LISP

Preliminary Programmer's Manual - Draft

January 7, 1960

J. McCarthy
R. Brayton
D. Edwards
P. Fox
L. Hodes
D. Luckham
K. Maling
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 difference in form however between functions and pseudo-functions except that pseudo-functions are not usually used as sub-expressions.

Page 12, Note that the compiler destroys the EXPR and S-expression 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.!

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

⟶ bFINbb ↵

to end the run, or

⟹ bCRDbb ↵

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)):
$:n_{i+1}$ ⟶ ⟶ bFINbb ↵

Page A5, lines 6 and 7 from the bottom should read:
cond$[x_1;x_2;\ldots;x_n]$: machine language; special form

The function 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

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 flexowriter, a package of routines to control the input-output aspects of the flexowriter 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 7094,-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 RFSE. The *apply* now in use is a function of three arguments,

apply[f;x;a]

where

f = an S- expression representing an S- function, f^*, of say n arguments,
x = a list of n arguments,
a = a list of pairs of the form, (atomic symbol, value), used to assign values to free variables.

The value of apply[f;x;a] is the value of the S- function $f^{*}[x]$ 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

cons[car[y];cdr[z]]

applied to the list

((A,B),(C,D)),

where no free variables need to be assigned. The arguments f,x, and a for the apply function, written in the form of S-expressions, are,

f: (LAMBDA,(Y,Z),(CONS,(CAR,Y),(CDR,Z))),
x: ((A,B),(C,D)),
a: ( )

and the value of 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: CAR,
x: ((A,B)),
a: ( )

where apply[f;x;a] = car[(A,B)] = A,

since 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: (A,B)

would be wrong. Note also that the LAMBDA definiton is not needed in this example since the specification of the single argument is clear. It would be correct but unnecessary to write

f: (LAMBDA,(Y),(CAR,Y)),
x: ((A,B)),
a: ( )

The exact descriptions of function formats acceptable to the APPLY operation are described further in Section 2.3.

Example 3:

As an example involving an a-list we have

f: (LAMBDA,(Y),(CONS,Y,Z)),

x: (A),

a: ((Z,(B)))

where apply[f;x;a] = λ[[y];cons[y;(B)]][A] = cons[A;(B)] = (A,B).

In this example, the λ-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 label definition is required,

subst[x;y;z] = [atom[z]→[y = z→x;T→z];
T→cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]

f: (LABEL,SUBST,(LAMBDA,(X,Y,Z),(COND,(
(ATOM,Z),(COND,((EQ,Y,Z),X),(T,Z))),(T,
(CONS(SUBST,X,Y,(CAR,Z)),(SUBST,X,Y,(CDR,Z))))))),

x: ((A,B),D,((D,C),D))

a: ( )

where apply[f;x;a] = subst[(A,B);D;((D,C),D)] = (((A,B),C),(A,B))

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

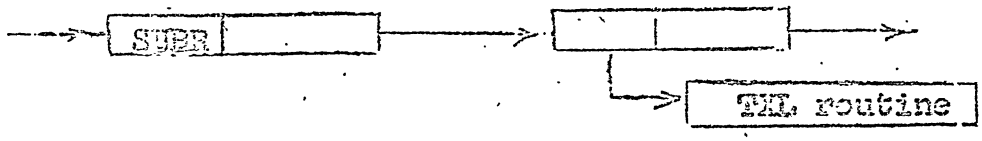
A program for the LISP system consists of sequences of f;x;a triplets strung together. The APPLY operator automatically operates on each triplet in turn and returns with the value of the triplet. The details for submitting and running such a program are given in Section 3.

2.2 Definitions of Functions in LISP

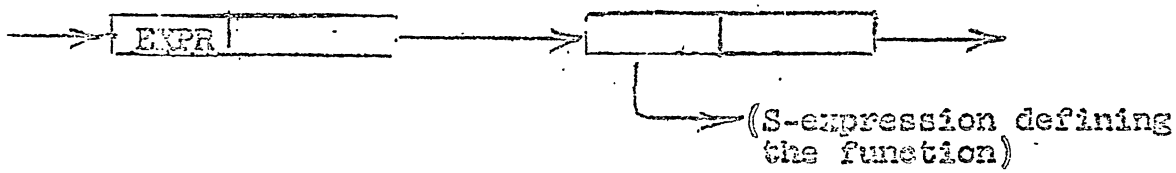
In RFSE, functions are connected to their names only

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 M-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 ~~used~~ put on an association list for storage

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

$$ff[x] = [atom[x] \rightarrow x; T \rightarrow ff[car[x]]], \quad \text{cf.p.132 of RFSE,}$$

and

$$alt[x] = [null[x] \wedge null[cdr[x]] \rightarrow x, T \rightarrow cons[car[x]; alt[cdr[cdr[x]]]]].$$

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

```
f:   DEFINE,
x:   (((FF,(LAMBDA,(X),(COND((ATOM,X),X),
     (T,(FF,(CAR,X)))))),(ALT,(LAMBDA,(X),
     (COND,((O,(NULL,X),(NULL,(CDR,X))),X),
     (T,(CONS,(CAR,X),(ALT,(CDR,(CDR,X))))))))))),
a:   ( )
```

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 pseudo-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~~ described by an atomic symbol ~~(atomic function)~~, or by a list ~~(compound function)~~. In the second example in Section 2.1 above, CAR is ~~an~~ atomic ~~function~~, whereas the LAMBDA expression is a compound function.

There is the wrong distinction

Atomic Functions

If a function is atomic, the association list for the atomic symbol is searched by the APPLY operator for either SUBR or EXPR (see Section 2.2 just preceeding). 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 x of arguments. If neither SUBR or EXPR is found on the association list for the function, then the a-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 x of arguments.

Compound Functions

If a function is compound, the first element of the list, or S-expression, for it may be any of LAMBDA, LABEL, or FUNARG.

If the list begins with LAMBDA for example (LAMBDA,(Y), (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 APPLY operator takes the names of the dummy variables and pairs them with the values given on the list x 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 a-list. Then the third element of the LAMBDA-list, the form for the function, is evaluated using the enlarged a-list of pairs of assigned values.

The third element of the LAMBDA 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 APVAL or APVAL1 on its association list- cf. Section 4), and if so, the value of the atomic form is the constant, i. e. caar if the list to which APVAL or APVAL1 points. If the atomic form is not a constant, the a-list is searched for the most recent pairing of this variable on the a-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 LAMBDA 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 x 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~~, For example

apply[(FUNARG,f,b);x;a] = apply[f;x;b].

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 FSUBR or FEXPR in place of SUBR 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, andthen the expresion 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).

PROG 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 70^{4} 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

4.21+10 is used for 4.2×10^{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,

(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 PNAME.

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,

(PROG,L,sequence of statements)

where PROG signals to the interpreter that a program for sequential execution is to follow, and where 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 L may be a null list; this is quite legal.

The functions available for changing program variables are SET, SETQ, LOC and LOCQ. To set a program variable V equal to an expression E, the functional statement

(SET,(QUOTE,V),E)

can be executed. However, since it is annoying to have to write the QUOTE all the time, a special form SETQ has been introduced whereby

(SET,(QUOTE,V),E) is entirely equivalent to (SETQ,V,E).

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

Thus

(RPLACA,(LOC,(QUOTE,V)),E)
(RPLACA,(LOCQ,V),E)
(SET,(QUOTE,V),E)
(SETQ,V,E)

are all equivalent in their effect.

