Tim Hart

LISP I

PROGRAMMER'S MANUAL

March 1, 1960

Artificial Intelligence Group

J. McCarthy
R. Brayton
D. Edwards
P. Fox
L. Hodes
D. Luckham
K. Maling
D. Park
S. Russell

COMPUTATION CENTER
and
RESEARCH LABORATORY OF ELECTRONICS
Massachusetts Institute of Technology
Cambridge, Massachusetts

Preface

LISP I is a programming system for the IBM 704 for computing with symbolic expressions. It has been used for symbolic calculations in differential and integral calculus, electric circuit theory, mathematical logic, and artificial intelligence.

This manual contains a full description of the features of LISP I as of March 1960. The system has a central core based on a class of recursive functions of symbolic expressions which should be studied first and if possible used before the more peripheral features are tried. This core is described in Chapters 2 and 3, and LISP programs can be written and run using this core provided someone familiar with the operational aspects of the system i.e. loaders, tapes etc. is available. Later, the advanced features will be found useful although they are less neat, and less carefully described.

This manual applies also to a version of LISP I being prepared for the IBM 709.

Contents

1. Introduction

The current basic LISP system uses about 12,000 of the 32,000 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 S-expressions (see Chapter 2) for the interpreter part of the system. A list of available functions is included in the manual, and other functions may be defined by the user. Chapters 2 and 3 of the manual should give the user enough information to enable him to use the basic system.

Enlargements of the basic system are available for various purposes. The compiler version of the LISP system can be used to compile S-expressions into machine code. Values of compiled functions are computed about 60 times faster than the S-expressions for the functions could be interpreted and evaluated. The LISP-compiler system uses about half of the 32,000 memory.

For on-line operation using M.I.T's Flexowriter-704 system, the basic LISP system can be augmented by a special package of routines to control the input-output aspects of the Flexowriter. The Flexowriter version of LISP I does not contain the compiler option.

Descriptions of the various features of the LISP system 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.

2. Recursive Functions of Symbolic Expressions[1,2]

The LISP I programming system is based on a class of functions of symbolic expressions which we now proceed to describe.

2.1 Functions and Function Definitions

We shall need a number of mathematical ideas and notations concerning functions in general. Most of the ideas are well known but the notion of conditional expression is believed to be new, and the use of conditional expressions permits functions to be defined recursively in a new and convenient way.

a. Partial Functions

A partial function is a function that is defined only on part of its domain. Partial functions necessarily arise when functions are defined by computations because for some values of the arguments, the computation defining the value of the function may not terminate. However, some of our elementary functions will be defined as partial functions.

---

1 This chapter is taken from the Quarterly Progress Report No. 53, Research Laboratory of Electronics, M.I.T., April 15, 1959: "Recursive Functions of Symbolic Expressions and Their Computation by Machine" by John McCarthy.

2 An article on the same subject by McCarthy is to appear in the April 1960 issue of the Communications of the Association for Computing Machinery.

b. Propositional Expressions and Predicates

A propositional expression is an expression whose possible values are T (for truth) and F (for falsity). We shall assume that the reader is familiar with the propositional connectives $\wedge$ ("and"), $\vee$ ("or"), and $\sim$("not"). Typical propositional expressions are

$x < y$

$(x<y) \wedge (b=c)$

x is prime

A predicate is a function whose range consists of the truth values T and F.

c. Conditional Expressions

The dependence of truth values on the values of quantities of other kinds is expressed in mathematics by predicates, and the dependence of truth values on other truth values by logical connectives. However, the notations for expressing symbolically the dependence of quantities of other kinds on truth values is inadequate, so that English words and phrases are generally used for expressing these dependences in texts that describe other dependences symbolically. For example, the function $|x|$ is usually defined in words.

Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

$$(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n)$$

where the p's are propositional expressions and the e's are expressions of any kind. It may be read, "If p_1 then e_1, otherwise if p_2 then e_2,..., otherwise if p_n then e_n," or "p_1 yields e_1,...,p_n yields e_n."

We now give the rules for determining whether the value of $(p_1 \rightarrow e_1,\ldots,p_n \rightarrow e_n)$ is defined, and if so what its value is. Examine the p's from left to right. If a p whose value is T is encountered before any p whose value is undefined is encountered, then the value of the conditional expression is the value of the corresponding e (if this is defined). If any undefined p is encountered before a true p, or if all p's are false, or if the e corresponding to the first true p is undefined, then the value of the conditional expression is undefined. We now give examples.

$(1<2 \rightarrow 4, 1>2 \rightarrow 3) = 4$
$(2<1 \rightarrow 4, 2>1 \rightarrow 3, 2>1 \rightarrow 2) = 3$
$(2<1 \rightarrow 4, T \rightarrow 3) = 3$
$(2<1 \rightarrow \frac{0}{0}, T \rightarrow 3) = 3$
$(2<1 \rightarrow 3, T \rightarrow \frac{0}{0})$ is undefined
$(2<1 \rightarrow 3, 4<1 \rightarrow 4)$ is undefined

Some of the simplest applications of conditional expressions are in giving such definitions as

$|x| = (x<0 \rightarrow -x, T \rightarrow x)$
$\delta_{ij} = (i=j \rightarrow 1, T \rightarrow 0)$
$\mathrm{sgn}(x) = (x<0 \rightarrow -1, x=0 \rightarrow 0, T \rightarrow 1)$

d. Recursive Function Definitions

By using conditional expressions we can, without circularity, define functions by formulas in which the defined function occurs. For example, we write

$n! = (n=0 \rightarrow 1, T \rightarrow n\cdot(n-1)!)$

When we use this formula to evaluate 0! we get the answer 1; because of the way in which the value of a conditional

expression was defined, the meaningless expression $0\cdot(0-1)!$ does not arise. The evaluation of $2!$ according to this definition proceeds as follows:

$$\begin{aligned} 2! &= (2=0 \rightarrow 1, T \rightarrow 2\cdot(2-1)!) \\ &= 2\cdot 1! \\ &= 2\cdot(1=0 \rightarrow 1, T \rightarrow 1\cdot(1-1)!) \\ &= 2\cdot 1\cdot 0! \\ &= 2\cdot 1\cdot(0=0 \rightarrow 1, T \rightarrow 0\cdot(0-1)!) \\ &= 2\cdot 1\cdot 1 \\ &= 2 \end{aligned}$$

We now give two other applications of recursive function definitions. The greatest common divisor, gcd(m,n), of two positive integers m and n is computed by means of the Euclidean algorithm. This algorithm is expressed by the recursive function definition:

$$\gcd(m,n)=(m>n \rightarrow \gcd(n,m), \mathrm{rem}(n,m)=0 \rightarrow m, T \rightarrow \gcd(\mathrm{rem}(n,m),m))$$

where rem(n,m) denotes the remainder left when n is divided by m.

The Newtonian algorithm for obtaining an approximate square root of a number a, starting with an initial approximation x and requiring that an acceptable approximation y satisfy $|y^2-a|<\epsilon$, may be written as

$$\mathrm{sqrt}(a,x,\epsilon) = (|x^2-a|<\epsilon \rightarrow x, T \rightarrow \mathrm{sqrt}(a,\tfrac{1}{2}(x+\tfrac{a}{x}),\epsilon))$$

The simultaneous recursive definition of several functions is also possible, and we shall use such definitions if they are required.

There is no guarantee that the computation determined by a recursive definition will ever terminate and, for example, an attempt to compute n! from our definition will only succeed if n is a non-negative integer. If the computation does not terminate, the function must be regarded as undefined for the given arguments.

The propositional connectives themselves can be defined by conditional expressions. We write

$$p \wedge q = (p \rightarrow q, T \rightarrow F)$$
$$p \vee q = (p \rightarrow T, T \rightarrow q)$$
$$\sim p = (p \rightarrow F, T \rightarrow T)$$
$$p \supset q = (p \rightarrow q, T \rightarrow T)$$

It is readily seen that the right-hand sides of the equations have the correct truth tables. If we consider situations in which p or q may be undefined, the connectives $\wedge$ and $\vee$ are seen to be noncommutative. For example, if p is false and q is undefined, we see that according to the definitions given above $p \wedge q$ is false, but $q \wedge p$ is undefined. For our applications this noncommutativity is desirable, since $p \wedge q$ is computed by first computing p, and if p is false q is not computed. If the computation for p does not terminate, we never get around to computing q. We shall use propositional connectives in this sense hereafter.

e. Functions and Forms

It is usual in mathematics - outside of mathematical logic - to use the word "function" imprecisely and to apply it to forms such as y^2+x. Because we shall later compute with expressions for functions, we need a distinction between functions and forms and a notation for expressing this distinction. This distinction and a notation for describing it, from which we deviate trivially is given by Church.[1]

---

[1] A. Church, The Calculi of Lambda-Conversion (Princeton University Press, Princeton, N.J., 1941).

Let f be an expression that stands for a function of two integer variables. It should make sense to write f(3,4) and the value of this expression should be determined. The expression y^2+x does not meet this requirement; $y^2+x(3,4)$ is not a conventional notation, and if we attempted to define it we would be uncertain whether its value would turn out to be 13 or 19. Church calls an expression like y^2+x a form. A form can be converted into a function if we can determine the correspondence between the variables occurring in the form and the ordered list of arguments of the desired function. This is accomplished by Church's λ-notation.

If $\mathcal{E}$ is a form in variables $x_1,\ldots,x_n$, then $\lambda((x_1,\ldots,x_n),\mathcal{E})$ will be taken to be the function of n variables whose value is determined by substituting the arguments for the variables $x_1,\ldots,x_n$ in that order in $\mathcal{E}$ and evaluating the resulting expression. For example, $\lambda((x,y),y^2+x)$ is a function of two variables, and $\lambda((x,y),y^2+x)(3,4) = 19$.

The variables occurring in the list of variables of a λ-expression are dummy or bound, like variables of integration in a definite integral. That is, we may change the names of the bound variables in a function expression without changing the value of the expression, provided that we make the same change for each occurrence of the variable and do not make two variables the same that previously were different. Thus $\lambda((x,y),y^2+x)$, $\lambda((u,v),v^2+u)$ and $\lambda((y,x),x^2+y)$ denote the same function.

We shall frequently use expressions in which some of the variables are bound by λ's and others are not. Such an expression may be regarded as defining a function with parameters. The unbound variables are called free variables.

An adequate notation that distinguishes functions from forms allows an unambiguous treatment of functions of functions. It would involve too much of a digression to give examples here,

but we shall use functions with functions as arguments later in this manual.

Difficulties arise in combining functions described by λ-expressions, or by any other notation involving variables, because different bound variables may be represented by the same symbol. This is called collision of bound variables. There is a notation involving operators that are called combinators for combining functions without the use of variables. Unfortunately, the combinatory expressions for interesting combinations of functions tend to be lengthy and unreadable.

f. Expressions for Recursive Functions

The λ-notation is inadequate for naming functions defined recursively. For example, using λ's, we can convert the definition

$$sqrt(a,x,\epsilon) = (|x^2-a|<\epsilon \rightarrow x, T \rightarrow sqrt(a,\tfrac{1}{2}(x+\tfrac{a}{x}),\epsilon))$$

into

$$sqrt = \lambda((a,x,\epsilon),(|x^2-a|<\epsilon \rightarrow x, T \rightarrow sqrt(a,\tfrac{1}{2}(x+\tfrac{a}{x}),\epsilon)))$$

but the right-hand side cannot serve as an expression for the function because there would be nothing to indicate that the reference to sqrt within the expression stood for the expression as a whole.

In order to be able to write expressions for recursive functions, we introduce another notation: label(a,ℰ) denotes the expression ℰ, provided that occurrences of a within ℰ are to be interpreted as referring to the expression as a whole. Thus we can write

$$label\{sqrt,\lambda[(a,x,\epsilon),(|x^2-a|<\epsilon \rightarrow x, T \rightarrow sqrt(a,\tfrac{1}{2}(x+\tfrac{a}{x}),\epsilon))]\}$$

as a name for our sqrt function.

The symbol a in label(a,ℰ) is also bound, that is, it may be altered systematically without changing the meaning of the expression. It behaves differently from a variable bound by a λ, however.

2.2 Symbolic Expressions

We shall first define a class of symbolic expressions in terms of ordered pairs and lists. Then we shall define five elementary functions and predicates, and build from them by composition, conditional expressions, and recursive definitions an extensive class of functions of which we shall give a number of examples. We shall then show how these functions can be expressed as symbolic expressions, and we shall define a universal function apply that allows us to compute from the expression for a given function its value for given arguments. Finally, we shall define some functions with functions as arguments and give some useful examples.

a. A Class of Symbolic Expressions

We shall now define the S-expressions (S stands for symbolic). They are formed by using the special characters

.
)
(

and an infinite set of distinguishable atomic symbols. For atomic symbols, we shall use strings of capital Latin letters and digits. Examples of atomic symbols are

A
ABA
APPLEPIENUMBER3

There is a twofold reason for departing from the usual mathematical practice of using single letters for atomic symbols. First, computer programs frequently require hundreds of distinguishable symbols that must be formed from the 47 characters that are printable by the IBM 704 computer. Second, it is convenient to allow English words and phrases to stand for atomic entities for mnemonic reasons. The symbols are atomic in the sense that any substructure they may have as sequences of characters is ignored. We assume only that different symbols can be distinguished.

S-expressions are then defined as follows:

1. Atomic symbols are S-expressions.

2. If e_1 and e_2 are S-expressions, so is $(e_1 \cdot e_2)$.

Examples of S-expressions are

AB

(A·B)

((AB·C)·D)

An S-expression is then simply an ordered pair, the terms of which may be atomic symbols or simpler S-expressions. We can represent a list of arbitrary length in terms of S-expressions as follows. The list

$$(m_1, m_2, \ldots, m_n)$$

is represented by the S-expression

$$(m_1 \cdot (m_2 \cdot (\ldots (m_n \cdot \text{NIL}) \ldots)))$$

Here NIL is an atomic symbol used to terminate lists.

