LISP
Preliminary Programmer's Manual - Draft

January 7, 1960

J. McCarthy
R. Brayton
D. Edwards
P. Fox
L. Hoddes
D. Luckham
K. McHale
D. Park
S. Russell

M.I.T. Computation Center
and
Research Laboratory of Electronics
Preface and Acknowledgements

This manual was mainly written by Fox on the basis of information supplied by McCarthy and the authors of specific sections of LISP.

The overall design of the system is the work of McCarthy. Certain ideas from Fortran, Gelernter's FLPL, Newell, Shaw and Simon's IPL, and of N. Rochester were used.

The apply operator was written by Russell starting from a preliminary LISP version by McCarthy.

The print and read programs were written by McCarthy and Maling, respectively.

The garbage collector was written by Edwards.

The parts of the system dealing with algebraic calculations and floating-point numbers were written by N. Rochester, S. Goldberg and Edwards.

The compiler was written by Brayton with the assistance of Park.

The flexo-system was written by Edwards and Luckham.

All the listed authors contributed to the collection of functions available with the system.

The secretarial work was done by Mrs. Marcia Webber.
Additions and Corrections to LISP Preliminary Manual
January 29, 1960

Page 2, last two lines should read:
a = a list of pairs of the form, (atomic symbol, value),
used to assign values to free variables.

Page 5, The function define is really a pseudo-function (see
below). Note that define makes the use of label desirable
only when the labeled function is itself the result of a
computation and is not to be put on an association list
for storage reasons.

Page 6, A further distinction should be made between functions
and pseudo-functions. In LISP a function is evaluated
for its value as such, e.g. car [(A,B,C)] = A, whereas a
pseudo-function is wanted for its effect rather than its
value, e.g. define, compile, etc. There is no differ­
ence in form however between functions and pseudo-func­
tions except that pseudo-functions are not usually used
as sub-expressions.

Page 12, Note that the compiler destroys the EXPR and S-expres­
sion on the association list of the function it compiles.
If these are to be saved one should compile a copy of
the function. Compile is a pseudo-function.

Page 17, line 3 should read:
of each triplet should be punched on cards separated by
one or more spaces (blanks).

Page 17, line 5 from the bottom:
That is, room 26-265 at M.I.T.!
Additions and Corrections

Page 22, line 9 should read:
There is a free-storage counter, called the CONS counter, associated with the LISP system.

Page 25, lines 10,12,13,19,24 and Page 26, line 7
The name trackless should be changed to tracklist.

Page 25, line 17 should read:
Further, the function must not be one of the ones built-in to the APPLY operator.

Page 28, replace the last four lines by:
direction cards (or type-ins). One can then type either

\[ \rightarrow b\text{FIN} \]

To end the run, or

\[ \rightarrow b\text{GRD} \]

to return control to the card-reader for the next direction card, e.g. TST,SET,FLX,FIN

Page 30, line 11 should read (b is a blank (space)):

\[ n_{i+1} \rightarrow \rightarrow b\text{FIN} \]

Page A5, lines 6 and 7 from the bottom should read;

\[ \text{cond} \left[ x_1; x_2; \ldots; x_n \right] : \text{machine language; special form} \]

The function \text{cond} has a variable number of arguments each one of which is a pair of expressions the first of which is propositional.
Contents

1. Introduction

2. Programming in LISP
   2.1 The APPLY Operator
   2.2 Definitions of Functions in LISP
   2.3 Functions Appropriate to APPLY
   2.4 Numbers in LISP
   2.5 The Program Feature
   2.6 The Compiler

3. Running LISP Programs
   3.1 Card Punching and Submitting a Run
   3.2 Error Indications
   3.3 Tracing Options
   3.4 The Flexwriter System

4. List Structures
   4.1 General Description
   4.2 Association Lists
   4.3 The Garbage Collector

5. Exercises with Solutions

6. Functions Available in the LISP System
This manual is a preliminary version of a programmer's manual for the IBM 704 LISP programming system. The material included in the manual is subject to change dependent on the evolution of the LISP system itself; as it stands, the manual refers to the system as of January 1, 1960.

In the interests of quick publication, the LISP system will not be described from scratch, but rather the manual will assume knowledge of the report Recursive Functions of Symbolic Expressions and their Computation by Machine by John McCarthy which appeared in the M.I.T. Research Laboratory of Electronics Quarterly Progress Report No. 53. This article, which will be referred to as RFSE, gives the theoretical basis of the system, but is light on practical details.

The current LISP system uses about half of the 32K memory of the 704. It includes routines for reading and for printing, and it contains many LISP functions either coded in machine language or given as lists of S-expressions for the interpreter part of the system. A list of these available functions is included in this manual and of course other functions may be defined by the user. There is also a provision made in the LISP system for compiling S-expressions into machine code to avoid the time-consuming operation of interpreting. Finally, for on-line operation using the flexwriter, a package of routines to control the input-output aspects of the flexwriter has been appended to the system. Descriptions of these various features are given in the manual together with an explanation of the procedure to be followed in submitting, running, and debugging a program written in the LISP language.

The kinds of computations for which LISP is suitable are primarily those involving symbolic formula evaluation, particularly of a recursive nature. The heuristics to be tried in puzzle-solving or game-playing programs can be phrased nicely in LISP language. Some warning should be given here...
however about the time-consuming aspects of using the interpretive mode of LISP for heuristic branching calculations. A great deal of computer time can be absorbed unless the compiler is used in preference to the interpreter in these cases. LISP is not suitable, as it stands, for integer or floating-point arithmetic, but a new version, LISP II, which is under study will be more capable in these areas. LISP also is not a good way to simulate, for example, a Turing machine on the 704, -FORTRAN is better here. Types of programs particularly suitable for LISP are the differentiation and simplification programs now available, and the integration program now being worked on, all of which are done by symbolic formula evaluation.

2. Programming in LISP

In this section are included descriptions of the main features of the LISP system with which a programmer should be familiar before attempting to write down a LISP program. The various ways of defining and evaluating symbolic expressions for functions are given, and some discussion is added on the use of numbers in LISP. The Program Feature, which is a somewhat FORTRAN-like feature, is explained, and finally the LISP compiler is described.

2.1 The Apply Operator

The basis of LISP programming is the APPLY operator which is a synthesis of the apply and eval functions described on page 135 of RPSE. The apply now in use is a function of three arguments,

\[
\text{apply}([f;x;a])
\]

where

\( f \) = an S-expression representing an S-function,
\( f^n \), of say \( n \) arguments,
\( x \) = a list of \( n \) arguments,
\( a \) = a list of pairs of the form, (S-expression, atomic symbol), used to assign values to free variables.
The value of \( \text{apply} [f; x; a] \) is the value of the S-function \( f \) represented by \( f \), where any free variables in the S-expression have been evaluated by using the list \( a \).

**Example 1:**

As a simple first example consider the function
\[
\text{cons} \left[ \text{car} [y] ; \text{cdr} [x] \right]
\]
applied to the list
\[
((A,B),(C,D)),
\]
where no free variables need to be assigned. The arguments \( f, x, \) and \( a \) for the \text{apply} function, written in the form of S-expressions, are,
\[
f: \quad (\text{LAMBDA}, (Y,Z), (\text{CONS}, (\text{CAR}, Y), (\text{CDR}, Z))),
\]
\[
x: \quad ((A,B),(C,D)),
\]
\[
a: \quad ( )
\]
and the value of \( \text{apply} [f; x; a] \) is \((A,D)\).

**Example 2:**

Some care must be exercised in writing the lists \( x \) and \( a \) correctly. In the example,
\[
f: \quad \text{CAR},
\]
\[
x: \quad ((A,B)),
\]
\[
a: \quad ( )
\]
where \( \text{apply} [f; x; a] = \text{car} [(A,B)] = A \), since \( \text{car} \) is a function of one variable, the list \( x \) must be written as \((A,B))\) where \((A,B)\) is the single argument. The list,
\[
x: \quad (A,B)
\]
would be wrong. Note also that the LAMBDA definition is not needed in this example since the specification of the single argument is clear. It would be correct but unnecessary to write
\[
f: \quad (\text{LAMBDA}, (Y), (\text{CAR}, Y)),
\]
\[
x: \quad ((A,B)),
\]
\[
a: \quad ( )
\]
The exact descriptions of function formats acceptable to the \text{APPLY} operator are described further in Section 2.3.
Example 3:
As an example involving an a-list we have
\[ f: \quad (\text{LAMBDA}, (y), (\text{CONS}, y, z)), \]
\[ x: \quad (a), \]
\[ a: \quad ((z, (b))) \]
where \( \text{apply}[f; x; a] = \lambda [y; \text{CONS}[y; (b)]][a] = \text{CONS}[a; (b)] = (a, b) \).
In this example, the \( \lambda \)-specified argument, \( y \), is given as \( A \) by the argument list \( x \), and the free variable, \( z \), is set by the a-list to \( b \).

Example 4:
The following example involves a function used recursively so that a \text{label} definition is required,
\[ \text{subst}[x; y; z] = \begin{cases} \text{atom}[z] \rightarrow [y = z \rightarrow x; T \rightarrow z] ; \\ T \rightarrow \text{cons}[\text{subst}[x; y; \text{car}[z]]; \text{subst}[x; y; \text{cdr}[z]]] \end{cases} \]
\[ f: \quad (\text{LABEL}, \text{SUBST}, (\text{LAMBDA}, (x, y, z), (\text{COND}, \text{ATOM}, z), (\text{COND}, ((EQ, y, z), x), (T, z))), (T, (\text{CONS}(\text{SUBST}, x, y, (\text{CAR}, z)), (\text{SUBST}, x, y, (\text{CDR}, z)))))), \]
\[ x: \quad ((a, b), d, ((d, c), d)) \]
\[ a: \quad (\ ) \]
where \( \text{apply}[f; x; a] = \text{subst}[(a, b); d; ((d, c), d)] = (((a, b), c), (a, b)) \)

Note: The system for translating \( \lambda \)-expressions into \( S \)-expressions given on page 135 of RTSP would give rise to \((\text{QUOTE}, T)\) in lieu of \( T \) in the above expression. It is simpler to be able to write \( T \) (\( \text{car} \)) and so for the cases of truth (or falsity). The choice has been made in the 704 \text{LISP} system to require the \( T \) (or \( F \)) usage. \((\text{QUOTE}, T)\) and \((\text{QUOTE}, F)\) are to be replaced by \( T \) and \( F \) respectively, and only the latter expressions will work.

A program for the \text{LISP} system consists of sequences of \( f; x; a \) triplets strung together. The \( \text{APPLY} \) operator automatically operates on each triplet in turn and returns with the value of the triplet. The details for submitting and running such a program are given in Section 3.

2.2 Definitions of Functions in \text{LISP}

In \text{RTSP}, functions are connected to their names only
through the LABEL of operation. In the current LISP system, there are two further ways a function may be defined:

The first of these relates to functions defined in the system by machine-language subroutines. Such a subroutine for a function may be already available as part of the LISP system itself, in which case it appears on the list of functions given in Section 6, or it may have been produced by the LISP compiler. In either case, if a machine-language subroutine defines a function, the association list for the name of the function will contain the indicator 'SUBR', and a transfer to the routine in the following structure,

If, on the other hand, a function is defined by means of a LISP S-expression (representing an N-expression) which is to be interpreted during the running of a LISP program, the indicator 'EXPR' is on the association list for the name of the function. 'EXPR points to the S-expression defining the function, as shown below:

A function defined in this way by an EXPR may be already available in the LISP system itself, in which case it appears on the list of functions given in Section 6, or it may be defined by the user by using the define function. Define is a function of one variable which is a list of pairs. The first member of each pair is a function name, and the second member is an expression defining a function with that name. The APPLY operator acting on define creates the EXPR structure shown above, for each function defined. If EXPR is

These facilities make the use of label desirable only when the function used is itself the result of a computation and is not to be stored, put on an association list, or saved.
already on the association list for a function, it is changed to point to the new S-expression, so that the define is really a redefine.

Suppose for example that the two new functions, \(ff\) and \(alt\) are to be defined where

\[
\begin{align*}
ff[x] &= \left[\text{atom}[x] \rightarrow x; \text{T} \rightarrow ff[\text{car}[x]]\right], \text{ cf. p. 132 of RFSE,} \\
\text{and} \\
alt[x] &= \left[\text{null}[x] \text{null}\left[\text{cdr}[x]\right] \rightarrow x, \text{T} \rightarrow \text{cons}\left[\text{car}[x], \text{alt}[\text{cdr}[x]]\right]\right]
\end{align*}
\]

These will be defined by letting the APPLY operator evaluate the following triplet, \(f;x;a\), where \(x\) is a list of two pairs.

\[
\begin{align*}
f &:\text{DEFINE,} \\
x &:\left(\left(\left(\left(\text{ff}, \text{(LAMBDA, (x), (COND, (\text{ATOM, x}, x), (T, (\text{ff}, (\text{CAR, x})))})}, (\text{ALT}, \text{(LAMBDA, x), (COND, ((0, (\text{NULL, x}), (\text{NULL, (CDR, x)})}, x), (T, (\text{CONS, (CAR, x), (ALT, (\text{CDR, (CDR, x)})})})})\right)\right)\right)\right) \\
a &:\left(\right)
\end{align*}
\]

After APPLY has evaluated this triplet, the DEFINE statement will have put the respective definitions, labelled by EXPR on the association lists of the atomic symbols \(FF\) and \(ALT\). Of course, DEFINE itself is one of the functions already defined and available as part of the LISP system.

2.3 Functions Appropriate to APPLY

Some further amplification of the forms the function \(f\) for the APPLY operator can assume might be in order.

First of all, the function \(f\) can be either an atomic symbol (atomic function) or a list (compound function). In the second example in Section 2.1 above, \(\text{CAR}\) is an atomic function, whereas the LAMBDA expression is a compound function.

Atomic Functions

If a function is atomic, the association list for the atomic symbol is searched by the APPLY operator for either \(\text{SUER}\) or \(\text{EXPR}\) (see Section 2.2 just preceding). If either
is found, the function which it points to, represented by a subroutine or an S-expression respectively, is used to evaluate the atomic function of the list \( \pi \) of arguments. If neither \texttt{SUBR} or \texttt{EXPR} is found on the association list for the function, then the \( \alpha \)-list is searched for the function's atomic symbol, and the expression paired with the symbol is used to evaluate the atomic function of the list \( \pi \) of arguments.

\textbf{Compound Functions}

If a function is compound, the first element of the list, or S-expression, for it may be any of \texttt{LAMDA}, \texttt{LABEL}, or \texttt{FUNARG}.

If the list begins with \texttt{LAMDA} for example \((\text{LAMDA}, (Y), (\text{CAR}, Y))\) the second element of the list is a list of the names of the dummy or bound variables for the function, and the third (last) element of the list is a form for the function to be evaluated. The \texttt{APPLY} operator takes the names of the dummy variables and pairs them with the values given on the list \( \pi \) of arguments. If the two lists are not of the same length an error stop occurs. Otherwise the list of pairs is added to the front of the \( \alpha \)-list. Then the third element of the \texttt{LAMDA}-list, the form for the function, is evaluated using the enlarged \( \alpha \)-list of pairs of assigned values.