The following is an example of a program using the program feature. It is a version of 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,

(LAMBDA,(L,P,FN,U),(PROG,(LT),

(SETQ,LT,L),

TEST,(GO,(COND,((NULL,LT),(QUOTE,RETU)),(T,(QUOTE,CONT)))),

CONT,(GO,(COND,((P,LT),(QUOTE,RETF)),(T,(QUOTE,STEP)))),

STEP,(SETQ,LT,(CDR,LT)),

(GO,(QUOTE,TEST)),

RETU,(RETURN,(U)),

RETF,(RETURN,(FN,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

(LABEL,NAME,(LAMBDA 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

((,LXD,0,4),(,TX1,00007,4,-1),(,TRA,2+5),(00008,BSS,0).

In this example, the objects beginning with G are atomic symbols generated for use within the Compiler. The BSS,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 BSS,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 TXL 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$\supset$F2.

2) If F1$\supset$F2 and F2$\supset$F1, then F1 is "equivalent" to F2, and the relation is written as F1$\sim$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 five functions to be compiled and they are related as follows,

F3$\supset$F2 , F3$\sim$F4

F4$\supset$F1 , F4$\supset$F5,

functions should be compiled in the order

1. COMPILE ((F1,F2)) , ( )
2. COMPILE ((F5)) , ( )
3. COMPILE ((F3,F4)) , ( ).

The functions caar,cadr,...,eddar,edddr 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 PROG the argument of any GO must be an atomic symbol; it cannot be a conditional expression. To get around the restriction, conditional statements are themselves allowed as action statements provided they give rise to a subsequent action, e. g. (COND,(P1,(GO,A)),(P2,(RETURN,X)),(P3,(SETQ,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 PROG for search given in the previous section must be revised for the Compiler to the following

```
(LAMBDA,(L,P,FN,U),(PROG,(LT),
        (SETQ,LT,L),
TEST,(COND,((NULL,LT),(RETURN,(U))),((P,LT),(RETURN,(FN,L)))),
        (SETQ,LT,(CDR,LT)),
        (GO,TEST)
        ) )
```

Macros can be defined for the Compiler. They are defined by using the APPLY operator on an attrib[n;s] where n the name of the macro, and s is an S-expression whose

first element is "EXP". The second element in the $\mathcal{S}$-expression is a list of dummy variables of the form (X,Y,...,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 $\mathcal{S}$-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)
```
ATTRIB
(COUNT1,(EXP,((X,Y),(),CLA,Y),(),ADD,ONE),
(),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, (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)
```
ATTRIB
(CAR,(EXP,((X,Y),(),LXD,Y,4),(),CLA,0,4),
(),PAX,0,4),(),SXD,X,4))
()
```

In this example, the value of car is actually stored at X, though the function is still a function of one argument.

3)
```
ATTRIB
(UNEQ,(EXP,((X,Y,Z),((),CLA,Y),(),SUB,Z),
(),TZE,*+2),(),CLA,ONE),(),STO,X)))))
()
```

Note that the SAP notation *+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.

```
4,1,4,2,4,3,4,4,4,5
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,LNA,LNQ
PAX,PDX,PXD
STA,STD,STO,STQ,SXD
TIX,TNX,TNZ,TRA,TSX,TXI,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 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 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 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 submitted 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 f, x, 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 NYBOL1 followed by six cards labelled LCON. Copies of the LCON deck may be found in room 26-265 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 and 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 flexowriter mode of operation (see Section 3.4) is called into control.

CRD: Control is returned from the flexowriter 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

3. The sets of triplets, f;x;a, which the apply function is to operate on.
4. A STOP card which contains the word STOP followed by a large number of right parentheses followed by the word STOP again,

   STOP)))))))STOP

   This information may be placed anywhere on the card. It is used to guarantee the safe completion of the reading program.
5. Cards such as in (2,3,4) above, repeated as often as desired.
6. A FIN card
7. Two blank cards for the card reader.

3.1.2 Operating Instructions

Tapes used:

TAPE NUMBER	USE
2	Off-line printed output (suppressed by sense switch 5 down)
4	Off-line BCD input (only if sense switch 1 up)
8	Temporary storage
9	LISP system tape

Sense switches used:

SWITCH NUMBER	USE
1	UP: On-line BCD input DOWN: Off-line BCD input (tape 4)
3	UP: Suppress on-line printing DOWN: Print on-line (see sense switch 4)
4	UP: Single space on-line printing DOWN: Double space on-line printing
5	UP: Write all printing on tape 2 for off-line printing DOWN: No off-line printing

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 26-265 and on the bulletin board in the 704 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 APPLY operator which 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 APPLY operator:

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

A2 FUNCTION OBJECT HAS NO DEFINITION -APP2-
I. e. The function has neither SUBR not EXPR on its association list, nor is it paired with something on the a-list (as a LABEL would have done).

A3 CONDITIONAL UNSATISFIED -EVCON-
I. e. none of the conditional expressions in a conditional were evaluated as true

A5 SETQ GIVEN ON A NON-EXISTENT PROGRAM VARIABLE -EVAL-
I. e. (see the Program Feature, Section 2.5) The program 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 UNBOUND VARIABLE USED -EVAL-
I. e. a free or unbound variable has neither APVAL nor APVAL1 (see Section 4.2) on its association list to signal its value, nor is the variable paired with something on the a-list.

A10 FUNCTION OBJECT HAS NO DEFINITION -EVAL-
I. e. a function does not have EXPR, FEXPR, SUBR, nor FSUBR on its association list, nor is the function paired with something on the a-list

A11 GO TO A POINT NOT LABELLED -INTER-
I. 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-
I. e. the interpreter didn't find a RETURN statement in a program using the program feature

A13 TOO MANY ARGUMENTS -SPREAD-
I. 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 expression, for example T->ERROR, then A14 is printed out followed by the argument of ERROR, if one has been assigned. It is not necessary to assign an argument, but it is possible to assign a single argument, for example

(T, (ERROR, (QUOTE, NO)))

Errors due to computer inadequacies:

B1 CUT OF PUBLIC PUSH DOWN LIST -SAVE-
I. e. (see page 144 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 of non-terminating

B2 FREE STORAGE COUNTER PANIC STOP. PRESS START TO CONTINUE
There is a free storage counter associated with the LISP system. Every time a register from free storage is used, the counter is reduced by one. The counter is not changed 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.

B3 DIVIDE IRROR - 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
I. e. (see The Compiler, Section 2.6) more than 10,000 symbols of the form G0000-G9999 have been needed during compilation. Uncommon error - impossible error (out of free storage first)

Errors in list structures:

C2 OBJECT GIVEN TO DESC AS LIST
I. e. (see desc in Section 6) in 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-

F3 2ND 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 lsit 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. one 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 FLVAL ASKED TO FIND VALUE OF NON-OBJECT

F6 FLVAL 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.
-GARBAGE COLLECTOR-

I. e. there is simply no more memory space available.

G2 TOO MUCH MARKING OF NON-LIST STRUCTURE. PANIC STOP.
-GARBAGE COLLECTOR-

This error is given after G3 below has occurred ten times. The leriency allowed here has been inserted to permit a great deal of information to be gleened 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 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.

03 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.

04 END OF FILE ON INPUT
Same as 01

05 ERROR IN READING TAPE
This is a machine error in reading either tape drive 8 or 9.

Errors during printing:

P1 PRIM ASKED TO PRINT NON-OBJECT
I. e. the printing program tried to print an item which was not an atomic symbol (object).

P2 PRIM ASKED TO PRINT UNPRINTABLE OBJECT
I. e. the printing program tried to print an atomic symbol which did not have PNAME on its association list.

