained by a dotted pair in the template:

```
'(X Y . Z) ← '(A B (C D) E F)
```

would set

X to A
Y to B
Z to ((C D) E F).

Suppose L = (A B (C D) E F). The list L itself could be used as the template:

```
L ← '(1 2 (3 4 5 6 7)(8 9))
```

would set

A to 1	In this case A, B, C, D, E and F must all be declared SPECIAL.
B to 2	
C to 3	
D to 4	
E to (8 9)	
F to NIL.	

Finally, a "match anything" symbol is available for use in the template: an underbar (_). This symbol will match any amount of list structure necessary to make the template match the RHS. Example:

```
'(_ X) ←• '(A B (C D) E F)
```

would set

X to F	Because the template specifies that the value for the variable X should occur as the last element in the RHS. The underbar matches (A B (C D) E) in this case.

Using the underbar symbol in a template causes the evaluation of the decomposition assignment expression to proceed differently: previously, any variable would be set if it was in a template position compatible with the corresponding RHS position. Using underbars, however, may require that the template structure match EXACTLY the RHS structure. Consider the example above in which X and Y failed to get set. We could now write:

```
'(_ (X Y) Z _) ←• '(A B (C D) E F)
```

which would set

X to C
Y to D
Z to E.

Note: neither '(_ (X Y) Z) nor '((X Y) Z _) would work, because in the first case the RHS would have to be in the form(...(* *) *), which it isn't; and in the second case it would have to be in the form ((* *) * ...), which it also isn't.

The user should experiment with the decomposition assignment to make sure he understands its operation.

Examples of assignment expressions:

Regular assignments:

```
X ← A + 10                     →       (SETQ X (PLUS A 10))
X ← Y ← Z ← NIL                →       (SETQ X (SETQ Y (SETQ Z NIL)))
X ← A + B ← 10                 →       (SETQ X (PLUS A (SETQ B 10)))
X ← A * B ← C + D              →       (SETQ X (TIMES A (SETQ B (PLUS C D))))
NOT ATOM X ← READ()            →       (NOT (ATOM (SETQ X (READ))))
NOT ATOM X & READ()            →       (AND (NOT (ATOM X)) (READ))
NULL A ← B ← FOO(X)            →       (NULL (SETQ A (SETQ B (FOO X))))
NULL A | B | FOO(X)            →       (OR (NULL A) B (FOO X))
NULL(A | B | FOO(X))           →       (NULL (OR A B (FOO X)))
A ← BEGIN
       NEW TEMP;               →       (SETQ A (PROG(TEMP)
       TEMP ← READ();                                (SETQ TEMP (READ))
       PRINT 'START;                                  (PRINT (QUOTE START)),
       RETURN TEMP                                    (RETURN TEMP)))
    END
```

Array assignments:

```
X(1) ← A + 10                  →       (STORE (X 1) (PLUS A 10))
A(I,J) ← FOO(X)                →       (STORE (A I J) (FOO X))
A(0) ← A(1) ← NIL              →       (STORE (A 0) (STORE (A 1) NIL))
X(1) ← A + B(0) ← 10           →       (STORE (X 1) (PLUS A (STORE (B 0) 10)))
A(I,FOO(J),K+1) ← T            →       (STORE (A I (FOO J) (ADD1 K)) T)
A(1,2,3,4,5) ← 'FIVE_D         →       (STORE (A 1 2 3 4 5) (QUOTE FIVE_D))
NOT ATOM X(1) ← READ()         →       (NOT (ATOM (STORE (X 1) (READ))))
NOT ATOM X(1) & READ()         →       (AND (NOT (ATOM (X 1))) (READ))
NULL A(1) ← B ← FOO(X)         →       (NULL (STORE (A 1) (SETQ B (FOO X))))
NULL A(1) | B | FOO(X)         →       (OR (NULL (A 1)) B (FOO X))
NULL(A(1) | B | FOO(X))        →       (NULL (OR (A 1) B (FOO X)))
```

Index assignments:

Suppose L = (A B (C D) E F).

```
L[1] ← 1          would set      L = (1 B (C D) E F)
L[2] ← 1          would set      L = (A 1 (C D) E F)
L[3] ← 1          would set      L = (A B 1 E F)
L[3,1] ← 1        would set      L = (A B (1 D) E F)
L[8] ← 1          would set      L = (A B (C D) E F NIL NIL 1)
L[2,3,2] ← 1      would set      L = (A (NIL NIL (NIL 1)) (C D) E F)
```

Decomposition assignments:

Suppose L = (A B (C D) E F).

```
                                          would set
L ←• '(1 2 3)                               A = 1
                                            B = 2
                                            C = D = E = F = NIL

L ←• '(1 2 (3) 4)                           A = 1
                                            B = 2
                                            C = 3
                                            E = 4
                                            D = F = NIL

L ←• '(1 2 (3 4) 5 6)                       A = 1
                                            B = 2
                                            C = 3
                                            D = 4
                                            E = 5
                                            F = 6

'(X Y Z)          ←• '(A B (C D) E F)       X = A
'(X Y Z _)        ←• '(A B (C D) E F)       Y = B
'(_ X Y Z _)      ←• '(A B (C D) E F)       Z = (C D)

'(_ X Y Z)        ←• '(A B (C D) E F)       X = (C D)
                                            Y = E
                                            Z = F

'(_ (X _) Y Z) ←• '(A B (C D) E F)          X = C
                                            Y = E
                                            Z = F

'(_ X)            ←• '(A B (C D) E F)       X = F

'(_ (_ X) _)      ←• '(A B (C D) E F)       X = D

'(X _ Y)          ←• '(A B (C D) E F)       X = A
                                            Y = F
```

Again, I wish to emphasize that if a decomposition assignment expression is used in a compiled program, all the variables in the template (the left-hand side) must be declared SPECIAL.

SEMANTICS - SECTION 3.12

```
<function_call>  ::=  <identifier>             ( <argument_list> )
                  |   <LAMBDA_expression> ;    ( <argument_list> )

<argument_list>  ::=  <expression> (, <expression>)*
                  |   <empty>
```

A functioncall is an identifier (a function name) or a LAMBDA expression (a function body) followed by an argument list enclosed in parentheses. An argument list is any number of expressions, possibly none, separated by commas.

Little need be said about this. Essentially the only difference between this and the LISP way of writing function calls is that the function name has been brought outside the parentheses. Also the arguments are separated by commas. The arguments may be any arbitrary expressions.

Examples of function calls:

```
FOO(X)                      →        (FOO X)
FOO(X,Y,Z)                  →        (FOO X Y Z)
FOO()                       →        (FOO)
FOO(A+B, C)                 →        (FOO (PLUS A B) C)
FOO(IF A THEN B ELSE C)     →        (FOO (COND (A B) (T C)))
```

The same function calls, written as LAMBDA expressions:

```
LAMBDA(L); FOO(L); (X)      →        ((LAMBDA (L) (FOO L)) X)

LAMBDA (A,B,C);
  FOO(A,B,C);               →        ((LAMBDA (A B C)(FOO A B C)) X Y Z)
(X,Y,Z)

LAMBDA(); FOO(); ()         →        ((LAMBDA NIL (FOO)))

LAMBDA (X,Y,Z);
  FOO(X+Y, Z);              →        ((LAMBDA (X Y Z) (FOO (PLUS X Y) Z)) A B C)
(A,B,C)

LAMBDA (L); FOO(L);         →        ((LAMBDA (L) (FOO L)) (COND (A B) (T C)))
(IF A THEN B ELSE C)
```

, SEMANTICS - SECTION 3.13

```
<index_expression>   ::= <simple_expression> [ <argument_list> ]

<argument_list>      ::= <expression> (, <expression>)*
                       | <empty>
```

An index expression is a simple expression, followed by an argument list enclosed in square brackets [], An argument list is any number of expressions (possibly none) separated by commas,

The MLISP index expression fills a critical deficiency in LISP: there is no easy way to reference an arbitrary cell in a list, CAR will obtain the first element, CADR the second, CADDADDDDR the third element in the fifth element of the list, etc, But CADDADDDDR is neither (1) very understandable, not (2) variable, The latter is important since it occasionally happens that the user does not know until run-time which element of a list he will wish to access,