The third element of the \texttt{LAMDA} list, the form for the function may itself be either an atomic symbol (atomic form) or a list (compound form). If it is an atomic form, its association list is searched to see if it is a constant (signalled by either \texttt{APVAL} or \texttt{APVALI} on its association list—cf. Section 4), and if so, the value of the atomic form is the constant, i.e., \texttt{CAR} if the list to which \texttt{APVAL} or \texttt{APVALI} points. If the atomic form is not a constant, the \( \alpha \)-list is searched for the most recent pairing of this variable on the \( \alpha \)-list, and the value given by this pairing is the value of the form.
If, on the other hand, the form which is the third element in the LISPDA expression is a compound form instead of an atomic form, its first element may be either a function or an atomic symbol. If the element is a function, the rest of the elements in the form are evaluated and the first element, the function, is applied to these values together with the current a-list of pairs. If the first element is an atomic symbol, it is either handled exactly as the atomic function above, or, if it is in special form, it is handled separately as explained below under the note on special forms.

If the first element of a compound function is LABEL (see Example 4 above), the second element of the list is the name of the function, and the third (last) element of the list is its definition. The APPLY operator pairs the second element with the third and adds the pair to the front of the a-list. The third element defining the function is then applied to the list a of arguments using the enlarged a-list, and the result is the value of the function.

If the first element of a compound function is FUNARG, the second element of the list is a function definition, and the third (last) element of the list is a particular list of pairs, an a-list to be used in evaluating the function in place of the a-list of the APPLY operator. In other words, \[ \text{apply} \left[ \left( \text{FUNARG, } x, b \right); x; a \right] \equiv \text{apply} \left[ x; x; b \right]. \]

**Note on Special Forms:**

Certain compound forms are classified under the heading of special forms and are evaluated differently from the usual case. If the first element of a compound form is one of the special forms, the rest of the elements in the compound form are treated in special ways described below under the particular cases. A special form is signalled by either FSUSER or FEXPR in place of SUER or EXPR on its association list.

QUOTE is a special form that prevents its argument from being evaluated. The value of a list beginning with QUOTE is always the second element of the list.
COND is a special form which is a conditional expression. The part of the list following COND is a list of pairs, and for each pair the first element is the proposition and the second element of the pair is the corresponding expression. The propositions are evaluated until one is found that is true, and then the expression corresponding to this proposition is evaluated and taken as the value of the entire conditional. If there are no propositions that are true, an error occurs. The error routine is entered.

AND is a special form to test if all of the propositional expressions following AND in the list are true. The propositions are evaluated until one is found that is false or until the end of the list is reached. The value of AND is respectively F (zero) or T (one).

OR is a special form to test if any of the propositional expressions following OR in the list are true. The propositions are evaluated until one is found that is true or until the end of the list is reached. The value of OR is respectively T (one) or F (zero).

PRG is a special form described under the Program Feature in Section 2.5.

Other special forms will be pointed out in the list of functions available in LISP given below.

2.4 Numbers in LISP

The 704 LISP system ultimately will be able to handle both integers and floating-point numbers, but at present, integers are not allowed.

Floating-point numbers are designated by a decimal point appearing within the number, e.g.

0.42, 120.05, 6., .003

An exponent indication may also be used, for example:

4.214E10 is used for 4.214 * 10^{10}; or
26.-2 is used for 26 * 10^{-2}. 

No spaces can appear within the characters representing a floating-point number.

Such numbers can be read in within a LISP program, for example the number 0.6 can be used in the form to be evaluated,

\[ \text{(PLUS,0.6,X)} \]

(by implication X in this expression stands for a number which is tied to it for example on the \(a\)-list).

An association list for a floating-point number always has the designation FLOAT on the list in place of NAME.

2.5 The Program Feature

The program feature in LISP allows sequences of operations to be expressed in LISP language. The effect is rather like a FORTRAN program with LISP statements. The statements may be any form to be evaluated, or any atomic symbol, or any list beginning with GO or RETURN. The atomic symbols are used as location markers within the program.

When a list beginning with GO is encountered, the rest of that list is evaluated under the assumption that the value will be one of the location-marking atomic symbols. Then a search is made of the program for a symbol equal to this value, and when it is found the statements immediately following the locating symbol are executed. A conditional expression may be used in a GO list, allowing conditional branching.

If a list beginning with RETURN is encountered, the rest of the list is evaluated and the resultant value is the final value of the entire program.

The other statement forms appearing in a program are evaluated in sequence but the resulting values are ignored. This implies that these forms are important mainly for their actions, such as changing lists of various sorts, rather than for their values.

A program form has the structure,

\[ \text{(GO,GO,sequence of statements)} \]
where PROG signals to the interpreter that a program for sequential execution is to follow, and where \textit{L} is a list of program variables which are handled in a manner similar to that for dummy or free variables. These program variables are initially made equal to NIL, but their values may be changed at any point in a computation. It is also even possible to change program variables of some higher level PROG provided that the program variable to be changed has been previously mentioned as a program variable in a PROG. For some programs \textit{L} may be a null list; this is quite legal.

The functions available for changing program variables are \texttt{SET}, \texttt{SETEQ}, \texttt{LOC} and \texttt{LOCQ}. To set a program variable \texttt{V} equal to an expression \texttt{E}, the functional statement
\begin{equation}
\texttt{(SET, (QUOTE, V), E)}
\end{equation}
can be executed. However, since it is annoying to have to write the \texttt{QUOTE} all the time, a special form \texttt{SETQ} has been introduced whereby
\begin{equation}
\texttt{(SET, (QUOTE, V), E)} \text{ is entirely equivalent to } \texttt{(SETQ, V, E)}.
\end{equation}

To provide even more flexibility, the function \texttt{LOC} has been provided. \texttt{LOC} is a function of one argument, the name of a program variable, and the value of \texttt{LOC} is the location whose address points to the value of the program variable. \texttt{LOCQ} again is a special form, available to avoid the need for \texttt{QUOTE}.

Thus
\begin{align*}
\texttt{(HPLACA, (LOC; (QUOTE, V)), E)} & \\
\texttt{(HPLACA, (LOCQ, V), E)} & \\
\texttt{(SET, (QUOTE, V), E)} & \\
\texttt{(SETQ, V, E)} & 
\end{align*}

are all equivalent in their effect.

The following is an example of a program using the program feature. It is a version of \texttt{search}. Although the entire program is in the form of a large list, for ease of reading it has been spaced out below in the form of a
sequential program, with the location-marking atomic symbols set out to the left. The program variable is LT. The LAMBDA expression for this program for the APPLY operator is,

\[(\text{LAMBDA}, (\text{L}, \text{P}, \text{EQ}, \text{U}), (\text{PROC}, (\text{LT})),
\quad (\text{SEQ}, \text{LT}, \text{L}),
\quad \text{TEST}, (\text{GO}, (\text{COND}, ((\text{NULL}, \text{LT}), (\text{QUOTE}, \text{RETU})), (\text{T}, (\text{QUOTE}, \text{CONT})))),
\quad \text{CONT}, (\text{GO}, (\text{COND}, ((\text{P}, \text{LT}), (\text{QUOTE}, \text{RETF})), (\text{T}, (\text{QUOTE}, \text{STEP})))),
\quad \text{STEP}, (\text{SEQ}, \text{LT}, (\text{CDR}, \text{LT})),
\quad (\text{GO}, (\text{QUOTE}, \text{TEST})),
\quad \text{RETU}, (\text{RETURN}, (\text{U})),
\quad \text{RETF}, (\text{RETURN}, (\text{FM}, \text{L}))\)
\]

2.6 The Compiler

The LISP Compiler is itself a function which is available to the APPLY operator, and it is used like any normal LISP function. Compile is a function of one argument which is a list of function definitions, each one of which must be of the form

\[(\text{LABEL}, \text{NAME}, (\text{LAMBDA} \text{ Expression}))\).

The Compiler creates a binary program for the function (or functions) and the value of the compile is a list of names of the functions compiled. A SAP program is printed out for each function compiled (on-line if switch 3 is down, otherwise off-line). After a function has been compiled, it can be used as if it had been defined, but of course it will run much faster than it would have as an interpreted expression.

The Compiler proceeds in three stages:
1. Generation of LISP-SAP
2. Generation of Binary Program
3. Definition.

LISP-SAP is SAP in list form, for example
\[
\left(\text{LAMBDA}, \text{EQ}, \left(\text{EX1}, 1, 0, 1, -1\right), \left(\text{EXA}, \text{E5}\right), \left(\text{G0008}, \text{ESS}, 0\right)\right).
\]
In this example, the objects beginning with 0 are atomic symbols generated for use within the Compiler. The ESS,0 in the last element above is used as it is in SAP to tag symbols which need to have a memory location assigned to them, but no actual space reserved for them, i.e. the usual location-field SAP symbol.

After the Compiler has created the LISP-SAP program for a function, the binary program is generated from LISP-SAP in two passes. The first pass collects all symbols associated with ESS,0 and assigns them locations in memory. The second pass assembles each instruction into memory and assigns to any still unassigned symbols it finds, locations in memory following the generated instructions.

The definition stage of the Compiler appends to the association list of each compiled function the item SUBR pointing to a TH instruction to the appropriate compiled program. This happens even if a SUBR had already been on the list—the new SUBR has precedence and acts as the current definition of the function.

If a programmer has a collection of functions which he wants to compile and if some of the functions use each other as subfunctions, a certain order of compilation must be followed. The rule can be expressed in terms of the two following relationships.

1) If F1 and F2 are functions, and F1 uses F2 as a subfunction, the relation is written as F1>F2.
2) If F1>F2 and F2>F1, then F1 is "equivalent" to F2, and the relation is written as F1~F2.

The functions to be compiled by any given compile statement must be either unrelated in the above sense or equivalent, and any subfunctions which they use must already have been compiled.

For example if there are 4 functions to be compiled and they are related as follows,

\[ F1>F2, \quad F3=F4, \quad F1>F3, \quad F5>F2, \]

functions should be compiled in the order
1. COMPILE ((F1,F2)) , ( )
2. COMPILE ((F5)) , ( )
3. COMPILE ((F3,F4)) , ( )

The functions car,cadr,...,caddr are available to the compiler, so it is possible to write simply (CADAR,X) in lieu of (CAR,(CDR,(CAR,X))). If a string of A's and D's of length greater than three is required, it is again necessary to form the function by composition, i.e. (CDADR(CAR,X)) for (CDADAR,X).

The Compiler will accept any function definition which is acceptable to the interpreter, except that the Program feature is different insofar as the GO statement is concerned. For the compiler version of FROG the argument of any GO must be an atomic symbol; it cannot be a conditional expression. To get around the restriction, conditional statements themselves allowed as action statements provided they give rise to a subsequent action, e.g.

(COND,(P1,(CONS,A)),(P2,(RETURN,X)),(P3,(SEQ,B,(CAR,B))))).

Furthermore, conditional expressions as used here do not need to include a true (T) action. If none of the conditions are satisfied, the next statement in the program following the conditional statement is executed.

Thus the example of the FROG for search given in the previous section must be revised for the Compiler to the following

(LANED,F1,F2,FN,U),(PRC2,(LT),
(SEQ,L,L),
TEST,(COND,((CONS,L),(RETURN,U)),((P,L),(RETURN,(FN,L)))),
(SEQ,L,T,CDR,LT)),
(GO,TEST)
)

Macros can be defined for the Compiler. They are defined by using the APPEND operator on an attribute where n is the name of the macro, and s is an S-expression those
first element is "EXP". The second element in the $\xi$-expression is a list of dummy variables of the form \((X,Y,\ldots,Z)\) where the first dummy variable stands for the location in which the value of the macro (in the LISP sense) is to be stored, and where the rest of the dummy variables are the working variables. The first variable must always be included even if it is not used explicitly as in the first example below. The rest of the $\xi$-expression is a LISP-SAP definition of the macro. The extra null list, ( ), in each instruction below is required by a vagary in the read program.

Three examples follow

1) \[ \text{ATTRIB} \]
\[
\text{(COUNT,"EXP",((X,Y),()),CLA,Y),((),ADD,ONE),}
\]
\[
\text{((),STO,Y))}
\]

In this example, even though the dummy variable \(X\) is not used explicitly it must be included. The effect of the macro is such that whenever the function, COUNT, of one argument, say \(A\), \((\text{COUNT},A)\) is mentioned at some later time in a function to be compiled the above sequence of instructions is compiled as an open subroutine in the compiled program with \(A\) substituted for \(Y\).

2) \[ \text{ATTRIB} \]
\[
\text{(CAR,\text{EXP},((X,Y),()),LKD,Y,4),((),CLA,0,4),}
\]
\[
\text{((),PAX,0,4),((),SHD,X,4))}
\]
\[
\text{())}
\]

In this example, the value of \(\text{car}\) is actually stored at \(X\), though the function is still a function of one argument.

3) \[ \text{ATTRIB} \]
\[
\text{(UNEQ,"EXP",((X,Y,Z),()),CLA,Y),((),SUB,Z),}
\]
\[
\text{((),TZE,=2),((),CLA,ONE),((),STO,X)))}
\]
\[
\text{())}
\]

Note that the SAP notation \(\equiv\text{n}\) is used here. It is allowed only for \(n = 1,2,3,4,5\) and not for \(n = 0\) or any other integer, nor for \(n = \) a negative integer.
The principal symbols and instructions that are acceptable to the Compiler are listed below:

2,3,4,5,6,7,8,9,10,11,12,13,14,15, 18
ONE,ZERO
X,Y,Z
ADD,ALS,ARS
ESS
CLA,COM
LDQ,LMA,LXD
PAX,PDX,PKD
SIA,STD,STO,STQ,SXD
SIX,TIX,TMZ,TRA,TSX,TXH,TXL,TZE
X,Y,Z

3. Running a LISP Program

In this section, various aspects of running a LISP program are discussed. In Section 3.1, we discuss how to punch cards, put together a running deck, and submit a run. The error stops word and indicators are listed in Section 3.2. In Section 3.3, the debugging aids and tracing program are described. Finally, Section 3.4 encompasses a brief discussion of the current state of the LISP flexo system.

3.1 Submitting a Run

As stated in the previous section, the basis of LISP programming is the APPLY operator, which is based on the function \texttt{apply[\(f;x;a\)]}. Here \(f\) is an S-expression for an S-function, \(x\) is a list of arguments, and \(a\) is a list of pairs. \(f\) must have been defined by the \texttt{DEFINE} function or by one of the other ways mentioned previously. A LISP program consists of sets of triplets, \(f;x;a\), which are punched on cards and slotted in the appropriate order in a deck.
Card Punching

Columns 1-72 inclusive of the punched card are read by the LISP read program. The S-expression for the first, and a of each triplet should be punched on cards separated by commas. There are no rules about the location of the expressions on the card; a new card may be started at any point in an expression, and the punching on a given card may stop at any column before column 72. In other words, boundaries are ignored. The read program reads until it finds the correct number of final parentheses.

The current version of the read program interprets a blank (space) as a comma, so that no blanks are allowed within atomic symbols. The read program ignores redundant commas so that a double blank (bb) for example is read as a single comma. A new version of the read program will allow single blanks within atomic symbols and will ignore leading and trailing blanks.