Errors during read-in:

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

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

R3 OBJECT INSIDE AN INPUT LIST IS ILLEGAL -READ1-
I. e. an illegal character appeared in some atomic symbol (object).

R4 ILLEGAL CHARACTER -RDA-
The illegal characters are,

. in the wrong context	*	/	
+ " " " "	=	-	(ECD 14 punch)
- " " " "	$		

R5 END OF FILE -RDA-
The tape or card-reader ran out of cards

R6 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

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;l;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 everytime the function eval is entered, a complete printout is given of the arguments of eval and its associated a-list of pairs. The printouts proliferate in a remarkable way when TRACE is used, so that vast reams of paper are involved, and the time increases by a factor of 60 or so. Trackless 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 version of the LISP-Flexo system is described here, but is subject 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 Teager's memo, Programming Manual for the M.I.T. Flexowriter Monitor Interpreter System of Time-Shared Program Testing.

The LISP-Flexo System enables the user to read in function definitions from cards or to type them in on the Flexowriter, 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,

GETOPFLX 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

IOFLIP(READ),(,)))))))),

where the extra parentheses are used as insurance. See below for a description of the function IOFLIP. 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 cards for the card reader

In the following description of the Flexowriter system, the three symbols ↵ , → , and b will be used, where

↵	indicates carriage return
→	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-APPLY operator. The Flexowriter then types out GO and waits for an input. The system at this time expects aLISP 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),() ↵

After the carriage return, cards containing the triplets for the APPLY 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

→ bCRDbb ↵

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

---

* 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 over-writing allowed, and the Flexo-APPLY operator can be asked to operate on any set of consecutive buffers, e. g. 0 through 6, or 4 through 9.

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

→ 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 → ((A,B).C) ↵

The Flexowriter 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 number, 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,

→ RLN ↵

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 each 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 Flexowriter 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 read program as used in the TEN-Mode remembers any incomplete lists it may have been reading at the end of the previous RLN 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 previous footnote), into two consecutive registers n_i and n_{i+1},

:n_i → STOP␣

:n_{i+1} → → bFINbb␣

Then one does a RLN of these two buffer registers as described above and this will end the run completely. One can instead replace the above FIN by a CDR 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 <u>any</u> GO, into three consecutive registers n_i, n_{i+1}, and n_{i+2}

:n_i →)))))))␣ (several)

:n_{i+1} → STOP␣

:n_{i+2} → → bFINbb␣ (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,

→ ONE␣

<u>Errors</u>

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 Flexowriter 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-Mode the read program goes on to start reading a new triplet in the register following that in which the error occurred, in the Sequence-Mode the read program starts reading the next line typed in.

Interrupts

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

The functions IOFLEX (X) AND IOFLIP (Y)

The two new functions IOFLEX and IOFLIP 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:

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

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

Y = READ	flips control between reading from cards or reading from type-ins on the Flexowriter
Y = PRINT	flips control between printing on the on-line printer or printing by Flexowriter type-outs
Y = 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 Flexowriter 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 Flexowriter. The tab will not be processed, but the Flexowriter keyboard will lock. At this point, pressing the START button on the card reader will get the 704 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.

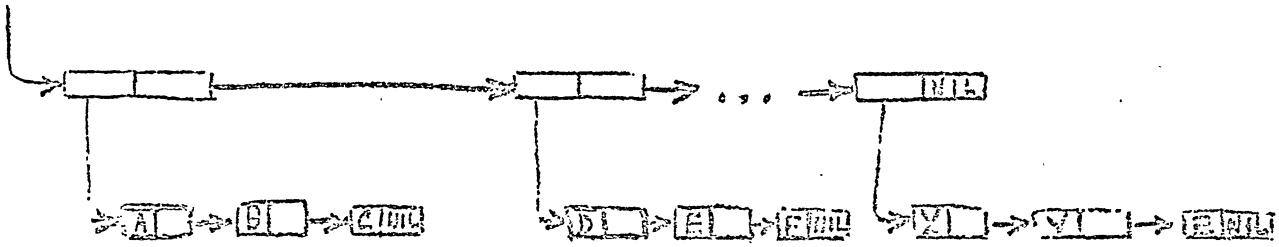
4.1 Construction of List Structures

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

Assume we are given a list of the form

$$\ell_1 = ((A,B,C),(D,E,F),\ldots,(X,Y,Z)),$$

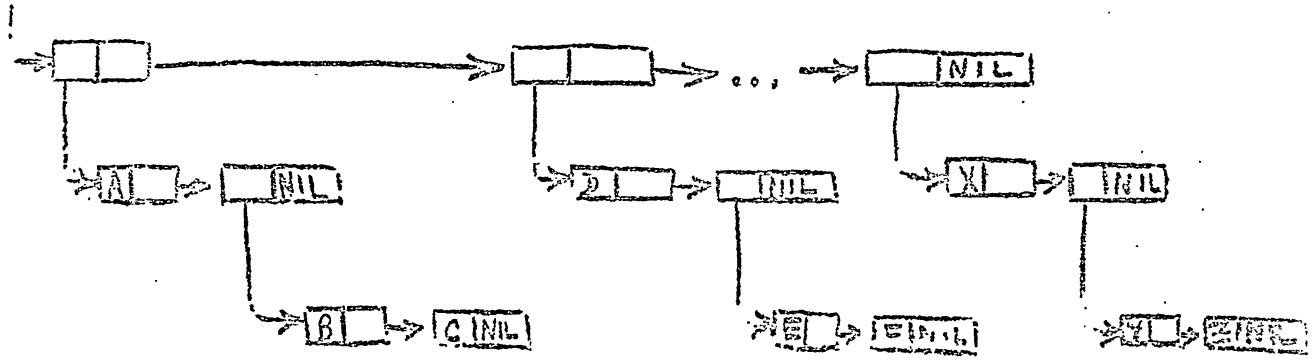
which is represented as



and that we wish to construct a list of the form

$$\ell_2 = ((A,CB,C)),(D,(E,F)),\ldots,(X,(Y,Z)))$$

which is represented as



Consider first the part (A,(B,C)) of the second list, ℓ_2. This may be constructed from A, B, and C by the operation cons[A;cons[cons[B;cons[C;NIL]];NIL]]. Another way of writing the same thing is as,

list[A;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 = car[t] = A

B = car[cdr[t]] = car[(B,C)] = B = cadr[t]

C = car[cdr[cdr[t]]] = car[cdr[(B,C)]] = car[(C)] = C caddr[t].

The first step in obtaining ℓ_2 from ℓ_1 is to define a function, grp, of three arguments which creates (X,(Y,Z)) from a list of the form (X,Y,Z)

```
define[(grp;λ[[t];cons[car[t];cons[cons[cadr[t];cons[caddr[t];
        NIL]];NIL]]])]
```

Then grp is used on the list ℓ_1 assuming ℓ_1 to be of the form given. For this purpose a new function, mltgrp, is defined recursively,

```
define[(mltgrp;λ[[ℓ];[null[ℓ]→NIL;T→cons[grp[car[ℓ]];
        mltgrp[cdr[ℓ]]]]])]
```

So mltgrp applied to the list ℓ_1 takes each threesome, (x,y,z), in turn and applies grp to it to put it in the new form, (x,(y,z)) until the list ℓ_1 has been exhausted and the new list ℓ_2 achieved.