Since many of the symbolic expressions with which we deal are conveniently expressed as lists, we shall introduce a list notation to abbreviate certain S-expressions. We have

1. (m) stands for $(m \cdot \text{NIL})$.

2. $(m_1, \ldots, m_n)$ stands for $(m_1 \cdot (\ldots (m_n \cdot \text{NIL}) \ldots))$.

3. $(m_1, \ldots, m_n \cdot x)$ stands for $(m_1 \cdot (\ldots (m_n \cdot x) \ldots))$.

Subexpressions can be similarly abbreviated. Some examples of these abbreviations are

((AB,C),D) for ((AB·(C·NIL))·(D·NIL))
((A,B),C,D·E) for ((A·(B·NIL))·(C·(D·E)))

Since we regard the expressions with commas as abbreviations for those not involving commas, we shall refer to them all as S-expressions.

b. Functions of S-expressions and the Expressions That Represent Them

We now define a class of functions of S-expressions. The expressions representing these functions are written in a conventional functional notation. However, in order to clearly distinguish the expressions representing functions from S-expressions, we shall use sequences of lower-case letters for function names and variables ranging over the set of S-expressions. We also use brackets and semicolons, instead of parentheses and commas, for denoting the application of functions to their arguments. Thus we write

car[x]
car[cons[(A·B);x]]

In these M-expressions (meta-expressions) any S-expressions that occur stand for themselves.

c. The Elementary S-functions and Predicates

We introduce the following functions and predicates:

1. atom

atom[x] has the value of T or F, accordingly as x is an atomic symbol or not. Thus

atom[X] = T

atom[(X·A)] = F

2. eq

eq[x;y] is defined if and only if either x or y is atomic. eq[x;y] = T if x and y are the same symbol, and eq[x;y] = F otherwise. Thus

eq[X;X] = T

eq[X;A] = F

eq[X;(X·A)] = F

3. car

car[x] is defined if and only if x is not atomic. car[$(e_1 \cdot e_2)$] = e_1. Thus

car[X] is undefined.

car[(X·A)] = X

car[((X·A)·Y)] = (X·A)

4. cdr

cdr[x] is also defined when x is not atomic. We have cdr[$(e_1 \cdot e_2)$] = e_2. Thus

cdr[X] is undefined.

cdr[(X·A)] = A

cdr[((X·A)·Y)] = Y

5. cons

cons[x;y] is defined for any x and y. We have cons[e_1;e_2] = $(e_1 \cdot e_2)$. Thus

cons[X;A] = (X·A)

cons[(X·A);Y] = ((X·A)·Y)

car, cdr and cons are easily seen to satisfy the relations

car[cons[x;y]] = x

cdr[cons[x;y]] = y

cons[car[x];cdr[x]] = x, provided that x is not atomic.

The names "car" and "cons" will come to have mnemonic significance only when we discuss the representation of the system in the computer. Compositions of car and cdr give the subexpressions of a given expression in a given position. Compositions of cons form expressions of a given structure out of parts. The class of functions which can be formed in this way is quite limited and not very interesting.

d. Recursive S-functions

We get a much larger class of functions (in fact, all computable functions) when we allow ourselves to form new functions of S-expressions by conditional expressions and recursive definition.

We now give some examples of functions that are definable in this way.

1. ff[x]

The value of ff[x] is the first atomic symbol of the S-expression x with the parentheses ignored. Thus

ff[((A·B)·C)] = A

We have

ff[x] = [atom[x] → x;T → ff[car[x]]]

We now trace in detail the steps in the evaluation of ff[((A·B)·C)]:

ff[((A·B)·C)] = [atom[((A·B)·C)] → ((A·B)·C);T → ff[car[((A·B)·C)]]]
= [F → ((A·B).C);T → ff[car[((A·B)·C)]]]
= [T → ff[car[((A·B)·C)]]]
= ff[car[((A·B)·C)]]
= ff[(A·B)]
= [atom[(A·B)] → (A·B);T → ff[car[(A·B)]]]
= [F → (A·B);T → ff[car[(A·B)]]]
= [T → ff[car[(A·B)]]]

$= ff[car[(A\cdot B)]]$

$= ff[A]$

$= [atom[A] \rightarrow A;T \rightarrow ff[car[A]]]$

$= [T \rightarrow A;T \rightarrow ff[car[A]]]$

$= A$

2. subst[x;y;z]

This function gives the result of substituting the S-expression x for all occurrences of the atomic symbol y in the S-expression z. It is defined by

$$subst[x;y;z] = [atom[z] \rightarrow [eq[z;y] \rightarrow x;T \rightarrow z];T \rightarrow cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]$$

As an example, we have

$$subst[(X\cdot A);B;((A\cdot B)\cdot C)] = ((A\cdot(X\cdot A))\cdot C)$$

3. equal[x;y]

This is a predicate that has the value T if x and y are the same S-expression, and has the value F otherwise. We have

$$equal[x;y] = [atom[x]\wedge atom[y]\wedge eq[x;y]]\vee[\sim atom[x]\wedge\sim atom[y]\wedge equal[car[x];car[y]]\wedge equal[cdr[x];cdr[y]]]$$

It is convenient to see how the elementary functions look in the abbreviated list notation. The reader will easily verify that

(i) $car[(m_1,m_2,\ldots,m_n)] = m_1$

(ii) $cdr[(m_1,m_2,\ldots,m_n)] = (m_2,\ldots,m_n)$

(iii) $cdr[(m)] = NIL$

(iv) $cons[m_1;(m_2,\ldots,m_n)] = (m_1,m_2,\ldots,m_n)$

(v) $cons[m;NIL] = (m)$

We define

$$null[x] = atom[x]\wedge eq[x;NIL]$$

This predicate is useful in dealing with lists.

Compositions of car and cdr arise so frequently that many expressions can be written more concisely if we abbreviate

cadr[x] for car[cdr[x]],

caddr[x] for car[cdr[cdr[x]]], etc.

Another useful abbreviation is to write list[e_1;e_2;...;e_n] for cons[e_1;cons[e_2;...;cons[e_n;NIL]...]]. This function gives the list, (e_1,...,e_n), as a function of its elements.

The following functions are useful when S-expressions are regarded as lists.

1. append[x;y]

append[x;y] = [null[x] → y;T → cons[car[x];append[cdr[x];y]]]

An example is

append[(A,B);(C,D,E)] = (A,B,C,D,E)

2. among[x;y]

This predicate is true if the S-expression x occurs among the elements of the list y. We have

among[x;y] = ~null[y]∧[equal[x;car[y]]∨among[x;cdr[y]]]

3. pair[x;y]

This function gives the list of pairs of corresponding elements of the lists x and y. We have

pair[x;y] = [null[x]∧null[y] → NIL;~atom[x]∧~atom[y] → cons[
list[car[x];car[y]];pair[cdr[x];cdr[y]]]]

An example is

pair[(A,B,C);(X,(Y,Z),U)] = ((A,X),(B,(Y,Z)),(C,U))

4. assoc[x;y]

If y is a list of the form ((u_1,v_1),...,(u_n,v_n)) and x is one of the u's then assoc[x;y] is the corresponding v. We have

assoc[x;y] = [caar[y] = x → cadar[y];T → assoc[x;cdr[y]]]

An example is

assoc[X;((W,(A,B)),(X,(C,D)),(Y,(E,F)))] = (C,D)

5. sublis[x;y]

Here x is assumed to have the form of a list of pairs

$((u_1,v_1),\ldots,(u_n,v_n))$, where the u's are atomic, and y may be any S-expression. The value of sublis[x;y] is the result of substituting each v for the corresponding u in y. In order to define sublis, we first define an auxiliary function. We have

```
sub2[x;z] = [null[x] → z;eq[caar[x];z] → cadar[x];T → sub2[
            cdr[x];z]]
```

and

```
sublis[x;y] = [atom[y] → sub2[x;y];T → cons[sublis[x;car[
              y]];sublis[x;cdr[y]]]]
```

We have

```
sublis[((X,(A,B)),(Y,(B,C)));(A,X·Y)] = (A,(A,B),B,C)
```

---

e. Representation of S-Functions by S-Expressions

S-functions have been described by M-expressions. We now give a rule for translating M-expressions into S-expressions, in order to be able to use S-functions for making certain computations with S-functions and for answering certain questions about S-functions.

The translation is determined by the following rules in which we denote the translation of an M-expression $\mathcal{E}$ by $\mathcal{E}^*$.

1. If $\mathcal{E}$ is an S-expression $\mathcal{E}^*$ is (QUOTE, $\mathcal{E}$).

2. Variables and function names that were represented by strings of lower-case letters are translated to the corresponding strings of the corresponding upper-case letters. Thus car* is CAR, and subst* is SUBST.

3. A form $f[e_1;\ldots;e_n]$ is translated to $(f^*,e_1^*,\ldots,e_n^*)$. Thus $\{cons[car[x];cdr[x]]\}^*$ is (CONS,(CAR,X),(CDR,X)).

4. $\{[p_1 \rightarrow e_1;\ldots;p_n]\}^*$ is $(COND,(p_1^*,e_1^*),\ldots,(p_n^*,e_n^*))$.

5. $\{\lambda[[x_1;\ldots;x_n];\mathcal{E}]\}^*$ is $(LAMBDA,(x_1^*,\ldots,x_n^*),\mathcal{E}^*)$.

6. $\{label[a;\mathcal{E}]\}^*$ is $(LABEL,a^*,\mathcal{E}^*)$.

With these conventions the substitution function whose M-expression is label[subst;λ[[x;y;z];[atom[z] → [eq[y;z] → x;T → z];T → cons[subst[x;y;car[z]];subst[x;y;cdr[z]]]]]] has the S-expression

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

This notation is writable and somewhat readable. It can be made easier to read and write at the cost of making its structure less regular. If more characters were available on the computer, it could be improved considerably.

f. The Universal S-Function apply[1]

There is an S-function apply with the property that if f* is an S-expression for an S-function f and args is a list of arguments of the form (arg1,...,argn), where arg1,...,argn are arbitrary S-expressions, then apply[f*;args] and f[arg1;...;argn] are defined for the same values of arg1,...,argn, and are equal when defined. For example,

λ[[x;y];cons[car[x];y]][(A,B);(C,D)] = apply[(LAMBDA,(X,Y),(CONS,(CAR,X),Y));((A,B),(C,D))] = (A,C,D)

---

[1] Note that the APPLY operator for the 704 version of LISP I as described and used in the rest of the manual is not identical with this apply function.

The S-function apply is defined by

```
apply[f;args] = eval[cons[f;appq[args]];NIL]
```

where

```
appq[m] = [null[m] → NIL;T → cons[list[QUOTE;car[m]];appq[
          cdr[m]]]]
```

and

```
eval[e;p] = [
  atom[e] → [assoc[e;p];
  atom[car[e]] → [
    eq[car[e];QUOTE] → cadr[e];
    eq[car[e];ATOM] → atom[eval[cadr[e];p]];
    eq[car[e];EQ] → eq[eval[cadr[e];p];eval[caddr[e];p]];
    eq[car[e];COND] → evcon[cdr[e];p];
    eq[car[e];CAR] → car[eval[cadr[e];p]];
    eq[car[e];CDR] → cdr[eval[cadr[e];p]];
    eq[car[e];CONS] → cons[eval[cadr[e];p];eval[caddr[e];p]];
      T → eval[cons[assoc[car[e];p];evlis[cdr[e];p]];p]];
    eq[caar[e];LABEL] → eval[cons[caddar[e];cdr[e]];cdr[e]];
      cons[list[cadar[e];car[e]];p]];
    eq[caar[e];LAMBDA] → eval[caddar[e];append[pair[cadar[e];
      evlis[cdr[e];p];p]]]]
```

and

```
evcon[c;p] = [eval[caar[c];p] → eval[cadar[c];p];T → evcon[
             cdr[c];p]]
```

and

```
evlis[m;p] = [null[m] → NIL;T → cons[list[eval[car[m];p];
             evlis[cdr[m];p]]]]
```

We now explain a number of points about these definitions.

1. apply itself forms an expression representing the value of the function applied to the arguments, and puts the work of evaluating this expression onto a function eval. It uses appq to put quotes around each of the arguments, so that eval will regard them as standing for themselves.

2. eval[e;p] has two arguments, an expression e to be evaluated, and a list of pairs p. The first item of each pair is an atomic symbol, and the second is the expression for which the symbol stands.

3. If the expression to be evaluated is atomic, eval evaluates whatever is paired with it first on the list p.

4. If e is not atomic but car[e] is atomic, then the expression has one of the forms (QUOTE,e) or (ATOM,e) or (EQ,e_1,e_2) or (COND,(p_1,e_1),...,(p_n,e_n)), or (CAR,e) or (CDR,e) or (CONS,e_1, e_2) or (f,e_1,...,e_n) where f is an atomic symbol.

In the case (QUOTE,e) the expression e, itself, is taken. In the case of (ATOM,e) or (CAR,e) or (CDR,e) the expression e is evaluated and the appropriate function taken. In the case of (EQ,e_1,e_2) or (CONS,e_1,e_2) two expressions have to be evaluated. In the case of (COND,(p_1,e_1),...,(p_n,e_n)) the p's have to be evaluated in order until a true p is found, and then the corresponding e must be evaluated. This is accomplished by evcon. Finally, in the case of (f,e_1,...,e_n) we evaluate the expression that results from replacing f in this expression by whatever it is paired with in the list p.