Only letters and floating-point numbers may be used in lists to be read. The characters, left parenthesis (()), right parenthesis ()), and comma (,) are used for punctuation, but any other special characters will cause an error stop.

The read program puts any atomic symbols it reads into the list of atomic symbols uniquely (without duplications).

Deck Format

A LISP program must be preceded by a deck labelled LCON which calls the LISP system from tape. The LCON deck consists of five cards labelled NBROLL followed by six cards labelled LCON. Copies of the LCON deck may be found in room 36-265 in the drawer labelled "Utility Decks".

There are five possible direction cards that can be used after the LISP system has been called by the LCON. These cards have the direction (TST, SET, FLX, CRD, or FIN) in columns 8-10 of the card and have the following effect:
TST: The lists of sets of triplets are read in until the STOP card (see 4. of the deck format below) is reached. The lists read are put out onto tape 2 for off-line printing together with a list of the atomic symbols in the machine following the reading. Control is then sent to the APPLY operator which operates on each triplet in turn, putting out on tape 2 the triplet f;x;a followed by the value of APPLY[\(f;x;a\)]. If any errors are found, an error indication is printed out. After all the triplets have been evaluated, the memory is restored to its state at the beginning of this TST.

SET: This card works the same way as TST except that at the end of the APPLY operator the state of the memory at that point is read out onto tape 8 as the new "base" image for the memory. Future TST cards will restore the memory to this new "base" state. If more SET cards are used, the functions following them will be compounded into the "base" image. If an error occurs during the evaluation of a SET triplet the new base image for that SET is not written out on tape.

Note: The variable field (columns 12-72) for both SET and TST cards should contain the problem number, the programmer's number, and name, and any other identification desired.

FLX: The flexwriter mode of operation (see Section 3.4) is called into control.

CRD: Control is returned from the flexwriter back to the card reader.

FIN: The LISP program is terminated.

A running deck (assuming no use of FLX or CRD) has the following format.

1. The LCON deck
2. A TST or SET card
1. The sets of triplets, \( f;e;e \), which the apply function is to operate on.

2. A STOP card which contains the word STOP followed by a large number of right parentheses followed by the word STOP again, 

\[
\text{STOP}}))))))))STOP
\]

This information may be placed anywhere on the card. It is used to guarantee the safe completion of the reading program.

3. Cards such as in (2, 3, 4) above, repeated as often as desired.

4. A FIN card

5. Two blank cards for the card reader.

3.1.2 Operating Instructions

Tapes used:

<table>
<thead>
<tr>
<th>TAPE NUMBER</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Off-line printed output (suppressed by sense switch 5 down)</td>
</tr>
<tr>
<td>4</td>
<td>Off-line ECD input (only if sense switch 1 up)</td>
</tr>
<tr>
<td>8</td>
<td>Temporary storage</td>
</tr>
<tr>
<td>9</td>
<td>LISP system tape</td>
</tr>
</tbody>
</table>

Sense switches used:

<table>
<thead>
<tr>
<th>SWITCH NUMBER</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UP: On-line ECD input</td>
</tr>
<tr>
<td></td>
<td>DOWN: Off-line ECD input (tape 4)</td>
</tr>
<tr>
<td>3</td>
<td>UP: Suppress on-line printing</td>
</tr>
<tr>
<td></td>
<td>DOWN: Print on-line (see sense switch 4)</td>
</tr>
<tr>
<td>4</td>
<td>UP: Single space on-line printing</td>
</tr>
<tr>
<td></td>
<td>DOWN: Double space on-line printing</td>
</tr>
<tr>
<td>5</td>
<td>UP: Write all printing on tape 2 for off-line printing</td>
</tr>
<tr>
<td></td>
<td>DOWN: No off-line printing</td>
</tr>
</tbody>
</table>

All the above have immediate effect.
Performance request card indications

1) Production run
2) Switch 1 down
3) LISP system tape on tape drive 9. The current number of this tape is posted in room 25-265 and on the bulletin board in the 70% scheduler's room.
4) Machine tape on tape drive 8.
5) Output tape (usually machine tape) on tape drive 2.
6) Tapes 8 and 9 are rewound by the program.
7) Operating instructions are:
   CLR (clear)
   LCD (Load cards)

The deck should be submitted along with its performance request card to the "regular runs" file in the scheduler's office.

3.2 Error Indications

Below are listed the print outs that occur on-line or off-line depending on sense switch 3 when an error is found in a LISP program. Generally, after an error is found control is returned to the APPLX operator who starts to operate on the next triplet. After errors which are more drastic such as no input data available or no more free storage available, the run is terminated. The word bracketed by dashes in the printouts below is the name of the LISP system subroutine involved, e.g. -APP2-in A1 below, and need not concern the user.

Errors during the operation of the APPLX operator:

A1 TOO MANY ARGUMENTS FOR A FUNCTION -APP2-
I. e. More are more than the ten arguments which the current APPLX operator is built to handle.

A2 FUNCTION OBJECT HAS NO DEFINITION -APP2-
I. e. The function has neither SUER nor EXPR on its association list, nor is it paired with something on the g-list (as a LABEL would have done).
A3 COND: XLINAL UNSATISFIED -EVCON-
X. e. none of the conditional expressions in a con-
ditional were evaluated as true
A5 SETG GIVEN ON A NON-EXISTENT PROGRAM VARIABLE -EVAL-
X. e. (see the Program Feature, Section 2.5) The pro-
gram variable is not within this program, nor is it in
some previous program in this run
A6 LOCQ GIVEN ON A NON-EXISTENT PROGRAM VARIABLE -EVAL-
See A5 above
A7 SET GIVEN ON A NON-EXISTENT PROGRAM VARIABLE -EVAL-
See A5 above
A8 LOC GIVEN ON A NON-EXISTENT PROGRAM VARIABLE -EVAL-
See A5 above
A9 UNEVALUATED VARIABLE USED -EVAL-
X. e. Q free or unbound variable has neither APVAL nor
APVAL (see Section 4.2) on its association list to sig-
nal its value, nor is the variable paired with something
on the a-list.
A10 FUNCTION OBJECT HAS NO DEFINITION -EVAL-
X. e. Q function does not have EXPR, FEXPR, SUER, nor
FUSER on its association list, nor is the function
paired with something on the a-list
A11 GO TO A POINT NOT LABELLED -INTER-
X. e. (see The Program Feature, Section 2.5) There is no
location-marking atomic symbol whose value corresponds
to the value given by evaluating the rest of the GO list
A12 Ran out of statements -INTER-
X. e. the interpreter didn’t find a RETURN statement in
a program using the program feature
A13 TOO MANY ARGUMENTS -SPREAD-
X. e. There are more than the ten arguments which the
current APPLY operator is built to handle
A14 APPLIED FUNCTION CALLED ERROR
If the APPLY operator reaches an "ERROR" in an expres-
sion, for example T->ERROR, then A14 is printed out fol-
lowed by the argument of ERROR, if one has been assigned.
It is not necessary to assign an argument, but it is pos-
sible to assign a single argument, for example

(T, (ERROR, (QUOTE, NO)))
Errors due to computer inadequacies:

E1 OUT OF PUBLIC PUSH DOWN LIST -SAVE-
1. e. (see page 184 of RFSE) The program has run out of space allotted to this purpose. Currently about 1000 registers are allotted, and if they are used up the recursion being done is either too "deep" for the given capacity or non-terminating.

E2 FREE STORAGE COUNTER PANIC STOP. PRESS START TO CONTINUE
There is a free storage counter associated with the LI SP system. Every time a register from free storage is used, the counter is reduced by one. The counter is not charged by the garbage collector operation. Thus to some extent the counter indicates how a program is operating—how much storage it keeps using up and so on, or, for a program whose operation is known, how long it has been running. The counter is initially set to 100,000, and when it has reached zero, the computer stops. When the start button has been pressed to restart the computer, the counter is cut out.

E3 DIVIDE ERROR - OCTAL TO DECIMAL CONVERTER -
This error comes only from computer malfunctioning or from some of the program having been written over.

C1 LIST OF COMPILER-GENERATED SYMBOLS EXHAUSTED
1. e. (see The Compiler, Section 2.6) More than 10,000 symbols of the form 00000-99999 have been needed during compilation. (Uncommon error — impossible error (cut))

C2 OBJECT GIVEN TO DESC AS LIST
1. e. (see desc in Section 6) The desc [x; y], x is an atomic symbol (object) instead of the required list of A's and D's.

F1 UNEQUAL LENGTH LISTS -MAP2-
This error can occur only during the differentiation function, and it implies a machine error.

F2 1ST ARG. LIST TOO SHORT -PAIR-
F2 and F3 occur when a list of pairs is being created out of two lists, for example when a list of free variables and their values is being appended to the a-list. The items still remaining on the larger list are printed out following the error indication.

F4 CANNOT PAIR OBJECTS. PLEASE USE LIST -PAIR-
I. e. (F2) of the lists to be paired is, erroneously, an atomic symbol (object). This usually happens when the user gives a single free variable in a LAMBDA expression as an atomic symbol instead of as a list.

F5 FINAL ASKED TO FIND VALUE OF NON-OBJECT

F6 FINAL ASKED TO FIND VALUE OF NON-FLOATING POINT NUMBER
F5 and F6 occur when the program is looking for the value of a floating-point number, and finds either that it is not an atomic symbol (object) (F5) or a floating-point number (F6)

Errors during the operation of the garbage collector:

G1 I HAVE FAILED TO FIND ANY GARBAGE. PANIC STOP.
-CARBAGE COLLECTOR-
I. e. There is simply no more memory space available.

G2 TOO MUCH MARKING OF NON-LIST STRUCTURE. PANIC STOP.
-CARBAGE COLLECTOR-
This error is given after G3 below has occurred ten times. The lefieancy allowed here has been inserted to permit a great deal of information to be gleaned from one run rather than to allow an immediate stop on an early error.

G3 MARKING IN NON-LIST AREA AT OCTAL/
This error occurs when illegal list structure is found during the garbage collector phase.

Errors during the operation of the direction cards on input:

O1 NO INPUT DATA -OVERLORD-
Overlord is the EXTERMINATE LISP routine which operates from the direction cards. This error arises from an end-of-file found in the wrong place, due to wrong input data.
AN ERROR HAS OCCURRED IN THE PRECEDING SET
I. e. An error has been found somewhere in the program
following the last SET direction card. This SET, see
Section 3.1, will not create a new base image of the
memory on tape.
END OF FILE ON INPUT
Same as 02.
ERROR IN READING TAPE
This is a machine error in reading either tape drive 8 or 9.

Errors during printing:

01 PRIM ASKED TO PRINT NON-OBJECT
I. e. the printing program tried to print an item which
was not an atomic symbol (object).

02 PRIM ASKED TO PRINT UNPRINTABLE OBJECT
I. e. the printing program tried to print an atomic sym-
bol which did not have PNAME on its association list.

Errors during read-in:

R1 ILLEGAL PUNCHING IN ON-LINE DATA -RX-
Sic

R2 1ST OBJECT ON INPUT LIST ILLEGAL -READ-
This error may arise when there is an error in the num-
ber of parentheses on the previous list read.

R3 OBJECT INSIDE AN INPUT LIST IS ILLEGAL -READ-
I. e. an illegal character appeared in some atom
atomic symbol (object).

R4 ILLEGAL CHARACTER -RDA-
The illegal characters are,
|   in the wrong context |   |
| + " " " " " | = | (BSD 14 punch) |
| " " " " " | $ |

R5 END OF FILE -RDA-
The tape or card-reader ran out of cards.
3.6 NUMBER TOO LARGE IN CONVERSION

The conversion program can take a floating-point number \( x \), of up to nine digits and an exponent, provided

\[ |x| < 10^{38} \]

3.3 Tracing Options

There are two tracing options available in the LISP system to help the user find his program errors. Of course, as with any tracing system, it is a slow way to find the trouble. The first tracer is a LISP function called `trackless`, described below, and the second tracer is a special trace which is a desperation measure to be avoided.

`trackless` function

The LISP function `trackless` \([x]\) is a function of one argument which is a list of the LISP functions to be traced. Each function mentioned in the list must be a function which has `EXPR`, `FEXPR`, `SUBR`, or `FSUBR` on its association list. Further, the function must not be an atomic function (see Section 2.3).

Whenever one of the functions on the list is encountered during the running of the LISP program, `trackless` gives a print out (on or off line depending on sense switch 3) of the name of the function, its type, e.g. `SUBR`, its arguments, and finally its value. Thus the path of computation followed by the program is recorded.

Once `trackless` has been called into play, it is in control until the next SET or TST direction card is read in the input data.

`TRACE`

The LISP system includes a further tracing option but its use is not recommended. A completely exhaustive trace of any triplet, \( f;1;a \) on which `apply` operates may be affected by preceding the triplet with a card on which `TRACE` has been punched (in any five contiguous columns). The `TRACE`
card affects only the triplet directly following it. The
effect of the TRACE is such that every time the function eval
is entered, a complete printout is given of the arguments of
eval and its associated e-list of pairs. The printouts pro-
liferate in a remarkable way when TRACE is used, so that vast
amounts of paper are involved, and the time increases by a fac-
tor of 50 or so. Tracless is preferable.

3.4 The Flexowriter System

The possibility of running LISP programs on line using
the Flexowriter as access to the 704, and time-sharing with
an "innocent victim" is under development. The current ver-
sion of the LISP-Flexo system is described here, but is sub-
ject to change. The reader who might wish first of all to
get familiar with the Flexowriter system, can find a write-up
of it, though for a different purpose, in Herb Tegger's memo,
Programming Manual for the M.I.T. Flexowriter Monitor Inter-
preter System of Time-Shared Program Testing.

The LISP-Flexo System enables the user to read in func-
tion definitions from cards or to type them in on the Flexo-
writer, and to control the operation of LISP via Flexowriter
type-ins.

Operating Instructions

To use the LISP-Flexo System on the M.I.T. 704 one must
1) Turn on the two power switches for the Flexowriter
2) Turn off the alarm clock
3) Turn off the back interrupt switch
4) Put the LISP-Flexo System tape on tape drive 10

The output will be either on the on-line printer or on the
Flexowriter. The input deck to be put into the card reader
has the following format,
GETCPFX card to call the M.I.T. Automatic Operator program and the Flexowriter programs in from the M.I.T. System tape.

GETLISP card to get the LISP-Flexo System in from tape 10.

Binary or octal correction cards currently needed for the LISP-Flexo System (obtainable along with the three GET---cards from Room 26-265)

GETLISP2 card to transfer into operating program

FLX direction card--see Section 3.1

Cards containing the triplets on which the APPLY operator is to operate, provided the triplets are to be read in rather than typed in on the Flexowriter. The last triplet must be followed by two cards of the form

```
ICFLIP(READ),(1,1)()()
```

where the extra parentheses are used as insurance. See below for a description of the function ICFLIP. If no triplets are to be read in, no cards need be placed here between the above FLX card and the FIN card below.