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

4.2 Association lists

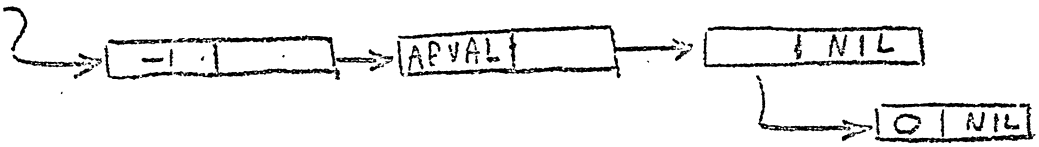
Every atomic symbol in the list of atomic symbols and every number on the list of floating-point numbers (integers are included within the list of atomic symbols) has an association list (after known locally as a "property list").

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 assigned to a floating-point number when the number is put on

the list of numbers, and to an atomic symbol when it is put on the list of atomic symbols.

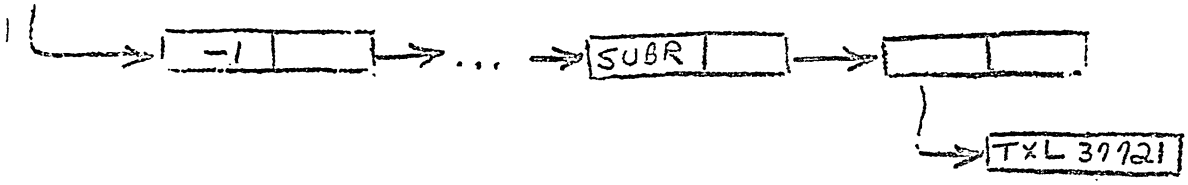
An association list has 77777 (i. e. minus 1) in the address section of the first word. The rest of the list carries other information such as possibly the print name, or the value of the integer, or an indication (FLOAT) that the list is associated with a floating-point number, etc. Two kinds of value indicators are used on association lists, APVAL and APVAL1. Both indicate a value, as in the association list for NIL

NIL

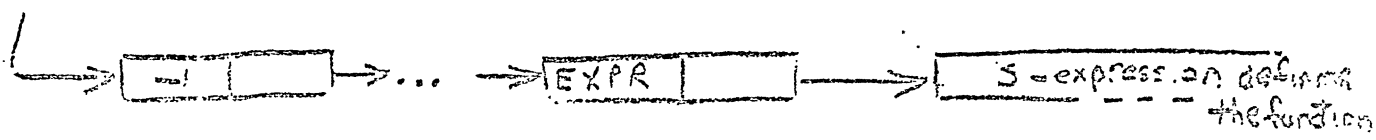


but APVAL acts like a stop to the garbage collector (see Section 4.3 below) whereas APVAL1, which is used by attrib, allows the garbage collector to look ahead through the part of list pointed to by APVAL1.

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 TXL 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 FSUBR 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 EXPR followed by the associated S-expression on the association list of a function. Compile (see Section 2.6) puts SUBR on the association list of the function compiled, and the SUBR points to the TXL 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, $\overline{\quad}$, to represent the 2's (8's) complement of an octal address.

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

$$\overline{x},,\overline{y}$$

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

location	entry	representing
24657	053120052632	$\overline{25146},,\overline{24658}$

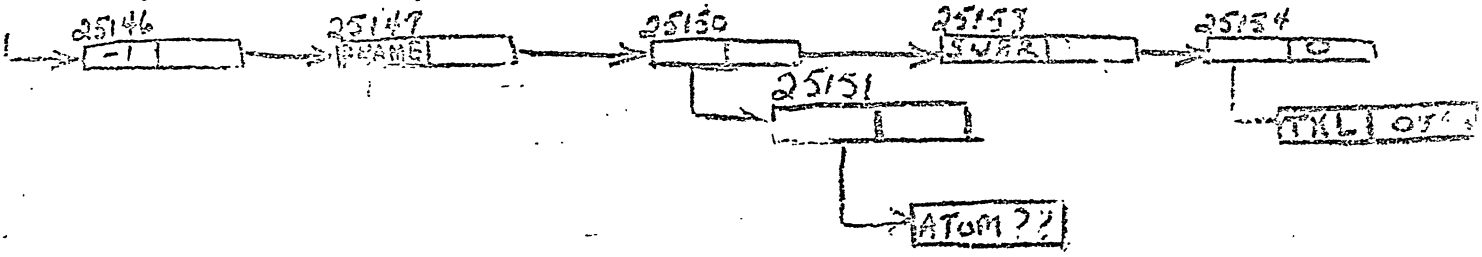
The association lists which appear in the following are

	atomic symbol	location of association list
	atom	25146
	constant 1	25201
	subr	26705
	pname	26231
(integer)	int	25665
	apval	25134

Then for atom the association list looks as follows:

location	entry		representing
25146	052631	077777	-1, ,25147
25147	052630	051547	26231, ,25150
25150	052625	052627	25151, ,25153
25151	000000	052626	25152, ,0
25152	216346	447777	BCD ATOM?? (77 ≡ ?)
25153	052624	051073	26705, ,25154
25154	000000	052623	25155, ,0
25155	700000	010762	TXL 10762 (location of SAP subroutine for atom

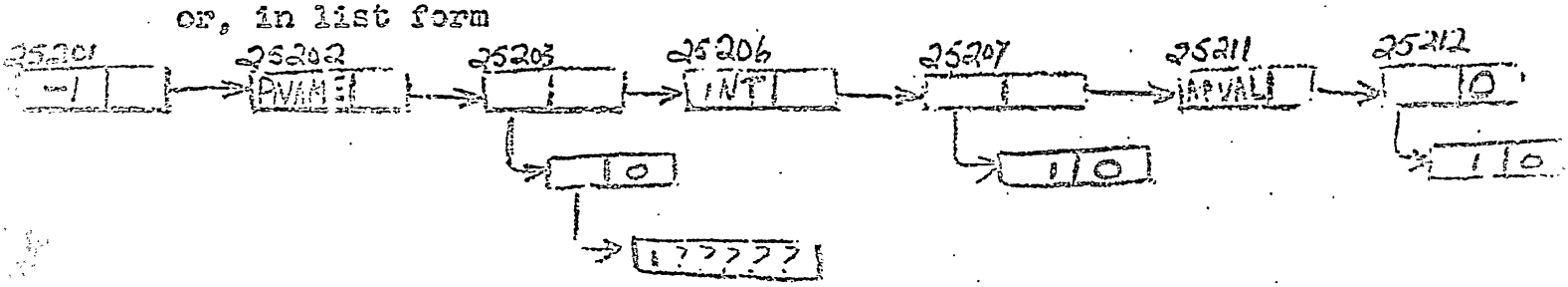
or, in list form,



For the constant 1, the association list looks as follows

location	entry		representing
25201	052576	077777	-1, ,25202
25202	052575	051547	26231, ,25203
25203	052572	052574	25204, ,25205
25204	000000	052573	25205, ,0
24205	017777	777777	BCD 1?????
25206	052571	052113	25665, ,25207
25207	052567	052570	25210, ,25211
25210	000000	000001	1, ,0
25211	052566	052644	25134, ,25212
25212	000000	052565	25213, ,0
25213	000000	000001	1, ,0

or, in list form



In this case, the constant 1 is tagged both as an integer INT 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-cdr 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-cdr 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:

and	desc	mapcon	pick	rplac(a)(d)
append	equal	maplist	print	sassoc
attrib	error	neone	printprop	search
ccns	format	not	prop	select
cond	inst	null	punchl	smplfy
copy	list	or	read	sublis
define	map	pair	remprop	subst

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

The arguments of 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 and is false (zero) or true (one) respectively.

app2[f;x;a] : machine-language

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

```
app2[f;x;a] = select[f;[CAR;caar[x]];[CDR;cdar[x]];
                       [CONS;cons[car[x];cadr[x]]];
                       [LIST;x];
```

```
search[f;λ[[j];[car[j] = SUBR∨car[j] = FSUBR]];
    λ[[j];[car[j] = SUBR → app3[cadr[j];x];
    T → apply[cadr[j];x;a];
    λ[[j];apply[car[sassoc[f;a;error]];x;a]]]]
```

app3[f;x] : machine-language

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

append[x;y] : machine-language.