5. The evaluation of ((LABEL,f,$\mathcal{E}$),e_1,...,e_n) is accomplished by evaluating ($\mathcal{E}$,e_1,...,e_n) with the pairing (f,(LABEL, f,$\mathcal{E}$)) put on the front of the previous list p of pairs.

6. Finally, the evaluation of ((LAMBDA,(x_1,...,x_n),$\mathcal{E}$), e_1,...,e_n) is accomplished by evaluating $\mathcal{E}$ with the list of pairs ((x_1,e_1),...,(x_n,e_n)) put on the front of the previous list p.

The list p could be eliminated, and LAMBDA and LABEL expressions evaluated by substituting the arguments for the variables in the expression ξ . Unfortunately, difficulties involving collisions of bound variables arise, but they are avoided by using the list p.

g. Functions with Functions as Arguments

There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions. One such function is maplist[x;f] with an S-expression argument x and an argument f that is a function from S-expressions to S-expressions. We define

maplist[x;f] = [null[x] → NIL;T → cons[f[x];maplist[cdr[x];f]]]

The usefulness of maplist is illustrated by formulas for the partial derivative with respect to x of expressions involving sums and products of x and other variables. The S-expressions that we shall differentiate are formed as follows.

1. An atomic symbol is an allowed expression.

2. If $e_1,e_2,\ldots,e_n$ are allowed expressions, (PLUS,$e_1,\ldots,e_n$) and (TIMES,$e_1,\ldots,e_n$) are also, and represent the sum and product, respectively, of $e_1,\ldots,e_n$.[1]

This is, essentially, the Polish notation for functions, except that the inclusion of parentheses and commas allows functions of variable numbers of arguments. An example of an allowed expression is

---

[1] For more exact information on arithmetic functions see Section 9.4.

(TIMES,X,(PLUS,X,A),Y), the conventional algebraic notation for which is X(X+A)Y.

Our differentiation formula, which gives the derivative of y with respect to x, is

```
diff[y;x] = [atom[y] → [eq[y;x] → ONE;T → ZERO];eq[car[y];
            PLUS] → cons[PLUS;maplist[cdr[y];λ[[z];diff[
            car[z];x]]]];eq[car[y];TIMES] → cons[PLUS;maplist[
            cdr[y];λ[[z];cons[TIMES;maplist[cdr[y];λ[[w];~eq[
            z;w] → car[w];T → diff[car[w];x]]]]]]]]
```

The derivative of the allowed expression, as computed by this formula, is (PLUS,(TIMES,ONE,(PLUS,X,A),Y),(TIMES,X,(PLUS,ONE,ZERO),Y),(TIMES,X,(PLUS,X,A),ZERO))

Besides maplist, another useful function with functional arguments is search, which is defined as

```
search[x;p;f;u] = [null[x] → u;p[x] → f[x];T → search[cdr[x];p;f;u]]
```

The function search is used to search a lsit for an element that has the property p, and if such an element is found, f of that element is taken. If there is no such element, the function u of no argument is computed.

3. LISP Primer

The features of LISP described in this section permit the user to define a number of S-functions and then compute the results of applying them to arguments.

3.1 Definition of Functions

In order to define functions we punch the following (using columns 1-72 of as many cards as are necessary):

```
DEFINE ((
(name of first function, definition of first function),
(name of second function, definition of second function),
        .
        .
        .
(name of last function, definition of last function)
)) ()
```

For example, if we wish to define the functions ff, alt, and subst given by

```
   ff[x] = [atom[x] → x;T → ff[car[x]]]
  alt[x] = [null[x]Vnull[cdr[x]] → x;T → cons[car[x];alt[
           cdr[cdr[x]]]]]
subst[x;y;z] = [atom[z] → [eq[y;z] → x;T → z];T → cons[subst[
           x;y;car[z]];subst[x;y;cdr[z]]]]
```

we write

```
DEFINE ((
(FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X))))))
(ALT LAMBDA (X) (COND ((OR (NULL X) (NULL (CDR X))) X)
(T (CONS (CAR X) (ALT (CDR (CDR X)))))))))
```

```
(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)))))))
))    ()
```

A few discrepancies between the situation as it is and what might be expected from the previous chapter should be noted

1. The commas in writing S-expressions may be omitted. This is an accident.

2. According to the definition of apply in the previous chapter one would expect to have to write (QUOTE T) in designating the left-over case in a conditional expression. For convenience, our apply allows (and in fact requires) that T be written instead.

3. The predicates null $\vee$ $\wedge$ and $\sim$ are built-in. We write (NULL X) and (OR p_1 p_2 ... p_n) and (AND p_1 p_2 ...p_n) and (NOT p).

4. The dot notation e.g. (A·B) is not allowed in LISP I.

Function definitions may involve other functions which are either built-in or are defined earlier or later.

3.2 The Use of Functions

After the cards which define functions we can put cards which cause their values to be computed and printed for given arguments. This is done by writing a triplet

function
list of arguments
p-list

In the simplest case the p-list is null and is written (). For example, in order to compute subst[(X,A);X;((X,A),X)] we write

```
SUBST    ((X A) X ((X B) X))    ()
```

The answer (((X A) B) (X A)) will be printed.

3.3 Debugging

The main debugging tool is the pseudo-function tracklist. If you put a card

TRACKLIST $(f_1\ f_2\ \ldots\ f_n)$ ()

after the cards which define the functions $f_1,\ldots,f_n$ and before the functions are used in computations, the computer will print the arguments and values of $f_1,\ldots,f_n$ each time they are used recursively in a computation. tracklist acts as a tracing program, but usually it is not necessary to trace more than one or two functions.

3.4 The Wang Algorithm for the Propositional Calculus

As an extended example of the use of a succession of function definitions to define an algorithm we give a LISP formulation of an algorithm for deciding whether a formula is a theorem of the propositional calculus published recently by Hao Wang.[1] Readers completely unacquainted with propositional calculus should probably skip this example.

---

[1] Wang, Hao. "Toward Mechanical Mathematics", IBM Journal of Research and Development, Vol. 4, No. 1, January 1960

(1) The Wang Algorithm[2]. We quote from pages 5 and 6 of Wang's paper.

"The propositional calculus (System P)

Since we are concerned with practical feasibility, it is preferable to use more logical connectives to begin with when we wish actually to apply the procedure to concrete cases. For this purpose we use the five usual logical constants $\sim$ (not), $\&$ (conjunction, $\vee$ (disjunction), $\supset$ (implication, $\equiv$ (biconditional), with their usual interpretations.

A propositional letter P, Q, R, M or N, et cetera, is a formula (and an "atomic formula"). If ϕ, ψ are formulae, then $\sim\phi, \phi\,\&\,\psi, \phi \vee \psi, \phi \supset \psi, \phi \equiv \psi$ are formulae. If π, ρ are strings of formulae (each, in particular, might be an empty string or a single formula) and ϕ is a formula, then π, ϕ, ρ is a string and $\pi \rightarrow \rho$ is a sequent which, intuitively speaking, is true if and only if either some formula in the string π (the "antecedent") is false or some formula in the string ρ (the "consequent") is true, i.e., the conjunction of all formulae in the antecedent implies the disjunction of all formulae in the consequent.

There are eleven rules of derivation. An initial rule states that a sequent with only atomic formulae (proposition letters) is a theorem if and only if a same formula occurs on both sides of the arrow. There are two rules for each of the five truth functions--one introducing it into the antecedent,

---

2 This example is an excerpt from Memo 14 of the Artificial Intelligence Project--R.L.E. and M.I.T. Computation Center, by John McCarthy, The Wang Algorithm for the Propositional Calculus Programmed in LISP.

one introducing it into the consequent. One need only reflect on the intuitive meaning of the truth functions and the arrow sign to be convinced that these rules are indeed correct. Later on, a proof will be given of their completeness, i.e., all intuitively valid sequents are provable, and of their consistency, i.e., all provable sequents are intuitively valid.

P1. Initial rule: if λ, ζ are strings of atomic formulae, then $\lambda \rightarrow \zeta$ is a theorem if some atomic formula occurs on both sides of the arrow.

In the ten rules listed below, λ and ζ are always strings (possibly empty) of atomic formulae. As a proof procedure in the usual sense, each proof begins with a finite set of cases of P1 and continues with successive consequences obtained by the other rules. As will be explained below, a proof looks like a tree structure growing in the wrong direction. We shall, however, be chiefly interested in doing the step backwards, thereby incorporating the process of searching for a proof.

The rules are so designed that given any sequent, we can find the first logical connective, i.e., the leftmost symbol in the whole sequent that is a connective, and apply the appropriate rule to eliminate it, thereby resulting in one or two premises which, taken together, are equivalent to the conclusion. This process can be repeated until we reach a finite set of sequents with atomic formulae only. Each connective-free sequent can then be tested for being a theorem or not, by the initial rule. If all of them are theorems, then the original sequent is a theorem and we obtain a proof; otherwise we get a counterexample and a disproof. Some simple samples will make this clear.

For example, given any theorem of "Principia", we can automatically prefix an arrow to it and apply the rules to look for a proof. When the main connective is $\supset$, it is simpler, though not necessary, to replace the main connective by an arrow and proceed. For example:

*2.45. $\vdash : \sim(PVQ) \cdot \supset \cdot \sim P$,
*5.21. $\vdash : \sim P \mathbin{\&} \sim Q \cdot \supset \cdot P \equiv Q$

can be rewritten and proved as follows:

*2.45. $\sim(PVQ) \rightarrow \sim P$ (1)
(1) $\rightarrow \sim P, PVQ$ (2)
(2) $P \rightarrow PVQ$ (3)
(3) $P \rightarrow P, Q$
VALID

*5.21. $\rightarrow \sim P \mathbin{\&} \sim Q \cdot \supset \cdot P \equiv Q$ (1)
(1) $\sim P \mathbin{\&} \sim Q \rightarrow P \equiv Q$ (2)
(2) $\sim P, \sim Q \rightarrow P \equiv Q$ (3)
(3) $\sim Q \rightarrow P \equiv Q, P$ (4)
(4) $\rightarrow P \equiv Q, P, Q$ (5)
(5) $P \rightarrow Q, P, Q$
VALID
(5) $Q \rightarrow P, P, Q$
VALID

P2a. Rule $\rightarrow\sim$: If $\phi, \zeta \rightarrow \lambda, \rho$, then $\zeta \rightarrow \lambda, \sim\phi, \rho$.
P2b. Rule $\sim\rightarrow$: If $\lambda, \rho \rightarrow \pi, \phi$, then $\lambda, \sim\phi, \rho \rightarrow \pi$.

P3a. Rule $\rightarrow \mathbin{\&}$: If $\zeta \rightarrow \lambda, \phi, \rho$ and $\zeta \rightarrow \lambda, \psi, \rho$, then $\zeta \rightarrow \lambda, \phi \mathbin{\&} \psi, \rho$.
P3b. Rule $\mathbin{\&} \rightarrow$: If $\lambda, \phi, \psi, \rho \rightarrow \pi$, then $\lambda, \phi \mathbin{\&} \psi, \rho \rightarrow \pi$.

P4a. Rule $\rightarrow V$: If $\zeta \rightarrow \lambda, \phi, \psi, \rho$, then $\zeta \rightarrow \lambda, \phi V \psi, \rho$.
P4b. Rule $V \rightarrow$: If $\lambda, \phi, \rho \rightarrow \pi$ and $\lambda, \psi, \rho \rightarrow \pi$, then $\lambda, \phi V \psi, \rho \rightarrow \pi$.

P5a. Rule $\rightarrow\supset$: If $\zeta,\phi\rightarrow\lambda,\psi,\rho$, then $\zeta\rightarrow\lambda,\phi\supset\psi,\rho$.
P5b. Rule $\supset\rightarrow$: If $\lambda,\psi,\rho\rightarrow\pi$ and $\lambda,\rho\rightarrow\pi,\phi$, then $\lambda,\phi\supset\psi,\rho\rightarrow\pi$.
P6a. Rule $\rightarrow\equiv$: If $\phi,\zeta\rightarrow\lambda,\psi,\rho$ and $\psi,\zeta\rightarrow\lambda,\phi,\rho$, then $\zeta\rightarrow\lambda,\phi\equiv\psi,\rho$.
P6b. Rule $\equiv\rightarrow$: If $\phi,\psi,\lambda,\rho\rightarrow\pi$ and $\lambda,\rho\rightarrow\pi,\phi,\psi$, then $\lambda,\phi\equiv\psi,\rho\rightarrow\pi$."

(2) The LISP Program. We define a function theorem [s] whose value is truth or falsity according to whether the sequent s is theorem.

The sequent

s: $\varphi_1,\ldots,\varphi_n\rightarrow\psi_1,\ldots,\psi_m$

is represented by the S-expression

s*: (ARROW,$(\varphi_1^*,\ldots,\varphi_n^*)$,$(\psi_1^*,\ldots,\psi_m^*)$)

where in each case the ellipsis ... denotes missing terms, and where φ^* denotes the S-expression for φ.

Propositional formulae are represented as follows:

1. For "atomic formulae" (Wang's terminology) we use "atomic symbols" (LISP terminology).

2. The following table gives our "Cambridge Polish" way of representing propositional formulae with given main connectives.