The MLISP index expression eliminates these objections, L[5,3] is the same as CADDADDDDR, but it is a good deal more readable, Furthermore, the index arguments may contain variables, in fact expressions: L[N], L[I,J,K], L[2*N], etc, The index expression, then, is a generalized version of CAR, (A generalized version of CDR also exists (SUFLIST) and is explained in SECTION 5.2 ,)

When index expressions are compiled, they are expanded by macros into highly optimized code:

```
        L[5,3]                    (CADDAR (CDDDDR L))
```

This insures that the execution of index expressions will be very efficient in compiled programs, In interpreted programs, it is more efficient to issue a call on a run-time function,

Examples of index expressions:

Suppose L = (A B (C (D E) F) G H),

```
L[1]                = A            →       (CAR L)
L[2]                = B            →       (CADR L)
L[3]                = (C (D E) F)  →       (CADDR L)
L[3,1]              = C            4       (CAADDR L)
L[3,2,1]            = D            →       (CAAR (CDADDR L))
'(A B C)[3]         = C            →       (CADDR (QUOTE (A B C)))
GET(X,'VALUE)[2]                   →       (CADR (GET X (QUOTE VALUE)))
(L1 @ L2)[1,2]                       →     (CADAR (APPEND L1 L2))
(FOR NEW I IN L COLLECT <CAR I>)[2*N, M/3 + 1]
```

. SEMANTICS - SECTION 3.14

```
<list_expression>   ::=  < <argument_list> >

<argument_list>     ::=  <expression> (, <expression>)*
                      |  <empty)
```

A list expression is a left angled bracket (<), followed by an argument list, followed by a right angled bracket (>), An argument list is any number of expressions, possibly none, separated by commas,

This is the MLISP equivalent of the LISP LIST function: <A,B,C> Is translated Into (LIST A BC),<A,B,<C,D>,E,F> Into (LIST A B (LIST C D) E F), etc, Angled brackets are used to make lists concise and to cut down further' on the number of parentheses needed, As with function calls, the arguments inside the list brackets may be any arbitrary expressions,

Examples of list expressions:

```
<>                          •       (LIST)
🖫✌✇                         →       (LIST A)
<A,B,C>                     →       (LIST A B C)
<A,<B>,C>                   →       (LIST A (LIST B) C)
<'A,B CONS C CONS D>        •       (LIST (QUOTE A) (CONS B (CONS C D)))
<X+10, <<Y>>, NIL>          →       (LIST (PLUS X 10) (LIST (LIST Y)) NIL)
<IF A THEN <B> ELSE C>      →       (LIST (COND (A (LIST B)) (T C)))
```

. SEMANTICS - SECTION 3.15

```
<quoted_expression>   ::=  ' <s-expression>

<s-expression>        ::=  <atom>
                       |   ()
                       |   ( <s-expression> . <s-expression> )
                       |   ( <s-expression> {(,) <s-expression>}* )
```

A quoted expression is the quote mark (') followed by an s-expression. An MLISP s-expression IS just the same as a LISP s-expression, except that each identifier in it must be a legal MLISP identifier. In particular, any special characters (characters which are not MLISP letters or digits) must have the literally character (?) in front of them.

Note that there is one fewer level of parentheses needed with the MLISP quoted expression than with the LISP QUOTE function. This is part of the effort to cut down on the number of parentheses required.

Examples of quoted expressions:

```
'A                          →     (QUOTE A)
'NIL                        →     (QUOTE NIL)
'0                          →     (QUOTE NIL)
'(A B C)                    →     (QUOTE (A B C))
'(A,B,C)                    →     (QUOTE (A B C))
'(a b c)                    →     (QUOTE (a b c))
'(A.B)                      →     (QUOTE (A.B))
'(A B ?*C?* O E)            →     (QUOTE (A B *C* D E))
'(A, 16.0, (E,F), 0)        →     (QUOTE (A 16.0 (E,F) 0))
'(A (B,C) ?= D , E)         →     (QUOTE (A (B,C) = D , E))
```

. SEMANTICS - SECTION 3.16

```
<identifier>            ::=  <letter> ( ⊂<letter>, <digit>⊃ )*

<letter>                ::=  ⊂A, B,  C,..., Z, a, b, c,..., z, _, :, !⊃
                         |   <literally_character> <any_character except %>

<literally_character>   ::=  ?
```

An identifier is an MLISP letter followed by any number of MLISP letters or digits. An MLISP letter is any of the upper or lower case letters of the alphabet, or an underbar (_), colon (:) or exclamation point (!), or any character except % preceded by the literally character. The literally character is a question mark (?). The comment character (%) may not be included because LISP 1.6 won't allow it to be used as anything except the start or end of a comment.

Underbar, colon and exclamation point are considered to be letters so that the user can easily create unusual names for variables. The literally character is a flag to the translator to take the next character literally and consider it to be a letter, even if the next character would ordinarily have a different meaning to MLISP. This enables the user to include virtually any character except % in variable names. However, the user must be sure that his LISP system won't object to any of the characters in his identifiers. Note: all of the functions and variable names used by the MLISP translator begin with an ampersand (&), so it is unwise to use such names.

Examples of identifiers:

```
X                                    →     X
X1                                   •     X1
AVERYLONGSTRINGOFLETTERS             →     AVERYLONGSTRINGOFLETTERS
A_VERY_LONG_STRING_OF_LETTERS        →     A_VERY_LONG_STRING_OF_LETTERS
x                                    →     x
x1                                   →     Xl
averylongstringofletters             →     averylongstringofletters
a_very_long_string_of_letters        →     a_very_long_string_of_letters
UPPER_and_lower_case_IDENTIFIER      →     UPPER_and_lower_case_IDENTIFIER

DSK:                                 →     DSK:
TTY:                                 →     TTY:
!SYSTEM_VARIABLE_357a                →     !SYSTEM_VARIABLE_357a
33                                   4     ?
?)                                   •     )
?1                                   →     1   (an identifier, not a number)
AB?*C?*DE                            →     AB*C*DE
?*! : _?#?@?$?≠?(?)?[?]?<?>          →     *!:_#@$≠()[]<>
```

, SEMANTICS - SECTION 3,17

```
<number>          ::=  <integer>
                   |   OCTAL <octal_integer>
                   |   <real>

<integer>         ::=  <digit> (<digit>)*

<digit>           ::=  ⊂0,1,2,3,4, 3, 6, 7, 8, 9⊃

<octal_integer>   ::=  <octal_digit> (<octal_digit>)*

<octal_digit>     ::=  ⊂0,1,2, 3, 4, 3, 6, 7⊃

<real>            ::=  <integer> <exponent>
                   |   <integer> , <integer> (<exponent>)

<exponent>        ::=  E (⊂+, -⊃) <integer>
```

Three types of numbers are permitted in MLISP: integers (base 10), integers preceeded by the reserved word OCTAL (base 8), and real numbers (base 10 again), An integer is any sequence of digits, A real number is either an integer followed by an exponent or two integers separated by a decimal point, optionally followed by an exponent, An exponent is the letter E, optionally followed by a plus or minus sign, followed by an integer, There should never be spaces between any of the parts of a number, except after the word OCTAL,

All numbers are taken to be decimal numbers unless preceeded by the word OCTAL, Octal numbers are included because they are used in many computer applications, Exponents provide a compact way of representing very large or very small real numbers, Only integer exponents are allowed, but they may be either positive or negative,

Plus (+) and minus (-) signs are not part of the syntax tar numbers (except in the exponent), Plus and minus signs are delimiters, and they are treated as either prefix or infix operators by the translator, Examples of numbers:

1 → 1
10 → 10
145968 → 145968

987,005 → 9,87005E2
13E+4 → 1,3E5
0,1 → 1,0E-1
0,000123E-5 → 1,23E-9

OCTAL 10 → 8 (decimal)
OCTAL 144 → 100 (decimal)
OCTAL 777777 → 262143 (decimal)

-145,12 → (MINUS 1,4512E2)
X-145,12 → (DIFFERENCE X 1,4512E2)
+98765,43210 → 9,87654321E4
x+98765,43210 → (PLUS X 9,87654321E4)

Note: .1 is not allowed; use 0,1 instead.