FIN direction card--see Section 3.1

2 blank fields for the card reader.
In the following description of the Flexowriter system, the three symbols $\downarrow$, $\rightarrow$, and $b$ will be used, where

- $\downarrow$ indicates carriage return
- $\rightarrow$ indicates a tab
- $b$ indicates a blank (space)

When cards are read in on the card reader the FLX card transfers control to the Flexo-APPLEX operator. The Flexowriter then types out GO and waits for an input. The system at this time expects an LISP program to be typed in on the Flexowriter. However, if there are cards in the card reader which contain the LISP program, the type-in to send control to the reader is

```
IOFLIP(READ),()$\downarrow$
```

After the carriage return, cards containing the triplets for the APPLEX operator will be read in from the card reader and applied.

If the previous method, that of a typed-in program is to be used, there are two possible modes of operation, the Sequence-Mode and the TEN-Mode. In the Sequence-Mode the
entire S-expression for a triplet $f;x;a$ for the APPLY operator is typed in sequentially, and at the end of the three lists the APPLY operator operates on them. This is satisfactory theoretically but in practice, due to the difficulty of typing S-expressions, it is inconvenient since an error anywhere in the expression requires a retyping of the entire expression, and if the error was in a previous line (see Errors below) it cannot be deleted at all. The second mode, the TEN-Mode was developed to allow the user to type in S-expressions in small fragments which can be individually corrected. Below we describe in more detail the use of the two modes.

Sequence-Mode

When the FLX card of the input has transferred control to the Flexo-APPLY operator, and the Flexowriter has typed out GO, an S-expression for an APPLY triplet, $f;x;a$, should be typed-in on the Flexowriter. The typing is done by typing-in up to 72 characters (fewer if desired) followed by a carriage return. The Flexowriter then types out STOP and digests the current information to date. If a full triplet has not been completed, the Flexowriter types out GO as a request for more of the triplet. If a triplet has been completed, the APPLY operator takes over and performs the triplet, typing out the answer followed by a GO requesting the next triplet.

To stop the computation, one may respond to an initial GO, or to a GO following the use of the APPLY operator, or to a GO following an error* (but not to a GO at other points, such as in the middle of list type-ins), by typing in STOP. Control is then returned to the control routine which reads direction cards (or type-ins). One can then type in

```
> bGRDb32
```

to return control to the card-reader for the next direction card, e.g., TST, SET, FLX, FXN.

---

* Such a GO will hereafter be referred to as a "fresh" GO.


TEN-Mode

There are ten buffers for S-expressions set aside for use by the TEN-Mode of operation. Pieces of an S-expression can be typed into these buffers in any sequence, with overwriting allowed, and the Flexo-APLIX operator can be asked to operate on any set of consecutive buffers, e.g. 0 through 6, or 8 through 9.

To use the TEN-Mode, one may respond to a "fresh" GO (see preceding footnote) by typing in

\[ \rightarrow \text{TEN} \]

Then type in a number from 0 to 9 representing one of the ten buffers, and then a tab, and then a (piece of a) S-expression and then a carriage return, as in

\[ 8 \rightarrow ((A,B),C) \]

The Flexwriter will then type out a colon, :, and one can type into another buffer register in the same way as above by typing a buffer-register, a tab, the information, and finally a carriage return.

When an entire triplet has been typed in, the APPLY operator can be called for by typing in, in response to the colon type-out, the read-line direction,

\[ \rightarrow \text{RIN} \]

Then two numbers followed by a carriage return must be typed in, \( n_1 \) \( n_2 \), e.g. 01, or 69, or 99. All the buffer registers, \( n_1 \) to \( n_2 \) inclusive, are then turned over to the read program, which reads them in, printing out the number of call register as it is read. When a complete triplet \( f;x;a \) has been found by the read program the APPLY operator then operates on this triplet, prints out the answer, and then returns control to the read program. The read program reads ahead until it finds either three more lists for a triplet or until it has read \( n_2 \). In the first case, the APPLY operator takes over as above. In the second case, the Flexwriter types out GO. The typed-in response to this GO may be either another read-line direction, or a type-in
into a buffer register as described previously. Note that
the real program as used in the TEN-Mode remembers any in-
complete lists it may have been reading at the end of the
previous RUN request, and considers this information to be
the initial part of the next material read.

To stop the operation of the TEN-Mode one types (if no
error has occurred), in response to a "fresh" GO (see pre-
vious footnote), into two consecutive registers \( n_1 \) and
\( n_{i+1} \):

\[
: n_1 \rightarrow \text{STOP} \\
: n_{i+1} \rightarrow \rightarrow \text{FINDB}
\]

Then one does a RLN of these two buffer registers as de-
scribed above and this will end the run completely. One can
instead replace the above FIN by a CRD if one wishes to send
control back to the card reader.

To stop the operation of the TEN-Mode after an error
has occurred, one types , in response to any GO, into three
consecutive registers \( n_1 \), \( n_{i+1} \), and \( n_{i+2} \):

\[
: n_1 \rightarrow \{\text{carriage} \} \\
: n_{i+1} \rightarrow \text{STOP} \\
: n_{i+2} \rightarrow \rightarrow \text{BFINDB} \quad \text{(or CRD in place of FIN)}.
\]

Then one does a RLN of these three buffer registers as above.

To get from the TEN-Mode back to the Sequence-Mode one
types in, in response to a GO,

\[
: \rightarrow \text{ONE}.
\]

Errors

Typing errors may be erased by hitting the backspace key
which will erase the preceding character, or by hitting the
underline key, which will erase the entire line. Several
characters at a time can be erased by using the backspace key
the appropriate number of times, but lines above the current
line can never be recovered for erasure.

Extra right parentheses give a read error.
The error type-outs described in Section 3.2 appear on the Flexewriter without the sentence describing the error, for example, 

**ERROR NUMBER :A2:**

and this type-out is followed by the argument of the error if it had one. The read program does not stop at the error; in the TEN-Node the read program goes on to start reading a new triplet in the register following that in which the error occurred, in the Sequence-Node the read program starts reading the next line typed in.

**Interrupts**

At any time except when the Flexewriter is itself typing out a message one may type in any of the monitor directions QUE, ENT, EXIT, LDC, EXIT, TEN.

**The functions I0FLEX (**X**) AND I0FLIP (**Y**)**

The two new functions I0FLEX and I0FLIP have been added to the LISP-Flexo System. They are functions of one argument and a null a list, and they have the following effects:

I0FLEX (**X**), () is a predicate which has the value true (\(T\)) or false (\(F\)). If \(X = \text{READ}\) the predicate asks if the Flexewriter is in the reading (type-in) mode; if \(X = \text{PRINT}\) the predicate asks if the Flexewriter in the printing (type-out) mode; and if \(X\) is something else an error message followed by \(F\) is typed out.

I0FLIP (**Y**), () is an operative function which changes control as follows:

- \(Y = \text{READ}\) flips control between reading from cards or reading from type-ins on the Flexewriter
- \(Y = \text{PRINT}\) flips control between printing on the on-line printer or printing by Flexewriter type-outs
- \(Y = \text{anything else}\) flips the last pair that was flipped. It assumes initially that this was a READ.
Note: At certain times the LISP-Flexo System may hang up trying to read the card reader. How it gets into this trouble, and how to get out of it are described below.

An input of the form IOFLIP,(READ),() given when operating in the Flexewriter type-in control will select the card reader and cause the read program to try to read in cards containing lists of triplets. If no cards of this form are in the reader, an EOF (end-of-file) error from the read program occurs, and the card reader is selected again. The card reader will not be in ready status, and, if the START button on the card reader is selected another EOF error will occur. To get out of this cycle, one can type a tab on the Flexewriter. The tab will not be processed, but the Flexewriter key card will look. At this point, pressing the START button on the card reader will get the card back on the line so that the interrupt (from the tab type-in) will be processed and return control to the innocent victim.

NOTE: AN EXAMPLE OF ACTUAL FLEXO TYPING USING ALL THE OPTIONS SHOULD GO HERE: P.F.

4. List Structures

The list structures used by LISP are discussed in part 4 of RTSE. Descriptions are given there of (a) the representation of S-expressions by list structures, and (b) the particular association lists associated with each atomic symbol which is on the list of atomic symbols, and finally (c) the free-storage list. In Section 4.1 below we shall try to clarify list structures by showing how a particular kind of list can be constructed using LISP. In Section 4.2 we describe association lists, and in Section 4.3 we shall describe how the "garbage collector" restores unused storage to the free-storage list.
4.1 Construction of List Structures

The problem we shall pose to illustrate list structures is the following:

Assume we are given a list of the form

\[ \mathcal{L}_1 = \{(A,B,C),(D,E,F),\ldots,(X,Y,Z)\}, \]

which is represented as

and that we wish to construct a list of the form

\[ \mathcal{L}_2 = \{(A,(B,C)),(D,(E,F)),\ldots,(X,(Y,Z))\}, \]

which is represented as

Consider first the part \((A,(B,C))\) of the second list, \(\mathcal{L}_2\). This may be constructed from \(A\), \(B\), and \(C\) by the operation

\[ \text{cons}\left[A;\text{cons}\left[\text{cons}\left[B;\text{cons}\left[C;\text{NIL}\right]\right]\right]\right];\text{NIL}. \]

Another way of writing the same thing is as

\[ \text{list}[A;\text{list}[B;C]] \]

In any case, given a list, \(t\), of three atomic symbols,

\[ t = (A,B,C), \]

the arguments \(A\), \(B\), and \(C\) which to be used in the operation of
form \((A, (B, C))\).
The arguments \(A\), \(B\), and \(C\) are found from
\[
A = \text{car}\[t]\] = A \\
B = \text{car}\[\text{cdr}\[t]\] = \text{car}\[(B, C)] = B = \text{cadr}\[t]\] \\
C = \text{car}\[\text{cdr}\[\text{cdr}\[t]\]]] = \text{car}\[\text{cadr}\[(B, C)]\] = \text{car}\[c\] = C = \text{caddr}\[t]\].
\]

The first step in obtaining \(r_2\) from \(r_1\) is to define a
function, \(\text{grp}\), of three arguments which creates \((X, (Y, Z))\)
from a list of the form \((X, Y, Z)\)
\[
\text{define}\[(\text{grp}; \lambda [t]; \text{cons}[\text{car}[t]; \text{cons}[\text{cadr}[t]; \text{cons}[\text{caddr}[t]; \text{NIL}]; \text{NIL}]]; \text{NIL}]]\]
\]

Then \(\text{grp}\) is used on the list \(r_1\) assuming \(r_1\) to be of the
form given. For this purpose a new function, \(\text{mltgrp}\), is de-
defined recursively.
\[
\text{define}\[(\text{mltgrp}; \lambda [l]; \text{null}[l] \rightarrow \text{NIL}; T \rightarrow \text{cons}[\text{grp}[\text{car}[l]]; \text{mltgrp}[\text{cdr}[l]]]); \text{NIL}]]\]
\]
So \(\text{mltgrp}\) applied to the list \(r_1\) takes each threesome, \((x, y, z)\),
in turn and applies \(\text{grp}\) to it to put it in the new form,
\((x, (y, z))\) until the list \(r_1\) has been exhausted and the new
list \(r_2\) achieved.

Note: In general any list which is read in can have
only \(\text{NIL}\) (zero) in the final doceas. That is, the dot
notation \((A\cdot B)\) of RESE cannot be read by the current read
routine, so that the only way to get \((A\cdot B)\) is by a \text{cons}
within the machine.

4.2 Association lists

Every atomic symbol in the list of atomic symbols and
every number on the list of floating-point numbers (integers
are included within the list of atomic symbols) has an as-
sociation list (after known locally as a "property list").

The floating-point numbers are put on the list of
floating-point numbers either as they are read-in or as they
are generated. Neither this list, nor the list of atomic
symbols, has any duplications. An association list is as-
signed to a floating-point number when the number is put in
the list of numbers, and to an atomic symbol when it is put on the list of atomic symbols.

An association list has 77777 (i.e., minus 1) in the address section of the first word. The rest of the list carries other information such as possibly the print name, or the value of the integer, or an indication (FLOAT) that the list is associated with a floating-point number, etc. Two kinds of value indicators are used on association lists, AFVAL and AFVALL. Both indicate a value, as in the association list for NIL

NIL

but AFVAL acts like a stop to the garbage collector (see Section 4.3 below) whereas AFVALL, which is used by attrib, allows the garbage collector to look ahead through the part of list pointed to by AFVALL.

On the association lists of these atomic symbols which represent functions, SUBR indicates that a SAP subroutine for performing the function. SUBR points to a NIL to the subroutine, e.g.

On the other hand, EXPR, on an association list for a function points to the S-expression for the function.
The special forms discussed at the end of Section 2.3 have FS'UBR or FEXPR on their association lists.

Either attrib or define (see Section 2) may be used to put something on an association list. Attrib puts the item at the end of the list, and define puts it at the beginning. Thus define can be used to redefine an atomic symbol. Define puts FEXPR followed by the associated S-expression on the association list of a function. Compile (see Section 2.6) puts SBR on the association list of the function compiled, and the SBR points to the THL to the compiled subroutine.

Just as a matter of interest we give below the association lists for atom and for the constant 1, just as they are represented in the 704. We use the bar notation, $\bar{e}$, to represent the 2's (8's) complement of an octal address.

Proceeding all the individual association lists is the list of all atomic symbols. Each register of this list has an entry of the form

$$x, \bar{y}$$

where $x$ is the address of the first register of the association list for this atomic symbol, and $y$ is the address of the next entry for the next atomic symbol in the list. Thus for atom whose association list starts in 25146 the entry in the list of atomic symbols is

<table>
<thead>
<tr>
<th>location</th>
<th>entry</th>
<th>representing</th>
</tr>
</thead>
<tbody>
<tr>
<td>24657</td>
<td>053120052632</td>
<td>25146, 24658</td>
</tr>
</tbody>
</table>

The association lists which appear in the following are:

<table>
<thead>
<tr>
<th>atomic symbol</th>
<th>location of association list</th>
</tr>
</thead>
<tbody>
<tr>
<td>atom</td>
<td>25146</td>
</tr>
<tr>
<td>constant 1</td>
<td>25201</td>
</tr>
<tr>
<td>subr</td>
<td>26705</td>
</tr>
<tr>
<td>prame</td>
<td>26231</td>
</tr>
<tr>
<td>(integer)</td>
<td>25665</td>
</tr>
<tr>
<td>arval</td>
<td>25134</td>
</tr>
</tbody>
</table>
Then for atom the association list looks as follows:

<table>
<thead>
<tr>
<th>location</th>
<th>entry</th>
<th>representing</th>
</tr>
</thead>
<tbody>
<tr>
<td>25146</td>
<td>052631 077777</td>
<td>-1, 25147</td>
</tr>
<tr>
<td>25147</td>
<td>052630 051547</td>
<td>26231, 25150</td>
</tr>
<tr>
<td>25150</td>
<td>052623 052627</td>
<td>25151, 25153</td>
</tr>
<tr>
<td>25151</td>
<td>000000 052626</td>
<td>25152, 0</td>
</tr>
<tr>
<td>25152</td>
<td>216346 447777</td>
<td>BD ATOM?? (77 ???)</td>
</tr>
<tr>
<td>25153</td>
<td>052624 051073</td>
<td>26705, 25154</td>
</tr>
<tr>
<td>25154</td>
<td>000000 052623</td>
<td>25155, 0</td>
</tr>
<tr>
<td>25155</td>
<td>700000 010762</td>
<td>TXL 10762 (Location of SAP)</td>
</tr>
</tbody>
</table>

or, in list form,

For the constant 1, the association list looks as follows:

<table>
<thead>
<tr>
<th>location</th>
<th>entry</th>
<th>representing</th>
</tr>
</thead>
<tbody>
<tr>
<td>25201</td>
<td>052576 077777</td>
<td>-1, 25202</td>
</tr>
<tr>
<td>25202</td>
<td>052575 051547</td>
<td>26231, 25203</td>
</tr>
<tr>
<td>25203</td>
<td>052572 052574</td>
<td>25204, 25206</td>
</tr>
<tr>
<td>25204</td>
<td>000000 052573</td>
<td>25205, 0</td>
</tr>
<tr>
<td>24025</td>
<td>017777 777777</td>
<td>BD DESIGNER</td>
</tr>
<tr>
<td>25206</td>
<td>052571 052113</td>
<td>25665, 25207</td>
</tr>
<tr>
<td>25207</td>
<td>052567 052570</td>
<td>25210, 25211</td>
</tr>
<tr>
<td>25210</td>
<td>000000 000001</td>
<td>1, 0</td>
</tr>
<tr>
<td>25211</td>
<td>052566 052644</td>
<td>25134, 25212</td>
</tr>
<tr>
<td>25212</td>
<td>000000 052565</td>
<td>25213, 0</td>
</tr>
<tr>
<td>25213</td>
<td>000000 000001</td>
<td>1, 0</td>
</tr>
</tbody>
</table>

or, in list form

The text contains a table and a diagram describing the association list for the constants -1 and 1, including their entries and their corresponding representations.
In this case, the constant 1 is tagged both as an integer \& IN of value 1, and as a constant which when evaluated (APVAL) yields the value 1.