1. $\sim\varphi$	becomes	(NOT,φ^*)
2. $\varphi\,\&\,\psi$	becomes	(AND,φ^*,ψ^*)
3. $\varphi\vee\psi$	becomes	(OR,φ^*,ψ^*)
4. $\varphi\supset\psi$	becomes	(IMPLIES,φ^*,ψ^*)
5. $\varphi\equiv\psi$	becomes	(EQUIV,φ^*,ψ^*)

Thus the sequent

$\sim P\,\&\,\sim Q\rightarrow P\equiv Q,RVS$

is represented by

(ARROW,((AND,(NOT,P),(NOT,Q))),((EQUIV,P,Q),(OR,R,S)))

The S-function theorem [s] is given in terms of auxiliary functions as follows:

```
theorem [s] = th1[NIL;NIL;cadr[s];caddr[s]]

th1[a1;a2;a;c] = [null[a] → th2[a1;a2;NIL;NIL;c];T →
                  member[car[a];c]∨[atom[car[a]] →
                  th1[[member[car[a];a1];T → cons[car[
                  a];a1]];a2;cdr[a];c];T → th1[a1;[
                  member[car[a];a2] → a2;T → cons[
                  car[a];a2]];cdr[a];c]]]

th2[a1;a2;c1;c2;c] = [null[c] → th[a1;a2;c1;c2];atom[
                      car[c]] → th2[a1;a2;[member[car[
                      c];c1] → c1;T → cons[car[c];c1]];
                      c2;cdr[c]];T → th2[a1;a2;c1;[
                      member[car[c];c2] → c2;T → cons[
                      car[c];c2]];cdr[c]]]

th[a1;a2;c1;c2] = [null[a2] → ∼null[c2]∧thr[car[c2];
                   a1;a2;c1;cdr[c2]];T → thℓ[car[a2];
                   a1;cdr[a2];c1;c2]]
```

th is the main predicate through which all the recursions take place. theorem, th1 and th2 break up and sort the information in the sequent for the benefit of th. The four arguments of th are:

a1: atomic formulae on left side of arrow
a2: other formulae on left side of arrow
c1: atomic formulae on right side of arrow
c2: other formulae on right side of arrow

The atomic formulae are kept separate from the others in order to make faster the detection of the occurrence of formula on both sides of the arrow and the finding of the next formula to reduce. Each use of th represents one reduction according

to one of the 10 rules. The formula to be reduced is chosen from the left side of the arrow if possible. According to whether the formula to be reduced is on the left or right we use thℓ or thr. We have

```
thℓ[u;a1;a2;c1;c2] = [
    car[u] = NOT → th1r[cadr[u];a1;a2;c1;c2];
    car[u] = AND → th2ℓ[cdr[u];a1;a2;c1;c2];
    car[u] = OR → th1ℓ[cadr[u];a1;a2;c1;c2]∧th1ℓ[
              caddr[u];a1;a2;c1;c2];
    car[u] = IMPLIES → th1ℓ[caddr[u];a1;a2;c1;c2]∧th1r[
              cadr[u];a1;a2;c1;c2];
    car[u] = EQUIV → th2ℓ[cdr[u];a1;a2;c1;c2]∧th2r[
              cdr[u];a1;a2;c1;c2];
    T → error[list[THL;u;a1;a2;c1;c2]]]

thr[u;a1;a2;c1;c2] = [
    car[u] = NOT → th1ℓ[cadr[u];a1;a2;c1;c2];
    car[u] = AND → th1r[cadr[u];a1;a2;c1;c2]∧th1r[
              caddr[u];a1;a2;c1;c2];
    car[u] = OR → th2r[cdr[u];a1;a2;c1;c2];
    car[u] = IMPLIES → th11[cadr[u];caddr[u];a1;a2;c1;c2];
    car[u] = EQUIV → th11[cadr[u];caddr[u];a1;a2;c1;c2]∧
              th11[caddr[u];cadr[u];a1;a2;c1;c2];
    T → error[THR;a1;a2;c1;c2]]]
```

The functions th1ℓ, th1r, th2ℓ, th2r, th11 distribute the parts of the reduced formula to the appropriate places in the reduced sequent.

These functions are

```
th1ℓ[v;a1;a2;c1;c2] = [atom[v] → member[v;c1]∨
    th[cons[v;a1];a2;c1;c2];T → member[v;c2]∨
    th[a1;cons[v;a2];c1;c2]]
```

th1r[v;a1;a2;c1;c2] = [atom[v] → member[v;a1]∨
th[a1;a2;cons[v;c1];c2];T → member[v;a2]∨
th[a1;a2;c1;cons[v;c2]]]

th2ℓ[v;a1;a2;c1;c2] = [atom[car[v]] → member[car[v];c1]∨
th1ℓ[cadr[v];cons[car[v];a1];a2;c1;c2];T → member[
car[v];c2]∨
th1ℓ[cadr[v];a1;cons[car[v];a2];c1;c2]]

th2r[v;a1;a2;c1;c2] = [atom[car[v]] → member[car[v];a1]∨
th1r[cadr[v];a1;a2;cons[car[v];c1];c2];T → member[
car[v];a2]∨
th1r[cadr[v];a1;a2;c1;cons[car[v];c2]]]

th11[v1;v2;a1;a2;c1;c2] = [atom[v1] → member[v1;c1]∨
th1r[v2;cons[v1;a1];a2;c1;c2];T → member[v1;c2]∨
th1r[v2;a1;cons[v1;a2];c1;c2]]

Finally the function member is defined by

member[x;u] =∼null[u]∧[equal[x;car[u]]∨member[x;cdr[u]]]

(3) The LISP Program as Written in S-expressions. In this section we give the translation of the functions of the preceding section into S-expressions. Note that spaces are used in place of commas.

We have

```
DEFINE ((
(THEOREM (LAMBDA (S) (TH1 NIL NIL (CADR S) (CADDR S))))
```

```
(TH1 (LAMBDA (A1 A2 A C) (COND ((NULL A)
(TH2 A1 A2 NIL NIL C)) (T
(OR (MEMBER (CAR A) C) (COND ((ATOM (CAR A))
(TH1 (COND ((MEMBER (CAR A) A1 ) A1)
(T (CONS (CAR A) A1))) A2 (CDR A) C))
(T (TH1 A1 (COND ((MEMBER (CAR A) A2) A2)
(T (CONS (CAR A) A2))) (CDR A) C))))))))

(TH2 (LAMBDA (A1 A2 C1 C2 C) (COND
((NULL C) (TH A1 A2 C1 C2))
((ATOM (CAR C)) (TH2 A1 A2 (COND
((MEMBER (CAR C) C1) C1) (T
(CONS (CAR C) C1))) C2 (CDR C)))
(T (TH2 A1 A2 C1 (COND ((MEMBER
(CAR C) C2) C2) (T (CONS (CAR C) C2)))
(CDR C))))))

(TH (LAMBDA (A1 A2 C1 C2) (COND ((NULL A2) (AND (NOT (NULL C2))
(THR (CAR C2) A1 A2 C1 (CDR C2)))) (T (THL (CAR A2) A1 (CDR A2)
C1 C2)))))

(THL (LAMBDA (U A1 A2 C1 C2) (COND
((EQ (CAR U) (QUOTE NOT)) (TH1R (CADR U) A1 A2 C1 C2))
((EQ (CAR U) (QUOTE AND)) (TH2L (CDR U) A1 A2 C1 C2))
((EQ (CAR U) (QUOTE OR)) (AND (TH1L (CADR U) A1 A2 C1 C2)
(TH1L (CADDR U) A1 A2 C1 C2) ))
((EQ (CAR U) (QUOTE IMPLIES)) (AND (TH1L (CADDR U) A1 A2 C1
C2) (TH1R (CADR U) A1 A2 C1 C2) ))
((EQ (CAR U) (QUOTE EQUIV)) (AND (TH2L (CDR U) A1 A2 C1 C2)
(TH2R (CDR U) A1 A2 C1 C2) ))
(T (ERROR (LIST (QUOTE THL) U A1 A2 C1 C2)))
  )))
```

```
(THR (LAMBDA (U A1 A2 C1 C2) (COND
((EQ (CAR U) (QUOTE NOT)) (TH1L (CADR U) A1 A2 C1 C2))
((EQ (CAR U) (QUOTE AND)) (AND (TH1R (CADR U) A1 A2 C1 C2)
(TH1R (CADDR U) A1 A2 C1 C2) ))
((EQ (CAR U) (QUOTE OR)) (TH2R (CDR U) A1 A2 C1 C2))
((EQ (CAR U) (QUOTE IMPLIES)) (TH11 (CADR U) (CADDR U)
A1 A2 C1 C2))
((EQ (CAR U) (QUOTE EQUIV)) (AND (TH11 (CADR U) (CADDR U)
A1 A2 C1 C2) (TH11 (CADDR U) (CADR U) A1 A2 C1 C2) ))
(T (ERROR (LIST (QUOTE THR) U A1 A2 C1 C2)))
  )))

(TH1L (LAMBDA (V A1 A2 C1 C2) (COND
((ATOM V) (OR (MEMBER V C1)
(TH (CONS V A1) A2 C1 C2) ))
(T (OR (MEMBER V C2) (TH A1 (CONS V A2) C1 C2) ))
  )))

(TH1R (LAMBDA (V A1 A2 C1 C2) (COND
((ATOM V) (OR (MEMBER V A1)
(TH A1 A2 (CONS V C1) C2) ))
(T (OR (MEMBER V A2) (TH A1 A2 C1 (CONS V C2))))
  )))

(TH2L (LAMBDA (V A1 A2 C1 C2) (COND
((ATOM (CAR V)) (OR (MEMBER (CAR V) C1)
(TH1L (CADR V) (CONS (CAR V) A1) A2 C1 C2)))
(T (OR (MEMBER (CAR V) C2) (TH1L (CADR V) A1 (CONS (CAR V)
A2) C1 C2)))
  )))
```

```
(TH2R (LAMBDA (V A1 A2 C1 C2) (COND
((ATOM (CAR V)) (OR (MEMBER (CAR V) A1)
(TH1R (CADR V) A1 A2 (CONS (CAR V) C1) C2)))
(T (OR (MEMBER (CAR V) A2) (TH1R (CADR V) A1 A2 C1
(CONS (CAR V) C2))))
  )))

(TH11 (LAMBDA (V1 V2 A1 A2 C1 C2) (COND
((ATOM V1) (OR (MEMBER V1 C1) (TH1R V2 (CONS V1 A1) A2 C1
C2)))
(T (OR (MEMBER V1 C2) (TH1R V2 A1 (CONS V1 A2) C1 C2)))
  )))

(MEMBER (LAMBDA (X U) (OR (AND (NOT (NULL U))
(EQUAL X (CAR U))) (MEMBER X (CDR U)))))
))    ()
```

This causes the functions mentioned to be defined.

In our test run we next gave

TRACKLIST ((TH)) ()

which caused the arguments and values of the function th to be printed each time the function came up in the recursion. Accidentally, it turns out that these arguments essentially constitute a proof in Wang's style of the theorem.

In order to apply the method to the sequent

$p \rightarrow pVq$

we write

```
THEOREM
((ARROW,(P),((OR,P,Q))))
()
```

The APPLY operator, for each theorem and argument, evaluates the proposition and gives the answer as either true (T)

or false (F).

Thus the LISP function theorem in itself suffices to apply the Wang algorithm to any trial proposition and determine whether or not the proposition is a tautology.

4. The LISP Programming System

In this section are included descriptions of the main features of the LISP system. 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.

4.1 The APPLY Operator

The basis of LISP programming is the APPLY operator which is a synthesis of the apply and eval functions described in Chapter 2. The APPLY operator is the interpreter part of the LISP system in the sense that S-expressions representing functions and their arguments are read, interpreted and evaluated by apply. The apply now in use is a function of three arguments,

apply[f;x;p]

where

f = an S-expression representing an S-function f of n arguments; $n \geq 0$.
x = a list of n arguments
p = a list of pairs, each of the form (atomic symbol, value), used to assign values to free variables.

The value of apply[f;x;p] is the value of the S-function, f, evaluated at x, where the values assigned to any free variables are the values paired with the corresponding variables on the p-list.

Example 1:

As a simple first example consider the function

$$\lambda[[y;z];cons[car[y];cdr[z]]]$$

applied to the list

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

where no free variables need to be assigned. The arguments f,x, and p 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))
p: ( )

and the value of apply[f;x;p] is (A,D).

Note that the p-list must be included even though it is the NIL list.

Example 2:

Some care must be exercised in writing the lists x and p correctly. In the example,

f: CAR
x: ((A,B))
p: ( )

where apply[f;x;p] = 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 definition is not needed in this example since the specification of the arguments is clear. It would be correct but unnecessary to write

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

Example 3:

The following example, on the other hand, involves two arguments.

f: CONS
x: ((A),(B))
p: ( )

giving

apply[f;x;p] = cons[(A);(B)] = ((A),B)

The exact descriptions of function formats acceptable to the APPLY operator are discussed further in Section 4.3.

Example 4:

As an example involving a p-list we have

f: (LAMBDA,(Y),(CONS,Y,Z))
x: (A)
p: ((Z,(B)))

where apply[f,x,p] = 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 p-list to (B).

Example 5:

The following example involves a recursive function so that a label definition is required:

```
subst[x;y;z] = [atom[z] → [eq[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))
        p:   ( )
```

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

Note: The system for translating M-expressions into S-expressions given in Chapter 2 would give rise to (QUOTE,T) in lieu of T in the above S-expression. It is simpler to be able to write T (or F) and so in LISP I the latter usage is required. (QUOTE,T) and (QUOTE,F) must 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;p 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 Chapter 5.

4.2 Definitions of Functions in LISP

In Chapter 2 functions are connected to their names only through the use of the form LABEL. In the current LISP system, there are two further ways a function can be defined:

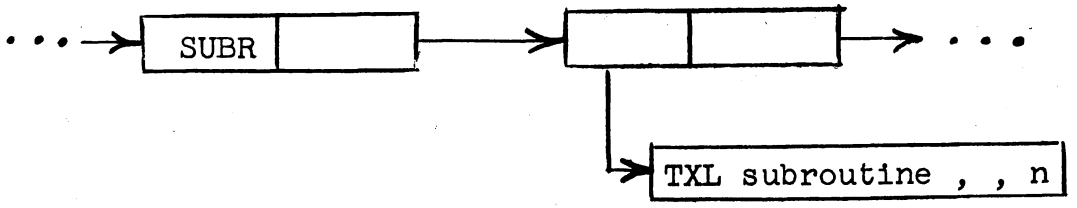
The first of these relates to functions defined in the system by machine-language subroutines. Such a subroutine for a function may be already available as part of the LISP system itself, in which case it appears among the functions given in Section 9, or it may have been produced by the LISP compiler.

