STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-195

STAN-CS-73-356

MLISP 2

BY

DAVID CANFIELD SMITH

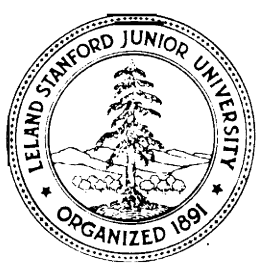
HORACE J. ENEA

SUPPORTED BY

NATIONAL INSTITUTE OF MENTAL HEALTH

MAY 1973

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
MEMO AIM-195

MAY 1973

COMPUTER SCIENCE DEPARTMENT
REPORT NO. CS-356

MLISP2

by

David Canfield Smith

Horace J. Enea

ABSTRACT: MLISP2 is a high-level programming language based on LISP. Features:

1. The notation of MLISP.
2. Extensibility---the ability to extend the language and to define new languages.
3. Pattern matching----the ability to match input against context free or sensitive patterns.
4. Backtracking--the ability to set decision points, manipulate contexts and backtrack.

This research is supported by Grant PHS MH 06645-12 from the National Institute of Mental Health.

Reproduced in the USA. Available from the National Technical Information Service, Springfield, Virginia 22151

TABLE OF CONTENTS

SECTION 1

Introduction

MLISPZ is a high-level, LISP-based programming language recently developed at the Stanford Artificial Intelligence Project. The language has been operational for: two years, and the developmental phase of it has been completed. This is a final report on the language.

MLISP2 is specially tailored for writing translators for other languages. To this end, two powerful control structures have been added to an ordinary LISP base: pattern matching and backtracking. This report serves the dual purpose of explaining our particular version and use of these control structures, as well as serving as a users' manual for anyone wanting to write MLISPZ programs,

Actually, MLISP2 is a transitional language. Laurence Tesler and the authors are presently implementing a language called LISP70 which will include and (for most applications) supersede MLISP2, MLISP and LISP. Therefore, perhaps it is worthwhile to briefly justify the current report, Many of the concepts developed in MLISPZ are being used in LISP70, though some of them are undergoing extensive revisions. But more importantly, MLISPZ is an extremely

effective translator writing system. It clearly isolates some general principles that may profitably be incorporated in other systems. This report concentrates on the nature of these principles, how they are implemented, and how they are most effectively used.

Since this report emphasizes the research aspects of MLISP2, the users' manual aspect necessarily is somewhat incomplete. This report is not a complete description of the MLISP2 language. Rather it is a supplement to the MLISP users' manual [6], and it only discusses in depth the differences (mainly additions) between MLISP2 and MLISP.

History

MLISP2 is the latest in a continuing development of list-processing programming languages. The progression, based on capabilities, is:

LISP → MLISP → MLISP2 → (LISP70),

where LISP70 has not been completed at the time of this writing. MLISP [2,6] is a programming language based on LISP [4]. MLISP programs are translated to LISP and then executed or compiled to LAP. The advantage of MLISP over LISP is primarily notational: the MLISP notation makes it easier to write and understand LISP programs. In addition, certain list-processing deficiencies in LISP are remedied (see the MLISP manual). MLISP2 is an extension of MLISP, originally created for the following reasons:

Section 1 Introduction

1. To make the syntax of MLISP more easily modifiable.
2. To provide a vehicle for easily implementing compilers for other languages.
3. To add backtracking as a control structure, making MLISP a more useful language for heuristic problems.

The MLISPZ and MLISP languages are separate and have separate capabilities. Since MLISP is simply a more convenient notation for LISP, it is suitable for exactly the same tasks as LISP. MLISPZ preserves the list-processing capabilities of LISP, but it has a substantially modified and augmented environment tailored for efficient backtracking and pattern matching, This extra overhead is unnecessary for simple list-processing tasks.

MLISPZ is mostly upwardly compatible with MLISP; MLISPZ differences are mainly in the form of additions to MLISP. We classify the differences between MLISP2 and MLISP as either "major" or "minor." The major changes modify the control structure or execution environment of MLISP: they substantially alter its capabilities, For example, the SELECT expression (backtracking) and the LET expression (extensibility) are major changes, The minor changes are modifications to MLISP, hopefully in the form of improvements, which do not substantially alter its capabilities but which make it more convenient to use. For example, the DOT notation and the RECOMPILE expression are minor changes.

The main productions needed to understand the ML/SP2 language are PROGRAM, EXPRESSION, PRIMARY, SIMPEX and BASIC. The main production needed to understand the extensibility mechanism in the language is LET. The main production needed to understand the backtracking facility is SELECT.

SECTION 2

Backtracking

MLISP2 makes heavy use of backtracking [3,7]. The pattern matcher (section 7) uses backtracking in its parsing algorithm. The SELECT expression (section 8) provides a means for the user to incorporate backtracking into his algorithms. Therefore, it is necessary to describe exactly what backtracking means in the MLISP2 language.

As was pointed out in [7], there is no universal agreement on the meaning of backtracking. Every implementation has produced a slightly different interpretation, Our view most closely follows Floyd's theoretical system [3] in its goals, though not in its implementation. Typically in heuristic programs there are points where several alternative strategies might be tried, with no certain knowledge of which one will be successful, In this situation the programmer wants to be able to try one out; but if it is unsuccessful; he wants to be able to pretend he had never tried it, select another alternative, and try that out. In this way he will either find a successful strategy or run out of alternatives. This is data backtracking, the restoration of changes to variables, Bobrow points out [1] that it is also sometimes useful to have

control and data backtracking separately programmable, though MLISP2 does not implement this.

Model

The state of a computation at any point is completely represented by a "state vector" consisting of the values of all variables in the program, plus system variables like the program counter, I/O pointers, etc. Every time a computation is begun with the same state vector, the results are identical. A "decision point" is a Point in the computation at which a copy of the state vector is saved (in memory or on secondary storage). In MLISP2 not the entire state vector is saved, just the "incremental state vector" -- those values that have changed since the vector was last saved, The process of restoring a copy of the state vector, thus wiping out all changes to variables since the copy was made, is called "backtracking". The complete state of a computation is restored to its value at the decision point, just as if nothing had been executed beyond that point. The only exception is that the program counter may be changed. so that execution picks up at a different place in the code.

The MLISP2 programmer may cause backtracking by invoking the intrinsic function FAILURE(). Executing FAILURE will cause the state

vector to be restored to the value it had when the last decision point was encountered. There is no failing to labels, as in some systems. Rather "failure" in MLISP2 means (loosely): "All I know is that I do not have the values I need to be successful. Therefore, back up to the last guy who had a choice to make, and let him choose some other alternative." This is entirely consistent with our state view of backtracking. FAILURE asserts that a state which cannot succeed has been reached. Unlike Floyd's system, there is no SUCCESS function in MLISP2; success is the absence of failure.

There are two final elaborations that have to be made. We stated above that upon failure the state vector is restored to the value it had when the last decision point was encountered. This is not entirely true. It is possible to change the saved copy of the state vector, thus changing the values that will be restored when a failure occurs. In this way, an unsuccessful alternative may pass back to the decision point information that may be useful in trying other alternatives. The MLISP2 notation for this is

<variable>{<context>}← <value>

A small integer called a "context number" is associated with each decision point. If no decision points have been set, the context number is zero. Every time a decision point is set, the context number is incremented by one. Every time a decision point is deleted, the context number is decremented by one. The intrinsic

function CONTEXT() returns the context number currently in effect, the "current context". Thus contexts may be manipulated by functions, For example,

X(CONTEXT()-1) ← 20

sets the value of X to 20 and also sets the value X had in the last copy of the state vector to 28. Therefore, as soon as a failure occurs, the value that will be restored to X will be 20. Setting variables in context actually sets the variable to the value in all contexts from the current one back to the specified one, This value will be restored to the variable whenever a failure occurs, unless the current context falls below the specified context, Therefore, setting a variable in context zero is a global set, since the current context can never become less that zero,

The other elaboration concerns implementation. Much of the discussion above is conceptual in nature and is not to be confused with the way MLISP2 implements it. The MLISP2 implementation is discussed in detail in [7]. One point that should be brought out here is that the amount of space required to store backtracking contexts may become quite large if many decision points are set. To manage this, an intrinsic function FLUSH() is included in MLISP2 to flush old contexts out of the system, Whenever the program reaches a point at which it is certain it will not have to backtrack, it should execute FLUSH. (This function should be used carefully, though, as it is possible to delete information that should have been saved.)

SECTION 3

Suntax conventions in this report

The following are meta-linguistic symbols used in this report to present the syntax of MLISP2.

SYMBOLS	MEANING
::=	Standard BNF symbols.
'	LITERAL. Any symbol preceeded by a quote mark or any identifier standing alone is a literal, i.e. stands for itself. Examples of literals: IF THEN ELSE '10 '(')
<>	NONTERMINAL. Any element enclosed in angled brackets is a nonterminal, or in some cases a description in English. Examples: <PROGRAM> <EXPRESSION> <PRIMARY>
[]*	REPEAT. Elements enclosed in square brackets followed by a (Kleene) star may occur repeatedly. Example: "[A]*" means "repeat A zero or more times"
[]	OPTIONAL. Elements enclosed in square brackets with no star or vertical bars are optional, Example: "[A]" means "optionally A"
[..|..]	ALTERNATIVES. Elements sepatated by vertical bars inside of square brackets are alternatives, one of which must be present. Example: "[A|B |C]" means "A or B or C"
[..|..]*	REPEAT OF ALTERNATIVES. This should be clear.

`[../..]*` REPEAT WITH SEPARATORS. If the repetition brackets `[]*` contain a slash "/", then the elements before the slash are repetition elements and the elements after the slash arc; separators for them, At most one slash will occur.
Example: "`[A/',]*`" means "repeat zero or more A's separated by commas"

SECTION 4

<PROGRAM?, <EXPRESSION>

<PROGRAM> ::= [<EXPRESSION> ';]* _EOF_

<EXPRESSION> ::= [<PREFIX>]* <PRIMARY>
[<INFIX>[<PREFIX>]* <PRIMARY>]*

<PREFIX> ::= <any id or delimiter declared a prefix> ['⊗]

<INFIX> ::= <any id, or any delimiter declared an infix> ['⊗]

<ID> ::= <any identifier not marked as a LITERAL>

Suntax

An MLISP2 PROGRAM is a sequence of expressions, each followed by a semicolon, ending with the literal _EOF_ (signifying end of file). An EXPRESSION is zero or more prefix operators, followed by a PRIMARY, followed any number of times by a triple composed of an infix operator, zero or more prefix operators, and another PRIMARY, Prefix operators must be defined to be a prefix (see the DEFINE expressioni, but any two-argument function may be used as an infix. Prefix and infix operators may be followed by the vector operator "⊗" (see the discussion of vectors in the MLISP manual). An ID is any

identifier which does not have the property LITERAL; identifiers yet marked as LITERALs when they are used without a quote mark ' in syntax patterns (section 7).