4.3 The Garbage Collector

During the running of a LISP program whenever the free-storage list has been exhausted a retrieval of storage is initiated. The program which retrieves the storage is a SAP-coded program called the Garbage Collector.

Any piece of list structure that is accessible to programs in the machine is considered an active list and is not touched by the Garbage Collector. These lists are accessible to the program through certain fixed sets of base registers such as the registers in the list of atomic symbols, the registers in the push-down list, the registers which contain partial results of the LISP computation in progress, etc. The list structures involved may be arbitrarily long but each register which is active must be connected to a base register through a car-sadr chain of registers. Any register that cannot be so reached is not accessible to any program and therefore its contents are no longer of interest.

The non-active, i.e. available, registers are reclaimed for the free-storage list by the Garbage Collector as follows. First every active register which can be reached through a car-sadr chain is marked by setting its sign negative. (Whenever a negative register is reached in a chain during this process, the Garbage Collector knows that the rest of the list involving that register has already been marked.) Then the Garbage Collector sweeps through the lists again, collecting all registers with a positive sign into a new free-storage list, and restoring the original signs of the active registers, until all the lists have been swept.

Thus the programmer does not have to keep track of and erase unwanted lists. The process is carried out automatically as the need arises. Partial experience has indicated that about a third of the running time is taken up by the Garbage Collector.
6. Functions Available in LISP

In this section, a brief description is given of all the functions available in the LISP system as of January 1, 1960. A brief account is given for each function explaining the form of its arguments, and its value. Whether the function is in machine language (otherwise it must be interpreted by the APPLY operator), and whether it is a special form is also included. In some cases, an M-expression for the function in terms of more basic functions is appended.

Since the number of functions is quite large (~90) it is suggested that the user might first look at the following selected list:

<table>
<thead>
<tr>
<th>and</th>
<th>desc</th>
<th>mapcon</th>
<th>pick</th>
<th>replace <a href="d">a</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>equal</td>
<td>maplist</td>
<td>print</td>
<td>sassee</td>
</tr>
<tr>
<td>attrib</td>
<td>error</td>
<td>none</td>
<td>printprop</td>
<td>search</td>
</tr>
<tr>
<td>assoc</td>
<td>format</td>
<td>not</td>
<td>prop</td>
<td>select</td>
</tr>
<tr>
<td>cond</td>
<td>inst</td>
<td>null</td>
<td>punchl</td>
<td>supply</td>
</tr>
<tr>
<td>copy</td>
<td>list</td>
<td>or</td>
<td>read</td>
<td>sublis</td>
</tr>
<tr>
<td>define</td>
<td>map</td>
<td>pair</td>
<td>remprop</td>
<td>subst</td>
</tr>
</tbody>
</table>

\[ \text{and}[x_1, x_2, \ldots, x_n] : \text{machine-language} ; \text{special form} \]

The arguments of \text{and} are evaluated in sequence, from left to right, until one is found that is false, or until the end of the list is reached. The value of \text{and} is false (zero) or true (one) respectively.

\[ \text{app2}[f; x; a] : \text{machine-language} \]

The function \text{app2} is the \text{apply} operator (see \text{apply}) in the case when \( f \) is atomic. Presumably \text{apply} will not be used as such by the reader, but for the record its modus operandi (and that of \text{app2} are included herein).

\[ \text{app2}[f; x; a] = \text{select}[f; \text{[CAR; caar}[x]]; \text{[CDR; cadr}[x]]; \]
\[ \text{[CONS; cons[car[x]; cadr[x]]]; [LIST;x]} ; \]
\[
\text{search} \left[ f; \lambda \left[ j; \text{car} \left[ j \right] = \text{SUPER} \rightarrow \text{car} \left[ j \right] = \text{FSUPER} \right] \right] ; \\
\lambda \left[ j; \text{car} \left[ j \right] = \text{SUPER} \rightarrow \text{app3} \left[ \text{cadr} \left[ j \right]; x \right] ; \\
T \rightarrow \text{apply} \left[ \text{cadr} \left[ j \right]; x; a \right] ; \\
\lambda \left[ j; \text{apply} \left[ \text{car} \left[ \text{assoc} \left[ f; a; \text{error} \right] \right] ; x; a \right] \right]
\]
\]

\[\text{app3} \left[ f; x \right] \quad \text{machine-language}\]

In the case when \text{app2} (see above) in the \text{apply} function finds an atomic function specified by a machine-language subroutine, \text{app3} applies that subroutine to the list \( x \) of arguments.

\[\text{append} \left[ x; y \right] \quad \text{machine-language}\]

The function \text{append} combines its two arguments into one list. The value of \text{append} is the resultant list. For example

\[
\text{append} \left[ \left( A, B \right), \left( C \right) \right] = \left( A, B, C \right), \\
\text{append} \left[ \left( \left( A \right), \left( C, D \right) \right) \right] = \left( \left( A \right), \left( C, D \right) \right)
\]

\[\text{append} \left[ x; y \right] = \left[ \text{null} \left[ x \right] \rightarrow y; T \rightarrow \text{cons} \left[ \text{car} \left[ x \right]; \text{append} \left[ \text{cdr} \left[ x \right]; y \right] \right] \right]
\]

See RUSE page 133.

\[\text{apply} \left[ f; x; a \right] \quad \text{machine-language}\]

The reader should have no occasion to use this function because essentially it gets done for him, but it operates as follows,

\[
\text{apply} \left[ f; x; a \right] = \left[ \text{atom} \left[ f \right] \rightarrow \text{app2} \left[ f; x; a \right] ; \\
\text{car} \left[ f \right] = \text{LABEL} \rightarrow \text{eval} \left[ \text{cadrr} \left[ f \right] \right]; \text{append} \left[ \text{pair} \left[ \text{cadrr} \left[ f \right]; x \right]; a \right] ; \\
\text{car} \left[ f \right] = \text{LABEL} \rightarrow \text{apply} \left[ \text{cadrr} \left[ f \right]; x \right]; \text{append} \left[ \text{pair} \left[ \text{cadrr} \left[ f \right]; \text{null} \left[ f \right] \right] \right] ; \\
\text{cadrr} \left[ f \right]; a \right] ; \\
\text{car} \left[ f \right] = \text{FNAME} \rightarrow \text{apply} \left[ \text{cadrr} \left[ f \right]; x; \text{cadrr} \left[ f \right] \right] ; \\
T \rightarrow \text{apply} \left[ \text{eval} \left[ f; a \right]; x; a \right] \right]
\]

\[\text{atom} \left[ f \right] \quad \text{machine-language}\]

The argument of \text{atom} is evaluated and the value of \text{atom} is true \text{(one)} or false \text{(zero)} depending on whether the argument is or is not an atomic symbol. In list terminology
(see Section 4) the argument is an atomic symbol: \( \text{car}(k) = -1 \).
See RFSE page 131.

\[ \text{attrib}(k; e) : \text{machine-language} \]

The function \( \text{attrib} \) concatenates its two arguments by changing the last element of its first argument to point to the second argument. Thus it is commonly used to tack something into the end of an existing association list. The value of \( \text{attrib} \) is the second argument. For example

\[ \text{attrib}(\text{FF}, (\text{EXPR}, (\text{LAMBDA}, (k), (\text{COND}, ((\text{ATOM}, x), x), (\text{T}, (\text{FF}, (\text{CAR}, x)))))))) \]

would put \( \text{EXPR} \) followed by the \( \text{LAMBDA} \) expression for \( \text{FF} \) onto the end of the association list for \( \text{FF} \).

\( \text{attrib} \) can be used to define a function provided no other definition comes earlier on the function's association list, but in general it is better to use \( \text{define} \).

\[ \text{ indiv}(k) : \lambda (k); \text{cadr}(k) \]

This function is used as a function whose form may later change in \( \text{csearch} \).

\[ \text{car}(k) : \text{machine-language} \]
See RFSE page 131.

**Examples:**

\[ \text{car}([(A,B)]) = A \]
\[ \text{car}([(A,B)]) = (A,B) \]

\[ \text{cdr}(k) : \text{machine-language} \]
See RFSE page 131.

**Examples:**

\[ \text{cdr}([(A,B)]) = (B) \]
\[ \text{cdr}([(A,B)]) = \text{NIL} = () \]
The function `compab` is a predicate whose value is true (one) if the absolute value of the difference between x and y is less than z, and whose value is false (zero) otherwise. x, y, and z are all floating-point numbers.

```
compab [x, y, z] : machine-language
```

The function `compsearch` composes an S-expression which will be interpreted by the APPLY operator as a search. The purpose of this new function is to speed up the operation of the search on its recursive paths. The search, at each recursion, has the APPLY operator rebind all of the variables x, d, f, and u, whereas generally only the x need be revised. The search in `compsearch` rebinds only the x at each recursion.

```
compsearch [x, d, f, u]
```

The function `compsearch` composes an S-expression which will be interpreted by the APPLY operator as a search. The purpose of this new function is to speed up the operation of the search on its recursive paths. The search, at each recursion, has the APPLY operator rebind all of the variables x, d, f, and u, whereas generally only the x need be revised. The search in `compsearch` rebinds only the x at each recursion.

```
(compsearch [x, d, f, u])
```

Note that `compsearch` avoids the rebinding of d and f by substituting the uniquely generated atomic symbol for T1 into d and f.

As an example, assume that a function, called `FINDPNAME`, is to be defined for the interpreter, where `FINDPNAME` is to search any list for PNAME and have the portion of the list pointed to by PNAME as its value.

Below are examples of how this function may be defined (using the APPLY operator), by using DEFINE, and then by using COMPSEARCH.
CONJPSEARCH
(FINDNAME,
(LAMBDA, (3), (EQ, (CAR, 3), (QUOTE, PHASE))),
(LAMBDA, (3), (CADR, 3)),
(LAMBDA, ()), ERROR),
()}

cons[x; y] : special form

The function cons evaluates the items on the list x, and concatenates the values. The value of cons is the final concatenated list. The items on the list x must either be found on the list of pairs (the a-list), y, or have AFVAL or AFVALI on their association lists.

Example: let x = (X,Y,Z) and y = ((X,W),(Z,R)) and let y have AFVAL on its list pointing to the value \( \overline{Y} \), then

\[
\text{cons}[x;y] = (X,\overline{Y},Z)
\]

cons is equivalent to

\[
\text{map}[x;\lambda {[i] \text{eval} [\text{car} [i]; y]}]
\]

cond[x] : machine language; special form

The argument of cond is a list of pairs. The first element of each pair is the proposition and the second element of the pair is the corresponding expression. The propositions are evaluated until one is found that is true. The expression corresponding to this proposition is evaluated and taken as the value of the entire conditional.
Except for its use with the compiler at least one of the
propositions must be true or an error will occur.
See RRSE page 136.

CONS is equivalent to (a is the a-list of pairs)

\[
\text{label[evcond; } \lambda \left( [x;a] : \text{null}[x] \rightarrow \text{error}; \text{eval}[\text{car}[x]; a] \rightarrow \text{eval}[\text{cadr}[x]; a]; T \rightarrow \text{evcond}[\text{cdr}[x]; a] \right) \]
\]

\[
\text{CONS} \equiv \text{QUOTE}. \quad \text{QUOTE should be used.}
\]

\text{CONS} : \text{machine language}

The function \text{CONS} (construct word) is not strictly
grammatical in the LISP-sense. It takes the contents of the
AC and sends them to the next free storage location. As with
\text{CONS} the value of \text{CONS} is the (location of) stored word.

\text{COPY} \equiv \text{QUOTE}.

\text{COPY} \equiv \text{QUOTE}.

The function \text{COPY} is a function of one argument. Its
value is the location of the copied list.

COPY is equivalent to

\[
\text{label[COPY; } \lambda \left( [x] : \text{null}[x] \rightarrow \text{NIL}; \text{atom}[x] \rightarrow x; \right.

T \rightarrow \text{cons}[\text{COPY}[\text{car}[x]]; \text{COPY}[\text{cdr}[x]]]\right)
\]

\text{COUNTER} : \text{machine language}

The function \text{COUNTER} is a function of no arguments. Its
value is identically zero. Its effect is to turn on the CONS
counter. If the counter is already on, \text{COUNTER} has no effect.

The CONS counter is a counter which is incremented every-
time a word is constructed by the LISP system and put into
free storage. This counter is to some extent a measure of
the length of a program and an indicator of the amount of free storage it is using up.

\text{cpl}[x] : \text{machine language}

The function \text{cpl} copies a one-level list \( x \) into free storage, and returns with the location of the copied list as its value.