In either case, if a machine-language subroutine defines a function, the association list for the name of the function (see Section 6.2) contains the indicator 'SUBR' to indicate that a subroutine exists. SUBR in turn points to a transfer instruction of the form

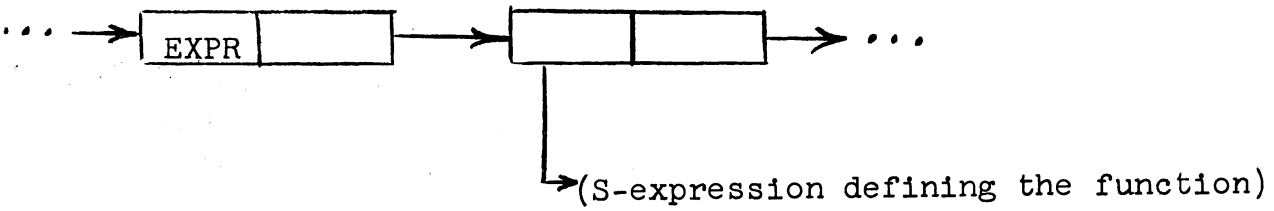
TXL subroutine , , n

where n is the number of arguments of the function.

Thus the relevant part of the association list has the structure,



The other way a function can be defined is by means of a LISP S-expression (representing an M-expression), which is to be interpreted during the running of a LISP program. In this case the indicator 'EXPR' is on the association list for the name of the function. EXPR points to the S-expression defining the function, as follows:



A function defined by an EXPR may be already available in the LISP system itself, in which case it appears among the functions given in Section 9, or it may be defined by the user by using the define function. Define is a pseudo-function (see Section 4.3) whose effect is to assign definitions of the EXPR sort to functions. The atomic symbol for the name of each function to be defined must be paired with the S-expression defining that function. A list of such pairs is then given as the argument for define. The APPLY operator acting on define of this argument creates, for each function defined, the EXPR structure shown above.

If EXPR is 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.

Consider for example the two functions, ff and alt discussed in Chapter 3,

$$ff[x] = [atom[x] \rightarrow x; T \rightarrow ff[car[x]]],$$

and

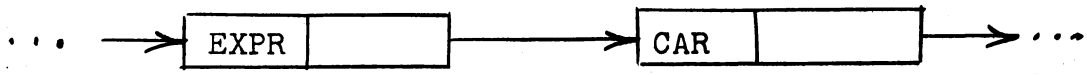
$$alt[x] = [null[x] \vee null[cdr[x]] \rightarrow x, T \rightarrow cons[car[x]; alt[cdr[cdr[x]]]]].$$

These are defined by letting the APPLY operator evaluate the following triplet, f;x;p, 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,((OR,(NULL,X),
     (NULL,(CDR,X))),X),(T,(CONS,(CAR,X),
     (ALT,(CDR,(CDR,X)))))))))))
p:   ( )
```

Note that the argument x starts with three parentheses: the first parenthesis starts the list x, the second indicates that x is a list of one argument, and the third starts the first pair. After APPLY has evaluated this triplet, the define function will have put the respective definitions, labelled by EXPR, on the association lists of the atomic symbols FF and ALT. Of course, define itself is one of the functions already defined and available as part of the LISP system.

Incidentally, a function can be defined in terms of another already-defined function--in which case the S-expression becomes simply the name of the established function. For example suppose one wishes to define the function FRST by the pair (FRST,CAR). After the define has been applied, the association list for FRST will contain the structure



where CAR points to the already-established association list for CAR. Chains of definitions of this sort are perfectly legal in LISP I.

The function 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.

4.3 Functions Appropriate to APPLY

Various kinds of functions can be used in LISP programming. One way of classifying functions is to divide them into functions and pseudo-functions. A function in LISP is evaluated for its value as such, e.g. car[(A,B,C)] = A, whereas a pseudo-

function is used for its effect rather than its value, e.g. define or compile. There is no difference in form however between functions and pseudo-functions.

In this section the classifying distinction will be made instead on the basis of the form of the functions; there are atomic functions and compound functions. If the f of the f; x;p triplet for the APPLY operator can be stated as an atomic symbol, the function is called atomic. All other legal forms for f are grouped under the heading of compound functions and are discussed below. An atomic function is not necessarily a simple or trivial function; it might be defined by a complicated subroutine or S-expression occurring on its association list.

Atomic Functions

When the APPLY operator evaluates a function which is atomic, that is, represented by an atomic symbol, it searches the association list of the atomic symbol for either SUBR or EXPR (see Section 4.2 just preceeding). If either is found, the function description which it points to, represented by a subroutine or an S-expression respectively, is used to evaluate the atomic function of x (the list of arguments). If neither SUBR or EXPR is found on the association list for the function, then the p-list is searched for the function's atomic symbol, and the expression paired with the symbol is used to evaluate the atomic function of x.

Compound Functions

If a function is compound it is represented by an S-expres-

sion whose first element may be LAMBDA, LABEL, or FUNARG.[1]

Before treating the different types in detail we recall from Chapter 2 that "If $\mathcal{E}$ is a form in (the) variables $x_1,\ldots,x_n$, then

$$\lambda((x_1,\ldots,x_n),\mathcal{E})$$

will be taken to be the function of n variables whose value is determined by substituting the arguments for the variables $x_1,\ldots,x_n$ in that order in $\mathcal{E}$ and evaluating the resulting expression."

For LABEL, on the other hand, from Chapter 2 we have "In order to be able to write expressions for recursive functions we introduce another notation: label(a,$\mathcal{E}$) denotes the expression $\mathcal{E}$, provided that occurrences of a within $\mathcal{E}$ are to be interpreted as referring to the expression as a whole."

FUNARG has not been discussed previously; it was introduced for convenience in writing the 704 system. Generally speaking, the use of FUNARG is internal to the system and need not concern the user. Its purpose is to tie the correct p-list of pairs to the function on which apply is operating. The following equivalence holds exactly:

$$\text{apply}[(\text{FUNARG},f,q);x;p] = \text{apply}[f;x;q],$$

where q is the list of pairs to be used in place of p in evaluating f[x].

---

[1] If some different element is used here, or if parentheses have been used incorrectly, the APPLY operator will usually fail.

When the APPLY operator evaluates a compound function, it determines first of all which of the three possible cases is involved.

If the S-expression for a compound function begins with LAMBDA, the triplet for the APPLY operator will be

f: (LAMBDA,(X1,X2,...,XN),ξ)
x: (Y1,Y2,...,YN)
p: (list of pairs)

The APPLY operator pairs the dummy variables[1], X1,X2,...,XN, with the values given on the list of arguments, x. 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 p-list. Then the third element of the LAMBDA list, the form ξ for the function, is evaluated using the enlarged p-list of pairs of assigned values.

The form ξ in the LAMBDA expression can be either an atomic symbol or a longer S-expression. The following cases can occur:

ξ = atomic symbol: The APPLY operator searches the association list of the atomic symbol to see if it represents a constant (signalled by either 'APVAL' or 'APVAL1' on the association list--cf. Chapter 6), and if so the value of f is that constant. If the atomic symbol does not represent a constant, the p-list is searched for the most recent pairing of this atomic symbol, and the value given by this pairing is the value of f. Otherwise an error is indicated, (see Chapter 8).

---

[1] The list of dummy variables can be the null list, ( ).

$\mathcal{E}$ = (atomic function, arguments): Cf. Example 4, Section 4.1--the APPLY operator evaluates the atomic function as described previously, and the result is the value of f.

$\mathcal{E}$ = (special form, arguments): See the description below of special forms. The value of f is the value given by APPLY after evaluating the special form.

$\mathcal{E}$ = (atomic symbol): Note the difference between this $\mathcal{E}$ which is enclosed in parentheses and the first $\mathcal{E}$ given above. The APPLY operator performs the function given within the parentheses. For example $\mathcal{E}$ = (READ), when operated on by APPLY would cause a list to be read from cards.

$\mathcal{E}$ = (LABEL,t,$\mathcal{F}$): The APPLY operator carries out the evaluation of the compound LABEL function as described below. The value of f as given by APPLY is the labelling name, t. Recall that in all the LAMBDA cases, the dummy variables have been paired and put on the p-list before the form $\mathcal{E}$ is evaluated.

If the S-expression for a compound function begins with LABEL, the triplet for the APPLY operator will be

f: (LABEL,name,definition)
x: (list of arguments)
p: (list of pairs)

(See Example 5, Section 4.1) The APPLY operator pairs the atomic symbol for the name of the function (the second element in the LABEL list), with the definition and adds the pair to the front of the p-list. It then applies the definition to the list of arguments x using the enlarged p-list, and the result is the value of the function.

Note on Special Forms:

Certain forms are classified under the heading of special forms and are evaluated differently from the usual case. If the first element of an S-expression is one of the special forms, the rest of the elements in the S-expression 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 the association list. Note that special forms cannot be used for the atomic functions discussed above.

QUOTE is a special form of one argument 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 S-expression following COND is an arbitrary number of arguments, each of which is a pair. For each pair the first element is the proposition and the second element of the pair is the corresponding expression. The propositions are evaluated in sequence from left to right, until one is found that is true, and then the expression corresponding to this proposition is evaluated and taken as the value of the entire conditional. If there are no propositions that are true, an error occurs.

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

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

PROG is a special form described under the program feature in Section 4.5.

Other special forms are described in Section 9.2.

4.4 Numbers in LISP

LISP II will be able to handle both integers and floating-point numbers, but in LISP I integers are not allowed.

Floating-point numbers in LISP I can be added, multiplied and raised to a power. The three functions available for these purposes are sum, prdct, expt, and are described in Chapter 9.

The association lists for floating-point numbers always contain the designation 'FLO' for float. Floating-point numbers do not appear on the list of atomic symbols (see Section 6.2), but on a list of floating-point numbers. These list entries point to the association lists for the numbers themselves. If a number is negative the indicator, 'MINUS' precedes the association list for the number. The detailed structure of association lists for floating-point numbers is discussed in Section 6.2.

Numbers are brought into a LISP program by being defined within S-expressions, and they must always be quoted. For example the following expression is correct,

(SUM,(QUOTE,0.6),X)

where X stands for a floating-point number which is, for example, paired with X on the p-list.

The exact rules for punching floating-point numbers for the LISP I read program are:

1) A decimal point must be included but not as the first symbol:

Incorrect	Correct
6	6. or 6.0
.0	0.0
.42	0.42
.003	0.003

2) A plus sign or a minus sign may precede a number. The plus sign is not required.

3) Exponent indication may be used. The power of ten is preceded by a sign and may contain one or two digits. Thus

4.21+10 represents 4.21×10^{10}

26.-2 represents 26×10^{-2}

4) Numerical values must be less in absolute value then $2^{128} (\sim 10^{38})$.

5) Significance is limited to eight decimal digits.

As an example of the use of numbers in LISP I consider the function length which finds the number of elements in a list. Its definition, in S-expression form, is

```
(LABEL,LENGTH,(LAMBDA,(Y),(COND,((NULL,Y),(QUOTE,0.0)),(T,(SUM,
        (LENGTH,(CDR,Y)),(QUOTE,1.0))))))
```

4.5 The Program Feature

The program feature in LISP allows sequences of operations (statements) to be expressed in LISP language. The effect is rather like a FORTRAN program with LISP statements. Each statement may be a form to be evaluated, or an atomic symbol, or a 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 this symbol, and when it is found the statements following the locating symbol are executed. A conditional expression to allow conditional branching may be used in a GO list.

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 APPLY operator that a program for sequential execution is to follow. The list L is a list of program variables which are handled in a manner similar to that for dummy variables, although the two types must be distinguished. The sequence of statements is the program to be executed.

The program variables are set equal to NIL at each (non-recursive) entrance to the program, but their values may be changed at any point whatsoever in the computation. For example programs can occur within other (higher-level) programs and in such cases an inner program can change any program variable of a higher-level program as well as its own program variables. The functions available for changing program variables are set and setq. The function set is a function of two variables both of which are evaluated. Thus to set a program variable V equal

to the value of an expression E, the V must be quoted in the S-expression,

(SET,(QUOTE,V),E).

The function setq on the other hand treats its first argument as if quoted and evaluates its second argument only. Thus the S-expression becomes

(SETQ,V,E),

which is entirely equivalent to the previous, and usually more convenient.

The following is an example of a program using the program feature. It is a version of search, (see Chapter 2).

$$\text{search}[\ell;p;f;u] = [\text{null}[\ell] \rightarrow u;p[\ell] \rightarrow f[\ell];T \rightarrow \text{search}[\text{cdr}[\ell];p;f;u]]$$

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))
     ))
```

In the above note that a function of no arguments, such as u, must be put in parentheses in order to be evaluated. Cf. (U) in the line starting with RETU.

Since the program feature will be used frequently in the manual to describe LISP functions, and since M-expressions are easier to read than S-expressions, programs will usually be written down in M-type notation. Program variables will be given in lower case, but the atomic symbols used as location-marking symbols will be retained in capital letters. The expression

(SETQ,X,Y) becomes x = y.