, SEMANTICS - SECTION 3,18

```
<string>  ::=  "(<any_character except " or %>)* "
```

A string Is a string quote ("), followed by any sequence of characters except the string quote or %, followed by 4 second string quote,

Strings are a special MLISP data structure Introduced primarily to facilitate Input/output, Several string manipulation features are Included In MLISP to make string handling easy, These are described in SECTION 5.1, However, MLISP Is not a string-manipulation language; It is a list-processing and symbol-manipulation language. Most of the string-handling routines are fairly time consuming, requiring an execution time proportional to the length of the string(s) Involved'; Therefore, If possible limit string manipulation to Input/output operations, or at least to operations which are not performed often, If it Is necessary to do a lot of string manipulating, the user should consider using some other, more suitable, language, since MLISP processes strings inefficiently,

String are stored by LISP 1,6 as un-INTERNED (i,e, not on the OBLIST) atoms having a print name consisting of the characters in the string, and including both string quotes,

Examples of strings:

```
""                          →       (QUOTE "" )
"THISIS A STRING"           →       (QUOTE "THIS IS A STRING")
"This is also a string,"→           (QUOTE "This la alsoastring,")
"123,;=<>()?;←"             •       (QUOTE "123,:=<>()?;←")
"        "                  →       (QUOTE "        ")
```

USER OPERATION OF MLISP - SECTION 4.1

This section tells the user how to get an MLISP program running.

(A) Translating MLISP Programs

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

MLISP - a core image contains LISP and MLISP.

MLISPC - a core image containing LISP, MLISP, PPRINT (the "pretty-print" functions), and the LISP compiler.

These core images may be loaded by typing:

```
        R MLISP          or        R MLISP  <core_size>
and
        R MLISPC         or        R MLISPC <core_size>
```

The core size of MLISP is 25K, and of MLISPC 35K. These should be sufficient to handle all but the largest programs. If not, a larger core size will have to be specified.

The MLISP core image should be used if the user wants to translate his MLISP program and then execute it. The MLISPC version should be used only if the user wants to translate his MLISP program and then compile it or pretty-print out its LISP translation. For large (debugged) programs, the most efficient use of core is achieved by compiling the MLISP program with MLISPC, and then reading the compiled code into a "fresh" LISP system (i.e. containing nothing else but LISP). Compiling the program has the following advantages:

(1) The program runs about 10 times faster compiled than interpreted.

(2) MLISPC incorporates some elaborate macros which expand FOR-loops, WHILE-loops, UNTIL-loops and index expressions into highly optimized code. This further speeds up their execution. MLISP is very compiler oriented: by far the most efficient execution of MLISP meta-expressions is by compiled code.

(3) Compiled code requires less space than the corresponding list-structure interpreted code.

(4) Function definitions don't have to be marked by the garbage collector every time a garbage collection occurs (a significant time savings for large programs).

To avoid confusion, two facts should be kept In mind when using MLISP:

(a) In WRITING your program, you will be communicating with MLISP. All expressions in the program must be legal MLISP expressions.

(b) In RUNNING your program, you will be communicating with LISP. All expressions to be executed, read or printed must be legal LISP expressions.

After the user has loaded a core image by typing one of the two commands above, he may begin translating his MLISP program by calling the top level function named, you guessed it, "MLISP". "MLISP" Is an FEXPR which takes from 1 to 4 arguments. These arguments will be explained by examples. The full command is:

```
(MLISP (<device>) <file_name> (⊂T, NIL, NIL NIL⊃) )
```

where () and ⊂⊃ mean "optional" and "alternatives" respectively. A <device> is either a physical device like a disk or dec tape (e.g. DSK: or DTA1:) or a project-programmer pair representing a disk area (e.g. (1,DAV) represents [1,DAV]).

Examples of the too level function "MLISP":

.R MLISP *(MLISP FOO)	would translate and execute a program on the disk file FOO
.R MLISP *(MLISP DSK: FOO)	would do exactly the same thing
.R MLISP *(MLISP (1,DAV) FOO)	would translate and execute a program on DSK:FOO[1,DAV]
.R MLISPC *(MLISP FOO T)	would translate a program on DSK:FOO and compile It onto DSK:FOO,LAP
.R MLISPC *(MLISP FOO NIL)	would translate a program on DSK:FOO and pretty-print the LISP translation onto DSK:FOO.LSP
.R MLISPC *(MLISP FOO NIL NIL)	would do the same thing, except that the expansi on of all LISP and MLISP macro8 is suppressed, Ordinarily, all macros (FOR-loop macros, PLUS, etc,) are expanded before printing, which enables tha user to see exactly what coda will be executed,
.R MLISPC *(MLISP DSK: FOO NIL NIL)	would do exactly the same thing
.R MLISP *(MLISP (FOO.BA$_Z$))	would translate and execute aprogram on DSK:FOO.BA$_Z$
.R MLISP ● ☎💣●●💧♐❼ DTA1:(FOO,BAZ))	would translate and execute a program on DTA1:FOO,BAZ
.R MLISPC *(MLISP MTA0: (FOO,BAZ) T)	would translate a program on MTA0:FOO.BAZ and compile It onto DSK:FOO,LAP

(8) Translating Under Program Control

It is sometimes desireable to call the MLISP translator under program control, This is made possible "by the special MLISP function "MTRANS", a function of no arguments, Calling MTRANS has the following effects:

(1) An MLISP <expression> is read from the currently selected input device, The first character read should be the first character in the expression, An MLISP <expression> differs from an MLISP <program> only in that the last character need not be a period; it may be any suitable expression-stopping character, usually a semicolon (;),

(2) The LISP translation is returned as the value of MTRANS,

The function "MLISP" should not be called from within a program, since it has several side effects, which are generally undesireable in a program; for example, the function RESTART is redefined, MTRANS has no side effects,

Note: If MTRANS is called, the entire MLISP translator must be available, This means that programs using MTRANS should only be run interpreted,

(C) Loading Compiled Programs

There is a file called UTILS on the system area of the disk containing run-time functions. This file must be loaded if the user has either compiled his MLISP program onto a .LAP file or pretty-printed it onto a .LSP file. UTILS is already loaded into both the MLISP and MLISPC core images, so that if the user simply wants to translate and run his MLISP program, the run-time functions will be available.

To read in an MLISP program after it has been compiled by MLISPC, set up a LISP system with sufficient Binary Program Space to hold the compiled code, and then type:

```
(INC (INPUT DSK: (<file_name>.LAP) SYS: UTILS))
```

The file UTILS should always be read in last since one of the things it does is set IBASE and BASE (the input and output radicies for numbers) to 10 (i.e. decimal). Thereafter, all numbers read or written will be interpreted as decimal numbers. The user should be careful to set IBASE to 8 (i.e. octal) if he wants to read in more LAP code, since LAP expects its numbers to be in octal form, and then reset IBASE to 10 afterwards.

the following sequence would translate and compile an MLISP program on the disk file FOO, and then read in the compiled code:

```
.R MLISPC
*(MLISP FOO T)

... <MLISP and compiler typeout> ...

***END-OF-RUN***
*↑C


.R LISP <coresize>

ALLOC?   ... <allocation> ...

*(INC (INPUT DSK: (FOO.LAP) SYS: UTILS))

... <LAP typeout> ...

*
```

, USER OPERATION OF MLISP - SECTION 4,2

This section is only for those hardy souls attempting to reconstruct MLISP on a LISP 1,6 system, Below is the sequence of commands necessary to reassemble both the MLISP and the MLISPC core images. In SECTION 4,3 is listed the contents of the various MLISP source files,

To Reconstruct MLISP:

```
.R LISP 24

ALLOC? Y
FULL WDS=2000
BIN,PROG,SP=8000
SPEC,PDL=_
REG, PDL=_
HASH=_

AUXILIARY FILES?Y
SMILE?_
ALVINE?_
TRACE?,
LAP?Y
DECIMAL?_

*(INC (INPUT DSK:
   (MLISP,LAP)
   (RUNFN1,LAP)
   (RUNFN2,LAP)
   MINIT
   SETQS))

...<type out>...

*(SCANNER1INIT)
**SCAN$

LOADER 1K CORE

*(SCANNER2INIT)
NIL

*↑C

.SAVE QSK MLISP
```

To Reconstruct MLISPC:

```
.RLISP 34

ALLOC? Y
FULL WDS=3000
BIN,PROG,SP=23000
SPEC,PDL=_
REG, PDL=_
HASH=_

AUXILIARY FILES?Y
SMILE?_
ALVINE?_
TRACE?,
LAP?Y
DECIMAL?_

*(INC (INPUT DSK:
   (MLISP,LAP)
   (MACROS,LSP)
   (MACRO1,LAP)
   (RUNFN2,LAP)
   (PPRINT,LAP)
   (COMPLR,LAP)
   MINIT
   SETQS))

...<type out>...

*(SCANNER1INIT)
**SCAN$

LOADER 1K CORE

*(SCANNER2INIT)
NIL

*↑C

.SAVE OSK MLISPC
```

The correct core images will now be saved under "MLISP" and "MLISPC". A little explanation about these two sequences Is necessary. The underbar (_) In the first few lines represents a space; this merely instructs LISP to use the standard allocation. The line reading:

```
(INC (INPUT DSK: (MLISP,LAP) ,,,)
```

assumes that all of the LAP files listed have been compiled by the LISP compiler. The file COMPLR should be the LISP compiler itself.