\text{cpl} is equivalent to

\begin{align*}
\text{label}[\text{cpl};\lambda [x]; [\text{null}[x] \rightarrow \text{NIL};
\text{cons}[\text{cons}[[\text{car}[x]]];\text{cpl}[[\text{cdr}[x]]]]])],
\end{align*}

(here \text{car}[x] is the 36-bit contents of register \( n \))

\text{defl}[x;e] : \text{The function \text{defl} puts a pointer to the expression \( e \) on the rest of the list \( x \) from the point where it finds the attribute PRO (see below). If no such attribute is found, one is created. The value of \text{defl} is \( x \). In practice, PRO is usually paired with EXPR on the a-list.}

\text{defl} is equivalent to the following (where the Program Feature has been used)

\begin{align*}
\lambda [[x;e]; \overline{5}] \text{ where } \overline{5} \text{ is the following program with no program variables}.
\end{align*}

\begin{align*}
\text{begin} \text{[prop}[x; \text{PRO}; \text{function}[[\lambda ()]; \text{cdr}[[\text{attrib}[x; \text{PRO}, \text{NIL}]]]); e]] \text{; return}[[x]]
\end{align*}

Thus the association list will end up with PRO pointing to \( e \) as follows

\begin{align*}
\text{---} \rightarrow \text{PRO} \rightarrow \cdots
\end{align*}
define(x)

The argument of define, x, is a list of pairs,
\[
\{(u_1,v_1), (u_2,v_2), \ldots, (u_n,v_n)\}
\]
where each u is a name and each v is a \(\lambda\)-expression or a function. For each pair, define uses defi to make the EXPR on the association list for u point to v. The function define puts things on at the front of the association list. The value of define is the list of u's.

define is equivalent to
\[
\lambda [x] \text{deflist}(x; \text{EXPR})
\]

deflist(x; PRO)

The function deflist is used usually used to tie the EXPR-search required by define to the PRO-search in the program for defi.

deflist is equivalent to
\[
\lambda [x; PRO] \text{deflist}(x)
\]

deflist(x)

The argument of deflist is a list of pairs. The function deflist does a defi of each pair, and has as value the list of the first element of each pair.

Thus the functions deflist and deflist carry out the purpose of define by typing the PRO in defi to EXPR and by carrying out the entire list of definitions (by the recursive function deflist).

deflist is equivalent to
\[
\text{label}[\text{deflist}; \lambda [x] \langle \text{null}(x) \rightarrow \text{NIL}; T \rightarrow \text{cons}(\text{defi} \langle \text{car}(x) \rangle; \text{deflist}(\text{cdr}(x)))\rangle\rangle]
\]

def2(u; v)

The function def2 (descend) is a function of two arguments, the first of which must be a list of the atomic
symbol A (for \texttt{car}) and D (for \texttt{cdr}). The value of \texttt{desa} is the result of executing on the second argument the sequence of \texttt{car}'s and \texttt{cdr}'s specified by the first argument. The operation indicated by the first element of the list of A's and D's is executed first. Illegal list structure is not checked for.

\texttt{desa} is equivalent to
\begin{verbatim}
labeled{desa; \left( \begin{array}{ll}
\texttt{null} & \rightarrow \texttt{error}; \\
\texttt{car} \rightarrow \texttt{desa} \left[ \texttt{adr} \right]; \texttt{desa} \left[ \texttt{adr} \right]; \\
? \rightarrow \texttt{desa} \left[ \texttt{adr} \right]; \texttt{desa} \left[ \texttt{adr} \right] \end{array} \right]
\end{verbatim}

\texttt{diff;} : machine language

The function \texttt{diff} differentiates the algebraic expression \( y \), with respect to \( x \). The value of \texttt{diff} is the (unsimplified) algebraic expression, \( \frac{\partial y}{\partial x} \). Gradients must be provided on the association lists of all the functions used in the expression, \( y \), except for \texttt{PLUS} and \texttt{TIMES}. See the notation and reference given under \texttt{simplify}.

\texttt{distr;} : machine language

The function \texttt{distr} will distribute conditionally the products of sums appearing in the expression \( e \). The proposition \( p \) determines whether a given sum is to be distributed. Thus in the following example let the proposition \( p \) of a sum be that "if \( y \) is included in a sum, do not distribute this sum", then

\[(x+a)(y+z+b)(w+c)\] distributes to
\[(x)(y+z+b)+(2x)(y+z+b)+(aw)(y+z+b)+(2a)(y+z+b)\]

See the correct notation for sums and products for \texttt{LISP} under the function \texttt{simplify}.

\texttt{eq;} : machine language

The function \texttt{eq} has the value \texttt{true} (\texttt{one}) if \( x = y \) or \texttt{false} (\texttt{zero}) if \( x \neq y \).
\textbf{eql}[x;y] : machine language}

The function \texttt{eql} compares the one-level lists at \(x\) and \(y\). The value of \(\texttt{eql} x;y\) is true (one) if the two lists are identical, and false (zero) otherwise.

\texttt{eql} is equivalent to

\begin{align*}
\text{label}[\texttt{eql},x;y] & : [x = y \rightarrow 1; x = 0 \lor y = 0 \rightarrow 0; \\
& T \rightarrow \text{eql} [\text{car}[x], \text{car}[y]] \land \text{eql} [\text{cdr}[x], \text{cdr}[y]]]
\end{align*}

(Here \texttt{eql} is the 36-bit contents of register \(n\))

\textbf{equal}[x;y] : machine language

The function \texttt{equal} compares the lists \(x\) and \(y\) and has the value true (one) if the two lists are identical, and false (zero) otherwise.

\texttt{equal} is equivalent to

\begin{align*}
\text{label}[\texttt{equal},x;y] & : [x = y \rightarrow 1; x = 0 \lor y = 0 \rightarrow 0, \\
& \text{car}[x] = -1 \lor \text{car}[y] = -1 \rightarrow 0, \\
& T \rightarrow \text{equal} [\text{car}[x], \text{car}[y]] \land \text{equal} [\text{cdr}[x], \text{cdr}[y]]]
\end{align*}

\textbf{error}[x] : machine language

The function \texttt{error} of one (or no) argument causes an error print-out followed by a print-out of any argument \(x\) given.

\textbf{eval}(a;b) : machine language

The reader should have no occasion to use this function as such, since it is called in by the \texttt{APPLY} operator, but he might be interested in its modus operandi which is as follows,
\[
eval[e; b] = \begin{cases} \text{atom}[e] \rightarrow & \text{search}[e; \lambda[j]; \text{car}[j] = \text{APVAL} \lor \text{car}[j] = \text{APVAL}] ; \\
\lambda[j]; \text{cadr}[j] ; \\
\lambda[j]; \text{search}[b; \lambda[j]; \text{car}[j] = e] ; \\
\lambda[j]; \text{cadar}[j] ; \\
\lambda[j]; \text{error}] \end{cases} \\
\text{atom[car[e]]} \rightarrow \\
\text{search}[\text{cadar}[e]; \lambda[j]; \text{car}[j] = \text{FSUER} \lor \text{car}[j] = \text{FSUER} \lor \text{car}[j] = \text{EXPR} \lor \text{car}[j] = \text{EXPR}] ; \\
\lambda[j]; \text{select}[\text{car}[j] ; \\
[\text{FSUER}; \text{app3} \text{cadr}[j]; \text{list}[\text{cadr}[e]; b]] ; \\
[\text{FSUER}; \text{app3} \text{cadr}[j]; \text{evlis}[\text{cadr}[e]; b]] ; \\
[\text{EXPR}; \text{app3} \text{cadr}[j]; \text{list}[\text{cadr}[e]; b]; b] ; \\
\text{apply}[\text{cadr}[j]; \text{evlis}[\text{cadr}[e]; b]; b]] \end{cases} \\
\lambda[j]; \text{search}[b] ; \\
\lambda[j]; \text{car}[j] = \text{car}[e] ; \\
\lambda[j]; \text{apply}[\text{cadar}[j]; \text{evlis}[\text{cadr}[e]; b]; b] ; \\
\lambda[j]; \text{error}] \end{cases} \]
\]

\text{evlis} [x; b] : \quad \text{machine language}

The arguments of \text{evlis} are a list, \(x\), of expressions, and a list, \(b\), of bound variables. The function \text{evlis} constructs a list of the values obtained by evaluating each element of the list \(x\), using \text{eval} and the list \(b\), and the resultant list is the value of \text{evlis}.

\text{evlis} is equivalent to

\[
\lambda[x; b]; \text{maplist}[x; \lambda[j]; \text{eval}[\text{car}[j]; b]]
\]

\text{expT} [x; n] : \quad \text{machine language}

The value of \text{expT} is the floating-point number, \(x^n\), where \(x\) is given as a floating-point number, and \(n\) is given as a positive or negative floating-point integer.
\textbf{\texttt{fixnil}[x]} \\

The function \texttt{fixnil} copies the list \texttt{x}, replacing null lists on the list \texttt{x} by the pair of parentheses, \texttt{()}, so that null lists will print out as this rather than as blanks. \texttt{fixnil} is equivalent to

\[
\text{label[fixnil;\lambda[x];\texttt{maplist}[\lambda[\texttt{null}[\texttt{car}[j]\rightarrow()];\texttt{atom}[\texttt{car}[j]\rightarrow\texttt{car}[j];
T\rightarrow\text{\text{fixnil}}[\texttt{car}[j]]][]][]}]
\]

\textbf{\texttt{flval}[x]} : machine language

The function \texttt{flval} finds the address of the floating-point representative of the number represented by the association list \texttt{x}. The value of \texttt{flval} is the address of the floating-point number.

The program for \texttt{flval} is

\[
\text{flval}::=\begin{cases}
\text{car}[x] \neq -1 \rightarrow \text{error} \\
\text{BL} & \text{cdr}[x] = 0 \rightarrow \text{error} \\
& x = \text{cdr}[x] \\
& \text{car}[x] \neq \text{FLOAT} \rightarrow \text{go[BL]} \\
\text{return}[	ext{cdr}[x]]
\end{cases}
\]

\textbf{\texttt{form}[x]} : \lambda[G];\text{\texttt{caddr}[G]]

This function is used as a function whose form may later change in \texttt{compsearch}.

\textbf{\texttt{format}[x];f;v]} : \\

The function \texttt{format} has the value \texttt{x}. \texttt{x} is an atomic symbol, \texttt{f} is a list structure, and \texttt{v} is a list of variables occurring in \texttt{f}. The function \texttt{format} causes \texttt{x} and the variables of \texttt{v} to become functions which are available to the \texttt{APPLX} operator. The following example of its use is taken from an earlier version of the programmer's manual.
Consider format
\[
\text{SHAKESPEARE; (UNDER, GREENWOOD, TREE);} \\
\text{(GREENWOOD, TREE)}
\]

There are two variables involved, GREENWOOD and TREE. Then the execution of format generates three functions to which we could give arguments

- \text{shakespeare} \text{SPREADING, CHESTNUT}
- \text{greenwood} \text{BENEATH, SPREADING, CHESTNUT}
- \text{tree} \text{BENEATH, SPREADING, CHESTNUT}

Executing these functions in turn gives \text{(UNDER, SPREADING, CHESTNUT)}

and \text{SPREADING}

and \text{CHESTNUT} respectively.

Thus \text{shakespeare} has as argument a list \( u \) which must contain as many terms as \( v \) and substitutes in \( f \) for one occurrence of each variable in \( v \) the corresponding variable in \( u \).

\text{greenwood} and \text{tree} have as argument a list structure \( g \) and pick out the element in \( g \) which occupies a position corresponding to their's in \( f \).

\[
\text{format}[n; f; v] = \lambda [n; f; v]; \lambda [n; sublis([v]; [f; f]; [P; formatp[v]]); (EXPR, \text{LAMBDA}, V, \text{STELLS}, \text{LIST}, \text{P}, \text{CONST}, F)]] \\
\text{formatq}[n; f; v]]]]]
\]

\[\text{formatp[v]} = \lambda [\text{null}(v) \rightarrow \Lambda ; T \rightarrow \text{cons} \text{[car(v)]; (LIST, (CONST, X), X)}; \\
\text{formatq[cdr(v)]}]]]

\[\text{formatq[n; f; v]} = \lambda [\text{null}(v) \rightarrow n; T \rightarrow \Lambda (z); \text{[z = ERR \rightarrow error}; T \rightarrow \Lambda ](x; y); v] \\
\text{attrib[car(v)]; subst(z; R, (EXPR, \text{LAMBDA}, (X), \text{DESS}, (R, X)))]}]] \\
\text{formatq[n; f; cdr(v)]]}] [\text{pick[car(v); f]}]]]

\[
\text{function}[f; b] : \quad \text{machine language; special form}
\]

If the first element of an S-expression is \text{FUNCTION}, the second element is understood to be the function. A new list is constructed with first element \text{FUNCTION}, second element equal to \( f \), and third element equal to the current list of bound variables. i.e.

\[
\text{eval}[(\text{FUNCTION}, f); b] = (\text{FUNCTION}, f, b)
\]

Having such a list of bound variables carried along with the function insures that the proper values of the bound.
variables are used when the function is evaluated. Thus
\[ \text{apply}[(\text{FUNARG}, f, b); x; a] = \text{apply}[f; x; b] \]

function is equivalent to
\[ \text{list}[\text{FUNARG}; \text{car}[x]; b] \]

gensym : machine language

The function gensym has no arguments. Its value is a
new distinct and freshly created atomic symbol with a print
name of the form GOOD1, GOOD2, ... ,6999.

This function is useful for creating atomic symbols when
one is needed; each one is guaranteed unique.

greater[p; q] : machine language

The function greater is a predicate whose value is true
(one) if the fifteen-bit quantity \( p \) is greater than the fif-
teen-bit quantity \( q \). Otherwise, the value of greater is
false (zero). This function is useful for ordering atomic
symbols in a list, e.g. to put the list in some canonical
form for comparison with other such lists.

inst[x; y; z] : machine language

Here \( x \) is assumed to be an incomplete list of pairs
\( (u_1,v_1), (u_2,v_2), \ldots, (u_n,v_n) \), where the \( u \)'s are atomic and
where some \( v \)'s may be missing. The value of inst is zero if
\( z \) cannot be obtained as substis[\( \tilde{x}; y \)] where \( \tilde{x} \) is a completion
of \( x \) obtained by substituting pairs \( (u,v) \) with the same \( u \) for
the unpaired elements \( (u) \). If \( z \) can be obtained in this way,
\( \text{inst}[x; y; z] \) has as value the completed list \( \tilde{x} \). The purpose
of \( \text{inst} \) is to determine whether \( z \) is a substitution instance
of the expression \( y \).