Using these rules and other obvious extensions of them the above program is written as follows:

```
        program for search[ℓ;p;fn;u] with program variables ℓt:
        ℓt = ℓ
TEST    go[null[ℓt] → RETU;T → CONT]
CONT    go[p[ℓt] → RETF;T → STEP]
STEP    ℓt = cdr[ℓ]
        go[TEST]
RETU    return[u]
RETF    return[fn[ℓ]]
```

4.6 The Compiler

The LISP compiler is itself a pseudo-function which is available to the APPLY operator. The compiler is called in by the LISP function,

comdef[x],

where x is a list of names of the functions to be compiled. Each function on the list will be compiled into a binary machine program provided the function is defined on its association list by an indicator EXPR pointing to an S-expression.

The value of comdef is a list of the names of the functions it was able to compile.

The compiler proceeds in three stages

1) Generation of LISP-SAP
2) Generation of binary program
3) SUBR put on association list

LISP-SAP is SAP in list form, for example

((,LXD,0,4),(,TXI,G0007,4,-1),(,TRA,*+5),(G0008,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.

The LISP-SAP program for each function compiled is printed out, on-line or off-line depending on the sense-switch settings.

After the compiler has created the LISP-SAP program for a function, the binary program is generated from LISP-SAP in two passes. In the first pass all symbols associated with BSS,0 are assigned locations in memory. In the second pass each instruction is assembled into memory. Then any unassigned symbols found during the second pass are assigned locations in memory following the generated instructions.

When the binary program has been generated, the compiler puts on the functions association list the indicator SUBR pointing to a TXL to the binary program.

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.

If a function listed in comdef has SUBR on its association list already, the compiler ignores the request for compilation

and goes ahead after printing out

(function name) HAS BEEN ALREADY COMPILED

If a function has not been defined at all, i.e. has neither EXPR or SUBR on its association list, the compiler prints out

(function name) IS NOT DEFINED

and goes on.

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 should be followed. If a function f uses a function g as a subfunction, then g should be included in a comdef which comes before the comdef involving f except in the following special case: if a closed circle of function usage occurs, e.g.

f_1 uses f_2
f_2 uses f_3
.
.
.
f_n uses f_1,

then all of the functions in the circle must be compiled in the same comdef. Thus the functions listed in a given comdef should be either unrelated or related in this circular sense. Any other subfunctions on which they depend should have been compiled by a previous comdef.

Note: The above rule on order should be followed for maximum compiling efficiency. It is also possible to compile all functions in one comdef regardless of dependency. The one unbreakable rule is that if a function f uses a function g as a subfunction, g cannot be compiled in a comdef which comes after the one containing the f.

Another pseudo-function, compile[ℓ], is available to compile functions not previously defined. The argument ℓ of compile is a list of function definitions, each one of which must be of the form

(LABEL,NAME,(LAMBDA,(list of free variables),expression))

The functions caar,cadr,...,cddar,cdddr are available to the compiler, so it is possible to write simply (CADAR,X) in lieu of (CAR,(CDR,(CAR,X))) in defining a function. 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 APPLY operator, 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. However, in distinction to the usual program-feature, conditional statements are themselves allowed as action statements by the compiler provided they give rise to a definite 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)
     ))
```

In terms of the M-type notation introduced in the previous section this becomes:

compiler program for search[ℓ,p,fn,u] with program variable ℓt:

```
            ℓt = L
TEST        [null[ℓt] → return[u];p[ℓt] → return[fn[ℓ]]]
            ℓt = cdr[ℓt]
            go[TEST]
```

In writing programs for the compiler the function setq must be used -- set is not acceptable.

Amplification of the compiler system is under way. In particular a feature will be included to permit a user to write SAP-type programs defining functions. Such programs will represent a third way (in addition to EXPR and SUBR) of defining functions on their association lists, and should add flexibility to the system. SAP-program definitions of functions will be termed 'Macros' and will be described in a later memo.

Two examples of compiled functions are given below.

1) memlis[x;y]

The function memlis[x;y] is a predicate whose value is true if x is a member of the list y, and false otherwise. x can be an atomic symbol or a list.

In the program below the compiler has generated certain atomic symbols (G0005, G0006, G0008, G0009) as location references in the program, and other symbols to be used as temporary storage as indicated below.

G0001) Index Register 4 at entry to memlis subroutine
G0002) Storage for x
G0003) Storage for y
G0004) Storage for final answer (true or false)

G0007) Storage for the program variable M1
G0010) Temporary storage
G0011) Temporary storage

The arguments x and y are stored in the AC and MQ at entry, and the answer is in the AC at exit.

The subroutine for equal[a;b] expects its two arguments in the AC and MQ and comes back with either zero (false) or one (true) in the AC.

The resulting LISP-SAP program is listed below exactly as the compiler gives it on the output sheet, except for explanatory comments to the right of the instructions.

```
FUNCTION APPLY(F,L,A) HAS BEEN ENTERED, ARGUMENTS..
COMPILE
(((LABEL,MEMLIS,(LAMBDA,(X,Y),(PROG,(M1),(SETQ,M1,Y),M2,(COND,
((NULL,M1),(RETURN,F)),((EQUAL,(CAR,M1),X),(RETURN,T))),(SETQ,
M1,(CDR,M1)),(GO,M2))))))
(MEMLIS,BSS,0)
( ,SXD,G0001,4)              save index register 4
( ,STO,G0002)                store x
( ,STQ,G0003)                store y
( ,CLA,G0003)
( ,STO,G0007)                set M1 = y
(G0006,BSS,0)                location M2
( ,CLA,G0007)
( ,TNZ,G0008)                to G0008 if null[M1] = F, or
( ,CLA,$ZERO)                return with AC = F if null[M1] = T
( ,TRA,G0005)
(G0008,BSS,0)
( ,LXD,G0007,4)
( ,CLA,0,4)
( ,PAX,0,4)                  car[M1]
```

```
( ,SXD,G0011,4)
( ,LDQ,G0002)              x
( ,CLA,G0011)              car[M1]
( ,TSX,EQUAL,4)
( ,STO,G00010)             result of equal test (note that the
( ,CLA,G00010)             compiler stores everything it computes)
( ,TZE,G0009)              transfer if car[M1] ≠ x
( ,CLA,$ONE)               otherwise return with AC = T
( ,TRA,G0005)
(G0009,BSS,0)
( ,LXD,G0007,4)
( ,CLA,0,4)
( ,PDX,0,4)                cdr[M1]
( ,SXD,G0007,4)            set M1 = cdr[M1]
( ,TRA,G0006)
(G0005,BSS,0)
( ,STO,G0004)              answer (true (one) or false (zero))
( ,CLA,G0004)
( ,LXD,G0001,4)            restore original index register 4
( ,TRA,1,4)                return

END OF APPLY, VALUE IS ...
(MEMLIS)
```

2) maplist[x;g]

The function maplist[x;g] is more complicated than the previous example, due both to its use of a function g, and to the fact that maplist may itself be used recursively by g. The latter fact requires that the arguments be saved each time the routine is entered, and this is done by storing them in a public push-down list.[1] The current next available space in this

---

[1] See page 144 of the Quarterly Report referred to on the first page of Chapter 2.

push-down list is indicated by the (complement of the) compiler public push-down list indicator, $CPPI. This indicator is tested by the program against the last available space in the list, ($ENPDL, or end of push-down list), and when there is no more space available an error transfer to $NOPDL stops the computation.

Another use of storage is illustrated below by the cons function which sends the result of the cons to free storage. Each time the cons is used, a test is made of the indicator, $FREE, which gives the (complement of the) next free-storage location available. When free storage has been used up the subroutine $FROUT is used to call in the garbage collector to retrieve some storage. (See Section 6.3.)

The treatment of the function g depends on whether it is defined by a subroutine or by an S-expression to be interpreted. If there is a subroutine for g, g will be in the form of an instruction

TXL(subr),0,0

stored in location G0003, so that the TSX,G0003,4 in the program below amounts to a transfer to the subroutine. If g is defined by an S-expression, then g, as stored in G0003, will have a decrement pointing to the S-expression. The subroutine COMPAT below carries out the interpretation of the expression using the APPLY operator. Incidentally, the 1 in the line following the use of COMPAT has been set up by the compiler to indicate that g is a function of one argument.

The LISP-SAP program is the following:

```
FUNCTION APPLY(F,X,P) HAS BEEN ENTERED, ARGUMENTS..
COMPILE
(((LABEL,MAPLIST,(LAMBDA,(X,G),(PROG,(M1,M2),(COND,((NULL,X),
(RETURN,NIL))),(SETQ,M1,(CONS,(G,X),NIL)),(SETQ,M2,M1),A1,
(SETQ,X,(CDR,X)),(COND,((NULL,X),(RETURN,M1))),(RPLACD,M2,
(CONS,(G,X),NIL)),(SETQ,M2,(CDR,M2)),(GO,A1))))))
```

```
(MAPLIST,BSS,0)
( ,SXD,G0001,4)          save index register 4
( ,STQ,$ARG2)            store g in temporary storage to free MQ
( ,LXD,$CPPI,4)          compiler push-down list indicator
( ,TXH,*+2,4,$ENPDL)     transfer to *+2 if space available
( ,TSX,$NOPDL+1,4)       otherwise (no more left), stop on error
( ,LDQ,G0001)
( ,STQ,0,4)              current index register 4 → push-down list
( ,LDQ,G0003)
( ,STQ,1,4)              current g → push-down list
( ,LDQ,G0011)
( ,STQ,2,4)              current M2 → push-down list
( ,LDQ,G0002)
( ,STQ,3,4)              current x → push-down list
( ,LDQ,G0009)
( ,STQ,4,4)              current M1 → push-down list
( ,TIX,*+1,4,5)
( ,SXD,$CPPI,4)          update push-down list indicator
( ,LDQ,$ARG2)
( ,STO,G0002)            store x
( ,STQ,G0003)            store g
( ,CLA,G0002)
( ,TNZ,G0008)            null[x] = F
( ,CLA,$ZERO)            null[x] = T
( ,TRA,G0005)            return, value = NIL
(G0008,BSS,0)            null[x] = F
(G0007,BSS,0)            extra (unused) line due to compiler
( ,CLA,G0002)            x → AC
( ,LXD,G0003,4)          g
( ,TXH,*+3,4,0)          transfer if g is S-expression
( ,TSX,G0003,4)          otherwise use g as subroutine
( ,TRA,*+4)              returns here from subroutine for g
( ,SXD,*+2,4)            set up pointer to S-expression for COMPAT
```

```
(  ,TSX,COMPAT,4)
(  ,0,1)                    "1" means g has one argument
(  ,STO,G0010)              g(x)
(  ,LXD,$FREE,4)            next free-storage location
(  ,TXH,*+2,4,0)
(  ,TSX,$FROUT,4)           out of free storage--retrieve some
(  ,CLA,0,4)
(  ,STD,$FREE)              update next free-storage location
(  ,CLA,G0010)
(  ,ARS,18)
(  ,ADD,$ZERO)
(  ,STO,0,4)                cons[g(x);NIL]
(  ,SXD,G0009,4)            set M1 = cons[g(x);NIL]
(  ,CLA,G0009)
(  ,STO,G0011)              set M2 = M1
(G0006,BSS,0)               location A1
(  ,LXD,G0002,4)
(  ,CLA,0,4)
(  ,PDX,0,4)
(  ,SXD,G0002,4)            set x = cdr[x]
(  ,CLA,G0002)
(  ,TNZ,G0013)              null[x] = F
(  ,CLA,G0009)              null[x] = T
(  ,TRA,G0005)              return, value = M1
(G0013,BSS,0)
(G0012,BSS,0)               etc.
(  ,CLA,G0002)
(  ,LXD,G0003,4)
(  ,TXH,*+3,4,0)
(  ,TSX,G0003,4)
(  ,TRA,*+4)
(  ,SXD,*+2,4)
(  ,TSX,COMPAT,4)
```

```
( ,0,1)
( ,STO,G0015)              all as before, g(x)
( ,LXD,$FREE,4)
( ,TXH,*+2,4,0)
( ,TSX,$FROUT,4)
( ,CLA,0,4)
( ,STD,$FREE)
( ,CLA,G0015)
( ,ARS,18)
( ,ADD,$ZERO)
( ,STO,0,4)                all as before, cons[g(x);NIL]
( ,SXD,G0014,4)
( ,LXD,G0011,4)
( ,CLA,G0014)
( ,STD,0,4)                replacd[M2;cons[g(x);NIL]]
( ,LXD,G0011,4)
( ,CLA,0,4)
( ,PDX,0,4)
( ,SXD,G0011,4)            M2 = cdr[M2]
( ,TRA,G0006)              go to A1
(G0005,BSS,0)              return
( ,STO,G0004)              store answer temporarily
( ,LXD,$CPPI,4)
( ,TXI,*+1,4,5)            prepare to restore items
( ,SXD,$CPPI,4)            from the push-down list
( ,LDQ,0,4)
( ,STQ,G0001)              restore index register 4
( ,LDQ,1,4)
( ,STQ,G0003)              restore g
( ,LDQ,2,4)
( ,STQ,G0011)              restore M2
( ,LDQ,3,4)
( ,STQ,G0002)              restore x
```

```
( ,LDQ,4,4)
( ,STQ,G0009)                restore M1
( ,CLA,G0004)                answer → AC
( ,LXD,G0001)
( ,TRA,1,4)                  return