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

```
append[(A,B),(C)] = (A,B,C),
append[((A)),(C,D)] = ((A),C,D)
append[x;y] = [null[x] → y;T → cons[car[x];append[cdr[x];y]]]
```

See RFSE page 133.

apply[f;x;a] : machine-language

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

```
apply[f;x;a] = [atom[f] → app2[f;x;a];
    car[f] = LAMBDA → eval[caddr[f];append[pair[cadr[f];x];a]];
    car[f] = LABEL → apply[caddr[f];x;append[pair1[cadr[f];
        caddr[f]];a]];
    car[f] = FUNARG → apply[cadr[f];x;caddr[f]];
        T → apply[eval[f;a];x;a]]
```

atom[x] : machine-language

The argument of atom is evaluated and the value of atom is true (one) or false (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 if car[x] = -1. See RFSE page 131.

attrib[x;e] : machine-language

The function 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 ~~exact~~ association list. The value of attrib is the second argument. For example

attrib[FF,(EXPR,(LAMBDA,(X),(COND,((ATOM,X),X),(T,(FF,(CAR,X)))))))]

would put EXPR followed by the LAMBDA expression for FF onto the end of the association list for FF.

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 define.

bndv[x] : λ[[x];cadr[x]]

This function is used as a function whose form may later change in compsearch.

car[x] : machine-language

See RFSE page 131.

Examples:

car[(A,B)] = A

car[((A,B))] = (A,B)

cdr[x] : machine-language

See RFSE page 131.

Examples:

cdr[(A,B)] = (B)

cdr[((A,B))] = NIL = ()

compab [x;y;z] : machine-language

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.

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 spped 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 is equivalent to (in S-expression form)

```
(FUNARG,(LAMBDA,(X,D,F,U),(PROG,(T1),
   (SETQ,T1,(GENSYM)),
   (RETURN,(DEFL,X,(SEARCHF,X,T1,
      (SUBST,T1,(CAR,(BNDV,D)),(FORM,D)),
      (SUBST,T1,(CAR,(BNDV,F)),(FORM,F))
      (FORM,U)))))))
((PRO,EXPR))
  ))
```

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 FINDNAME 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.

```
DEFINE
(FINDNAME,(LAMBDA,(L),(SEARCH,L,
      (FUNCTION,(LAMBDA,(J),(EQ,(CAR,J),(QUOTE,PNAME)))),
      (FUNCTION,(LAMBDA,(J),(CADR,J))),
      (FUNCTION,ERROR)
      )))
or
COMPSEARCH
(FINDNAME,
(LAMBDA,(J),(EQ,(CAR,J),(QUOTE,PNAME))),
(LAMBDA,(J),(CADR,J)),
(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 bound on the list of pairs (the a-list), y, or have APVAL or APVAL1 on their association lists.

Example: let x = (X,Y,Z)
and y = ((X,W),(Z,R))
and let Y have APVAL on its list pointing to the value $\tilde{Y}$, then
cons[x;y] = (W,$\tilde{Y}$,R)

cons is equivalent to

```
λ[[x;y];mapcon[x;λ[[j];eval[car[j];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.

NOT cond has a variable number of arguments, each of which is a pair

(CONT'D)

Except for its use with the compiler at least one of the propositions must be true or an error will occur.
See RFSE page 136.

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

```
label[evcond;λ[[x;a];[null[x]→error;eval[caar[x];a]→
    eval[cadar[x];a];T→evcond[cdr[x];a]]]]
```

cons[x;y] : machine language

cons[x;y] = (x·y) , See RFSE page 131

The value of cons is the (location of the) stored word.

CONST ≡ QUOTE. QUOTE should be used.

consw : machine language

The function consw (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 cons the value of consw is the (location of the) stored word.

copy[x] : machine language

This function makes a copy of the list x. The value of copy is the location of the copied list.

copy is equivalent to

```
label[copy;λ[[x];[null[x]→NIL;atom[x]→x;
    T→cons[copy[car[x]];copy[cdr[x]]]]]]
```

count : machine language

The function count 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, count 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.

cpl[x] : machine language

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

cpl is equivalent to

label[cpl;λ[[x];[null[x]→NIL;T→
cons[gcmem[cwr[car[x]];cpl[cdr[x]]]]]],

(here cwr[n] is the 36-bit contents of register n)

defl[x;e] :

The function 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 defl is x. In practice, PRO is usually paired with EXPR on the a-list.

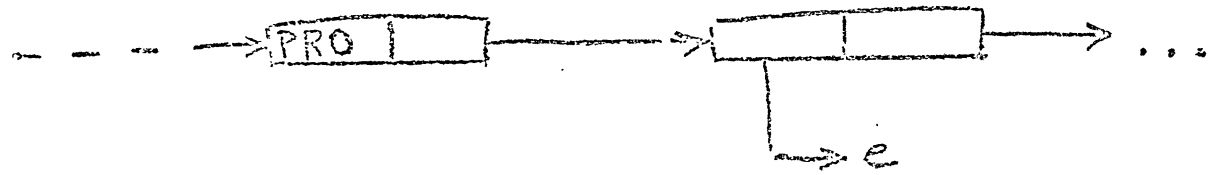
defl is equivalent to the following (where the Program Feature has been used)

λ[[x;e];S] where S is the following program with no program variables,

replaca[prop[x;PRO;function[λ[[()];cdr[attrib[x;(PRO,NIL)]]]];e]]

return[x]

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



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 λ-expression or a function. For each pair, define uses defl 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

λ[[x];deflist[x;EXPR]]

deflist[x;PRO]

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

deflist is equivalent to

λ[[x;PRO];deflisl[x]]

deflisl[x]

The argument of deflisl is a list of pairs. The function deflisl does a defl of each pair and has as value the list of the first element of each pair.

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

deflisl is equivalent to

```
label[deflisl;λ[[x];[null[x] → NIL;T →
    cons[defl[caarL;cadarL];deflisl[cdr[L]]]]]]
```

desc[x;y] : machine language

The function desc (descend) is a function of two arguments, the first of which must be a list of the atomic

symbols A (for car) and D (for cdr). The value of desc is the result of executing on the second argument the sequence of car's and 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.

desc is equivalent to

```
label[desc;λ[[x;y];[null[x]→y;atom[x]→error;
    car[x] = A→desc[cdr[x];car[y]];
    T→desc[cdr[x];cdr[y]]]]]
```

diff[y;x] : machine language

The function diff differentiates the algebraic expression y, with respect to x. The value of 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 PLUS and TIMES. See the notation and reference given under smplfy.

distrb
~~distmib~~ [e;p] : machine language

The function distrb 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+z) distributes to
(xw)(y+z+b)+(zx)(y+z+b)+(aw)(y+z+b)+(za)(y+z+b)

See the correct notation for sums and products for LISP under the function smplfy.

eq[x;y] : machine language

The function eq has the value true (one) if x = y or false (zero) if x ≠ y.

eql[x;y] : machine language

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

eql is equivalent to

label[eql;λ[[x;y];[x = y → 1;x = 0∨y = 0 → 0;
T → cwr[car[x]] = cwr[car[y]] ∧
eql[cdr[x];cdr[y]]]]]