The line **SCAN% ($ stands for ALTMODE) loads the MLISP scanner package, which must have been compiled by MACRO and be in .REL format.

If the machine language scanner Is not to be used, then the LISP scanner listed In SECTION 7.3 should be compiled by the LISP compiler and read In with the other LAP files. Note: all LAP files must be read before the file SETQS, because SETQS changes IBASE, the Input radix for numbers, from 8 (octal) to 10 (decimal). LAP expects IBASE to be 8.

If the LISP scanner Is used, the following lines should be omitted:

```
*(SCANNER1INIT)
**SCAN$

LOADER 1K CORE

*(SCANNER2INIT)
NIL
```

, USER OPERATION OF MLISP - SECTION 4.3

This is a reference file of the MLISP source files.

File	Contents
MLISP	The MLISP translator functions -- In LISP
MINIT	Initialization for the MLISP translator (reserved words, abbreviations, precedences, etc.) -- In LISP
SETQS	Initialization of the MLISP globally-defined atoms -- In LISP
RUNFN1	&FOR, &DO, &WHILE, &INDEX -- In LISP
RUNFN2	PRELIST, SUFLIST, STR, STRP, STRLEN, AT, CAT, SEQ, SUBSTR, PRINTSTR, NEQ, NEQUAL, LEQUAL, GEQUAL -- In LISP
MACROS	&FOR, &DO, &WHILE, &INDEX, NEQ, NEQUAL, LEQUAL, GEQUAL -- all macros, In MLISP
MACRO1	Macro-expanding functions for the file 'MACROS' -- In MLISP
PPRINT	Functions for pretty-printing LISP expressions -- In MLISP
MEXPR	LISP-to-MLISP convertor -- In MLISP
UTILS	RUNFN2.LAP, SETQS -- This may be assembled by compiling the file RUNFN2 and adding the file SETQS to it.
SCAN.MAC	The machine language scanner for MLISP -- In DEC MACRO

RUN-TIME FUNCTIONS - SECTION 5.1

This section describes the string-handling functions of MLISP. Other run-time functions available to the user are described in the next section. Strings are described in SECTION 3.18; they exist primarily to facilitate input/output. To make string handling easy, MLISP includes the following set of primitives.

STR (sexp) - "STRINGIFY"
This takes one argument, which may be any s-expression, and returns a string containing the charactors in that s-expression (including spaces and parentheses).

STRP (sexp) - "STRING PREDICATE"
This takes one argument, which may be any s-expression. Returns TRUE if the s-expression is a string, NIL otherwise.

STRLEN (string) - "STRING LENGTH"
This takes one argument, a string, and returns an integer equal to the number of characters in the string (not counting the string quotes).

AT (string) - "ATOMIZE"
This takes one argument, a string, and returns an atom having a printname made up of the characters in the string (not including the string quotes).

CAT (string1, string2) - "CONCATENATE"
This takes two arguments and returns a string made up of their concatenation. The arguments need not be strings. If either argument is not a string, it is first converted to one, and then the concatenation is carried out. CAT, being a function of two arguments, may be used as an infix:

```
  string1 CAT string2.
```

SEQ (string1, string2) - "STRING EQUAL"
This takes two arguments, both strings, and returns T if they are identical, NIL otherwise. The LISP function EQ cannot be used because strings are atoms which are not on the OBLIST. As with CAT, SEQ may be used as an infix:

```
     string1 SEQ string2.
```

SUBSTR (string, start, length) - "SUBSTRING"
This takes three arguments, the first being a string and the other two being integers. It returns a substring of the first

argument beginning with the character in position "start" (counting from 1) and continuing for "length" characters. "length" need not be a number; if it is not, then the rest of the string is taken,

PRINTSTR (string) - "PRINT STRING"

This takes one argument, a string, and prints it on the current output device without the string quotes, followed by a carriage return. The value of PRINTSTR is the value of its argument (the same as with PRINT).

Examples of the string-handling functions:

```
STR 'STRING                          =       "STRING"
STR "STRING"                         =       "STRING"
STR '(A (B,C) D)                     =       "(A (B , C) D)"

STRP "THIS IS A STRING,"             =       T
STRP '(THIS IS NOT ONE)              =       NIL
STRP ""                              =       T

STRLEN "THIS IS A STRING,"           =       17
STRLEN STR 'STRING                   =       6
STRLEN ""                            =       0

AT "STRING"                          =       STRING
AT "THIS IS A STRING,"               =       THIS/IS/A/ STRING/,
AT ""                                =       Illegal

STR AT "THIS IS A STRING,"           =       "THIS IS A STRING,"
AT STR 'THIS? IS? A? STRING?,        =       THIS/ IS/ A/ STRING/,

"THIS IS A " CAT "STRING,"           =       "THIS IS A STRING,"
"THIS IS A " CAT 'STRING?,           =       "THIS IS A STRING,"
"A PERIOD " CAT "(,)"                =       "A PERIOD (,)"
"A PERIOD "CAT'(?,)                  =       "A PERIOD (.)"
"A PERIOD " CAT <PERIOD>             =       "A PERIOD (.)"

"STRING" SEQ "STRING"                =       T
"STRING" SEQ STR 'STRING             =       T
"STRING" SEQ "STRING,"               =       NIL

SUBSTRV'THIS IS A STRING,",6,4)           =        "IS A"
SUBSTRVTHIS IS A STRING,",100,5)          =        ""
SUBSTR("THIS IS A STRING,",5,100)         =        " IS A STRING,"
SUBSTR("THIS IS A STRING,",5,'REST)       =        " IS A STRING,"

PRINTSTR "A STRING,"   prints   A STRING,       value = "A STRING,"
PRINT "A STRING,"      prints   "A STRING,"     value = "A STRING,"
```

. RUN-TIME FUNCTIONS - SECTION 5.2

This section describes some general-purpose routines that have been judged sufficiently useful to be included in the set of run-time functions available to the MLISP user. All of these functions are short and have been compiled, so that they require very little binary program space and almost no free storage. The functions NEQ, NEQUAL, LEQUAL and GEQUAL are expanded by macros when the MLISP program in which they occur is compiled. This makes using these functions in a compiled program very efficient.

PRELIST (list, integer) - "PREFIX OF LIST"

This takes two arguments, a list and an integer. PRELIST returns a list of the first "integer" elements of its first argument. If there are fewer than "integer" elements in it, PRELIST returns as many as it can (i.e., the whole list).

PRELIST may be abbreviated ↑ (up arrow): PRELIST(L,6) ≡ L↑6

SUFLIST (list, integer) - "SUFFIX OF LIST"

This takes the same two arguments as PRELIST: a list and an integer. SUFLIST returns a list formed by taking "integer" CDR's of its first argument. If it exhausts its first argument before it runs out of CDR's, it stops at NIL (i.e. it will return NIL).

SUFLIST is the "compliment" of PRELIST in the sense that:

```
PRELIST(L,N) @ SUFLIST(L,N) ≡ L
```

for all lists L and for all integers N. SUFLIST is a generalization of CDR:

```
COR L             ≡   SUFLIST(L,1)
COOR L            ≡   SUFLIST(L,2)
CODR   CDDDDR L   ≡   SUFLIST(L,6)
```

SUFLIST is more powerful than CDR because the second argument may be a variable (in fact, any expression), thereby permitting the user to defer until run-time his decision on how many CDR's to take.