END OF APPLY, VALUE IS ...
(MAPLIST)
```

5. Running a LISP Program

In this section, various aspects of running a LISP program are discussed. In Section 5.1, we discuss how to punch cards, put together a running deck, and submit a run. In Section 5.2, the tracing program is described. Section 5.3 encompasses a brief discussion of the current state of the LISP-Flexo system. The error indications given by the LISP I system when running a LISP program have been relegated to a late section (Chapter 8) since they are in the nature of a reference look-up list.

5.1 Submitting a Run

As stated in the previous section, the foundation of LISP programming is the APPLY operator, which is based on the function apply[f;x;p]. The function 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;p, which are punched on cards and submitted in the appropriate order in a deck.

Card Punching

Columns 1-72 (inclusive) of a punched card are read by the LISP read program. The S-expression for the f, x, and p of each triplet should be punched on cards with one or more spaces (blanks) between each member of the triplet. There are no rules about the location of the expression 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 73. In other words, card 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 for example is read as a single comma.

Deck Format

A LISP program to be run must be preceded by an LCON deck which calls the LISP system from tape. The LCON deck consists of five cards labelled NYBOL1 followed by six cards labelled LCON. If the LISP program is being run at the M.I.T. Computation Center, 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 following the LCON deck. These cards have the direction (TST, SET,FLX,CRD, or FIN) in columns 8-10 of the card and have the following effects:

TST: Subsequent triplets are read in until a STOP card (see 4. of the deck format below) is reached. The lists read are put out onto tape 2 for off-line printing together with a list of the atomic symbols in the machine. Control is then sent to the APPLY operator which operates on each triplet in turn, putting out on tape 2 the triplet f;x;p followed by the value of apply[f;x;p]. 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 immediately after all the triplets have been evaluated the state of the memory as it stands 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 triplet in a SET 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 5.3) is called into control.

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

FIN: The computer stops.

Any other card, such as a SAP REM card, may be included in a deck between a STOP card and the next direction card. The card will be printed out but will have no other effect. Such a card in any other position in the deck will cause trouble.

---

*The FLX and CRD direction cards are not understood by the basic LISP system; only the LISP-flexo system can interpret them.

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 sequence of triplets f;x;p, on which the apply function is to operate
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 within columns 1-72 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 (to prevent the card reader from hanging up)

Operating Instructions

Tapes used:

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

Sense switches used:

SWITCH NUMBER	USE
1	UP: Off-line card input (tape 4) DOWN: On-line card input
3	UP: Suppress on-line printing DOWN: Print on-line
5	UP: Write all printing on tape 2 for off-line printing DOWN: No off-line printing

Both on-line and off-line printing can be done at the same time. All the above switches take effect immediately.

If a LISP program is being run at M.I.T. the following indications should be given on the performance request card:

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.

5.2 Tracing Option

A LISP tracing function called tracklist is available to help the user locate his program errors. Of course, as with any tracing system, tracklist consumes both time and paper, but in some cases it may prove helpful.

The function tracklist is a function of one argument, x, 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 if tracklist is to trace a function which is on the "top-level" of APPLY, i.e. the f in apply[f;x;p], then the S-expression for f must start with LAMBDA or LABEL. Finally tracklist cannot trace the functions

car
cdr
cons
list

Whenever one of the functions included in the argument of tracklist is encountered during the running of the LISP program, tracklist 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.

If tracklist is called into action within a computation preceded by a SET direction card (see Section 5.1), it continues to trace its arguments for the rest of the run (provided the SET card is successful and determines a new "base" image). If tracklist occurs within a computation preceded by a TST direction card it continues only up to the STOP card.

5.3 The Flexowriter System (for M.I.T. users only)[1]

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 at M.I.T. The current version of the LISP-Flexo system is described here, but it is subject to change. The reader who might wish first of all to become familiar with the Flexowriter system, can find a write-up of it, although for a different purpose, in Professor Herbert M. 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 function definitions into the 704 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, at the option of the user. The input deck to be put into the card reader has the following format,

---

[1] The LISP-Flexo system does not contain the compiler option.

1. GETOPFLEX — card to call the M.I.T. Automatic Operator program and the Flexowriter programs in from the M.I.T. System tape

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

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

4. GETLISP2 — card to transfer into operating program

5. FLX — direction card--see Section 5.1

6. 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 <u>two</u> 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 between the above FLX card and the FIN card below.

7. FIN — direction card--see Section 5.1

8. 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 a LISP program to be typed 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 a program is to be typed in on the Flexowriter 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;p 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 correcting, and if the error was in a previous line (see Errors below) it cannot be deleted at all. The TEN-Mode was developed to obviate this difficulty and to allow the user to type S-expressions in small fragments independently erasable. 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, the S-expressions for an APPLY triplet, f;x;p, should be typed on the Flexowriter. The typing is done by typing up to 72 characters (fewer if desired) followed by a carriage return. The Flexowriter then types out STOP and digests the 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 ↵

The Flexowriter then types out STOP and returns control to the control routine which types out GO and expects a direction type-in. 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.

---

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

TEN-Mode

There are ten buffers of core storage for S-expressions set aside for use by the TEN-Mode of operation. Pieces of an S-expression can be typed into these buffers in any sequence, with overwriting allowed, and the Flexo-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 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;p has been found by the read program, the APPLY operator operates on the 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, provided no error has occurred, one types into two consecutive registers, in response to a "fresh" GO (see previous footnote),

$:n_1 \rightarrow$ STOP ↵

$:n_{i+1} \rightarrow$ bFINbb ↵

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

To stop the operation of the TEN-Mode after an error has occurred, one types, in response to <u>any</u> GO, into three consecutive registers n_i, n_{i+1}, and n_{i+2}

$:n_i \rightarrow$ several)))))) ↵

$:n_{i+1} \rightarrow$ STOP ↵

$:n_{i+2} \rightarrow\rightarrow$ bFINbb ↵ (or CRD in place of FIN).

Then one does a RLN of these three buffer registers.

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

→ ONE ↵

Errors

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

Extra right parentheses give a read error.

The error type-outs described in Chapter 8 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 read 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.

Other Flexowriter Monitor Type-ins

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

The Functions ioflex(x) and ioflip(y)

Two special functions ioflex and ioflip are available in the LISP-Flexo System. They are functions of one argument and a null p-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 is 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-in on the Flexowriter
y = PRINT	flips control between printing on the on-line printer or 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, and 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.

Example Flexowriter Run

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.

Flexowriter Sheet	Explanation
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;p, 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))) ()
:2   CONS  ((G,H),                   (type-ins)
: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
     RLN                             error
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)) ()
:4   IOFLEX(PRINT) ()                type-ins
: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                                 flips 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
```

6. List Structures

Much of the following is taken from the M.I.T. report cited in Chapter 2. In Section 6.1 the representation of S-expressions by list structures is described and an example of list construction is included. In Section 6.2 association lists for atomic symbols and for floating-point numbers are described. Finally, in Section 6.3, a discussion of the use of free storage and the operation of the "garbage collector" is given.

6.1 General Description

(1) Representation of S-expressions by List Structure

A list structure is a collection of computer words arranged as in Fig. 1a or 1b. Each word of the list structure is represented by one of the subdivided rectangles in the figure. The left box of a rectangle represents the address field of the word and the right box represents the decrement field. An arrow from a box to another rectangle means that the field corresponding to the box contains the location of the word corresponding to the other rectangle.

It is permitted for a substructure to occur in more than one place in a list structure, as in Fig. 1b, but it is not permitted for a structure to have cycles, as in Fig. 1c.

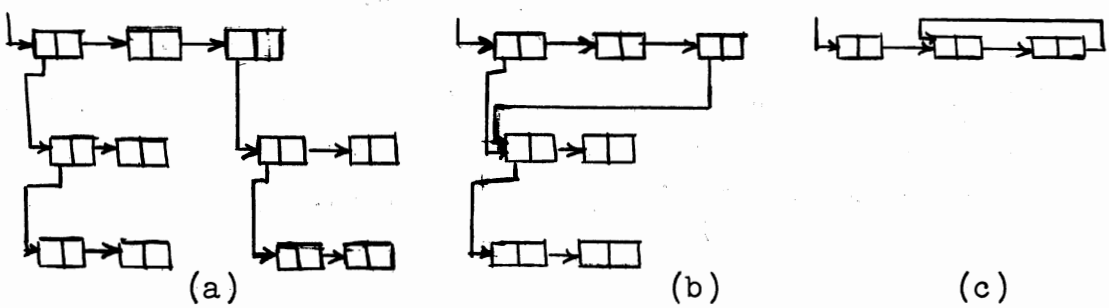


Fig. 1

An atomic symbol or a floating-point number is represented in the computer by a list structure of special form called the association list of the symbol. Such lists are described in Section 6.2 which follows.

An S-expression x that is not atomic is represented by a word, the address and decrement parts of which contain the locations of the subexpressions car[x] and cdr[x], respectively. In the list form of expressions the S-expression (A,(B,C),D) is represented by the list structure of Fig. 2.

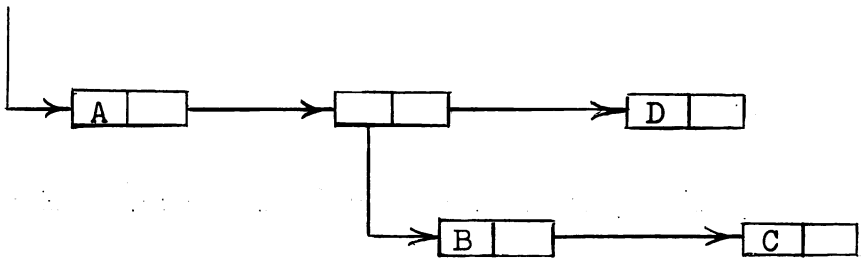


Fig. 2

When a list structure is regarded as representing a list, we see that each term of the list occupies the address part of a word, the decrement part of which points to the word containing the next term, while the last word has NIL in its decrement. The dot notation, e.g. (A·B), which is discussed in Section 2.2 is not allowed in LISP I; all lists and sublists must end with NIL.

An expression that has a given subexpression occurring more than once can be represented in more than one way. Whether the list structure for the subexpression is or is not repeated depends upon the history of the program. Whether or not a subexpression is repeated will make no difference in the results of a program as they appear outside the machine, although it will affect the time and storage requirements. For example, the S-expression in list form ((A,B),(A,B)) can be represented

by either the list structure of Fig. 3a or 3b.

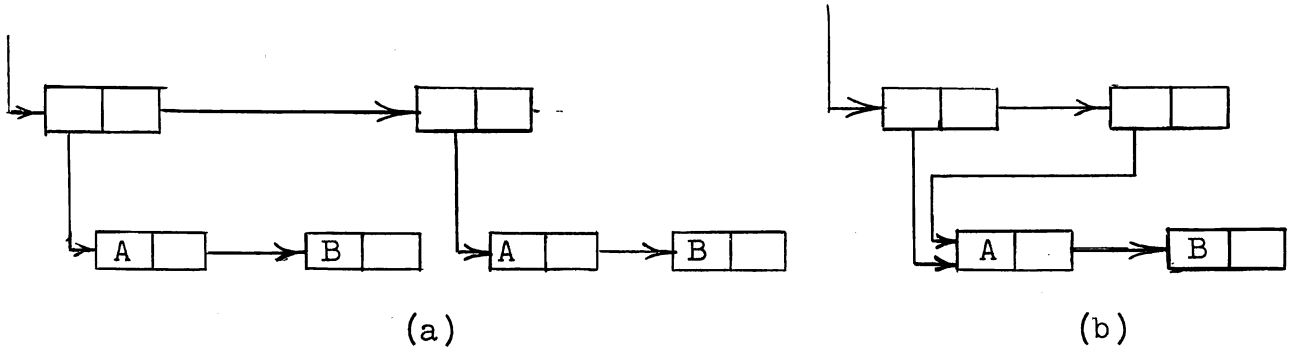


Fig. 3

The prohibition against circular structures is essentially a prohibition against an expression being a subexpression of itself. Such an expression could not exist on paper in a world with our topology. Circular list structures would have some advantages in the machine, for example, for representing recursive functions, but difficulties in printing them, and in certain other operations, make it seem advisable not to use them for the present.

The advantages of list structures for the storage of symbolic expressions are:

1. The size and even the number of expressions with which the program will have to deal cannot be predicted in advance. Therefore, it is difficult to arrange blocks of storage of fixed length to contain them.

2. Registers can be put back on the free-storage list when they are no longer needed. Even one register returned to the list is of value, but if expressions are stored linearly, it is difficult to make use of blocks of registers of odd sizes that may become available.

3. An expression that occurs as a subexpression of several expressions need be represented in storage only once.

(2) Construction of List Structures

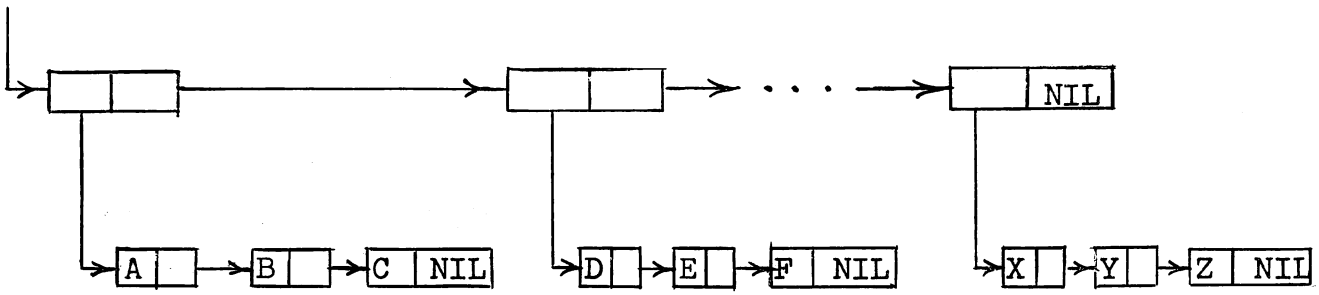
The following simple example has been included to illustrate the exact construction of list structures. Two types of structure are shown, and the recipe for using LISP to derive one from the other is given.

In the list structures below the word 'NIL' is shown in the decrement of the final element of lists and sublists. Strictly speaking in the current 704 representation of lists this decrement contains zero rather than the indicator NIL which would point to the association list for the atomic symbol NIL, but the replacement generally will not affect the user.