This is not the same as MLISP's definition of a PROGRAM, so this in itself is one of the minor changes to MLISP. In MLISP, a program is:

BEGIN [<EXPRESSION> ';]* END '.

In MLISP2 the enclosing BEGIN-END has been eliminated, and the period at the end has been replaced with the literal _EOF_.

Execution

When a program is parsed, each expression is translated and immediately evaluated. The value of the expression (if it is non-NIL) is printed on the teletype. Thus MLISP2 is a incremental, compile-and-execute type of translator, suitable for interactive programming in a time-shared environment. In fact one may regard MLISP2 as an elaborate terminal command language which will accept MLISP2 expressions one at a time from a teletype and execute them "on the spot," printing out each result. A trivial application of this capability might be to use MLISPZ as an adding machine: type "3+2;" and it will immediately print "5". _EOF_ may be typed at any time, terminating the session. Incremental translation/execution is an important capability in any time-sharing language.

In addition to being incremental, MLISP2 is also an extensible language. EXPRESSION makes use of an extensible production PRIMARY; at any time the programmer may enrich the MLISPZ language by making extensions to PRIMARY. Section 7 explains this in detail. In addition, PRIMARY or any other production in MLISPZ may be replaced entirely, including PROGRAM itself. Replacing PROGRAM produces a completely new language. In this way translators have already been produced for English, French, Logic [5], ALGOL, MLISPZ itself and others, some of them by programmers with no experience in translator writing. None of the MLISP? users have expressed much difficulty with their translators: in every case they were able to devote the bulk of their time to semantic applications (e.g. theorem proving strategies) rather than to the mechanics of the translation process.

Comments

1. One of the main reasons MLISP2 is successful as a translator writing tool is that it is an incremental extensible language, It fulfills Bobrow's recommendation [1]: "Reading a particular statement should be able to change the grammar at that time, for some defined scope." (his emphasis) The translators written in MLISPZ have all developed in this way, by adding a few productions at a time to the language. These can often be debugged independently. MLISP2 provides a rich environment for debugging

(c.f. RECOMPILE expression), In this way, MLISP2 enables the translator writer to easily subdivide the task of producing a translator, Furthermore, he always has something working, making progress easier to measure, When the set of productions is complete, PROGRAM is redefined, producing the new translator. The advantages of incremental programming are obvious to anyone who has had to write an "all or nothing" program, a large body of code which all had to be correct before anything would run.

2. MLISP2 is a successful extensible language because its syntax is simple and concise. Given the above definitions of PROGRAM and EXPRESSION, a programmer has little trouble comprehending the effects of an extension to PRIMARY. While extensible languages have been around for several years (and there is now a proliferation), all too frequently the extension mechanism has been couched in confusing notation and/or semantics, making them not at all "self evident." Self evident programming, the goal of COBOL and a host of successors, remains elusive, and MLISPZ does not attain it. But it has been the guiding principle in the design of MLISP2. Above all else, we have tried to make MLISP2 easy to use.

SECTION 5

<PRIMARY>

```
<PRIMARY>  ::=  <any MLISP expression>
            |   <an expression which is an "major" change to MLISP>
            |   <an expression which i s an "minor" change to MLISP>
```

Suntax

The production PRIMARY is an extensible production (section 7), and it is the principle means of extending the MLISP2 language. (BASIC is the other main extensible production.) In fact, we developed the MLISP2 language by first defining PRIMARY to be the same as in MLISP, and then extending it from time to time as we thought up new features we would like to have! The MLISP2 user may do the same thing: if he comes up with a useful language feature, he may add it at any time to PRIMARY or BASIC. The next few sections discuss the extensions the authors have made.

The major changes

<LET>
<SELECT>

The minor changes:

<SIMPEX>
<RECOMPILE>
<CASE>
<DEFINE>

SECTION 6

<SIMPEX>, <BASIC>

```
<PRIMARY>      ::=  <SIMPEX>

<SIMPEX>       ::=  <BASIC>  [<QUALIFIER>]*

<BASIC>        ::=  <ID>
                  |  [OCTAL]  <NUMBER>
                  |  <STRING>
                  |  ''  &-EXPRESSION>
                  |  '<  <ARGUMENTS> '>
                  |  '(  <EXPRESSION> ')

<QUALIFIER>    ::=  '( <ARGUMENTS> ')
                  |  '[  <ARGUMENTS> 'I
                  |  <DOT>
                  |  ['{ <EXPRESSION> '}]  '← <EXPRESSION>

<ARGUMENTS>    ::=  [<EXPRESSION> /',]*
```

Suntax

One of the alternatives of PRIMARY is SIMPEX. A SIMPEX (simple expression) is a basic expression, followed by zero or more qualifiers.

A BASIC expression is one of the following:

a. an ID (any identifier which is not a LITERAL)

b. a number (real or integer) optionally preceeded by the literal OCTAL

c. a string (a sequence of characters enclosed in double quotes ")

d. a quote mark ' followed by a LISP s-expression

e. arguments enclosed in broken brackets <>

f. an expression enclosed in parentheses ()

A QUALIFIER is one of the following:

a. arguments enclosed in parentheses 0

b. arguments enclosed in square brackets []

c. a DOT expression

d. an assignment arrow followed by an expression, optionally preceeded by an expression enclosed in braces {}.

ARGUMENTS are zero or more expressions separated by commas",".

SIMFEX

SIMPEX is a generalization of a production (also called SIMPEX) in the MLISP translator. The main difference is that in the MLISP2 version any number of qualifiers may appear after a basic expression, whereas in MLISP only a fixed number are allowed. Multiple qualifiers is just one of those constructions we thought it would be nice to have, so we added it one day.

Thus in MLISP2

```
FN(A,B,C)(X,Y,Z)
```

is allowed, while in MLISP it would have to be written

```
LAMBDA(FN1); FN1(X,Y,Z); (FN(A,B,C))  .
```

The association of qualifiers is to the left: e.g.

```
<basic> <qualifier1><qualifier2><qualifier3>
```

translates to

```
(((<basic> <qualifier1>) <qualifier2>) <qualifier3>)
```

or to be more concrete,

```
FN(A,B,C)(X,Y,Z)(1,2,3)
```

translates to

```
(((FN A B C) X Y Z) 1 2 3) ,
```

The other changes to SIMPEX are additions to the definition of QUALIFIER. In MLISP2 the DOT expression (see the next section) is allowed to be a qualifier, but not in MLISP. Also the brace notation {} on the left of the assignment operator "←" is allowed, and means the assignment is to take effect in a certain backtracking context (section 2). However, one restriction is that the vector operator "⊗" is not allowed to be used with the assignment operator "←", to simplify backtracking.

BASIC

BASIC is the other main extensible production in MLISP2, besides

PRIMARY. Intuitively, a basic expression is a "small unit" such as a single identifier or number, a "kernel" used to build larger expressions. It is not as useful as PRIMARY because fewer constructions intuitively seem primitive enough to be BASICs. But to demonstrate the type of extension it is reasonable to make, suppose you wanted to add complex numbers to the system, in the form

#<real part>,<imaginary part>I

e.g.

#3,2I

Then you could type

```
LET COMPLEX (*,REAL,*, IMAGINARY,*) BASIC =
        { '# [NUMBER] ' , [NUMBER] I }
        MEAN
        <whatever translation is desired>.
```

(This is a example of a LET expression, explained in section 7.)

Everything else in SIMPEX and BASIC is the same as in MLISP. PROGRAM, EXPRESSION, PRIMARY, SIMPEX and BASIC form the heart of the MLISP2 language. Understand them and you will have a good idea of what a legal MLISP2 program looks like, as well as how to change that definition.

SECTION

<DOT>

<QUALIFIER> ::= <DOT>

<DOT> ::= '. [<IDENTIFIER> | <BASIC>]

Syntax

One of the alternatives of QUALIFIER is the DOT expression. A DOT expression is a period ".", followed by either an identifier or a basic expression. In the first case, the identifier is quoted by MLISP2, while in the second case the value of the basic expression is used unquoted.

This is a notation for handling property lists. Ordinarily it means GET, but on the left side of an assignment operator "←" it means PUTPROP. (The value of the assignment operator is always the value of the right hand side.) While the dot notation is a seemingly trivial change, our experience has shown that it is capable of striking clarifications in a program.

Examples

A.B	translates to	(GETA (QUOTE B))
A.B ← C		(PUTPROP A C (QUOTE B))
A.'B		GET A (QUOTE B))
A.'B ← C		(PUTPROP A C (QUOTE B))
A.(B)		(GET A B)
A. (B) ← C		(PUTPROP A C B)
A. (B+C) ← D		(PUTPROP A D (PLUS B C))

Be careful about the association of qualifiers!

A.FN(X,Y,Z)	translates to	((GET A (QUOTE FN)) X Y Z)
A.(FN(X,Y,Z))		(GET A (FN X Y Z))

SECTION 7

<LET>

```
<PRIMARY>          ::=  <LET>

<LET>              ::=  LET  <IDENTIFIER> '(  <LET VARIABLES> ')
                          [<IDENTIFIER>]  '='(  <PATTERN>  '3
                          MEAN  <EXPRESSION>

<LET VARIABLES>    ::=  <LET VARIABLE> I ', <LET VARIABLE>]*

<LET VARIABLE>     ::=  <ID> | '*

<PATTERN>          ::=  [<PATTERN ITEM>]*

<PATTERN ITEM>     ::=  I'!]  ['#]  [ <LITERAL>
                                    | <NONTERMINAL>
                                    | <INLINE EXPR>
                                    |<META> 1

<LITERAL>          ::=  <IDENTIFIER>|'' <TOKEN>

<TOKEN>            ::=  <IDENTIFIER> | <NUMBER> | <STRING> | <DELIMITER>

<NONTERMINAL>      ::=  '< <IDENTIFIER> '>

<INLINE EXPR>      ::=  '[  <EXPRESSION> 'I

<META>             ::=  [<REP>| < O P T > |<ALT>]

{}                 ::=  [{}|  01   <Anywhere braces {} may be used in
                          patterns, parentheses () may be used instead>
```

Syntax

A LET expression is the literal LET, followed by an identifier which is the name of the production being defined, followed by a list of LET variables enclosed in parentheses, optionally followed by a second identifier which represents the name of a production to which this production will be added as an alternative, followed by an equal sign "=", followed by a syntax pattern enclosed in braces {}, followed by the literal MEAN, and finally followed by an expression which represents the semantics to be evaluated if the syntax is successfully matched. A LET VARIABLE is either an ID or an asterisk "*".