SUFLIST may be abbreviated ↓ (down arrow); SUFLIST(L,6) ≡ L↓6

NEQ (sexp1, sexp2) - "NOT EQ"

This takes two arguments, which may be any s-expressions, and returns TRUE if they are not EQ to each other, NIL otherwise. The LISP translation of X NEQ Y:

```
(NEQ X Y)
```

is expanded by macros to:

```
(NOT (EQ X Y))
```

if it is compiled.

NEQUAL (sexp1, sexp2) - "NOT EQUAL"

This takes two arguments, which may be any s-expressions, and returns TRUE if they are not EQUAL to each other, NIL otherwise. The LISP translation of X NEQUAL Y:

```
(NEQUAL X Y)
```

is expanded by macros to:

```
(NOT (EQUAL X Y))
```

If it is compiled. NEQUAL may be abbreviated ≠ (not-equal sign).

LEQUAL (number1, number2) - "LESS THAN OR EQUAL"

This takes two arguments, which should be numbers, and returns TRUE if the first argument is less than or equal to the second one, NIL otherwise. The LISP translation of X LEQUAL Y:

```
(LEQUAL X Y)
```

is expanded by macros to:

```
(NOT (GREATERP X Y))
```

if it is compiled. LEQUAL may be abbreviated ≤ (less-than-or-equal sign).

GEQUAL (number1, number2) - "GREATER THAN OR EQUAL"

This is the converse of LEQUAL. It takes two arguments, which should be numbers, and returns TRUE if the first argument is greater than or equal to the second one, NIL otherwise. The LISP translation of X GEQUAL Y:

```
(GEQUAL X Y)
```

is expanded by macros to:

```
(NOT (LESSP X Y))
```

If it is compiled. GEQUAL may be abbreviated ≥ (greater-than-or-equal sign).

Examples of these run-time functions:

```
'(A B C D E) PRELIST 3                  =       (A B C)
'(A B C D E) ↑ 3                        =       (A B C)
'(A B C D E) ↑ 10                       =       (A B C D E)
'(A 6 C D E) ↑ 0                        =       NIL

'(A B C D E) SUFLIST 3                  =       (D E)
'(A B C D E) ↓ 3                        =       (D E)
'(A B C D E) ↓ 10                       =       NIL
'(A B C D E) ↓ 0                        =       (A B C D E)

'(A B C D E) ↑ 3 @ '(A B c D E) ↓ 3      = (A B C DE)
'(A B C D E) ↑ 10 @ '(A B C D E) ↓ 10    = (A B C D E)
'(A B C D E) ↑ 0 @ '(A B C D E) ↓ 0      = (A B C 0 E)

'(A B C D E)↓0 ≡   '(A B C D E)                 = (A B C D E)
'(A B C D E)↓1 ≡   CDR '(A B c D E)             =(B C D E)
'(A B C D E)↓2 ≡   CDDR '(A B C D E)            = (C D E)
'(A B C D E)↓3 ≡   CDDDR '(A B C D E)           = (D E)
'(A B C D E)↓4 ≡   CDDDDR '(A 8 C D E)          = (E)
'(A B C D E)↓5 ≡   CDR  CDDDDR '(AB C D E)      = NIL

'A NEQ '8                               =       T
'A NEQ '(A)                             =       T
'A NEQ 'A                               a       NIL

'(A (B,C)) NEQUAL '(A (B C))            =       T
'(A (B,C)) ≠ '(A (B C))                 =       T
'A         ≠ '(A)                       =       T
'(A (B,C)) ≠ '(A (B,C))                 =       NIL

10 LEQUAL 20                            =       T
10 ≤ 20                                 =       T
10 ≤ 10                                 =       T
10 ≤ 0                                  =       NIL

10 GEQUAL 20                            =       NIL
10 ≥ 20                                 =       NIL
10 ≥ 10                                 =       T
10 ≥ 0                                  =       T
```

. SAMPLE MLISP PROGRAM - SECTION 6.1

```
BEGIN

% This program is included to provide an example of the MLISP
 language, It examines several ways of writing the function REVERSE
in MLISP, REVERSE was chosen because it is familiar to most people;
it reverses the top level of a list: REVERSE '(A B C) = (C B A),

The function REVERSE may be written in many ways in MLISP, Some of
the . ways shown here are not too efficient, but they do serve to
illustrate different MLISP expressions, The method used in each
function is explained in a comment included with the function, %


%#################################################################%
%#######           DEFINE ALL THE REVERSE [UNCTIONS        ########%
%#################################################################%


% REVERSE1 Just calls REVERSE1a with the list to be reversed and NIL.
The NIL initializes REVERSE1a's second argument, %

EXPR REVERSE1 (L);   REVERSE1a(L,NIL);


% REVERSE1a does all the work for REVERSE1, It uses an IF expression
and a recursive call on itself, The reverse of L is built up in the
second argument RL, %

EXPR REVERSE1a (L,RL);
   IF NULL L THEN RL ELSE REVERSE1a(CDR L,CAR L CONS RL);



% REVERSE2 also uses an IF expression and a recursive call on itself.
In this clever but inefficient version, the reverse of the rest of
the list L is APPEND'ed (@) to a list containing the first element, %

EXPR REVERSE2 (L);
   IF NULL L THEN NIL ELSE REVERSE2(CDR L) @ <CAR L>;



% REVERSE3 is an FEXPR; the arguments to it are unevaluated, It uses
a FOR expression as follows: I is set to each member of the list L
and then is CONS'ed onto the reversed list RL, REVERSE3 does not use
recursion, %
```

```
FEXPR REVERSE3 (L);
   BEGIN NEW RL;        % PROG variables are initialized to NIL.%
      RETURN FOR NEW I IN L DO RL ← I CONS RL;
   END;
```

% REVERSE4 is an example of a FOR expression using a numerical increment. In the operation of the loop, I is incremented from 1 to the length of L. For each value, the I'th element of L is obtained by the index expression L[I] and then is CONS'ed onto the reversed list RL. %

```
EXPR REVERSE4 (L);
   BEGIN NEW RL;
      RETURN FOR NEW I←1 TO LENGTH L DO RL ← L[I] CONS RL;
   END;
```

% PROG1 is like PROG2, except that PROG1's value is the value of its first (rather than its second) argument. This is not a reverse function, but is used by reverse functions which follow. %

```
EXPR PROG1 (A,B); A;
```

% REVERSE5 is another FEXPR. It uses a WHILE expression as follows: while there is still something left in L, the next element is taken off and CONS'ed onto the reversed list RL. This does not use recursion. %

```
FEXPR REVERSE5 (L);
   BEGIN NEW RL;
      RETURN WHILE L DO PROG1(RL ← CAR L CONS RL,L ← CDR L);
   END;
```