See RESE page 134.
\[ \text{inst is equivalent to} \]
\[
\text{label[inst; } \lambda \left( [x; y; z] \right) : \begin{cases} 
  x = 0 \rightarrow 0; \ y = 0 \land \ z = 0 \rightarrow 0; \ \text{car} [y] = -1 \\
  \rightarrow \text{search} [x; \lambda (j): \begin{cases} 
    \text{caar} [j] = y; \ \lambda (j); \ \text{cdr} [j] = 0 \rightarrow \\
    \text{naplist} [x; \lambda (k); \begin{cases} 
      k \neq j \rightarrow \text{car} [k]; \\
      T \rightarrow \text{cons} [y; \text{cons} (z; 0)] 
    \end{cases}]; \\
    \text{caar} [j] = z \rightarrow x; T \rightarrow 0; \\
    \begin{cases} 
      y = z \rightarrow x; T \rightarrow 0 \\
      T \rightarrow \text{inst[inst} \left( [x; \text{car} [y]; \text{car} [z]; \text{cdr} [y]; \text{cdr} [z]] \right) \right) \right) 
  \end{cases} \right) 
\]

\[ \text{inty}[x] : \text{machine language; special function} \]

The function inty finds the address of the value of the integer represented by the association list \( x \).

\[ \text{inty is equivalent to} \]
\[
\lambda \left( x \right); \ \text{caar} \left[ \text{prop} \left[ \text{car} [x]; \text{inty}; \text{error} \right] \right) 
\]

\[ \text{joins} [x; y; z] : \text{machine language} \]

The function joins is the same as the function inst except that for joins the lists \( y \) and \( z \) can be lists of y-lists and z-lists respectively.

\[ \text{joins is equivalent to} \]
\[
\text{label[joins; } \lambda \left( [x; y; z] \right) : \begin{cases} 
  z = 0 \rightarrow x; \ \text{car} [z] = 0 \rightarrow 0; \\
  \text{inst}[x; \text{car} [y]; \text{car} [z]] = 0 \rightarrow \rightarrow \\
  \text{joins}[x; y; \text{cons} [\text{car} [z]; \text{cdr} [z]]]; \\
  T \rightarrow \text{append} \left[ \text{joins} \left[ \text{inst} \left[ [x; \text{car} [y]; \text{car} [z]; \text{cdr} [y]; \text{cdr} [z]] \right) \right) \right) \right) \right) \right) 
\]

\[ \text{label[a}; b] : \text{machine language; special function} \]

The effect of label as the first element of a form is described in Section 2.3 of this manual.

\[ \text{label is equivalent to} \]
\[
\lambda \left( [a; b] \right); \text{eval} \left[ \text{cdr} [a]; \text{append} \left[ \text{list} [a]; b \right] \right) \right) 
\]
The predicate `larger` is true (one) if list $x$ is larger than list $y$, and false (zero) otherwise. Larger is used, as one may note in the definition below, to mean either that some pair of corresponding elements obey the greater relation, or, if this is not relevant, that the list $x$ is the longer.

$$\text{larger} \equiv \lambda [x; y]: \begin{cases} \text{null}(x) \rightarrow 0; \text{null}(y) \rightarrow 1; \\ \text{atom}(x) \land \text{atom}(y) \rightarrow \text{greater}(x; y); \\ \text{atom}(x) \rightarrow 0; \text{atom}(y) \rightarrow 1; \\ \text{larger}(\text{car}(x); \text{car}(y)) \rightarrow 1; \\ \text{larger}(\text{car}(y); \text{car}(x)) \rightarrow 0; \\ 0 \rightarrow \text{larger}(\text{adr}(x); \text{adr}(y)) \\ \end{cases}$$

The function `list` of any number of arguments has as value the list of its arguments.

For the `APPN` operator, if $a$ is the associated $a$-list, `list` is equivalent to (where $x$ represents the string of $x_i$'s above).

$$\lambda [x, a]: \text{maplist}([x]; \lambda [j]; \text{eval}([\text{car}(j); a])]$$

See REPE page 133.

The value of `loc` (or `locq`) is the location where the value of the program variable, $x$, is stored. See the Section on the Program Feature.

Note:

$$(\text{REPLACA}, (\text{LCOQ}, X), e) \equiv (\text{SEPQ}, X, e)$$
$$(\text{REPLACA}, (\text{LOC}, X), e) \equiv (\text{SEP}, X, e)$$
makable x

This function ("make car or cdr abler") uses the function
desc to speed up and improve functions such as car or caddr,
etc. The argument x is a list of pairs of the form, for ex-
ample,

\[
\{(\text{car}, (A, A)), (\text{cadr}, (D, A)), (\text{caddr}, (D, D, A)), \ldots \}
\]

The function makable takes this list and on the association
list of the first element of each pair puts EXPR followed by
the lambda expression,

\[
\lambda \langle \text{l} \rangle; \text{desc} [\text{WAY}; \text{j}],
\]

where the second list of the pair has been substituted for
WAY. The function funarg is used below to bind the PRO used
by defl to EXPR.

makable is equivalent to

funarg[λ \langle \text{j} \rangle; makelist \langle \text{j} \rangle; function \langle \text{l} \rangle; defl [\{(\text{car}[\text{car}[\text{k}]);
\text{subst}[\text{car}[\text{cdr}[\text{car}[\text{j}])]; (\text{QUOTE}, \text{WAY}); (\text{LAMBDA}, (\text{j}),
\text{DESC}, (\text{QUOTE}, \text{WAY}), \text{j}))\}]; \text{PRO}, \text{EXPR}]]

map[x; f]

The function map is like the function makelist except
that the value of map is nil, and map does not do a cons of
the evaluated functions. map is used only when the action
of doing the \text{f}(x) is important.

The program for map, (LT is the only program variable), is

\[
\begin{align*}
/ & \text{LT} = x \\
\text{LOP} & \text{go} \langle \text{null} \langle \text{LT} \rangle \rightarrow \text{END}; \text{T} \rightarrow \text{COMP} \\
\text{COMP} & \text{f} \langle \text{LT} \rangle \\
\text{LT} & = \text{cdr} \langle \text{LT} \rangle \\
\text{go} & \langle \text{LOP} \rangle \\
\text{END} & \text{return} \langle \text{nil} \rangle
\end{align*}
\]
mapcar\[x;f\] : machine language

The function mapcar has the value nil. It is like map except that it uses the function \( f \) on the list \( x \) until either the list is finished, or until a value of \( f[y] = 1 \) is found.

\[
\text{mapcar is equivalent to}
\]
\[
\text{label}[\text{mapcar}; \begin{cases} \text{f} & \text{x} = 0 \\
\text{null} \rightarrow \text{NIL} \rightarrow \text{cons}[f[x]; \text{mapcar}[\text{cdr}[x]; f]] & \text{else}
\end{cases}]
\]

mapcon\[x;f\] :

The function mapcon is like the function maplist except that the resultant list is a concatenated one instead of having been created by cons-ing.

\[
\text{mapcon is equivalent to}
\]
\[
\text{label}[\text{mapcon}; \begin{cases} \text{f} & \text{x} = 0 \\
\text{null} \rightarrow \text{NIL} \rightarrow \text{append}[f[x]; \text{mapcon}[\text{cdr}[x]; f]] & \text{else}
\end{cases}]
\]

maplist\[x;f\] :

The function maplist is a mapping of the list \( x \) onto a new list \( f[x] \).
See RESE page 138.

\[
\text{maplist is equivalent to}
\]
\[
\text{label}[\text{maplist}; \begin{cases} \text{f} & \text{x} = 0 \\
\text{null} \rightarrow 1 \rightarrow \text{cons}[f[x]; \text{maplist}[\text{cdr}[x]; f]] & \text{else}
\end{cases}]
\]

matrimenty\[x;y\] : machine language

This function has as value a matrix which is the product of the row matrix, \( x \), and the column matrix, \( y \). The two matrices are entered either as direct functional arguments in matrimenty or from being given on the a-list. For example, if the matrix

\[
A = \begin{pmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{pmatrix}
\]

is to be multiplied by itself. The two arguments, for the
The APPLY operator are in the form

\[ x = (\text{MATRIX}, (\text{ROW}1, A11, A12, A13), (\text{ROW}2, A21, A22, A23), (\text{ROW}3, A31, A32, A33)) \]
\[ y = (\text{MATRIX}, (\text{COL}1, A11, A21, A31), (\text{COL}2, A12, A22, A32), (\text{COL}3, A13, A23, A33)) \]

Actually any atomic symbols may be used in place of ROW or COL. It is only necessary that the rest of each sublist be the elements of the row matrix and the column matrix in the right order. See page 126 of the following reference where this function was developed,


\[ \text{neone}[x; y] : \text{ machine language} \]

The function \text{neone} concatenates its arguments without copying the first one. The operation is identical to that of \text{attrib} except that the value is the entire result, (i.e. the modified first argument, \( x \)).

The program for \text{neone} has the program variable \( M \) and is as follows,

\[
/ \quad X = 0 \quad \rightarrow \quad \text{return}[Y] \\
M = X \\
A2 \quad \text{cdr}[M] = 0 \quad \rightarrow \quad \text{go}[A1] \\
M = \text{cdr}[M] \\
\text{go}[A2] \\
A1 \quad \text{cdr}[M] = Y \\
\quad \rightarrow \quad \text{return}[X] \\
\]

\[ \text{not}[x] : \text{ machine language} \]

The function \text{not} is a predicate. Its value is true \((\text{one})\) if its argument is false \((X = \text{zero})\), and false \((\text{zero})\) if its argument is true \((X = \text{one})\).
The function **null** is a predicate. Its value is true (one) if its argument is zero, and false (zero) otherwise.

The arguments of **or** are evaluated in sequence, from left to right, until one is found that is true, or until the end of the list is reached. The value of **or** is true (one) or false (zero) respectively.

The function **pair** has as value the list of pairs of corresponding elements of the lists **x** and **y**. The arguments **x** and **y** must be lists of the same number of elements. They cannot be atomic symbols.

See RESE page 133.

The function **pick** finds the atomic symbol, **x**, in the list structure, **y**. The value of **pick** is a list of A's (for car) and D's (for cdr) which give the location of **x** in **y**. This value could be used for example as the first argument of **desc**.

**Example**

\[
pick[y; \{ ((u,v)), u \}] = (A,A,D,A)
\]

**pick** is equivalent to

\[
\text{label}[\text{pick}; \lambda[k;y]; [\text{null}[y] \rightarrow \text{NO}; \text{equal}[x;y] \rightarrow \text{NIL}; \\
\text{atom}[y] \rightarrow \text{NO}; T \rightarrow \lambda[j]; \text{equal}[j;\text{NO}] \rightarrow \\
\lambda[k]; \text{equal}[k;\text{NO}] \rightarrow \text{NO}; T \rightarrow \text{cons}[D;k];] \\
\text{pick}[x;\text{cdr}[y]]; T \rightarrow \text{cons}[A;\lambda]; ]\text{pick}[x;\text{car}[y]]]
\]

The function **prod** computes the floating-point product of the two floating-point numbers represented on the
association lists of \( x \) and \( y \). The value of \( \text{prdat} \) is the address of the atomic symbol containing the product.

\[
\text{print}[x] : \quad \text{machine language}
\]

The function \( \text{print} \) prints out (on-line or off-line depending on sense switch settings, see Section 3.1), its argument \( x \) if \( x \) is a legal list structure, and malfunctions if it is not. The value of \( \text{print} \) is always zero.

In the following equivalent expression, \( \text{print2} \) is a routine which prints up to six EOD characters when the characters are justified to the left, followed by the illegal character \( \text{T} \).

\[
\text{label[print;} x [k] ; \text{car}[x] = -1 \rightarrow \text{print}[x] ; T \rightarrow
\]

\[
\begin{array}{l}
[ \text{print2} [\text{EOD}]; \text{print}[\text{car}[x]]; T \rightarrow
\end{array}
\]

\[
[ \text{cdr}[x] = 0 \rightarrow \text{print2} [\text{EOD}]; T \rightarrow
\]

\[
[ \text{print2} [\text{EOD}]; \text{print}[\text{cdr}[x]]; T \rightarrow
\]

where

\( 74_8 = \text{EOD left-parenthesis, } "(" \)

\( 34_8 = \text{EOD right-parenthesis, } "\)"

\( 73_8 = \text{EOD comma, } ",\"

\[
\text{print1}[x] : \quad \text{machine language}
\]

The routine \( \text{print1} \) prints the association list of the atomic symbol \( x \). If \( x \) is not an atom, an error occurs.

The equivalent program for \( \text{print1} \) has the two program variables \( \text{ST} \) and \( \text{VAL} \), and is as follows,

\[
/ \quad \text{car } x \neq -1 \rightarrow \text{error}
\]

\[
\text{ST} = x
\]

\[
\text{AI} \quad \text{cdr}[x] = 0 \rightarrow \text{error}
\]

\[
x = \text{cdr}[x]
\]

\[
\text{car}[x] = \text{PRINT} \rightarrow \text{go [A3]}
\]

\[
\text{car}[x] \neq \text{FLOAT} \rightarrow \text{go [A]}
\]

\[
x = \text{car}[\text{cdr}[x]]
\]
VAL = FLOMAM[x]
replaced[cons[FNAME; cons[VAL; cdr[ST]]]; ST]

x = cdr[ST]

A3  x = car[cdr[x]]
A2  print2[car[cdr[x]]]
    cdr[x] = 0 --> return
    x = cdr[x]
\   go[A2]

printprop[x]

The function printprop prints the properties on the as-
association list of the atomic symbol, x. The parts of the
list printed to by any of FLO,SUE,FSUE,PHAME,APVAL, or INT
are not printed.

The function prog2[x;y] used by printprop has the value
y, though x may be used to effect some action. For example,
printprop[x] = prog2[print[list[PROPERTIES;OF;x]];printpl[cdr[x]]]

prints out
"PROPERTIES OF x",

and then goes on to printpl below.

printpl x   (see printprop x )

The function printpl, which does the printing of the
properties is equivalent to,

label[printpl; (x [x]); (null[x] -> NIL; T -> prog2[print[car[x]]; search[(FLO,SUE,FSUE,PHAME,APVAL,INT);
   (LAMDA, (x), (EQUAL, (CAR, (CAR, x))));
   (LAMDA, (x), (PRINTPL, (CAR, (CDR, x))));
   (LAMDA, (x), (PRINTPL, (CDR, x)))]])]

LISP[ed]: machine language; special form

The Program Feature is discussed in Section 3.5. For in-
terest, we include here the program which the interpreter
uses to work out a program given by the expression, e, and the e-list of pairs, b. There are five program variables, (pv is the list of program variables) t2, t3, pr, r, br

\[
\begin{align*}
t2 &= \text{maplist}(\text{car}[e]; \lambda[j]; \text{list}(\text{car}[j]; \text{NIL})) \\
t3 &= \text{append}(t2; b) \\
t3 &= pv \\
pr &= t2 \\
\text{attrib}(pv; t3) \\
pr &= \text{cdr}[e] \\
r &= pr \\
\text{ML} &= \text{go}([\text{atom}(\text{car}[e]) \rightarrow \text{ADV} ; \text{car}[e] = \text{GO} \rightarrow \text{GO} ; \\
&\text{car}[e] = \text{RETURN} \rightarrow \text{RET}; T \rightarrow \text{NORM}) \\
\text{NORM} &= \text{eval}(\text{car}[r]; br) \\
\text{ADV} &= r = \text{cdr}[r] \\
\text{go} &= \text{ML} \\
\text{RET} &= \text{return}(\text{eval}(\text{cadar}[r]; br)) \\
\text{GO} &= t2 = \text{eval}(\text{cadar}[r]; br) \\
&\text{search}(pr; \lambda[j]; \text{car}[j] = t2; \lambda[i]; r = \text{cdr}[i] ; \\
&\lambda[j]; \text{error}) \\
&\text{go}(\text{ML}) \\
\end{align*}
\]

\text{prop}[x;y;u] : machine language

The function \text{prop} searches the list (or atomic symbol) \( x \), for an item identical with \( y \). If such an element is found, the value of \text{prop} is the rest of the list beginning immediately after the element. Otherwise the value is \( u \). (\( u \) is a functional)

\text{prop} is equivalent to

\[
\text{label}([\text{prop}; \lambda [x;y;z]; \text{null}[x] \rightarrow u; \text{car}[x] = y \rightarrow \text{cdr}[x]; \\
T \rightarrow \text{prop}([\text{cdr}[x]; y; z])])
\]

\text{punchdef}[x] :

The function \text{punchdef} writes the definitions of the functions named in the list \( x \) out on tape 3 in SDC for off-line punching.
punchdef is equivalent to

\begin{verbatim}
label[punchdef;λ[x];map[x;λ[y];punchl[list[car y];
 extrad[car[propl[car y]];EXPR;
 λ[(nil)];error[conc[[NO,EXPR,ON,PROPERTY,LIST,OF];
 list[car y]];()];PUNCHDEF,CANNOT, CONTINUE]]]]]])]]])]]
\end{verbatim}

\textbf{punchl[x]} : machine language

The function \texttt{punchl} writes the list, \textit{x}, out onto tape 3 in ECD form for off-line punching. The resultant cards have the list information in consolidated form, i.e., extra blanks have been left out and commas inserted in the correct locations. All $72$ columns of the card are used.