A PATTERN is zero or more triples, each composed of an optional exclamation point "!", followed by an optional sharp sign "#", followed by one of

a. A literal: an identifier, or a quotemark ' followed by any token (identifier, number, string or delimiter), Iuentifiers not preceeded by the quote mark are marked with the property LITERAL (and become essentially reserved words).
b. A nonterminal: an identifier enclosed in broken brackets <>, representing a call on another production,
c. An inline expression: any MLISP2 expression enclosed in square brackets []. One special convention: if the expression is just a

single identifier, e.g. [FOO], then it is taken as the name of a function of no arguments, FOOO, rather than as a variable, Thus [FOO] and [FOO()] are equivalent,

d. A meta expression: REP, OPT or ALT.

Capabilities of the LET expression

The LET expression is composed of a syntactic pattern matcher and a semantic expression evaluator, It may be used to extend the language, to define entirely new languages, or as a limited pattern matcher, This is the core of the MLISP2 extensibility mechanism. The recognition algorithm is top down, depth first, and uses backtracking. The top-level production is PROGRAM. The pattern matcher is powerful enough to handle any context free or sensitive grammar. However, it is only capable of dealing with linear input, such as tokens from a file or from a linear list; it is not capable of handling structured input. It is designed primarily as a translator writing tool.

The LET expression, like all MLISP2 expressions, is fully incremental; at any time the user may type a LET expression on line and have it take effect immediately. The advantages of this for debugging a translator should be obvious: if the programmer discovers a bug in a production, he can type in a corrected version and try it

out right away. Furthermore, if he desires to have a new language construction in a program, the user simply includes the relevant productions at the head of his program, and the MLISP2 language will extend itself as his program is being translated.

In order to extend the definition of some production P,

1. P must already exist and must be an extensible production. An extensible production is any production whose syntax contains the meta expression ALT (section 7.4). "ALT" means that the syntax of the production consists a set of alternatives. This set may be extended at any time.
2. The production being added to the definition of P must contain the name of P in the identifier slot between the LET variables and the equal sign: e.g.

   LET FOO (X) P =

   adds FOO to the definition of P. A production may only be added as an alternative to one production: e.g. FOO cannot now be added to another production's definition, However, an extensible production P may have any number of alternatives added to its definition.

Using ALT in an increnienta1 manner is one of the most powerful capabilities of any extensible language. It makes MLISP2's extensibility very flexible.

Semantics

When a production is defined using LET, two functions are declared: one for the syntax and one for the semantics, The name of the syntax function is the production name with a sharp sign "#" appended on the end. The name of the semantics function is the name of the production. For example, if the production is

LET FOO(X,*,Y) = (A B C) MEAN PRINT <X,Y>

then the two functions are named FOO# and FOO, The definition of the semantics function FOO is

(LAMBDA(X Y) (PRINT (LIST X Y)))

Note that the LAMBDA variables are the non-* LET variables. When <FOO> is called in a pattern, a call to the syntax routine FOO# is compiled, The syntax routine always calls the semantics routine FOO as part of its definition. In addition, either of the two functions may be called like any other function: e.g. FOO#() or FOO(args). Thus the pattern matcher may be invoked from within an ordinary function.

An important point hare is that many extensible languages interpret their patterns. MLISP2 compiles its syntax into machine code, resulting in greater speed and code density. There are certain technical difficulties with compiling a general, incremental syntax processor. We hope to discuss our treatment of these problems in a later paper.

Execution

LET expressions are executed in three stages:

1. Match the syntax pattern against the input.
2. Bind the LET variables to the values of the pattern items.
3. Evaluate the semantics expression. This becomes the value of the production.

In more detail,

1. The matching of a syntax pattern proceeds as follows:
   a. A pattern is matched from left to right,
   b. Each item in a pattern interacts in a specified way with the input. This is explained in the following sections for each type of pattern item. Generally pattern items make some checks on the content of the input and cause the input pointer to be advanced.
   c. Each item in a pattern returns a value.
2. After the pattern has been completely matched, the LET variables are bound on a one-to-one positional basis to the pattern values, with two exceptions:
   a. LET variables which are asterisks "*" serve only as positional place-holders and do not receive values. (Thus they are not really variables at all.) If all of the LET variables are asterisks, then all of the pattern values are thrown away.

b. If there are more pattern values than LET variables, the last non-* variable is bound to a list of the remaining values.

3. After the entire pattern has been successfully matched against the input and the LET variables have all been bound, the semantics of the production are evaluated. The semantics consist of the expression after the MEAN, together with the non-* LET variables. It is exactly equivalent to

((LAMBDA <variables> <expression>) <pattern values>)

The value of this expression becomes the value of the production and is returned to whomever called it.

Examples of variable binding

```
(a) LET IF(*,E1,*,E2,E3) =
        { IF <EXPRESSION> THEN <EXPRESSION>
                {OPT ELSE <EXPRESSION>) }
        MEAN NIL:

this    - throw away the IF
        - binds E1 to the value of the first <EXPRESSION>
        - throws away the THEN
        - binds E2 to the value of the second <EXPRESSION>
        - binds E3 to the value of the OPT.


(b) LET PROGRAM(*) =
        { {REP 0 M {<EXPRESSION> ';}} _EOF_ }
        MEAN NIL:

this    - throws away all the values of the pattern items.
```

```
(c) LET FOO(X,Y,Z) =
        { A  B  C  D  E  F  G }
        MEAN NIL;
```

this - binds X to A
- binds Y to B
- binds Z to (C D E F G .

Example of LET semantics

```
(d)LET F O O (X,*,Y,*,Z) =
        { A  B  C  D  E  F  G }
        MEAN PRINT(X CONS Y CONS Z);
```

this - prints and has as its value the list (A C E F G).
- The semantics function is
(LAMBDA (X Y Z) (PRINT (CONS X (CONS Y Z)))).

The best examples of LET expressions, and indeed of all MLISP2 expressions, are provided by the productions in the MLISP2 translator, which are included in the appendix.

SECTION 7.1

Suntax description language

There are four types of constructions which can be used in syntax patterns: literals, nonterniinals, inline expressions and meta expressions. In addition, each of these constructions may be preceeded by either/both/none of an exclamation point "!" and a sharp sign "#". The meaning of each of these in a syntax pattern and the values they return will now be described in detail.

Our approach to a syntax description language is someuhat different than other approaches, a necessary consequence of our desire to make languages incrementally extensible. Rather than working with traditional BNF terms and analyzing grammars formally (e.g. as precedence, operator precedence, LR(k), etc. grammars), we have isolated a small set of primitives powerful enough to specify any context free or sensitive grammar and still maintain a good degree of efficiency. We obtain this flexibility by a pattern matching approach to language translationThe extremely useful control structure of backtracking is used to resolve ambiguities: if one syntax pattern will not match the input, the system is capable of backing up and trying others, until either one finally succeeds or no patterns are left (indicating a real syntax error). While it is

theoretically possible for this to require a time proportional to the length of the input cubed, in practice the types of grammars one writes for programming languages are almost always handled in linear time. In fact, even our grammar for English, a highly ambiguous language, produced a linear parser.

We believe that the primitives presented here: REP, OPT and ALT, together with literals, nonterminals and inline expressions, are primitives that should be included in any syntax description language.

!

The exclamation point feature "!" in patterns is a way of automatically generating error messages, An exclamation point in a pattern item signifies "it had better be there!"; otherwise there is an error. For example, in the pattern

IF <EXPRESSION> !THEN

the literal THEN had better occur in the input after the expression, or the error message "MISSING THEN" will be printed. The exclamation point is actually a macro that expands to an ALT (section 7.4); e.g.

!THEN

expands to

{ALT THEN|[ERROR("MISSING THEN")]}.

This expansion should be remembered when dealing with the value of a pattern item with an exclamation point in front of it. The value of the pattern item THEN is just THEN; the value of !THEN is (1 THEN), or an error.

#

The sharp sign feature "#" in patterns is a way of controlling the scanning of the input. It really has a meaning only with literals. If it is present, then after the literal is matched, the scanner will not advance over it. Ordinarily, after a literal in a

pattern is matched by a token in the input, the scanner advances to the next token automatically. The "#" feature is is useful if, for some reason, it is desired to temporarily discontinue scanning, For example, MLISP2 does not want to advance over the _EOF_ at the end of the program because there is nothing there to scan, so the pattern is written #_EOF_. Similady, MLISP2 wants to pause in scanning when it sees the literal OCTAL ipreceeding an octal number) in order to change the radix from 10 (decimal) to 8(octal) before scanning the number, so the octal pattern is written #OCTAL <OCTAL-NUMBER>.

LITERAL

Literals are constants in the syntax description language. If the next item in the pattern is a literal, then the next token in the input must be that literal, or the pattern fails.

VALUE = the literal.

NONTERMINAL

Nonterminals are the subroutine mechanism in the syntax description language. If the next item in the pattern is a nonterminal call on another production, then that production is evaluated as a subroutine.

VALUE = the value of the called production.

INLINE EXPRESSION

An inline expression is a piece of code evaluated "in line", during matching of a pattern. If the next item in the pattern is an inline expression, then that expression is immediately evaluated. This is an unusual and powerful feature in a pattern matcher, It provides a means for making a syntax context sensitive and also for increasing the its efficiency by making run-time tests,

To illustrate these capabilities, suppose a global variable FLAG exists in a program and a production P uses this variable to govern its execution:

```
LET P(X) = {{ALT  [FLAG  = 'FOO OR FAILURE01 ...
                | [FLAG  = 'BAZ OR FAILURE03 . . .
                | ...
           }} MEAN <whatever>
```

Then the matching of P is context sensitive. If the value of FLAG is FOO, then the first alternative will be tried. Otherwise FAILURE is executed, which causes processing to pass to the second alternative. Similarly, if the value of FLAG is BAZ, then the second alternative will be tried. Otherwise processing skips to the third alternative, and so on.