% REVERSE6 uses an UNTIL expression (DO-UNTIL). The operation of this UNTIL-loop is roughly the same as that of the WHILE-loop in REVERSE5. The one difference is that since the body of the loop gets executed before testing if there is anything in L, an initial test must be included to take care of the trivial case where REVERSE6 is called with NIL as its argument. This does not use recursion. %

```
EXPR REVERSE6 (L);
   IF NULL L THEN NIL ELSE
   BEGIN NEW RL;
      RETURN DO PROG1(RL ← CAR L CONS RL,L ← CDR L) UNTIL NULL L;
   END;



% REVERSE7 uses a standard LISP function, MAPCAR,together with a
LAMBDA expression. The operation of this is very similar to that of
REVERSE3. %

FEXPR REVERSE7 (L);
   BEGIN NEW RL;
      MAPCAR(FUNCTION(LAMBDA(I); RL ← I CONS RL), L);
      RETURN RL;
   END;



% Of all the methods presented, REVERSE8 is the most unique to MLISP.
It uses a numerical FOR-loop, as does REVERSE4; In addition it uses
index expressions on both the left and right sides of the assignment
operator (←). The index expression on the left side retrieves the
I'th position in the reversed list RL, into which is placed the
LEN-N+1'st element of L. LEN is the length of L. The first index
expression is used to obtain a "cell" or POSITION in RL, while the
second index expression is used to obtain the ELEMENT which occupies
a position in L. %

EXPR REVERSE8 (L);
   BEGIN NEW RL,LEN;
      LEN ← LENGTH L;
      FOR NEW N←1 TO LEN DO RL[N] ← L[LEN-N+1];
      RETURN RL;
   END;



% The LISP translation of this program is listed in the following
section. It has been printed using a program called PPRINT, an
s-expression formatting (pretty-printing) Program. This program is
written in MLISP and is included with the MLISP system. (All of the
files in the MLISP system are listed in SECTION 4.3.) Note that
FOR-loops, WHILE-loops and UNTIL-loops have been expanded by macros
into in-line code. %


END.
```

```
, SAMPLE MLISP PROGRAM - SECTION 6.2

(DEFPROP REVERSE3
 T
*FEXPR)

(DEFPROP REVERSES
 T
*FEXPR)

(DEFPROP REVERSE7
 T
*FEXPR)

(DEFPROP REVERSE1
 (LAMBDA (L) (REVERSE1a L NIL))
EXPR)

(DEFPROP REVERSE1a
 (LAMBDA (L RL)
  (COND((NULL L) RL) (T (REVERSE1a (CDR L) (CONS (CAR L) RL)))))
EXPR)

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

(DEFPROP REVERSE3
 (LAMBDA (L)
  (PROG (RL)
        (RETURN
         (PROG (&V &LST1 I)
               (SETQ &LST1 L)
          LOOP (COND ((NOT &LST1) (RETURN &V)) (T NIL))
               (SETQ I (CAR &LST1))
               (SETQ&V(SETQRL (CONS I RL)))
               (SETQ &LST1 (CDR &LST1))
               (GO LOOP)))))
FEXPR)

(DEFPROP REVERSE4
 (LAMBDA (L)
  (PROG (RL)
        (RETURN
         (PROG (&V &LST1 &UPPER1 I)
               (SETQ &LST1 1.)
               (SETQ&UPPER1 (LENGTH L))
          LOOP(COND ((@GREAT &LST1 &UPPER1) (RETURN &V)) (T NIL))
               (SETQ I &LST1)
               (SETQ &V (SETQ RL (CONS (CAR (SUFLIST L (SUB1 I),) RL)))
               (SETQ &LST1(ADD1 &LST1))
```

```
                (GO LOOP)))))
EXPR)

 (DEFPROP PROG1
  (LAMBDA (A B) A)
EXPR )

 (DEFPROP REVERSE5
  (LAMBDA (L)
   (PROG (RL)
         (RETURN
          (PROG (&V)
           LOOP  (COND (L (SETQ &V
                                (PROG1 (SETQ RL (CONS (CAR L) RL))
                                       (SETQ L (COR L)))))
                      (T (RETURN &V)))
                (GO LOOP)))))
FEXPR)

(DEFPROP REVERSE6
  (LAMBDA (L)
   (COND ((NULL L) NIL)
         (T (PROG (RL)
                  (RETURN
                   (PROG (&V)
                    LOOP (SETQ &V
                                (PROG1 (SETQ RL (CONS (CAR L) RL))
                                       (SETQ L (CUR I.))))
                         (COND ((NULL L) (RETURN &V))
                               (T (GO LOOP))))))))))
EXPR)

(DEFPROP REVERSE7
 (LAMBDA (L)
  (FROG (RL)
        (MAPCAR (FUNCTION (LAMBDA (I) (SETQ RL (CONS I RL)))) L)
        (RETURN RL)))
FEXPR)

 (DEFPROP REVERSE8
  (LAMBDA (L)
  (PROG (RL LEN)
        (SETQ LEN (LENGTH L))
        (PROG (&V &LST1 &UPPER1 N)
              (SETQ &LST1 1,)
              (SETQ &UPPER1 LEN)
         LOOP (COND ((*GREAT &LST1 &UPPER1) (RETURN 8V)) (T NIL))
              (SETQ N &LST1)
              (SETQ &V
                    (PROG2 (SETQ RL
                                 (&REPLACE RL
                                           (LIST N)
```

```
                                             (SETQ &M001
                                                   (CAR
                                                    (SUFLIST
                                                     L
                                                     (*DIF LEN N))))))
                            &M001))
               (SETQ &LST1(ADD1 &LST1))
               (GO LOOP) )
         (RETURN RL)))
EXPR )
```

THE MLISP SCANNER - SECTION 7.1

The set of routines that returns the next "token" (identifier, number, special character, string) in the input stream is generally called the "scanner" for a language. It is true of almost every language that the majority of compilation time is spent in the scanner, since every character in a program has to be read in individually and some sequence of tests made on it. This is the plight of MLISP, and the best that can be done is to make the scanner as fast and efficient as possible. Lynn Quam at Stanford has developed a super fast, table-driven READ function for LISP 1.6. To this he has added a set of machine language functions which may be used to specify the precise syntax for a token returned by READ. These routines actually modify READ's internal character tables, thus giving the user a completely general table-driven scanner. The scanner for MLISP was obtained in this way. It has increased translation speed by a factor of three (translation speed is now 30004000 lines/minute). It has decreased the size of the translator as well, since using READ does not require any additional LISP functions.

Since there is no formal writeup on Quam's READ-modifying functions, the following is a reproduction of (parts of) Quam's informal description.

> LISP now uses a table driven scanner, whose table may be modified by the user for the purpose of implementing scanners for other languages. For simplicity, the functions for constructing the scanner table initially give an ALGOL type scanner; that is, the ALGOL definitions for identifiers, strings and numbers. The ALGOL table may be deviated from by using additional functions to include additional characters in identifiers, and to specify delimiters for strings.

(SCANINIT comment_start comment-end string_start string_end literally)

SCANINIT sets up the LISP scanner to be an ALGOL-type scanner with the special delimiters for comments and strings. MLISP calls (SCANINIT % % " " ?).

(LETTER x)

LETTER specifies to the scanner that x is an extra-letter, and thus allows x to be in an identifier. MLISP calls (LETTER _), (LETTER :), (LETTER !).

(IGNORE x)

IGNORE specifies to the scanner that x is not to be returned as a delimiter from SCAN, but instead will be ignored. However, x will still function as a separator between identifiers and numbers. MLISP calls (IGNORE BLANK), (IGNORE CR), (IGNORE LF), (IGNORE FF), (IGNORE VT), (IGNORE TAB), (IGNORE ALTMODE).

(SCAN)

SCAN reads an atom or delimiter and sets the value of the global variable SCNVAL to the value read, and returns a number corresponding to the syntactic type read, as follows:

Syntactic Type	Value of SCAN	Value of SCNVAL
<identifier>	0	the uninterned identifier
<string>	1	the string
<number>	2	the value
<delimiter>	3	the ASCII numerical value of the delimiter

(SCANSET)

SCANSET modifies the LISP scanner In READ according to the user specifications.

(SCANRESET)

SCANRESET unmodifies the LISP scanner to its normal state, and must be called before REAP will work properly once SCANSET is used.

, THE MLISP SCANNER - SECTION 7.2

BEGIN

% This program presents a set of functions which is equivalent to the MLISP scanner. It is for the reference of users wanting to implement MLISP on a LISP system without Quam's READ-modifying functions. In order to use these functions, the function &SCAN in the MLISP translator should be replaced by the &SCAN function below, and the other functions added where convenient. The functions below are written in MLISP, so their LISP translations would actually be used.

The scanner below places only two restrictions on the LISP system!

(1) There must be a READCH function, which reads the next character in the input stream and returns that character as its value.

(2) There must be a READLIST function, which takes as its argument a list of single characters and concatenates them, to form an atom.

These two functions are taken to be primitives, and they are used below without further explanation. &SCAN sets the global variables &SCANTYPE and &SCANVAL as follows;

Syntactic Type	Value of &SCANTYPE	Value of &SCANVAL
<identifier>	0	the identifier
<string>	1	the string
<number>	2	the number
<delimiter>	3	the delimiter

!NEXT_CHAR is always set to the next character in the input stream after the current token has been obtained.