(Here cwr n is the 36-bit contents of register n)

equal[x;y] : machine language

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

equal is equivalent to

label[equal;λ[[x;y];[x = y → 1;x = 0∨y = 0 → 0,
car[x] = -1∨car[y] = -1 → 0,
T → equal[car[x];car[y]]∧equal[cdr[x];cdr[y]]]]]

error[x] : machine language

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

eval[e;b] : machine language

The reader should have no occasion to use this function as such, since it is called in by the APPLY operator, but he might be interested in its modus operandi which is as follows,

```
eval[e;b] = [atom[e] → search[e;λ[[j];car[j] = APVAL ∨ car[j] = APVAL1];
                λ[[j];caadr[j]];
                λ[[j];search[b;λ[[j];caar[j] = e];
                          λ[[j];cadar[j]];
                          λ[[j];error]]]];
    atom[car[e]] →
      search[cdar[e];λ[[j];car[j] = FSUBR ∨ car[j] = SUBR ∨ car[j] =
      FEXPR ∨ car[j] = EXPR];
         λ[[j];select[car[j];
             [FSUBR;app3[cadr[j];list[cdr[e];b]]];
             [SUBR;app3[cadr[j];evlis[cdr[e];b]]];
             [FEXPR;apply[cadr[j];list[cdr[e];b];b]];
            apply[cadr[j];evlis[cdr[e];b];b]]]
          λ[[j];search[b;
                λ[[j];caar[j] = car[e]];
                λ[[j];apply[cadar[j];evlis[cdr[e];b];b]];
                λ[[j];error]]]];
     T → apply[car[e];evlis[cdr[e];b];b]]
```

evlis[x;b] : machine language

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

evlis is equivalent to

```
λ[[x;b];maplist[x;λ[[j];eval[car[j];b]]]]
```

expt[x;n] : machine language

The value of 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.

fixnil[x]

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

fixnil is equivalent to

```
label[fixnil;λ[[x];maplist[x;[function;λ[[j];
    [null[car[j]]→();atom[car[j]]→car[j];
    T→fixnil[car[j]]]]]]]]
```

fival[x] : machine language

The function fival finds the address of the floating-point representative of the number represented by the association list x. The value of fival is the address of the floating-point number.

The program for fival is

```
      /    car[x] ≠ -1 → error
B1         cdr[x] = 0 → error
             x  = cdr[x]
           car[x] ≠ FLOAT → go[B1]
           return[cdar[x]].
```

foxm[x] : λ[[x];caddr[x]]

This function is used as a function whose form may later change in compsearch.

format[x;f;v] :

The function format has the value x. x is an atomic symbol, f is a list structure, and v is a list of variables occurring in f. The function format causes x and the variables of v to become functions which are available to the APPLY operator. The following example of its use is taken from an earlier version of the programmer's manual.

Consider format[SHAKESPEARE;(UNDER,GREENWOOD,TREE);
(GREENWOOD,TREE)]

There are two variables involved, GREENWOOD and TREE

Then the execution of format generates three functions to which we could give arguments

shakespeare [SPREADING;CHESTNUT]
greenwood [(BENEATH,SPREADING,CHESTNUT)]
tree [(BENEATH,SPREADING, CHESTNUT)]

Executing these functions in turn gives

(UNDER,SPREADING,CHESTNUT)
SPREADING
and CHESTNUT respectively

Thus 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.

greenwood and 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.

```
format[n;f;v] = λ[[n;f;v];[λ[[s;t];attrib[n;sublis[[[y;v];[F;f];[P;
      formatp[v]]];(EXPR,(LAMBDA,✓,(SUBLIS,(LIST,P),(CONST,F))))]]
      formatq[n;f;v]]]]]
formatp[v] = [null[v]→A;T→cons[subst[car[v];X;(LIST,(CONST,X),X)];
      formatp[cdr[v]]]]
formatq[n;f;v] = [null[v]→n;T→λ[[z];[z = NO→error;T→λ[[x;y];y]
      [attrib[car[v];subst[z;R;(EXPR,((LAMBDA,(X),(DESC,R,X))))]]]
      [formatq[n;f;cdr[v]]]]][pick[car[v];f]]]
```

function[f;b] : machine language; special form

If the first element of an S-expression is FUNCTION, the second element is understood to be the function. A new list is constructed with first element FUNARG, second element equal to f, and third element equal to the current list of bound variables. I.e.

eval[(FUNCTION,f);b] = (FUNARG,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

apply[(FUNARG,f,b);x;a] = apply[f;x;b] .

function is equivalent to

λ[[f;b];list[FUNARG;car[f];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 G0001, G0002,...,G9999.

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 fifteen-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 sublis[$\tilde{x}$;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, inst[x;y;z] has as value the completed list $\tilde{x}$. The purpose of inst is to determine whether z is a substitution instance of the expression y.

See RFSE page 134.

inst is equivalent to

```
label[inst;λ[[x;y;z];[x = 0→0;y = 0∨z = 0→0;ear[y] = -1
    →search[x;λ[[j];caar[j] = y];λ[[j];cdar[j] = 0→
        maplist[x;λ[[k];[k ≠ j→car[k];T→cons[y;cons[z;0]]]]];
        cadar[j] = z→x;T→0];[y = z→x;T→0]];
        T→inst[inst[x;car[y];car[z]];cdr[y];cdr[z]]]]]
```

intv[x] : machine language;special function

The function intv finds the address of the value of the integer represented by the association list x.

intv is equivalent to

```
λ[[x];caar[prop[car[x];INTV;error]]]
```

joinst[x;y;z] : machine language

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

joinst is equivalent to

```
label[joinst;λ[[x;y;z];[z = 0→x;car[z] = 0→0;
    inst[x;car[y];caar[z]] = 0→
    joinst[x;y;cons[cdar[z];cdr[z]]];
    T→append[joinst[inst[x;car[y];caar[z]];cdr[y];cdr[z]];
        joinst[x;y;cons[cdar[z];cdr[z]]]]]]]
```

label[a;b] : machine language; special function

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

label is equivalent to

```
λ[[a;b];eval[cadr[a];append[list[a];b]]]
```

larger[x;y] : machine language

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.

larger is equivalent to

```
label[larger ;λ[[x;y];[null[x]→0;null[y]→1;
    atom[x] ∧ atom[y]→greater[x;y];
    atom[x]→0;atom[y]→1;
    larger[car[x];car[y]]→1;
    larger[car[y];car[x]]→0;
    T→larger[cdr[x];cdr[y]]]]]
```

list[x_1;x_2;...;x_n] : machine language; special function

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

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

```
λ[[x,a];maplist[[x];λ[[j];eval[car[j];a]]]]
```

See RESE page 133.

loc[x] : machine language; special function
locq[x] : machine language; special function

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:

(REPLACA,(LOCQ,X),e) ≡ (SETQ,X,e)
(REPLACA,(LOC,X),e) ≡ (SET,X,e)

makeblr x

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

((caar,(A,A)),(cadr,(D,A)),(caddr,(D,D,A)),...).

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

λ[[j];dese[WAY;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.

makeblr is equivalent to

```
funarg[λ[[j];maplist[j;function[λ[[k];defl[((car[car[k]];
    subst[car[cdr[car[j]]];(QUOTE,WAY);(LAMBDA,(J),
    (DESC,(QUOTE,WAY),J))])))]]]];(PRO,EXPR)]
```

map[x;f]

The function map is like the function maplist 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 f(x) is important.