This illustrates the power of inline expressions in a syntax description language. It also illustrates how they can be used to increase the efficiency of pattern matching. Suppose the programmer

knows that if FLAG has certain values, the input can never match certain patterns. Then making tests like the ones in the example above insure that these patterns will never be tried. The patterns would have eventually failed anyway; the inline expressions just cause them to fail at the earliest possible moment, with a minimum of work being done.

VALUE = the value of the expression.

META EXPRESSIONS

Three meta expressions -- REP (repeat), OPT (optional) and ALT (alternatives) -- are included in MLISP2 to make it easier to specify syntax. These constructions reduce the number of productions required and make them clearer and more concise. It is surprising how much more powerful the syntax description language becomes with the inclusion of these three expressions. They make the language far more descriptive of the kinds of configurations to be expected in the input.

By way of contrast, in the Backus-Naur form (BNF) if one wishes to express the fact that some item I may occur repeatedly, he must write

```
P ::= I <P>
P ::= <empty>
```

while in MLISP2 one would write

{REP 0 M {I} }

In the first case the repetition is implicit, leaving the user to figure out exactly what the productions will handle; in the second case the repetition is explicit. The same is true for OPT and ALT. Consider

P ::= IF <E> THEN <E>
P ::= IF <E> THEN <E> ELSE <E>

versus

IF <E> THEN <E> {OPT ELSE <E>}

In the first case it requires a production-by-production analysis to discover that the ELSE clause of the IF expression may be left off; in the second case it is explicitly stated. This distinction becomes important when the number and complexity of productions are large. Explicit specification is very important in any "descriptive" language.

VALUE = the value of the meta expression.

SECTION 7.2

<REP>

```
<META>          ::=  <REP>

<REP>           ::=  '{ REP <INTEGER>  [<INTEGER> | M]  ['*]
                        '{  <PATTERN> '}  [<SEPARATORS>]  ' }

<SEPARATORS>    ::=  <PATTERN>
```

Suntax

A REP is the literal REP, followed by two integers (the second integer may be the literal M), optionally followed by an asterisk "*", followed by a syntax pattern enclosed in braces {}, and optionally followed by any number of separators. This whole thing is enclosed in braces {}.

Semantics

The REP expression causes a pattern to be matched repetedly. The number of times the pattern is matched depends on two things: (1) how many times the pattern occurs in the input, and (2) the values of the "repetition control numbers" which come after the word REP. The <number>s must be non-negative integers. The first number is the

minimum number of times that the pattern must occur in the input; the second is the maximum number of tinies that it may occur. Alternatively, the letter "M" may be substituted for the maximum and means "more": if M is used, the pattern may be repeated any number of tinies greater than or equal to the minimum number. For example, (REP 1 M ...) mean3 "repeat 1 or more times."

If the minimum number of repetitions of the pattern does not occur, the entire REP fails. REP always tries to match the maximum number of repetitions possible. In some cases too many repetitions may be matched, causing a later pattern to fail. In this case the tokens from one cycle of the pattern are returned to the input, Pattern matching then proceeds with the new, shorter, REP list (unless the number of repetitions falls below the minimum). More than one repetition may have to be given back before later patterns all succeed. If you want to supress this step-by-step backup, include the asterisk "*" in your REP, The asterisk means "either use all the REP cycles you got or give them ALL back!" Any failure into the REP after using this feature will cause all tokens matched by all cycles of the REP to be returned to the input. The number of repetition5 immediately becomes zero. If the minimum for the REP was greater than zero, the entire REP then fails. Otherwise the value of the REP becomes NIL. This is useful when you are certain there is no ambiguity between the REP pattern and later patterns, so that no REP

cycles will have to be given back. The advantage of using the asterisk is that REPs are more efficient, since not as much backtracking information has to be saved.

Separators may occur between repetitions of the REP pattern. Any pattern item or items may be used as separators. The value of a REP is a list of the values of the REP patterns: the values of the separators are discarded.

Evaluation

Evaluation of REPs proceeds as follows:

1. When a REP is encountered, one of two things happens.
   a. If the REP uses the asterisk "*" feature, then a single decision point (section 2) is created for the entire REP. The first time this decision point is failed to, it deletes itself. Subsequent failures will fail to whatever decision point preceeded the REP.
   b. If the REP does not use the asterisk feature, then a decision point is created for each cycle which the REP makes. Each of these decision points behaves like 1.a above, i.e. the first time it is failed to it deletes itself. The difference is that there are many of these decision points, one for each cycle through the REP. Therefore, each REP cycle can be backed up

over one at a time, whereas with the asterisk feature ALL of the REP cycles are backed up over at once.

2. Then the REP pattern is matched against the input.
3. The maximum REP number is now checked (unless it is M) to see if the REP pattern has been matched the maximum number of times allowed. If so, the REP exits returning a list of the pattern values matched.
4. If there are separators, they are matched against the input and their values thrown away. Then step 2 is executed again,
5. Finally, either the maximum number of cycles is reached, or the REP pattern or separators no longer match the input. Then the minimum REP number is checked. If the REP has not executed the minimum number of cycles, then the entire REP fails. If the minimum has been reached, the REP exits returning a list of the pattern values matched.

Examples

(a) {REP 1 3 {A B}}

```
input:              A B A B A B A B
value:              ((A B) (A B) (A B))
left in input:  A   B

input:              A C B
value:              REP fai Is because minimum (1) was not achieved
```

(b) {REP 0 M {<IDENTIFIER>} ',}

```
input:              A;
value:              ((A))
left in input:      .

input:              A, B, C; D, E, F
value:              ((A) (B) (C))
left in input:      ;D, E, F

input:              (
value:              NIL
left in input:      (
```

SECTION 7.3

<OPT>

<META> ::= <OPT>

<OPT> ::= '{ OPT <PATTERN> '}

Syntax

An OPT is the literal OPT followed by a syntax pattern, all enclosed in braces {}.

The OPT expression is just an abbreviation for (and a slightly more effecient implementation of) the special REP case {REP 0 1 ...}, i.e. "repeat zero or one time." This is one of the most frequent REP cases.

Evaluation

Evaluation of OPTs proceeds as follows:

1. When an OPT is encountered, it creates a decision point, This decision point may be failed to only once. The first time it is failed to, it deletes itself so that subsequent failures will fail to the previous decision point.

2. The OPT pattern is matched against the input. Two things might happen:
   a. The match is successful, in which case the OPT returns a list of the pattern values (leaving the decision point intact).
   b. The match is unsuccessful, i.e. one of the pattern items fails, in which case the OPT deletes its decision point and returns NIL.

3. If a later failure occurs, and if the OPT decision point is still intact, then (as in 2.b above) the OPT deletes its decision point and returns NIL,

Examples

(a) {OPT A B}

```
input:          A B C
value:          (A B)
left in input:  C

input:          A C B
value:          NIL
left in input:  A C B
```

(b) {OPT <IDENTIFIER> '(<IDENTIFIER> ')}

```
input:          CAR(A).B
value:          (CAR /( A /))
left in input:  .B

input:          CAR(1).B
value:          NIL
left in input:  CAR(1).B
```

SECTION

<ALT>

<META> ::= <ALT>

<ALT> ::= '{ ALT [<PATTERN> /'|]*'}

Suntax

An ALT is the literal ALT, fol lowed by zero or more syntax patterns separated by vertical bars "|", al l enclosed in braces {}.

Senian t i cs

ALT is the most interesting meta expression of MLISP2's pattern matching system. It specifies that the input may be matched by any of a set of alternatives. The powerful aspect of ALTs is that the set of alternatives may be dynami ca l l y extended at execution time. If the alternative being augmented is part of the MLISP2 translator, for example, then the effect is to extend the MLISPZ language. To i l lustrate this idea, consider the following pair of productions from the MLISP2 translator,

```
LET PRIMARY (X)=
        { {ALT} }
        MEAN X[2];
```

```
LET BEGIN (*,VARS,EXS,*)PRIMARY =
        ( BEGIN <DECLARATIONS> <EXPRESSIONS>    !ENO )
        MEAN
        'PROG CONS VARS CONS EXS;
```

The first production defines a PRIMARY in MLISP2. It says that initially a PRIMARY is an ALT with no alternatives, While this may seem useless, it serves the important function of providing a (null) set to which other productions may be added. For example, the second production defines a BEGIN-END block. It further indicates that it is to be added to the set of alternatives in the production PRIMARY, i.e. BEGIN is now to be considered as an example of a legal PRIMARY.

So what ? So now instead of having to define all of the alternatives in a static definition, the various parts of the production may be defined individually and dynamically! Productions which add themselves to other productions may be included in any prOgram. If you want some language feature for your particular program, you need only include a set of language extensions at the beginning of it. Immediately you have a new language with a tailor-made feature!

One wOrd Of caution: the syntax of MLISP2 is not context sensitive: additions to it should be unambiguous with the productions already there. In fact, they should probably be unambiguous in the first symbol: don't start any of your productions with any of the words that starts an MLISP2 production, such as

BEGIN, IF, FOR, WHILE, UNTIL, DO, COLLECT, SELECT, . . .

This restriction grows out of our desire to provide good error messages. If we start a production, such as BEGIN, and we get to a point in it where we expect a literal to appear in the input, and the literal isn't there, then we stop immediately and print an error message. The alternative is to back up out of the production, see if any other production can handle the input, and if not give an error message like "SYNTAX ERROR" or some such nonsense. Since the MLISP2 language is unambiguous, we can give much better error messages than that. Most PRIMARYs in MLISP2 begin with a unique LITERAL,

As mentioned above, only a productions containing an ALT is an extensible production. If the production contains more than one ALT, then there is an ambiguity as to which ALT is to be extended. To resolve this ambiguity, the following rule applies:

a. The outermost ALT lexically is the only one that may be extended,
b. If several ALTs are at the same lexical level, then the last one lexically is the only one that may be extended.

Evaluation

Evaluation of ALTs proceeds as follows:

1. When an ALT is encountered, it creates a decision point. It also creates the equivalent of a local own variable; this variable is

initialized to zero and represents the number of the alternative currently being tried.

2. To try the next alternative, the alt number is incremented by one, and then the pattern for that alternative is matched against the input.

3. If the pattern is successfully matched, the ALT exits returning a list of the pattern values with the alt number added to the front. If the pattern fails, the next step is executed.