%

```
SPECIAL !NEXT_CHAR,?&SCANTYPE,?&SCANVAL,?&X?&;


EXPR ?&SCAN ();
   IF NUMBERP !NEXT_CHAR THEN SCAN_NUMBER() ELSE
   IF LETTERP(!NEXT_CHAR) THEN SCAN_IDENTIFIER(NIL, !NEXT_CHAR) ELSE
   IF !NEXT_CHAR EQ DBQUOTE THEN SCAN_STRING(<DBQUOTE>,READCH()) ELSE
   IF IGNOREP(!NEXT_CHAR) THEN
      PROG2(DO NIL UNTIL ¬IGNOREP(!NEXT_CHAR ← READCH()), ?&SCAN()) ELSE
   IF !NEXT_CHAR EQ PERCENT THEN
      PROG2(DO NIL UNTIL READCH() EQ PERCENT & !NEXT_CHAR←READCH(),?&SCAN())
   ELSE SCAN_DELIMITER();
```

```
EXPR SCAN_IDENTIFIER (L,NEXT);
   IF NUMBERP NEXT | GET(NEXT,'LETTER) THEN
      SCAN_IDENTIFIER(NEXT CONS L, READCH()) ELSE
   IF NEXT EQ '?? THEN            % The MLISP literally character (?) %
      SCAN_IDENTIFIER(READCH() CONS SLASH CONS L, READCH())
   ELSE BEGIN
      ?&SCANTYPE ← 0;             % Identifier type, %
      ?&SCANVAL  ← READLIST REVERSE L;
      IF ?&X?& & GET(?&SCANVAL,'?&TRANS) THEN
      BEGIN     % This symbol has been DEFINE'ed as something else, %
         ?&SCANTYPE ← GET(?&SCANVAL,'?&TRANSTYPE);
         ?&SCANVAL  ← GET(?&SCANVAL,'?&TRANS);
      END;
      !NEXT_CHAR ← NEXT;          % Advanor !NEXT_CHAR, %
   ENDJ


EXPR SCAN-STRING (L,NEXT);
   IF NEXT NEQ DBQUOTE THEN SCAN_STRING(NEXT CONS L, READCH())
   ELSE BEGIN
      ?&SCANTYPE ← 1;             % String type, %
      ?&SCANVAL  ← READLIST REVERSE(DBQUOTE CONS L);
      !NEXT_CHAR ← READCH();      % Advance !NEXT_CHAR, %
   END;


EXPR SCAN_DELIMITER ();
   BEGIN
      ?&SCANTYPE ← 3;             %Delimiter typo. %
      ?&SCANVAL  ← !NEXT CHAR)    % Set ?&SCANVAL to the delimiter, %
      IF ?&X?& 8 GET(?&SCANVAL,'?&TRANS) THEN
      BEGIN     % This symbol has been DEFINE'ed as something else, %
         ?&SCANTYPE ← GET(?&SCANVAL,'?&TRANSTYPE);
         ?&SCANVAL  ← GET(?&SCANVAL,'?&TRANS);
      END;
      !NEXT_CHAR ← READCH();      % Advance !NEXT_CHAR, %
   END;


EXPR LETTERP (CHAR);    GET(CHAR,'LETTER) | CHAR EQ '??;


EXPR IGNOREP (CHAR);    GET(CHAR,'IGNORE);


EXPR SREAD (); PROG2(?&SCAN(),SREAD1());


EXPR SREAD1();
   IF ?&SCANVAL EQ LPAR & ?&SCANTYPE = 3 THEN                 % ( %
      PROG2(?&SCAN(),SREAD2())
   ELSE ?&SCANVAL;
```

```
EXPR SREAD2 0;
   IF ?&SCANVAL EQ RPAR 8 ?&SCANTYPE = 3 THEN NIL          % ) %
   ELSE BEGIN NEW X;
      X ← SREAD1();
      ?&SCAN();
      RETURN(X CONS SREAD3())
   END;

EXPR SREAD3 ();
   IF ?&SCANVAL EQ PERIOD & ?&SCANTYPE = 3 THEN            % , %
   BEGIN NEW X;                                 % We have a dotted pair (A,B) %
      X ← SREAD1();                             % Get the "B" part, %
     . ?&SCAN();                                % Get rid of the ) %
      RETURN X
   END
   ELSE SREAD2();
```

```
% Scanning numbers, %

EXPR SCAN-NUMBER ();
   BEGIN NEW !IVALUE,!ILENGTH,N,X;   SPECIAL !IVALUE,!ILENGTH;
      SCAN_INTEGER(!NEXT_CHAR, 0, 0);    % Stan an Integer, %
      N ← !IVALUE;                        % Save it, %

      IF !NEXT_CHAR EQ PERIOD THEN        % We have a decimal number, %
      BEGIN
         SCAN_INTEGER(READCH(), 0, 0);    % Scan the decimal part, %
         N ← N + !IVALUE/EXP(10,0,!ILENGTH);
      END;

      IF !NEXT_CHAREQ 'E THEN             % There is an exponent, %
      BEGIN
         !NEXT_CHAR ← READCH();           % See if there is a + or -. %
         IF !NEXT_CHAR EQ PLUSS THEN      % + %
            PROG2(X←10,0, !NEXT_CHAR←READCH()) ELSE
         IF !NEXT_CHAR EQ DASH THEN       % - %
            PROG2(X←0,10, !NEXT_CHAR←READCH())
         ELSE X←10,0;
         SCAN_INTEGER(!NEXT_CHAR,0,0);    % Now get the exponent, %
         N ← N * EXP(X,!IVALUE);
      END;
      % Now we've got the whole number, %
      ?&SCANTYPE ← 2;                     % Number type, %
      ?&SCANVAL  ← N;                     % Value of the number, %
                                          % !NEXT_CHAR is already set, %
   END;


EXPR SCAN-INTEGER (NEXT,N,LEN);           % Scan an Integer, %
   IF NUMBERP NEXT THEN SCAN_INTEGER(READCH(), N*IBASE+NEXT, LEN+1)
   ELSE BEGIN
      !IVALUE     ← N;                    % Value of the Integer, %
      !ILENGTH    ← LEN;                  % # digits In the Integer, %
      !NEXT_CHAR  . NEXT;                 % Advance !NEXT_CHAR. %
   END;


EXPR EXP (X,N);                           % An exponent function, %
   IF N = 0 THEN 1,0 ELSE                 % The exponent is 0, %
   IF N = 2*(N/2) THEN EXP(X*X, N/2)      % It is an even number, %
   ELSE X * EXP(X*X, (N-1)/2);            % Else odd, %
```

```
% Calling the following function will set up the propertylists
needed by the function above. %

EXPR SCANINIT ();
   BEGIN
      FOR NEW CHAR IN
         '(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
           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 _ ! !) DO
         PUTPROP(CHAR,T,'LETTER);
      FOR NEW CHAR IN <BLANK,CR,LF,FF,VT,TAB,ALTMODE> DO
         PUTPROP(CHAR,T,'IGNORE);

      !NEXT_CHAR ← BLANK;          % Start the scanner out with a blank. %
   END;


EXPR SCANSET ();          NIL;     % Dummy definitions. %

EXPR SCANRESET ();        NIL;


% The LISP translation of this program is listed in the following
section. It has been printed using a program called PPRINT, an
s-expression formatting (pretty-printing) program. This program is
written in MLISP and is included with the MLISP system. (All of the
files in the MLISP system are listed in SECTION 4.3.) Not. that
FOR-loops, WHILE-loops and UNTIL-loops have been expanded by macros
into in-line code. %

END.
```

```
. THE MLISP SCANNER - SECTION 7.3

(DEFPROP !NEXT_CHAR
 T
SPECIAL)

(DEFPROP &SCANTYPE
 T
SPECIAL)

(DEFPROP &SCANVAL
 T
SPECIAL)

(DEFPROP &X&
 T
SPECIAL)

(DEFPROP !IVALUE
 T
SPECIAL)

(DEFPROP !ILENGTH
 T
SPECIAL)

(DEFPROP 8SCAN
 (LAMBDA NIL
  (COND ((NUMBERP !NEXT_CHAR) (SCAN_NUMBER))
        ((LETTERP !NEXT_CHAR) (SCAN_IDENTIFIER NIL !NEXT_CHAR))
        ((EQ !NEXT_CHAR DBQUOTE) (SCAN_STRING (LIST DBQUOTE) (READCH)))
        ((IGNOREP !NEXT_CHAR)
         (PROG2 (PROG (&V)
                 LOOP (COND