\textbf{punchs[x]} : machine language

The function \texttt{punchs} writes the list, \textit{x}, out onto tape 3 for off-line punching. Each sublist of the top level of list \textit{x} appears as a separate card punched in SAP format. \texttt{punchs} is used with the LISP Compiler.

\textbf{quote} : machine language; special form

The value of a list beginning with \texttt{QUOTE} is always the rest of the list.

\textbf{rd} : machine language

The function \texttt{rd} of no arguments determines to which category, (letter, number, punctuation, etc.), the symbol on the input list belongs. \texttt{rd} is used by the function \texttt{read}.

\textbf{read} : machine language

The function \texttt{read} of no arguments reads one list from cards or type, depending on the sense switch settings. The value of \texttt{read} is the list it has read.
reduce\([m]\) : machine language

The function \texttt{reduce} reduces an \(n \times n\) matrix to an \((n-1) \times (n-1)\) matrix. The function has been used in electrical network reduction. See the references under \texttt{reduce\(n\!m\!n\)} below.

\texttt{reduce\(n\!m\!n\)} \([m;n]\) : machine language

This function uses the function \texttt{reduce} to reduce a square matrix \(m\) to an \(n\) by \(n\) matrix whose rank is less than that of \(m\). The argument \(n\) must be given in floating-point form. This function has been used in electrical network reduction, see


\texttt{remprop\([x,p]\)} :

The function \texttt{remprop} searches the list, \(x\), looking for all occurrences of the property \(p\). When such a property is found, its name and the succeeding element are removed from the list. The two "ends" of the list are tied together as indicated by the dashed line below.

\begin{center}
\begin{tikzpicture}
    \draw[->] (0,0) -- (0.5,0);
    \draw[->] (0.5,0) -- (1,0);
    \draw[->] (1,0) -- (1.5,0);
    \draw[->] (1.5,0) -- (2,0);
    \draw[->] (2,0) -- (2.5,0);
    \draw[->] (2.5,0) -- (3,0);
    \draw[->] (3,0) -- (3.5,0);
    \draw[->] (3.5,0) -- (4,0);
    \draw[->] (4,0) -- (4.5,0);
    \draw[->] (4.5,0) -- (5,0);
    \draw[->] (5,0) -- (5.5,0);
    \draw[->] (5.5,0) -- (6,0);
    \draw[->] (6,0) -- (6.5,0);
    \draw[->] (6.5,0) -- (7,0);
    \draw[->] (7,0) -- (7.5,0);
    \draw[->] (7.5,0) -- (8,0);
    \node at (0,-0.5) {\(x\)};
    \node at (8,-0.5) {\(\rightarrow\)};
    \node at (7.75,-1) {name of the property};
\end{tikzpicture}
\end{center}
The value of \texttt{remprop} is \texttt{NIL}.

\texttt{remprop} is equivalent to
\[ \lambda \left[ (x; p) ; \texttt{rempl} [x] \right] , \]
where
\[ \texttt{rempl} [x] = \begin{cases} \texttt{null} [x] \land \texttt{null} [\texttt{cdr} [x]] \rightarrow \texttt{NIL} ; \texttt{cdr} [x] = p \rightarrow \texttt{prog2} [\texttt{replad} [x]; \texttt{null} [\texttt{cdr} [x]] \rightarrow \texttt{NIL} ; T \rightarrow \texttt{addr} [x]]; \\ \texttt{rempl} [x]; \\ T \rightarrow \texttt{rempl} [\texttt{addr} [x]] \end{cases} , \]
where \texttt{prog2}[u; v] has the value \( v \), though \( u \) is used to effect an action.

\[ \texttt{replad} [x; y] \]  
\[ \texttt{replad} [x; y] \]

These functions replace the \( \{ \texttt{address}(a) (d) \} \) part of the first argument, \( x \), by the second argument, \( y \). I.e. \( y \) is stored in the \( (a, d, \text{ or } w \text{ part}) \) of the location pointed to by the first argument. The value of these functions is always zero.

\[ \texttt{sassoc}[x; y; u] : \]  
\[ \texttt{sassoc}[x; y; u] : \texttt{machine \ language} \]

The function \texttt{sassoc} searches \( y \), which is a list of lists, for a sublist whose first element is identical with \( x \). If such a sublist is found, the value of \texttt{sassoc} is the sublist with the first element removed. Otherwise the value is \( u \).

\texttt{sassoc} is equivalent to
\[ \texttt{label}[\texttt{sassoc}; \lambda \left[ (x; y; u); \texttt{null} [y] \rightarrow u; \texttt{caar} [y] = x \rightarrow \texttt{cdr} [y]; \texttt{T} \rightarrow \texttt{sassoc} [x; \texttt{addr} [y]; u] \right] ] . \]

\[ \texttt{search} [x; p; f; u] : \]  
\[ \texttt{search} [x; p; f; u] : \texttt{machine \ language} \]

The function \texttt{search} looks through the list, \( x \), for an
element that has the property, \( p \), and if such an element is found, the function, \( f \), of that element is the value of \( \text{search} \). If there is no such element, the function \( u[x] \) is taken as the value of \( \text{search} \).

\[ \text{search} \text{ is equivalent to} \]
\[
\lambda [x; \text{search}; \lambda [p; f; u] \mapsto (\text{null}[x] \mapsto u[x]; p \mapsto f[x]; \text{T} \mapsto \text{search}[\text{cdr}[x]; p; f; u])]
\]

\[ \text{search}[x; v; p; f; u] \]

See the function \( \text{compssearch} \); the function \( \text{search} \) is the fast search whose \( S \)-expression is set up by \( \text{compssearch} \).

\[ \text{search} \text{ is equivalent to} \]
\[
\lambda [x; v; p; f; u]; \text{subs}l[(\text{NAME}, x), (\text{VAR}, v), (\text{PF}, p), (\text{FF}, f), (\text{UF}, u)];
(\text{LAMDA}, (\text{VAR}), (\text{COND}, ((\text{NULL}, \text{VAR}), \text{UF}), (\text{PF}, \text{FF}),
(\text{null}, (\text{NAME}, (\text{CDR}, \text{VAR})))])]
\]

\[ \text{select}[x; a] : \text{ special form} \]

The function \( \text{select} \) can be considered to have an argument \( x \) of the form
\[
(\lambda ; p_1 \mapsto e_1; p_2 \mapsto e_2; \ldots; p_n \mapsto e_n; \text{T} \mapsto u)
\]
and an argument \( a \) which is a list of pairs, an \( a \)-list. The \( p_i \)'s are evaluated in sequence until one is found such that
\[
p_i = \text{S},
\]
and in that case the value of \( \text{select} \) is \( e_i \); if no such \( p_i \) is found, the value of \( \text{select} \) is \( u \).

\[ \text{select}[x; a] \text{ is equivalent to} \]
\[
\lambda [x; a]; \text{select}2[\text{eval}[\text{car}[x]; a]; \text{cdr}[x]]
\]

\( \text{eval}[\text{car}[x]; a] \) here will give the value of the \( S \)-above.

\[ \text{select}2[y; v] = (\text{null}[\text{cdr}[v] \mapsto \text{eval}[\text{car}[v]; a];
\]
\[ v = \text{eval}[\text{car}[v]; a] \mapsto \text{eval}[\text{car}[v]; a];
\]
\[ \text{T} \mapsto \text{select}2[v; \text{cdr}[v]]
\]
\[
\text{set}(x;y) : \text{ machine language}
\]
\[
\text{setq}(x;y) : \text{ machine language ; special form}
\]

The \text{set} and \text{setq} functions are used to change the values of the program variables when using the Program Feature (see Section 2.5). The program variables are initially bound to null lists.

Note that (in S-expression form),
\[
(\text{set}, (\text{QUOTE}, v), e) \equiv (\text{setq}, v, e)
\]

\[
\text{simplify}[x] : \text{ machine language}
\]

The function \text{simplify} takes the algebraic expression \(x\) and simplifies it. The resultant expression is the value of \text{simplify}.

The notation allowed in the algebraic expression is any compounding of the following modified polish notation:

- \((\text{TIMES}, A, B, C); \text{(any number (\(\geq 1\)) of arguments)}\)
- \((\text{SUM}, A, B, C); \text{(any number (\(\geq 1\)) of arguments)}\)
- \((\text{COS}, x)\)
- \((\text{SIN}, x)\)
- \((\text{MINUS}, x)\)
- \((\text{POWER}, x, y) \equiv x^y\)
- \((\text{RECIPE}, x) \equiv \frac{1}{x}\)
- \((\text{LOG}, B, X) \equiv \log_B x\)

For example,
\[
(a)(b)(\frac{1}{c}) - d^e \quad \text{is written as}
\]
\[
(\text{PLUS}, (\text{TIMES}, A, B, (\text{RECIPE}, C)), (\text{MINUS}, (\text{POWER}, D, E)))
\]

**speak**: machine language

The function `speak`, of no arguments, causes the contents of the GHS counter to be printed on-line or off-line depending on the sense-switch settings. See the function `count` for a description of this counter.

**sublis** \([x;y]\) : machine language

**Note**: \(x\) is a list of pairs,

\[((u_1,v_1),(u_2,v_2),\ldots,(u_n,v_n))\]

where the \(u\)’s are atomic. The value of `sublis[x;y]` is the result of substituting each \(v\) for the corresponding \(u\) in \(y\). See RFSE page 134.

`sublis` is equivalent to

\[
\text{label}\left(\text{sublis} ; \lambda \left[\left(x,y\right)\right] ; \begin{array}{l}
x = 0 \rightarrow y ; y = 0 \rightarrow 0 ; T \rightarrow \\
\text{search}[x ; \text{function} \left[\lambda \left[j\right] ; \text{equal} \left[y , \text{caar} \left[j\right]\right]\right] ;
\text{function} \left[\lambda \left[j\right] ; \text{cadar} \left[j\right]\right] ;
\text{car} \left[y\right] = -1 \rightarrow y ; T \rightarrow \text{cons} \left[\text{sublis} \left[\text{car} \left[y\right]\right] ; \text{sublis} \left[\text{cdr} \left[y\right]\right]\right]\right]
\end{array}\right)
\]

**subst** \([x;y;z]\) : machine language

The function `subst` has as value the result of substituting \(x\) for all occurrences of the atomic symbol \(y\) in the S-expression \(z\). See RFSE page 132.

`subst` is equivalent to

\[
\text{label}\left(\text{subst} ; \lambda \left[\left(x,y,z\right)\right] ; \begin{array}{l}
y = z \rightarrow \text{copy} \left[x\right] ; \text{car} \left[z\right] = -1 \rightarrow z ;
T \rightarrow \text{cons} \left[\text{subst} \left[\text{y} ; \text{car} \left[z\right]\right] ; \text{subst} \left[\text{y} ; \text{cdr} \left[z\right]\right]\right]\right]
\end{array}\right)
\]

**substr** \([x;y]\)

The proposition `substr` is true (one) if the list structure \(y\) is a substructure of the list structure \(x\), otherwise proposition is false (zero).
substr is equivalent to

\[
\text{label} \begin{cases} 
\text{substr} x [x; y]; & \begin{cases} x = y \rightarrow 1; \text{null} x \rightarrow 0; \text{atom} x \rightarrow 0 \\
\text{substr} \text{car} x [x; y] \rightarrow 1; T \rightarrow \text{substr} \text{cdr} x [x; y] \end{cases} 
\end{cases}
\]

\text{sum} [x; y] : machine language

The function \text{sum} will compute the sum of the two floating-point numbers whose association lists are \( x \) and \( y \). The value of \text{sum} is the address of the atomic symbol containing the sum.

tflop [x] : machine language

The proposition tflop is true (one) if \( x \) is the address of an association list containing a floating-point number. The value is false (zero) if \( x \) is not an association list, or if the list does not contain FLOAT.

tflop is equivalent to

\[
/ \text{car} x = -1 \quad \text{return} (0) \\
B1 x = 0 \quad \text{return} (0) \\
\text{car} x = \text{FLOAT} \quad \text{return} (1) \\
x = \text{cdr} x \\
\text{go} B1
\]

\text{uncount} : machine language

The function uncount, of no arguments, turns off the CONS counter. See the function count for a description of this counter.

END
Programmers Manual--Draft: Addendum to
Section 3.4 on the Flexowriter January 29, 1960

An example follows of an actual run with the Flexowriter. The Flexowriter types out in a different color from the type-in, and this is indicated below by underlining. At present the commas separating elements of lists can be replaced by blanks since the comma is inconveniently upper case. However, in the example the commas are used for clarity. Explanatory comments have been put in below in lower case to the right of the actual output.

The card-reader which was called in by IOFLXP(READ) below had cards in it defining the function RVSE (reverse a list), followed by a card IOFLXP(READ) to return to control to the Flexowriter.

```
FIX
GO CDR ((A,B,C)) ()
STOP

(B,C) GO CDR ((A,B,C))
STOP
GO () STOP

(B,C) GO TEN
  0 CAR ((A,B),(C)) ()
  1 CDR ((D,(E,F))) ()
  2 CONS ((G,H),
  3 (X,Y)) ()
```

the Flexowriter takes over, and requests a type-in (Sequence Mode) (type-in) the Flexowriter digests the information and finds the answer, and asks for more. (type-in)

the Flexowriter does not have a full triplet f;x;a, so asks for the rest answer (the TEN-Mode is entered) (type-ins)
: RLN
01
0
\[(A, B)\]
 1
\[((E, F))\]
 2
GO
RLN
22
2
GO
RLN
32
WRONG ORDER
R LN
33
3
\[((G, H), I, J)\]
 2
TEN
CAR ((A B)) ()
LINE NUMBER MISSING RETYPE
2  IOFLIP(READ) ()
:3  RVSE ((A, B, C, D)) ()
4  IOFLEX(PR INT) ()
:5  IOFLEX(NG) ()
:6  STOP
:7  FIN
: R LN
25
2
(read lines 0 and 1)
line 0
answer
line 1
answer
(line 2)
line 2
the Flexowriter asks for the rest
(line 3 to 2)
error
(line 3)-line 2 has been re-
membere d
answer
error indication
type-ins
(error indication)
(type-ins)
(read lines 2 through 5)
(RVSE, REV1)

READ

(D,C,B,A)

T

ERROR NUMBER : F 1:

NG

F

GO

RLN

67

67

FIN

go to card-reader to read
in definition of RVSE and
REV1 (used in RVSE)

flip back to flexowriter
read line 3 and find

answer to RVSE
read line 4

True-the Flexowriter is printing
read line 5

NG is not a valid argument
for XOFLEX, so

answer is False

(read line 6 and 7)

end of computation