4. If there are more alternatives to be tried, step 2 is executed again. If there are no more alternatives, the decision point is deleted, and the whole ALT fails.

5. If a subsequent failure returns into the ALT, step 4 is executed,

Examples

```
(a) {ALT  A  | B}

input:              A B C
value:              (1 A)
left in input:  B   C

input:              B C
value:              (2 B)
left in input:  C

'input:             C
value:              Fai Is
```

```
(b){ALT A ', B | <IDENTIFIER> '( <IDENTIFIER> ')| CAR }

input:              A,  B, C
value               (1 A /, B)
left in input:  ,   C

input:              CAR(A) .B
value:              (2 CAR /( A /))
left in input:.B

input:              CAR().B
value               (3 CAR)
left in input:  0 .B

input:              A: B; C
value:              Fai Is
```

SECTION 8

<SECT>

```
<PRIMARY>      ::=  <SELECT>

<SELECT>       ::=  SELECT [<EXPRESSION>]
                      FROM [<ID>':']<EXPRESSION>
                      [SUCCESSOR <EXPRESSION>]
                      CUNLESS <EXPRESSION>]
                      [FINALLY <EXPRESSION>]
```

sun tax

A SELECT expression is the literal SELECT, optionally followed by an expression, followed by the literal FROM, optionally followed by an ID and a colon, followed by an expression, optionally followed by any or all of the literal SUCCESSOR and an expression, the literal UNLESS and an expression, and the literal FINALLY and an expression,

Four of the five expressions in the SELECT expression are optional. If they are omitted, defaults are supplied. The default for the first expression is CAR, for the second CDR, for the third NULL, and for the fourth FAILURE. Thus

```
SELECT FROM '(A B C)
```

and

```
SELECT CAR(L) FROM L:'(A B C) SUCCESSOR CDR(L)
       UNLESS NULL(L) FINALLY FAILURE0
```

are exactly equivalent,

Semantics

The following explanation of SELECTs is largely reproduced from a paper by the authors titled "Backtracking in MLISP2" [7].

The meta expressions REP, OPT and ALT use backtracking in the MLISP2 syntax description language. The SELECT expression is the means for incorporating back tracking into ordinary functions. The logical form of the SELECT expression is

SELECT <value function> FROM <formal variable>:<domain>
SUCCESSOR<successor function>
UNLESS <termination condition>
FINALLY <termination function>

This is a generalization of Floyd's CHOICE function [3], though the two are functionally equivalent. However, the SELECT expression is a little more versatile and easy to use.

The four "functions" in SELECT are actually expressions which serve as the bodies of LAMBDA expressions having the fornial variable as its LAMBDA variable:

(LAMBDA (<formal variable>) <expression>)

The functions are defined as:

<value function> : <domain> → <value>

<successor function> : <domain> → <domain>

<termination condition> : <domain> → T or NIL

<termination function> : <domain> → <value>

Evaluation

The evaluation of a SELECT expression proceeds as follows:

1. Evaluate the domain expression to get an initial domain,
2. Set up a decision point,
3. Apply the termination condition to the domain, If the value is TRUE (non-NIL), delete the decision point and apply the termination function to the domin. Exit with this value as the value of the SELECT. (The termination function may call FAILURE). If the value of the termination condition is FALSE (NIL), proceed to the next step.
4. Apply the value function to the domain, and exit with this value as the value of the SELECT.
5. If a failure returns to the SELECT (the only way a SELECT may be reentered), apply the successor function to the domain to yield a new clOmain.
6. Go to step 3.

Floyd's CHOICE function is written:

```
EXPR CHOICE (N);
        SELECT I FROMI:1 SUCCESSOR I+1
                UNLESS I GREATERP N FINALLY FAILUREO;
```

CHOICE(10) gives ten choices. The initial domain is just the integer 1 (one). The value function is the identity function (LAMBDA (I)I). The successor function is addition by one (LAMBDA (I) (PLUS I 1)).

The termination condition is a check if the maximum has been exceeded (LAMBDA (I) (GREATERP I N)). The termination function propagates the failure (LAMBDA (I) (FAILURE)).

Examples

(a) SELECT FROM '(A B C)

This is the most primitive version of the SELECT expression. It gets and returns elements one at a time from a list. Every time it is failed to, it returns the next element in the list. If the list becomes exhausted, the failure propagates to the preceeding decision point.

(b) SELECT CAR(L) FROM L: '(A B C) SUCCESSOR CDR(L)
UNLESS NULL(L) FINALLY FAILURE 0

This is exactly the same as (a).

(c) SELECT FROM '(A B C) FINALLY NIL

This is the same as (a) except that if the list becomes exhausted, this will return NIL instead of failing.

SECTION 8

<RECOMPILE>

```
<PRIMARY>        ::=  <RECOMPILE>

<recompile>      ::=  RECOMPILE <IDENTIFIER> [', <IDENTIFIER>]*
                        IN  <file>  [<ppn>]  [TO <file>]

<file>           ::=  <file_spec> ['. <file_spec>]

<ppn>            ::=  '[ <file_spec> ', <file_spec> ']

<file_spec>      ::=  [<identifierr | <integer>]
```

Syntax

A RECOMPILE expression is the literal RECOMPILE, followed by one or more identifiers representing the names of functions or productions to be translated, followed by the literal IN and an input file name, optionally followed by the literal TO and and output file name. The input file name may include a project-programmer area, but the output file name may not. The output may not go to another project-programmer area to prevent the user from accidentally clobbering someone else's disk area.

Semantics

The RECOMPILE expression is an extremely useful feature of MLISP2. It enables selected functions in a file to be quickly translated. The functionsmay either be defined in core at once, replacing any definitions that existed, or their translations may be printed onto an output file. In either case, translation ceases as soon as ail the functions in the list have all been translated, without going all the way to the end of the file.

This feature substantially decreases debugging time by speeding up the test/correct/recompile/test loop. With the RECOMPILE feature, you can edit your file, change the function or functions containing the bug, and then recompile only those functions -- a much shorter task usually than recompiling your entire program. RECOMPILE will find and translate any function or production beginning with LET, EXPR, FEXPR, LEXPR or MACRO. It skips down to a specified function at scanner speed, tr-anslates the function, and then either exits or skips on to the next function.

Examples

(a) RECOMPILE FN1 IN IFILE;

(b) RECOMPILE FN1,FN2,FN3 IN IFILE.EXT;

(c) RECOMPILE FN1,FN2,FN3 IN IFILE.M2[1,FOO] TO OFILE.LSP;

SECTION 10

<CASE>

<PRIMARY> ::= <CASE>

<CASE> ::= CASE <EXPRESSION> OF
BEGIN [<EXPRESSION> ';]* END

syntax

The CASE expression is the literal CASE, followed by an expression which must evaluate to a positive integer (the case index), followed by the literals OF and BEGIN, followed by zero or more expressions each with a semicolon, followed by the literal END. The semicolon after the last expression is optional.

Semantics

Including this expression in MLISP2 remedies an obvious omission of MLISP; every good language should have a case expression. The MLISP2 version is pretty standard, The expression after the CASE computes an integer index, and then the corresponding expression after the BEGIN (counting from one) is evaluated and returned as the value of the case expression. Using an index greater than the number

of expressions will result in a run-time error. Using an index less that or equal to zero will execute the first case expression (i.e. as if "CASE 1 OF . . ." had been typed).

Examples

(a) X ← CASE 1 OF BEGIN 'A; 'B; 'C END;

X gets. the value A.

(b) CASE IF N=1 THEN 2 ELSE 3 OF
BEGIN PRINT "CASE 1"; PRINT "CASE 2"; PRINT "CASE 3";END;

This will print either "CASE 2" or "CASE 3" and return the string printed as its value. Case 1 will never be evaluated,

SECTION 11

<DEFINE>

```
<PRIMARY>          ::=  <DEFINE>

<DEFINE>           ::=  DEFINE [<DEFINE CLAUSE> /',]*

<DEFINE CLAUSE>    ::=  <I D> PREFIX [<TOKN>] [<NUMBER>]
                      |  <ID> <NUMBER> <NUMBER>
                      |  <ID> <TOKN> [<NUMBER> <NUMBER>]

<TOKN>             ::=  <any id or any delimiter except , or ;>
```

Syntax

A DEFINE expression is the literal DEFINE followed by zero or more define clauses separated by commas, A DEFINE CLAUSE is either

a. An id, followed by the literal PREFIX , optionally followed by a tokn and/or a number.

b. An id, followed by two numbers.

c. An id, followed by a tokn, optionally followed by two numbers,

A TOKN is any id or any delimiter except comma "," or semi colon ";".

Semantics

MLISP2's DEFINE expression is pretty much like MLISP's, although it is slightly less general. In MLISP one could define any symbol to be any other symbol; in MLISP2 only IDs may be given alternate definitions. For example, in MLISP

DEFINE ; →

is legal and means "translate all future occurrences of "→" to ";" in the program." In MLISP2 the first symbol must be an ID. Example:

DEFINE APFEND @

translates all future occurrences of "@" to APPEND. The DEFINE expression will be explained by examples,

(1) DEFINE NOT PREFIX ¬1000

This defines ROT to be a prefix (see section 6); it defines the symbol "¬" to be an abbreviation for it; and it defines its binding power to be 1000. Anytime "¬" occurs hereafter, it will be translated to NOT. Like MLISP, MLISP2 uses binding powers to implement its operator precedence hierarchy. Binding powers are explained in the MLISP manual [6]. Only right binding powers have to be defined for prefix operators. Most prefixes have a binding power of 1000; this is higher than the binding power of any infix. However, one difference between MLISP2 and MLISP is that in MLISP2

some prefixes (GO, RETURN, and all the print functions -- PRINT, PRINTSTR, PRINTTY, PRIN1, PRINC, TYO) have a binding power of zero, This means, effectively, that they take a whole expression as their argument, rather than just a primary. Examples:

```
PRINT CAR A CONS COR B
RETURN A+B*C/D-E
```

are translated to

```
(PRINT (CONS (CAR A) (CDR B)))
(RETURN (DIFFERENCE (PLUS A (QUOTIENT (TIMES B C) D)) E))
```

The advantages of defining a function to be a prefix are that it may be used without parentheses around its argument, and it may be used with the vector operator "⊗". For example, since CAR is a prefix, CAR L, CAR(L), CAR⊗ L and CAR⊗(L) are all legal.