                       ((NOT (IGNOREP (SETQ !NEXT_CHAR (READCH))))
                        (RETURN &V))
                       (T (GO LOOP))))
                (&SCAN)))
        ((EQ !NEXT_CHAR PERCENT)
         (PROG2 (PROG (&V)
                 LOOP (COND
                       ((AND (EQ(READCH) PERCENT)
                             (SETQ !NEXT_CHAR (READCH)))
                        (RETURN &V))
                       (T (GO LOOP))))
                (&SCAN)))
        (T (SCAN_DELIMITER))))
EXPR)

(DEFPROP SCAN_IDENTIFIER
 (LAMBDA (L NEXT)
  (COND
```

```
    ((OR (NUMBERP NEXT) (GET NEXT (QUOTE LETTER)))
     (SCAN_IDENTIFIER (CONS NEXT L) (READCH)))
    ((EQ NEXT (QUOTE ?))
     (SCAN_IDENTIFIER (CONS (READCH) (CONS SLASH L))(READCH)))
    (T (PROG NIL
             (SETQ &SCANTYPE 0,)
             (SETQ &SCANVAL (READLIST (REVERSE L)))
             (COND
              ((AND &X& (GET &SCANVAL (QUOTE &TRANS)))
               (PROG NIL
                     (SETQ &SCANTYPE (GET &SCANVAL (QUOTE &TRANSTYPE)))
                     (SETQ &SCANVAL (GET &SCANVAL (QUOTE &TRANS)))))
              (T NIL))
             (SETQ !NEXT_CHAR NEXT)))))
EXPR)

 (DEFPROP SCAN-STRING
  (LAMBDA (L NEXT)
   (COND
    ((NOT (EQ NEXT DBQUOTE)) (SCAN_STRING (CONS NEXT L) (READCH)))
    (T (PROG NIL
             (SETQ &SCANTYPE 1,)
             (SETQ &SCANVAL (READLIST (REVERSE (CONS DBQUOTE L))))
             (SETQ !NEXT_CHAR (READCH))))))
EXPR)

 (DEFPROP SCAN_DELIMITER
  (LAMBDA NIL
   (PROG NIL
         (SETQ &SCANTYPE 3,)
         (SETQ &SCANVAL !NEXT_CHAR)
         (COND
          ((AND &X& (GET &SCANVAL (QUOTE &TRANS)))
           (PROG NIL
                 (SETQ &SCANTYPE (GET &SCANVAL (QUOTE &TRANSTYPE)))
                 (SETQ &SCANVAL (GET 8SCANVAL (QUOTE &TRANS)))))
          (T NIL))
        . (SETQ !NEXT CHAR (READCH))))
EXPR)

 (DEFPROP LETTERP
  (LAMBDA (CHAR) (OR (GET CHAR (QUOTE LETTER)) (EQ CHAR (QUOTE ?))))
EXPR)

(DEFPROP IGNOREP
 (LAMBDA (CHAR) (GET CHAR (QUOTE IGNORE)))
EXPR)

(DEFPROP SREAD
 (LAMBDA NIL (PROG2 (&SCAN) (SREAD1)))
EXPR)
```

```
 (DEFPROP SREAD1
  (LAMBDA NIL
   (COND
    ((AND (EQ &SCANVAL LPAR) (EQUAL &SCANTYPE 3,))
     (PROG2 (&SCAN) (SREAD2)))
    (T &SCANVAL)))
EXPR)

 (DEFPROP SREAD2
  (LAMBDA NIL
   (COND
    ((AND (EQ &SCANVAL RPAR) (EQUAL &SCANTYPE 3.)) NIL)
    (T (PROG (X)
               (SETQ X (SREAD1))
               (&SCAN)
               (RETURN (CONS X (SREAD3)))))))
EXPR)

 (OEFPROP SREAD3
  (LAMBDA NIL
   (COND
    ((AND (EQ &SCANVAL PERIOD) (EQUAL &SCANTYPE 3,))
     (PROG (X) (SETQ X (SREAD1)) (&SCAN) (RETURN X)))
    (T (SREAD2))))
EXPR)

 (DEFPROP SCAN-NUMBER
  (LAMBDA NIL
   (PROG (!IVALUE !ILENGTH N X)
          (SCAN_INTEGER !NEXT_CHAR 0, 0,)
          (SETQ N !IVALUE)
          (COND ((EQ !NEXT CHAR PERIOD)
                  (PROG NIL-
                         (SCAN-INTEGER (READCH) 0, 0,)
                         (SETQ N
                                 (*PLUS N
                                         (*QUO !IVALUE
                                                (EXP 10,0 !ILENGTH))))))
                 (T NIL))
          (CON0 ((EQ {NEXT_ CHAR (QUOTE E))
                  (PROG NIL
                         (SETQ !NEXT_CHAR (READOH))
                         (COND
                          ((EQ !NEXT_CHAR PLUSS)
                           (PROG2 (SETQ X 10,0)
                                   (SETQ !NEXT_CHAR (READCH))))
                          ((EQ !NEXT_CHAR DASH)
                           (PROG2 (SETQ X 0,10000000)
                                   (SETQ !NEXT_CHAR (READCH))))
                          (T (SETQ X 10,0)))
                         (SCAN INTEGER !NEXT CHAR 0, 0,)
                         (SETQ N (*TIMES N (EXP X !IVALUE))))
```

```
                (T NIL))
        (SETQ &SCANTYPE 2,)
        (SETQ &SCANVAL N)))
EXPR)

 (DEFPROP SCAN_INTEGER
  (LAMBDA (NEXT N LEN?
   (COND ((NUMBERP NEXT)
          (SCAN INTEGER (READCH)
                        (*PLUS (*TIMES N IBASE) NEXT)
                        (ADD1 LEN)))
        (T (PROG NIL
                 (SETQ !IVALUE N)
                 (SETQ!ILENGTH LEN)
                 (SETQ !NEXT_CHAR NEXT)))))
EXPR)

 (DEFPROP    EXP
  (LAMBDA (X N)
   (COND ((EQUAL N 0,) 1,0)
        ((EQUAL N (*TIMES 2, (*QUO N 2,)))
         (EXP (*TIMES X X) (*QUO N 2,)))
        (T (*TIMES X (EXP (*TIMES X X) (*QUO (SUB1 N) 2,))))))
EXPR)

 (DEFPROP SCANINIT
  (LAMBDA NIL
   (PROG NIL
        (PROG (&V &LST1 CHAR)
              (SETQ &LST1
                    (QUOTE
                     (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
                      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
                      _ ! !)))
         LOOP (COND ((NOT &LST1) (RETURN &V)) (T NIL))
              (SETQ CHAR (CAR &LST1))
              (SETQ &V (PUTPROP CHAR T (QUOTE LETTER)))
              (SETQ &LST1 (CDR &LST1))
              (GO LOOP))
        (PROG (8V &LST1 CHAR)
              (SETQ &LST1 (LIST BLANK CR LF FF VT TAB ALTMODE))
         LOOP (COND ((NOT &LST1) (RETURN &V)) (T NIL))
              (SETQ CHAR (CAR &LST1))
              (SETQ &V (PUTPROP CHAR T (QUOTE IGNORE)))
              (SETQ &LST1 (CDR &LST1))
              (GO LOOP))
       (SETQ !NEXT_CHAR BLANK)))
EXPR)

 (DEFPROP SCANSET
  (LAMBDA NIL NIL)
EXPR)
```

```
(DEFPROP SCANRESET
 (LAMBDA NIL NIL)
EXPR)
```

. BIBLIOGRAPHY - SECTION 8

Enea, Horace, MLISP, Technical Report No, CS-92, Computer Science Department, Stanford University, 1968,

Hearn, Anthony C,, STANDARD LISP, Stanford Artificial Intelligence Laboratory Memo No, AI-90, Stanford University, 1969,

Hearn, Anthony C,, REDUCE, A PROGRAM FOR SYMBOLIC ALGEBRAIC COMPUTATION, Proc, SHARE XXXIV, 1970,

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,

Quam, Lynn, STANFORD LISP 1,6 MANUAL, Stanford Artificial Intelligence Laboratory Operating Note No. 28,3, Stanford University, 1969,

Weissman, Clark, LISP 1,5 PRIMER, Dickenson Publishing Company, Inc., Belmont, California, 1967.