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

```
/  LT = x
LOP  go[null[LT]→END;T→CONT]
CONT f[LT]
     LT = cdr[LT]
     go[LOP]
END return[nil]
```

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.

mapcar is equivalent to

label[mapcar;λ[[x;f];[x = 0→0;f[x]→0;mapcar[cdr[x];f]]]]

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.

mapcon is equivalent to

label[mapcon;λ[[x;f];[null[x]→NIL;T→append[f[x];mapcon[cdr[x];f]]]]]

maplist[x;f] :

The function maplist is a mapping of the list x onto a new list f[x] .

See RFSE page 138.

maplist is equivalent to

label[maplist;λ[[x;f];[null[x]→0;1→cons[f[x],maplist[cdr[x];f]]]]]

matrixmultiply[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 matrixmultiply 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

APPLY operator are in the form

x = (MATRIX,(ROW1,A11,A12,A13),(ROW2,A21,A22,A23),(ROW3,A31,A32,A33))
y = (MATRIX,(COL1,A11,A21,A31),(COL2,A12,A22,A32),(COL3,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,

> Goldberg, Solomon H., "Solution of an Electrical Network using a Digital Computer", S.M. Thesis, Course VI, MIT, August, 1959.

ncone[x;y] : machine language

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

The program for ncone has the program variable M and is as follows,

```
/     X = 0 → return[Y]
      M = X
A2    cdr[M] = 0 → go[A1]
      M = cdr[M]
      go[A2]
A1    cdr[M] = Y
\     return[X]
```

not[x] : machine language

The function not is a predicate. Its value is true (one) if its argument is false (X = zero), and false (zero) if its argument is true (X = one).

null[x] : machine language

The function null is a predicate. Its value is true (one) if its argument is zero, and false (zero) otherwise.

or[x_1;x_2;...;x_n] : machine language; special form

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.

pair[x;y] : machine language

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 RFSE page 133.

pick[x;y]

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[v;(((u,v)),w)] = (A,A,D,A)

pick is equivalent to

```
label[pick;λ[[x;y];[null[y]→NO;equal[x;y]→NIL;
    atom[y]→NO;T→λ[[j];[equal[j;NO]→
    λ[[k];[equal[k;NO]→NO;T→cons[D;k]]]
    [pick[x;cdr[y]];T→cons[A;j]]][pick[x;car[y]]]]]
```

prdct[x;y] : machine language

The function prdct computes the floating-point product of the two floating-point numbers represented on the

association lists of x and y. The value of prdct is the address of the atomic symbol containing the product.

print[x] : machine language

The function 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 print is always zero.

In the following equivalent expression, prin2 is a routine which prints up to six ECD characters when the characters are justified to the left, followed by the illegal character 77.

label[print;λ[[x];[car[x] = -1 → prinl[x];T →
[prin2[location of $7^4777777777$];print[car[x]];
[cdr[x] = 0 → prin2[location of $3^4777777777$];T →
[prin2[location of 7377777777];print[cdr[x]]]]]]]]

where 7^4_8 = ECD left-parenthesis, "("

3^4_8 = ECD right-parenthesis, ")"

73_8 = ECD comma, ","

prinl[x] : machine language

The routine prinl prints the association list of the atomic symbol x. If x is not an atom, an error occurs.

The equivalent program for prinl has the two program variables ST and VAL, and is as follows,

```
/    car x ≠ -1 → error
     ST = x
A1   cdr[x] = 0 → error
     x = cdr[x]
     car[x] = PNAME → go[A3]
     car[x] ≠ FLOAT → go[A1]
     x = car[cdr[x]]
```

```
        VAL = FLONAM[x]
        replaq[cons[PNAME;cons[VAL;cdr[ST]];ST]
        x = cdr[ST]
A3      x = car[cdr[x]]
A2      prin2[cwr[car[x]]]
        cdr[x] = 0 → return
        x = cdr[L]
        go[A2]
```

printprop[x]

The function printprop prints the properties on the association list of the atomic symbol, x. The parts of the list pointed to by any of FLO, SUBR, FSUBR, PNAME, 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;[λ[[x];[null[x] → NIL;T → prog2[print[car[x]];
    search[(FLO,SUBR,FSUBR,PNAME,APVAL,INT);
        (LAMBDA,(J),(EQUAL,(CAR,J),(CAR,X)));
        (LAMBDA,(J),(PRINTPl,(CDR,(CDR,X))));
        (LAMBDA,(J),(PRINTPl,(CDR,X)))]]]]]
```

prog[e;b] : machine language; special form

The Program Feature is discussed in Section 2.5. For interest, we include here the program which the interpreter

uses to work out a program given by the expression, e, and the a-list of pairs, b. There are five program variables, (pv is the list of program variables) t2, t3, pr, r, br

```
/       t2 = maplist[car[e];λ[[j];list[car[j];NIL]]]
        br = append[t2;b]
        t3 = pv
        pr = t2
        attrib[pv;t3]
        pr = cdr[e]
         r = pr
ML      go[[atom[car[r]]→ADV;caar[r] = GO→GO;
          caar[r] = RETURN→RET;T→NORM]
NORM    eval[car[r];br]
ADV     r = cdr[r]
        go[ML]
RET     return[eval[cadar[r];br]]
GO      t2 = eval[cadar[r];br]
        search[pr;λ[[j];car[j] = t2;λ[[j];r = cdr[j]];
          λ[[j];error]]
\       go[ML]
```

prop[x;y;u] : machine language

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

prop is equivalent to

```
label[prop;λ[[x;y;z];[null[x]→u;car[x] = y→cdr[x];
          T→prop[cdr[x];y;z]]]]
```

punchdef[x] :

The function punchdef writes the definitions of the functions named in the list x out on tape 3 in BCD for off-line punching.

punchdef is equivalent to