(2) DEFINE CONS 450 480

This defines the left and right binding powers of CONS to be 450 and 400 respectively. MLISP2 uses the same precedence system as MLISP. The binding powers of any operator can be found in the MLISP manual, or by examining the property list for the indicators &LEFT and &RIGHT. Then if you want to give your operator a higher precedence, simply define it with higher binding powers. Operators with no &LEFT or &RIGHT properties use the values under the atom

DEFAULT. Parentheses may be used to alter the precedence by grouping arguments i n any des i red order.

(3) DEFINE APPEND @ 450 400

This is just like example (2) above, except that it also defines "@" to be an abbreviation for APPEND, Actually it defines "@" to be an infix whose translation is APPEND, Any id may be used as an infix without defining it as such: however, delimiters must be explicitly defined. MLISP2's pre-defined delimiter infixes are (in their proper hierarchy):

* /	(TIMES, QUOTIENT)
+ -	(PLUS, DIFFERENCE)
@ ↑ ↓	(APPEND, PRELIST, SUFLIST)
= ≠ ≤ ≥ ∈	(EQUAL, NEQUAL, LEQUAL, GEQUAL, MEMBER)
& ∧	(AND)
| ∨	(OR)

The complete operator precedence hierarchy is in the MLISP manual,

SECTION 12

Primitive productions

<IDENTIFIER> ::= <LETTER> [<LETTER>| <DIGIT>]*

<LETTER> ::= [A|B|...|Z|a|b|...|z|<underbar>|? <any character>]

<ID> ::= <any identifier-not marked as a LITERAL>

<NUMBER> ::= [<INTEGER>| <REAL>]

<INTEGER> ::= <DIGIT> [<DIGIT>]*

<DIGIT> ::= [0|1|2|3|4|5|6|7|8|9]

<REAL> ::= <INTEGER> '. <INTEGER> [<EXPONENT>]
| <INTEGER> <EXPONENT>

<EXPONENT> ::= E [+|-|] <INTEGER>

<STRING> ::= '" [<any character except % or ">]*'"

<COMMENT> ::= '% <any characters except %>'%
| COMMENT <any characters except ; or
unpaired " or %>';

<NULL> ::= [<blank>|<tab>|<carriage return>|<line feed>
|<vertical tab>|<form feed>|<altmode>]

<DELIMITER> ::= <any character except letters, digits, nulls or %>

These lowest level productions are virtually identical to MLISP's definitions (with one exception), and they are repeated here merely for the sake of reference. The one exception is that the special characters colon ":" and exclamation point "!" are legal letters in MLISP but not in MLISP2. The only special character that is considered a letter in MLISP2 is underbar "_". However, any special character may be included in identifiers by preceeding them with the "literally" character: a question mark "?".

IDENTIFIER, NUMBER, STRING and DELIMITER are pretty standard. But remember that an ordinary variable or function name must be an ID; that is, it must be an IDENTIFIER which is not marked with the property LITERAL.

SECTION 13

Runtime functions

In addition to all the MLISP runtime functions available to the user, MLISP2 adds several functions for dealing with input and for backtracking.

PARSE

This function starts the MLISP2 parser. It initializes all necessary internal structure and then calls <PROGRAM>. This causes the current definition of PROGRAM to be executed as explained in section 7. There are several alternatives to the arguments to PARSE:

(PARSE)

This sets the input to the teletype. MLISP2 expressions may now be typed and evaluated on line. Typing _EOF_ will exit gracefully from this mode.

(PARSE SOURCE-FILE)

This sets the input to the specified file, MLISP2 expressions will now be accepted and evaluated from this file, The file should

end with _EOF_. The source file may have an extension, in which case it should be in the form (name . ext). Otherwise the filename should be an atom. If the filename is NIL, then input will be accepted from the teletype, just as in (PARSE), The source file may be preceeded by a project/programmer specification, which should be in the form (projprog). This is the same convention as for the LISP 1.6 INPUT function. Examples:

```
(PARSE FOO)
(PARSE (FOO.M2))
(PARSE (1 DAV) FOO)
(PARSE (1 DAV) (FOO.M2))
```

(PARSE SOURCE-FILE DEST_FILE)

This sets the input to the source file, and also sets up a destination file onto which the translation of the source file will be printed. Again the input file may have an extension and may be preceeded by a project/programmer specification. If the source file is NIL, input will be accepted from the teletype. The destination filename must be an atom; the translation is printed onto <name>.LSP . Again each top-level expression in the program is evaluated as it is translated. Examples:

```
(PARSE FOO BAZ)
(PARSE (1 DAV) (FOO.M2) BAZ)
```

(PARSE SOURCE-FILE DEST_FILE NIL)

This is the same as the above case, except that evaluation of the translated expressions is inhibited. In this mode MLISP2 acts like a compiler-conipiler, translating and printing out the translation without altering itself. This should be used whenever it is desired to print out a complete translator. Examples:

```
(PARSE FOO BAZ NIL)
(PARSE (1DAV)(FOO.M2) BAZ NIL)
```

(PARSE SOURCE-FILE PEST-FILE T)

This is the same as (PARSE SOURCE-FILE DEST_FILE).

Input functions

There are five predicates for checking the type of the next token in the input. These return T if the next token is of the specified type, NIL otherwise. The input is not changed,

EXPR ISIDENTIFIER()
EXPR ISSTRING()
EXPR ISNUMBER()
EXPR ISDELIMITER0
EXPR ISSEXPRESSION()

There are five corresponding functions for fetching the next token in the input, after first checking its type. These return the token if it is of the specified type, otherwise they execute FAILURE (). The input pointer is advanced over the token.

EXPR IDENTIFIER0
EXPR STRING0
EXPR NUMBER()
EXPR DELIMITER0
EXPR SEXPRESSION()

There are also several functions for manipulating the next token without regard to its type.

EXPR TOKEN()

This returns a dotted pair in the form (next-token . type). The token type is a small integer between 0 and 4:

0 - identifier type
1 - string type
2 - number type
3 - delimiter type

4 - sexpression type

The inputpointer is advanced over the token.

EXPR PEEK ()

This is the same as TOKEN except that it does not change the input. It just peeks ahead at the next token,

EXPR NEXT (ATOM!

This is a predicate. Its value is T if the next token in the input is EQ to its argument, otherwise NIL. The input is not changed.

EXPR PROPERTY(INDICATOR)

This checks if the next token in the input has a property under the specified indicator. If so, it returns the property, otherwise NIL. This first checks to make sure the next token is not a number, since GET of a number causes an error in LISP 1.6. The input is not changed.

Backtracking functions

EXPR FAILURE 0

This causes backtracking, as described section 2.

EXPR FLUSH0

This flushes old contexts out of the system, as described in section 2. Its value is NIL,

EXPR CONTEXT 0

This returns the current backtracking context (a small integer). This is useful in conjunction with the function below for manipulating contexts.

EXPR SET-CONTEXT (ATOM, PROPERTY, INDICATOR, CONTEXT)

This function maybe used to change the property list of an atom in a given backtracking context, If the indicator s VALUE, then the effect is to assign the property-list variable a valtie in the specified context. Actualy, tne property is changed in all contexts from the current one back to the specified one. That is to say, if a failure occurs, the property change will not be undone until the

failure happens in an earlier context than the specified one. Setting the property in context zero will insure that failure never undoes the change. The value ot SET-CONTEXT is the value of the second argument.

Other routines

EXPR ERROR (STRING)

This is the standard MLISP2 error handler. It prints out the error message which is its argument on the teletype, then enters the incore editor. The incore editor, which prints instructions when called, gives the user a chance to correct the input and resume translating, without having to begin all over again. After the input has been corrected, ERROR calls <PROGRAM> again. (This is not the ideal solution, but it permits recovery from some types of errors.)

EXPR FATAL-ERROR (STRING)

This is for non-recoverable errors. After printing the error message on the teletype, this returns to the top level of LISP.

EXPR WARNING(STRING,X)

This is just for warning the user about various conditions. It prints on the teletype first the string, then the second argument, then returns the second argument as its value,

EXPR PRINTTY(X)

This prints its argument onto the teletype, no matter what output file is currently selected. It does not change the selected output file. Its value is the value of its argument.

FEXPR LAPIN(L)

This just calls EVAL('DSKIN CONS L), after first setting the input radix to 8 (octal). Since the radix for numbers in MLISP2 is 10 (decimal) but LAP files (and some LISP files) are in octal, this is often useful. After doing the DSKIN, the input radix is reset to its old value.

SECTION 14

MLISP incompatibilities

Things that worked in MLISP that will not work in MLISP2

The changes that MLISP2 makes to the syntax of MLISP are primarily in the form of additions. In general, any legal MLISP EXPRESSION will be accepted by MLISP2. However, there are a few MLISP syntactic constructions which will not be accepted by MLISP2. These have all been mentioned above, but are summarized here,

1. A PROGRAM in MLISP is surrounded by a BEGIN-END pair and terminated with a period. In MLISP2 there is no enclosing BEGIN-END pair, and _EOF_ terminates the program.

2. MLISP2's DEFINE expression is slightly less general than MLISP's.

3. Exclamation point "!" and colon ":" are not legal letters in MLISP2, though they are in MLISP, However they may still be included in identifiers, as may any special character, by preceeding them with the "literally" character: a question mark "?".

4. The vector operator "⊗" may not be used with the assignment operator "←" in MLISP2.

SECTION 15

Appendix

```
% ********** Complete definition of MLISP2 in MLISP2 ********** %


LEJ PROGRAM (*) =
        { {REP 0 M * {<EXECUTED_EXPRESSION> '; [FLUSH]}} !#_EOF_ 3
        MEAN NIL;


LET EXECUTED-EXPRESSION (EX,*) =
        { <EXPRESSION> !#'; }
        MEAN
        IF NULL EX THEN NIL
        ELSE    BEGIN
                IF ?!DEFINE THEN TERPRI PRINT EVAL EX;
                IF ?!PRINT THEN PUTOUT(EX, T);
                END;


LET EXPRESSION (P,EX,L) =
        { <PREFIXES> <PRIMARY>
                {REP 0 M * {<INFIX> <PREFIXES> <PRIMARY> }}
        }
        MEAN
        IF P | L THEN HIER(P CONS EX CONS L, 0)[2] ELSE EX;


LET PRIMARY (X) =
        { {ALT} }
        MEAN X123;


LET SIMPEX (B,L)PRIMARY=
        { <BASIC> {REP 0 M * {<QUALIFIER>}} }
        MEAN
        IF NULL L THEN B
        ELSE FOR NEW I IN L DO B ← CASE CAR(I ← I[1]) OF
```

```
                BEGIN
                B CONS I[3];

                <'?&INDEX, 6, 'LIST CONS I[3]>;

                <'GET, B, CASE I[3,1]OF
                                BEGIN <'QUOTE, I[3,2]>;I[3,2]; END>;

                IF ATOM B THEN
                        TSETQ(B,I[4], IF I[2] THEN I[2,2] ELSE NIL)
                ELSE IF I[2] THEN
                        IF B[1] EQ 'GET THEN
                                <'SET-CONTEXT, B[2], I[4], B[3],
                                        I[2,2]>
                        ELSE <'SET_CONTEXT,B,I[4], '(QUOTE VALUE),
                                        I[2,2]>
                ELSE IF B[1] EQ 'GET THEN <'PUT, B[2], I[4], B[3]>
                ELSE IF B[1]EQ'?&INDEX THEN
                        <'PROG2, <'?&REPLACE, B[2],B[3],
                                        <'SETQ,B← GENSYMO, I[4]>>,
                                B>
                ELSE <'STORE, B,I[4]>;
                END:


LET BASIC (X) =
        { {ALT [ID]
             | [NUMBER]
             | [STRING]
             | #'' [SEXPRESSION]
             | '< <ARGUMENTS> !'>
             | '( <EXPRESSION> !')
             | #OCTAL [BEGIN NEW IBASE;IBASE←8;SCANNER();
                             RETURN NUMBERO; END]
        3 |
        MEAN
        IF X[1]EQ 3 THEN <'QUOTE,X[2]>
        ELSE IF X[1] EQ 4 THEN <'QUOTE, X[3]>
        ELSE IF X[1] EQ 5 THEN 'LIST CONS X[3]
        ELSE IF X[1] EQ 6 THEN X[3]
        ELSE IF. X[1] EQ 7 THEN Xt31
        ELSE X[2];


LET QUALIFIER (Q) =
        {{ALT'( <ARGUMENTS> !')
```

```
            |  '[ <ARGUMENTS> !']
            |  '.  {ALT [IDENTIFIER] | <BASIC>}
            |  {OPT  '{  <EXPRESSION>  !'I  } '←  <EXPRESSION>
        } }
MEAN Q;


LET LET (*,?!PROD,*,PARAM,*,ALT,*,*,SYNTAX,*,*,SEMANTICS) PRIMARY =
        { LET  [IDENTIFIER]  !'( <PARAMETERS> !')
                {OPT [IDENTIFIER])  !'= <LBR>  <PATTERN>  <RBR>
                !'MEAN <EXPRESSION>'
        }
        MEAN
        BEGIN NEW ARGS, NARGS, PUSHLIST, LAM, ?!PROD?#
                ?!CODE, ?!PC, ?!LAST, LOC, CONLISJ, GEN. REMDB;
        % Make a name for the syntax routine and check'it %
        ?!PROD?# ← SYNAM(?!PROD, ?!PROD.?!PROD?#);
        IF ?!PROD?#.SUBR THEN WARNING("PRODUCTION REDEFINED", ?!PROD)
        ELSE CHECKDEF(?!PROD);
        IF ?!PROD MEMQ ?!PRODUCTIONS THEN
                WARNING("PRODUCTION MULTIPLY-DEFINED" ?!PROD)
        ELSE ?!PRODUCTIONS{0} ← ?!PROD CONS ?!PRODUCTIONS;
        % Find the number of non-* arguments to semantics routine %
        NARGS ←0;
        FOR NEW P IN PARAM DO
                IF P[1,1] EQ 1 THEN
                        BEGIN
                        ARGS ← (IF P[1,2] THEN SPECIALDEC(P[1,3])
                                ELSE P[1,3]) CONS ARGS;
                        NARGS ←NARGS+1;
                        PUSHLIST ← 'PUSH CONS PUSHLIST;
                        END
                ELSE PUSHLIST ← NIL CONS PUSHLIST;
        % Syntax %
        LAPST(?!PROD?#);
        EPAT(SYNTAX, NARGS, LENGTH(PARAM), REVERSE(PUSHLIST),
                NARGS NEQ 0);
        EMIT(<'JCALL, NARGS, <'E,?!PROD>>);
        LAPFN(?!PROD?#);
        ?!PROD.CODE{0}← ?!COOE:
        IF ALT  THEN  ADALT(?!PROD,?!PROD?#, ALT ←ALT[1],
                SYNAM(ALT, ALT.?!PROD?#));
        % Semantics %
        LAM ← <'LAMBDA, REVERSE(ARGS), SEMANTICS>;
        IF ?!DEFINE THEN ?!PROD.EXPR{0} ← LAM
        IF ?!PRINT THEN PUTOUT(<'DEFPROP, ?!PROD, LAM  'EXPR>, T);
        PRINTTY ?!PROD;
```

```
        END;


LET PATTERN (L) =
        {  {REP i M * {
                {OPT '!}
                {OPT '#}
                {ALT [IDENTIFIER]
                  |  ''  [TOKEN]
                  |  '<  [IDENTIFIER]  !'>
                  |  '[  <EXPRESSION>  !']
                  |  <LBR>  <META>  <RBR>
        }}}  }
        MEAN L;


L E T META(X) =
        {{ALT 'R E P  [NUMBER]  {ALT [NUMBER] | 'M}  {OPT '*}
                      <LBR>  <PATTERN>  <RBR>  {OPT  <PATTERN>}
            |  'OPT  <PATTERN>
            |  'ALT  {REP  0  M * {<PATTERN>} '|}
        } |
        MEAN X;


LET BEGIN (*,VARS,EXS,*) PRIMARY =
        { BEGIN <DECLARATIONS> <EXPRESSIONS>  !END
        MEAN
        'PROG CONS VARS CONS EXS;


L E T  I F (*,E1,*,E2,E3) PRIMARY =
        { IF <EXPRESSION>  ! THEN  {REP 1  M * {<EXPRESSION>} ALSO}
                {OPT E  L  S  E {REP  1  M * {<EXPRESSION>} ALSO}}
        ;
        MEAN
        'COND   CONS (E1 CONS MAPCAR('CAR,E2))
                CONS  ( IF ¬E3  THEN  NIL
                        ELSE IF ¬CDR(E3 ← E3[2]) & - A T O M E3[1,1]
                                & E3[1,1,1] EQ 'COND THEN CDAAR E3
                        ELSE <'T CONS MAPCAR('CAR,E3)>);
```

```
LET FOR (L,D,EX,BE) PRIMARY =
        { {REP 1 M *
                { FOR {OPT NEW} ! [ID]
                        {ALT 'IN   <EXPRESSION>
                          | 'ON   <EXPRESSION>
                          | '←    <EXPRESSION>   !TO <EXPRESSION>
                                {OPT BY <EXPRESSION>}
                } }
                ! {ALT DO|COLLECT|'; [ID]}   <EXPRESSION>
                {OPT   {ALT WHILE|UNTIL}   <EXPRESSION>)
        }
        MEAN
        <'?&FOR,  <'QUOTE,  MAPCAR(FUNCTION(LAMBDA (I): <
                        IF I [2] THEN 'NEW ELSE 'OLD,'
                        I [3,2],
                        (I ← I [4]) [2],
                        IF I [1] EQ 3 THEN
                                <'?&RANGE, I [3], 1151,
                                        IF I [6] THEN I [6,2] ELSE 1>
                        ELSE I [3]>), Lb,
                <'QUOTE,  CASE D[2,1] OF'
                                BEGIN 'PROG2; 'APPEND; D[2,3]; END>,
                <'QUOTE,  EX>,
                <'QUOTE, IF BE THEN
                                IF BE[1,1] EQ 1 THEN <'NOT, BE[2]>
                                ELSE BE[2]
                          ELSE NIL>>;


LET WHILE (W,BE,D,EX) PRIMARY =
        { {ALT WHILE|UNTIL}   <EXPRESSION>
                ! {ALT DO|COLLECT}   <EXPRESSION>

        MEAN
        <'?&WHILE,
                <'QUOTE,  IF D[2,1] EQ 1 THEN 'PROG2 ELSE 'APPEND>,
                <'QUOTE,  IF W[1] EQ 1 THEN BE ELSE <'NOT,  BE>>,
                <'QUOTE,  EX>>;


LET DO (D,EX,W,BE) PRIMARY =
        { {ALT DO|COLLECT}   <EXPRESSION>
                ! {ALT WHILE|UNTIL}   <EXPRESSION>

        MEAN
        <'?&DO,  <'QUOTE,  IF D[1] EQ 1 THEN 'PROG2 ELSE 'APPEND>,
```

```
                <'QUOTE, EX>,
                <'QUOTE, IF W[2,1] EC 1 THEN <'NOT, BE> ELSE BE>>;


LET FNDEF (TYP,NAME,LAM) PRIMARY =
        { [FUNCTION_TYPE] [ID] <LAMBDA-BODY> }
        MEAN
        BEGIN
        CHECKDEF(NAME);
        IF TYP EQ 'LEXPR THEN
                IF LENGTH(LAM[2]) EQ 1 THEN
                        LAM ← <'LAMBDA, LAM[2,1], LAM[3]>
                        ALSO TYP ← 'EXPR
                ELSE ERROR("LEXPRS MUST TAKE ONE FORMAL ARGUMENT");
        IF ?!DEFINE THEN NAME.(TYP)(0) ← LAM;
        IF ?!PRINT THEN PUTOUT(<'DEFPROP, NAME, LAM, TYP>, T);
        PRINTTY NAME;
        END;


LET LAMBDA (*,LAM,ARGS) PRIMARY =
        { LAMBDA <LAMBDA-BODY> [OPT ';' ( <ARGUMENTS> !') ]}
        MEAN
        IF ARGS THEN LAM CONS ARGS[3] ELSE LAM


LET CASE (*,EX,*,*,EXS,*) PRIMARY =
        { CASE <EXPRESSION> !OF !BEGIN <EXPRESSIONS> !END }
        MEAN
        BEGIN NEW LABELS, L, LAB;
        FOR NEW E IN EXS DO