```
label[punchdef;λ[[x];map[x;λ[[j];punch8[list[car[j];
    fixnil[car[prop[car[j];EXPR;
    λ[[(nil)];error[conc[(NO,EXPR,ON,PROPERTY,LIST,OF);
    list[car[j]];(()),PUNCHDEF,CANNOT, CONTINUE]]]]]]]]]]
```

punch8[x] : machine language

The function punch8 writes the list, x, out onto tape 3 in BCD form foroff-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.

punchs[x] : machine language

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

quote : machine language; special form

The value of a list beginning with QUOTE is always the rest of the list.

rd : machine language

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

read : machine language

The function read of no arguments reads one list from cards or type, depending on the sense switch settings. The value of read is the list it has read.

reduce[m] : machine language

The function reduce reduces an n×n matrix to an (n-1)×(n-1) matrix. The function has been used in electrical network reduction. See the references under reducetonxn below.

reducetonxn[m;n] : machine language

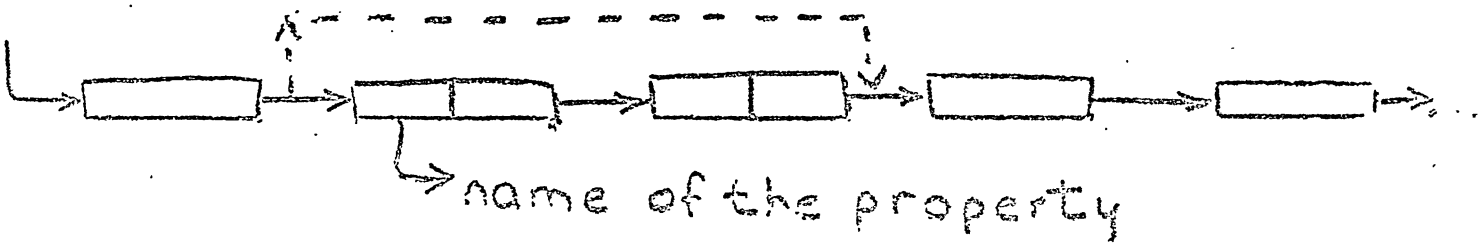
This function uses the function reduce to reduce a square matrix m to an n by n matrix whose rank is less than that of m. The argument n must be given in floating-point form. This function has been used in electrical network reduction, see

Edwards, D. J., "Symbolic Circuit Analysis with the 704 Electronic Calculator", S. B. Thesis, Course VI, MIT, June 1959.

Goldberg, Solomon H., "Solution of an Electrical Network using a Digital Computer", S. M. Thesis, Course VI, MIT, August, 1959.

remprop[x,p] :

The function 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,



The value of remprop is NIL.

remprop is equivalent to

$$\text{A}[[x;p];\text{rempl}[x]] \quad ,$$

where

$$\text{rempl}[x] = [\text{null}[x] \wedge \text{null}[\text{cdr}[x]] \rightarrow \text{NIL}; \text{cadr}[x] = p \rightarrow \text{prog2}[\text{replacd}[x;[\text{null}[\text{cddr}[x]] \rightarrow \text{NIL}; T \rightarrow \text{cddr}[x]]]; \text{rempl}[x]]; T \rightarrow \text{rempl}[\text{cdr}[x]]] \quad ,$$

where prog2[u;v] has the value v, though u is used to effect an action.

rplaca[x;y]
rplacd[x;y] } machine language

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

sassoc[x;y;u] : machine language

The function 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 sassoc is the sublist with the first element removed. Otherwise the value is u.

sassoc is equivalent to

$$\text{label}[\text{sassoc};\text{A}[[x;y;u];[\text{null}[y] \rightarrow u; \text{caar}[y] = x \rightarrow \text{cdar}[y]; T \rightarrow \text{sassoc}[x;\text{cdr}[y];u]]]] \quad .$$

search[x;p;f;u] :

The function 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 search. If there is no such element, the function u[x] is taken as the value of search.

search is equivalent to

label[search;λ[[x;p;f;u];[null[x]→u[x];p[x]→f[x];
T→search[cdr[x];p;f;u]]]]

searchf[x;v;p;f;u]

See the function compsearch; the function searchf is the fast search whose S-expression is set up by compsearch.

searchf is equivalent to

λ[[x;v;p;f;u];sublis[((NAME,x),(VAR,v),(PF,p),(FF.f),(UF,u));
(LAMBDA,(VAR),(COND,((NULL,VAR),UF),(PF,FF),
(T,(NAME,(CDR,VAR))))))]]]

select[x;a] : special form

The function select can be considered to have an argument x of the form

$$(\xi;p_1\rightarrow e_1;p_2\rightarrow e_2;\ldots;p_n\rightarrow e_n;T\rightarrow u)$$

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

$$p_i = \xi,$$

and in that case the value of select is e_i. If no such p_i is found, the value of select is u.

select[x;a] is equivalent to

λ[[x;a];select2[eval[car[x];a];cdr[x]]]

(eval[car[x];a] here will give the value of the ξ above),

where

select2[v;w] = [null[cdr[w]→eval[car[w];a];
v = eval[caar[w];a]→eval[cadar[w];a];
T→select2[v;cdr[w]]]

set[x;y] : machine language
setq[x;y] : machine language ; special form

The set and 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),

(SET,(QUOTE,V),e) ≡ (SETQ,V,e)

smplfy[x] : machine language

The function smplfy takes the algebraic expression x and simplifies it. The resultant expression is the value of smplfy.

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

(TIMES,A,B,C); (any number (>1) of arguments)
(SUM,A,B,C) ; (any number (>1) of arguments)
(COS,X)
(SIN,X)
(MINUS,X)
(POWER,X,Y) ≡ X^Y
(RECIP;X) ≡ $\frac{1}{X}$
(LOG,B,X) ≡ $\log_B X$

For example,

(a)(b)($\frac{1}{c}$)-d^e is written as

(PLUS,(TIMES,A,B,(RECIP,C)),(MINUS,(POWER,D,E)))

For further discussion of the smplfy function, see Goldberg, Solomon H., "Solution of an Electrical Network using a Digital Computer", S.M. Thesis, Course VI, MIT, August, 1959.

speak : machine language

The function speak, of no arguments, causes the contents of the CONS 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

Here 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

```
label[sublis;λ[[x;y];[x = 0→y;y = 0→0;T→
    search[x;function[λ[[j];equal[y;caar[j]]]];
        function[λ[[j];cadar[j]]];
    [car[y] = -1→y;T→cons[sublis[x;car[y]];sublis[x;cdr[y]]]]]]]
```

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

```
label[subst;λ[[x;y;z];[y = z→copy[x];car[z] = -1→z;
    T→cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]]]
```

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

```
label[substr;λ[[x;y];[x = y -> 1;null[x] -> 0;atom[x] -> 0
    substr[car[x];y] -> 1;T -> substr[cdr[x];y]]]]
```

sum[x;y] : machine language

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

tsflot[x] : machine language

The proposition tsflot 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.

tsflot is equivalent to

```
/   car x  = -1    return (0)
B1  x = 0    return (0)
    car x  = FLOAT    return (1)
    x = cdr x
\   go B1
```

unsount : machine language

The function unsount, 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 IOFLIP(READ) below had cards in it defining the function RVSE (reverse a list), followed by a card IOFLIP(READ) to return to control to the Flexowriter.

```
        FLX                          the Flexowriter takes over, and
 GO                                  requests a type-in (Sequence Mode)
CDR  ((A,B,C)) ()                    (type-in)
 STOP                                the Flexowriter digests the information

 (B,C)                               and finds the answer,
 GO                                  and asks for more.
CDR  ((A,B,C))                       (type-in)
 STOP

 GO                                  the Flexowriter does not have a full
()                                   triplet f;x;a ,so asks for the rest
 STOP

 (B,C)                               answer
 GO
    TEN                              (the TEN-Mode is entered)
0   CAR  (((A,B),C)) ()
:1  CDR  ((D,(E,F))) ()              } (type-ins)
:2  CONS  ((G,H),                    }
:3  (I,J)) ()                        }
```

```
: RLN
01                                  (read lines 0 and 1)
0                                   line 0

 (A,B)                              answer
1                                   line 1

 ((E,F))                            answer
 GO
     RLN                            (read line 2)
22
2                                   line 2
 GO
     RLN                            the Flexowriter asks for the rest
32                                  (read lines 3 to 2)
 WRONG ORDER                        error
     RLN
33                                  (read line 3)-line 2 has been re-
3                                   membered

 ((G,H),I,J)                        answer
 GO
     TEN
CAR  ((A B)) ()
 LINE NUMBER MISSING RETYPE         error indication
2    IOFLIP(READ) ()
:3   RVSE  ((A,B,C,D))  ()          type-ins
4    IOFLEX(PRINT) ()
:5   IOFLEX(NG) ()
:6   STOP
:7         FIN
:    RLN                            (read lines 2 through 5)
25
2
```

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

(RVSE,REV1)

READ                          flip back to flexowriter
3                             read line 3 and find

(D,C,B,A)                     answer to RVSE
4                             read line 4

T                             True-the Flexowriter is printing
5                             read line 5

ERROR NUMBER :F 1:            NG is not a valid argument
                              for IOFLEX, so
NG

F                             answer is False
GO
    RLN                       (read line 6 and 7)
67
67                            end of computation

          FIN
```