                PROG2( LABELS ← (LAB ← GENSYM()) CONS LABELS,
                        L ← <'RETURN, E> CONS LAB CONS L);
        RETURN 'PROG CONS NIL
                CONS <'GO, <'?&INDEX, <'QUOTE, REVERSE LABELS>,
                        <'LIST, EX>>>
                CONS REVERSE(L);
        END;


LET INLINE (*,L) PRIMARY =
        { #INLINE [SEXPR_LIST] }
        MEAN
        BEGIN NEW GEN, CDNLIST, LOC, REMDB, FN;
```

```
        CHECKDEF(FN ← L[1,2]);
        IF ?!DEFINE THEN
                BEGIN
                GEN ← GENSYMO;    CONLIST ← <NIL>:    LOC ← BPORG;
                FOR NEW I IN CDR L DO
                        IF ATOM I THEN I & DEFSYM(I,LOC)
                        ELSE DEPOSIT(LOC,GWD(I)) ALSO UPLOCO;
                DEFSYM(GEN, LOC);
                FOR NEW I IN CDRCONLIST DO
                        PROG2(DEPOSIT(LOC, GWD(I)), UPLOC());
                FN.(L[1,3])[0] ← NUMVAL(BPORG);
                BPORG ← LOC;
                END:
        IF ?!PRINT THEN
                OUTC(T,NIL) ALSO BASE4 ALSO TERPRI MAPC('PRINT,L)
                ALSO BASE40 ALSO OUTC(NIL,NIL);
        PRINTTY FN;
        END:


LET SELECT (*,VALFN,*,DOMAIN,SUCFN,TCOND,TERFN) PRIMARY =
        [ S E L E C T [OPT <EXPRESSION>]
                FROM     [ALT [ID]':  <EXPRESSION> |<EXPRESSION>]
                [OPT SUCCESSOR  <EXPRESSION>]
                [OPT UNLESS     <EXPRESSION>]
                [OPT FINALLY    <EXPRESSION>]

        MEAN
        BEGIN NEW VAR:
        CASE DOMAIN[1] OF
                BEGIN
                PROG2(VAR ← <DOMAIN[2]>,DOMAIN ←DOMAIN[4]);
                IF VALFN | SUCFN | TCOND | TERFN THEN
                        ERROR("VARIABLE NEEDED IN SELECT EXPRESSION")
                ELSE VAR ←<INTERNGENSYM()> ALSO DOMAIN ←DOMAIN[2];
                END;
        RETURN <'?&SLCT, DOMAIN,
                IF VALFN THEN <'FUNCTION,  <'LAMBDA, VAR, VALFN[1]>>
                ELSE  '(QUOTE CAR),
                IF SUCFN THEN <'FUNCTION, <'LAMBDA, VAR, SUCFN[2]>>
                ELSE '(QUOTE CDR),
                IF TCOND THEN <'FUNCTION, <'LAMBDA, VAR,TCOND[2]>>
                ELSE  '(QUOTE NULL),
                IF TERFN THEN <'FUNCTION, <'LAMBDA, VAR,TERFN[2]>>
                ELSE ' (QUOTE FAILURE)>;
        END;
```

```
LET RECOMPILE (*,FNS,*, IFILE,PPN,OFILE) PRIMARY =
        { RECOMPILE {REP 1   M {{IDENTIFIER}} ',}
                !'IN   <FILE>   {OPT   '  [  [TOKEN]  !',   [TOKEN]   !'] }
                {OPT   'TO   <FILE>  }
        }
        MEAN
        BEGIN NEW NFNS, N, ?!PRODUCTIONS, ?!PRINT, ?!DEFINE;
        IF OFILE THEN
                IF ¬'PPRINT.SUBR THEN
                        ERR PRINTSTR TERPRI
                                "USE MLISP2.PRI FOR PRINTING"
                ELSE OUTFILE('DSK?:,
                        IF ATOM OFILE ← OFILE[2] THEN OFILE
                        ELSE CAR OFILE,
                        IF ATOM OFILE THEN NIL ELSE CDR OFILE)
                        ALSO ?!PRINT ← T ALSO ?!DEFINE← NIL
        ELSE ?!PRINT ← NIL ALSO ?!DEFINE← T;
        FNS  ← MAPCAR('CAR, FNS);          % List of fns to reconipile %
        NFNS ←LENGTH(FNS);                 %# of fns to reconipile %
        N    ← 0:                          %# of fns conipiled so far %
        EVAL <'INPUT,  IF PPN THEN <PPN[2,1], PPN[4,1]> ELSE 'DSK?:,
                IFILE>;
        INC(T, NIL);
        UNTIL N EQ NFNS DO
                BEGIN NEW XI, X2, TX1,TX2;
                IF ((TX1 ← SCAN()) EQ ?!IDTYPE
                        & (Xi ← INTERN SCNVAL) MEMQ
                                '(LET EXPR FEXPR LEXPR MACRO)
                        & (TX2 ← SCAN()) EQ ?!IDTYPE
                        & (X2 ← INTERN SCNVAL) MEMQ FNS)
                |(X1 EQ 'INLINE
                        & (X2 ← SREAD())[2] MEMQ FNS
                        & TX2 ← ?!SEXPTYPE)
                |(X1 EQ 'SPECIAL
                        & (TX2 ← SCAN()) EQ ?!IDTYPE
                        6 x2 ← INTERN SCNVAL) THEN
                        BEGIN
                        % set token stack to contain only XI and X2 %
                        SET-TOKENS (X1,TX1, X2, TX2);
                        %Parse an <EXPRESSION> %
                        EXPRESSION?#(j ;
                        IF NEXT('?;) THEN FLUSH()
                        ELSE ERROR("ILLEGAL EXPRESSION RECOMPILED");
                        IF XI NEQ'SPECIAL THEN N ←N+1;
                        END
                ELSE SKIP-TO-SEMI (NIL);
```

```
            END;
        INC(NIL, T);
        TERPRI TERPRI IF ?!PRINT THEN FINISH-PRINTINGO:
        END;


LET DEFINE (*,L) PRIMARY =
        { OEFINE   {REP 1 M *{
                EIDENTIFIERI
                {ALT 'P R E F I X {OPT [TOK]}   {OPT [NUMBER]}
                  | [NUMBER]  !  [NUMBER]
                  |[TOK]{OPT       [NUMBER]    ![NUMBER]}
        }} ',} 3
        MEAN
        FOR NEW I IN L DO
                CASE i [2,1]OF
                        BEGIN

                        BEGIN
                        I [1].?&PREFIX{0}← I ill ;
                        I i11. ?&RIGHT{0}←
                                IF I [2,4] THEN I [2,4,1] ELSE 1000;
                        IF I [2,3] THEN
                                I [2,3,1].?&PREFIX{0}← I [1];
                        END;

                        BEGIN
                        I  [1].?&LEFT {0} ← I [2,2];
                        I [1].?&RIGHT{0} ← I [2,3,2];
                        END:

                        BEGIN
                        I  [2,2].?&INFIX{0} ← I [1];
                        IF I [2,3] THEN
                                        I [1].?&LEFT {0} ← I [2,3,1]
                                ALSO I [1].?&RIGHT {0} ← I [2,3,2,2];
                        END;

                        END;


LET COMMENT (*) PRIMARY =
        { COMMENT [SKIP-TO-SEMI (T)]}
        MEAN NIL:
```

```
LET SPECIALS (*,L) PRIMARY =
        { SPECIAL  <IDLIST> }
        MEAN
        MAPC('SPECIALDEC, L);


LET LAMBDA-BODY (*,VARS,PVARS,*,*,EX)=
        {!'(<VARIABLES>{OPT':<VARIABLES>}!')!';<EXPRESSION>}
        MEAN
        <'LAMBDA, VARS,
                IF PVARS THEN <'PROG,PVARS[2], <'RETURN, EX>>
                ELSE EX>;


LET DECLARATIONS (L)=
        {{REP 0 M*{{ALT NEW |     SPECIAL1   <IDLIST> !'; }} }
        MEAN
        FOR NEW I IN L COLLECT
                IF I[1,1] EQ 1 THEN I[2]
                ELSE MAPC('SPECIALDEC,I[2]);


LET EXPRESSIONS (L)=
        {{REP 0     M*{<EXPRESSION>}';  [FLUSH]} }
        MEAN
        MAPCAR('CAR, L);


LET IDLIST(L) =
        { {REP 0 M * {[ID]} ',} }
        MEAN
        MAPCAR('CAR, L);


LET VARIABLES (L)=
        {{REP 0 M *{{OPT SPECIAL)  [ID]} ',} }
        MEAN
        MAPCAR(FUNCTION(LAMBDA (X);
                IF CAR X THEN SPECIALDEC(X[2]) ELSE X[2]),L);
```

```
LET PARAMETERS (L) =
        { {REP 1 M * {{ALT  {OPT SPECIAL}  [ID] | '*}} ',} }
        MEAN L;


LET ARGUMENTS (L) =
        { {REP 0 M * {<EXPRESSION>}   } }
        MEAN
        MAPCAR('CAR, L);


LET PREFIXES (L) =
        { {REP 0 M { [PREFIX]  {OPT '@} }} }
        MEAN
        REVERSE(L);


LET INFIX (L) =
        { [INFIX1]  {OPT  @} }
        MEAN L;


LET FILE (NAM, EXT  =
        { [TOKEN]   OPT '. [TOKEN]} }
        MEAN
        IF EXT THEN NAM[1] CONS EXT[2,1] ELSE NAM[1];


LET LBR (*) =
        {{ALT '{ | '(  }        %  (  may be used instead of "{" %
        MEAN NIL;


LET RBR (*) =
        {!{ALT '} | ') }}       % ")" may be used instead of "}" %
        MEAN NIL;
```

SECTION 16

Bibliography

1. Bobrow, D.G. "Requirements for Advanced Programming Systems for List Processing" Comm. ACM 15, 7 (July 1972), 618-627.

2. Enea, H.J. MLISP Computer Science Department Report CS-92, Stanford University, 1968.

3. Floyd, R.W. "Nondeterministic Algorithms" J.ACM 14, 4 (Oct. 1967), 636-644.

4. McCarthy, J., Abrahams, P., Edwards, D., Hart T. and Levin, M. LISP 1.5 Programmer's Manual Massachusetts Institute of Technology, MIT Press, 1965.

5. Milner, R. Logic for Computable Functions, Description of a Machine Implementation Artificial Intelligence Project Memo AIM-169, Stanford University, 1972,

6. Smith, D.C. MLISP Artificial Intelligence Project Memo AIM-135, Stanford University, 1970.

7. Smith, D.C. and Enea, H.J. "Backtracking in MLISP2", submitted to the Third International Joint Conference on Artificial Intelligence, Stanford, 1973.

SECTION 17

Index