In the following example we assume that we have a list of the form

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

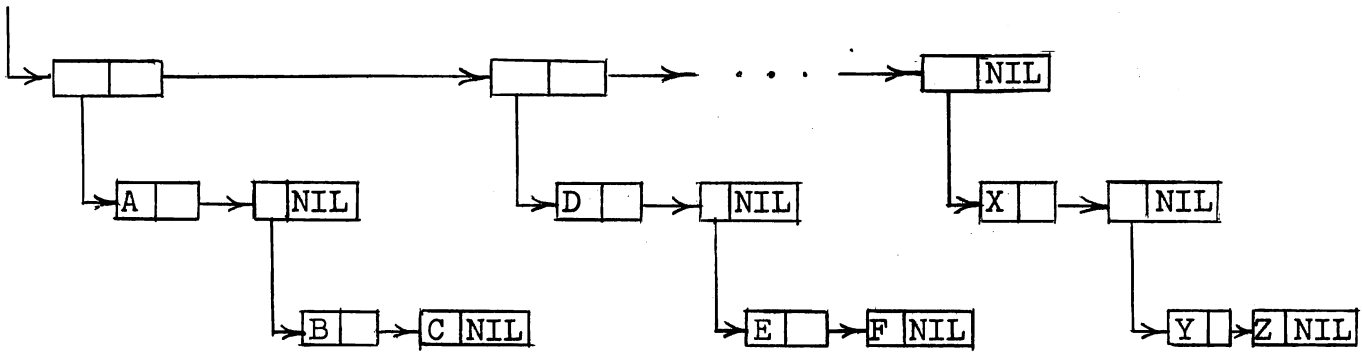
which is represented as



and that we wish to construct a list of the form

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

which is represented as



First we consider the typical substructure, (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]]

Or, using the list operation, we can write the same thing 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 to be used in the previous construction are found from

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

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: Any list which is read into the computer by LISP I can have only NIL (zero) in the final decrement. That is, the dot notation (A·B) cannot be read by the current read routine, so that the only way to get (A·B) is by a cons within the machine. Generally the use of such a cons to create a full machine word should be avoided.

6.2 Association Lists

Within the LISP I system there is a list of atomic symbols already available in the system. Each of these atomic symbols has an association list* associated with it, and in fact the atomic symbol on the list of atomic symbols points to

---

* In the local M.I.T. patois, association lists are also referred to as "property lists", and atomic symbols are sometimes called "objects".

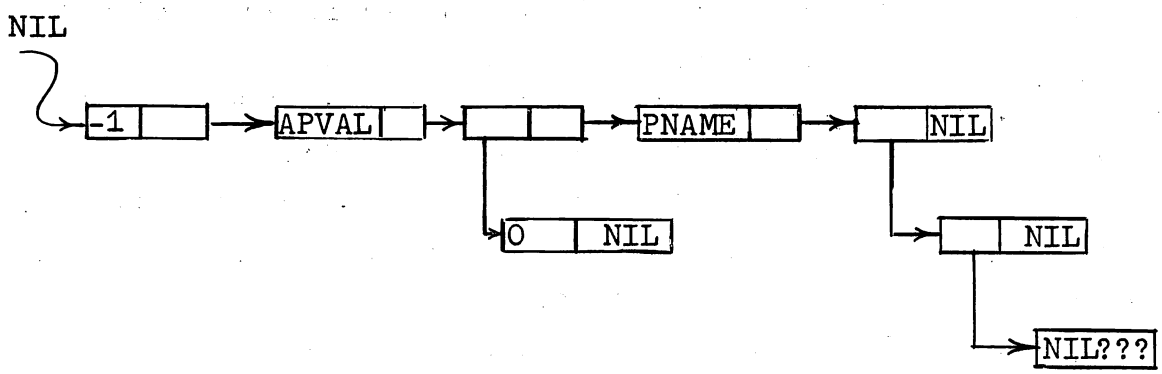
the location of the association list. Integers are included in the list of atomic symbols, but floating-point numbers are listed on a separate list which will be discussed below.

When an atomic symbol is read by the read program a check is made to see if it is already available on the list of atomic symbols. If it is not, it is added to the list and an association list is created for it.

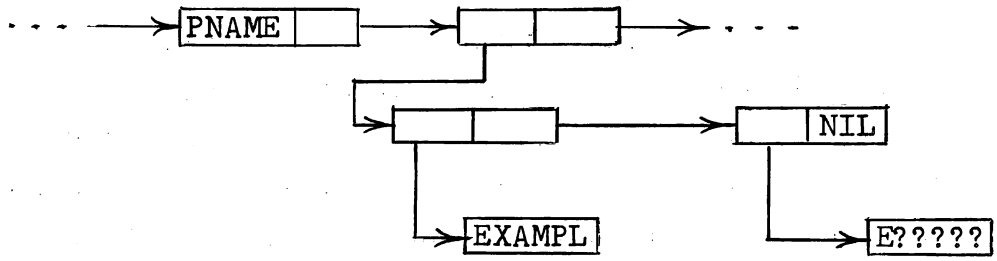
An association list is characterized by having the special constant 77777_8 (i.e. minus 1) in the address section of the first word. The rest of the list carries any other information that is to be associated with the atomic symbol. This information may include: the print name (signalled by the indicator 'PNAME'), that is, the string of letters and digits which represent the symbol outside the machine; a numerical value if the symbol represents a number; another S-expression if the symbol, in some way, serves as a name for it; or the location of a routine if the symbol represents a function for which there is a machine-language subroutine.

Two kinds of value indicators are used on association lists: APVAL and APVAL1. Both indicate a value, but APVAL points to a single word which has the value in the address part and NIL (zero) in the decrement part, whereas APVAL1 points to a list. The indicator APVAL acts like a stop to the garbage collector (see Section 6.3 below) whereas APVAL1 allows the garbage collector to look ahead through the part of the list pointed to by APVAL1. Whenever a programmer wishes to put some value on an association list for an atomic symbol, he may use the function attrib discussed in Chapter 9 to put a sublist of the form (APVAL,( )) on the association list.

For example, on the association list for NIL, APVAL is used as follows:



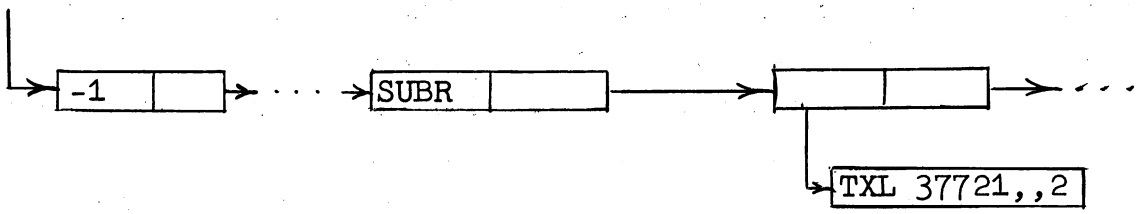
A similar structure holds for APVAL1 list-segments. The last word of a print name is filled out with a 6-bit combination (here shown as "?") that does not represent a character printable by the computer. Note incidentally that the print name is depressed two levels to allow for names longer than one register length, for example,



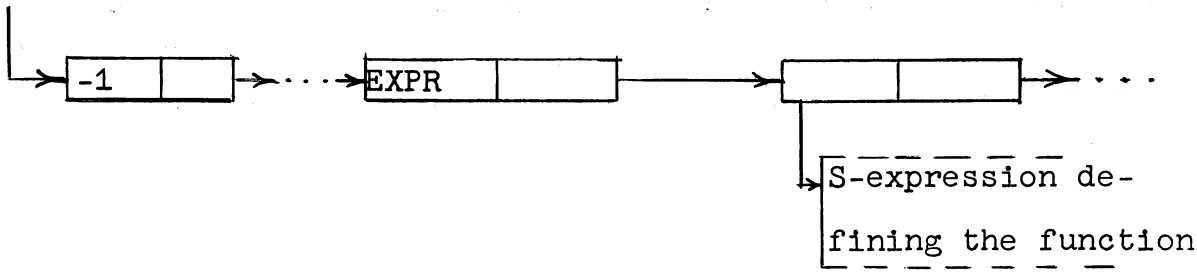
On the association lists for integers the indicator INT points to the integral value. The integral value is given as a full machine word with the value pushed to the right-hand end of the register.

On the association lists of those atomic symbols which represent functions, SUBR indicates a SAP subroutine for performing a function. SUBR points to a TXL to the subroutine,

and the decrement of the TXL instruction contains the number of arguments belonging to the function. Thus the following fragment of an association list belongs to a function of two arguments whose subroutine is located at 37721_8.



The indicator EXPR on an association list of a function points to the S-expression for the function, as in



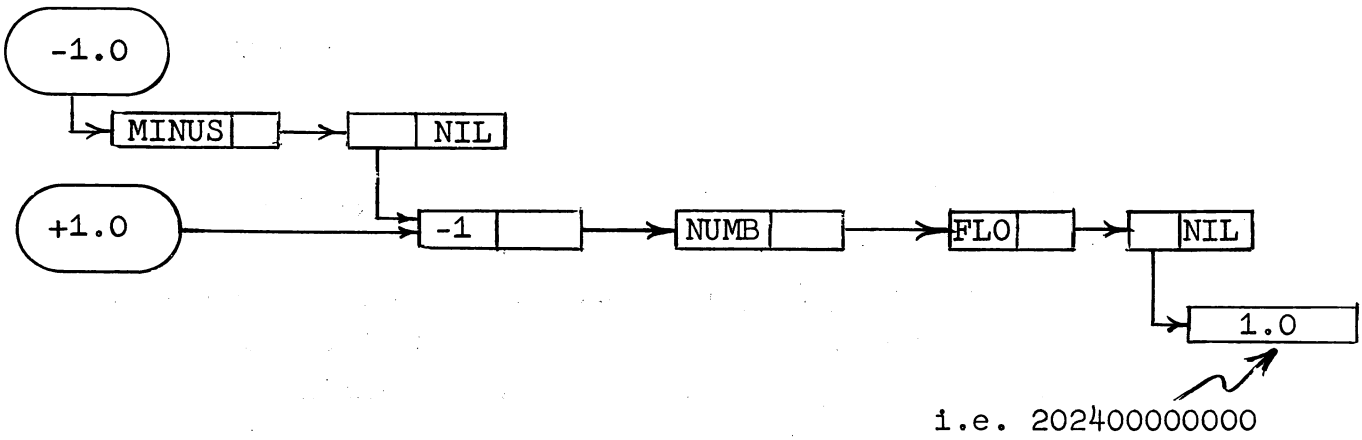
The special forms discussed at the end of Section 4.3 have FSUBR or FEXPR on their association lists.

Either attrib or define (see Chapter 4) 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. Comdef (see Section 4.6) puts SUBR on the association list of the function compiled, and the SUBR points to the TXL to the compiled subroutine.

Floating-Point Numbers

Floating-point numbers are listed in a separate list. A floating-point number is put on this list either as it is read in or as it is generated within a calculation. Neither this list nor the list of atomic symbols has any duplications. An association list is created for a floating-point number when the number is put on the list of numbers.

The association list for floating-point numbers contains the indicators NUMB and FLO (for floating-point), but the indicator PNAME is not put on the list until a request for printing the number is encountered. Only positive numerical values are put on the association lists for floating-point numbers; if a number is negative the association list for the number is preceded by the indicator MINUS, and the entry in the list of floating-point numbers points to the MINUS indicator. Thus we have



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, $\overline{\text{add}}$, to represent

the 2's (8's) complement of an octal address.*

Preceding all the individual association lists in the 704 LISP I system 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 in the list of atomic symbols. Thus for atom whose association list starts in 25146 the entry in the list of atomic symbols is

location	entry	representing
24657	053120052632	$\overline{25146}$, ,$\overline{24660}$

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

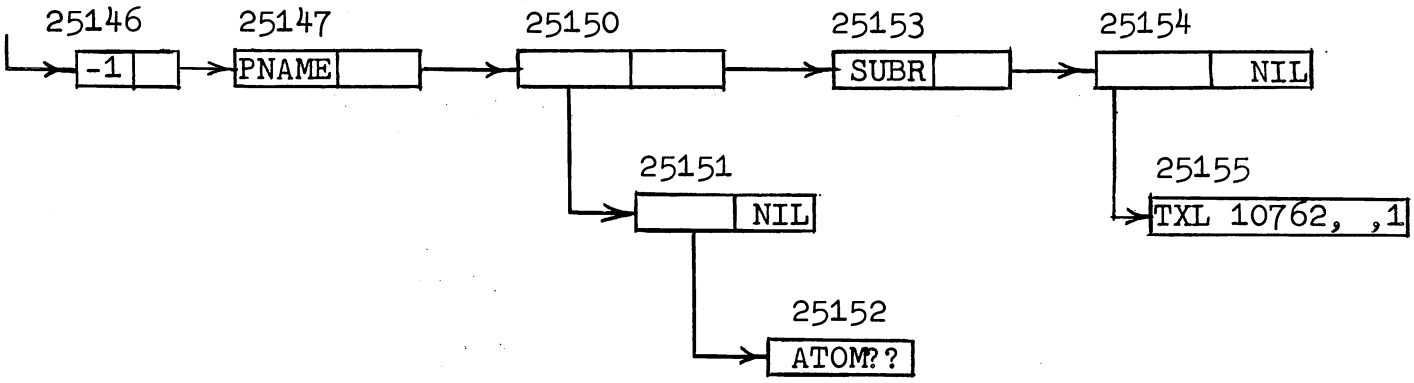
---

* The term "findex" is sometimes used for the 2's complement of an octal location.

Then for atom the association list looks as follows:

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

or, in list form,

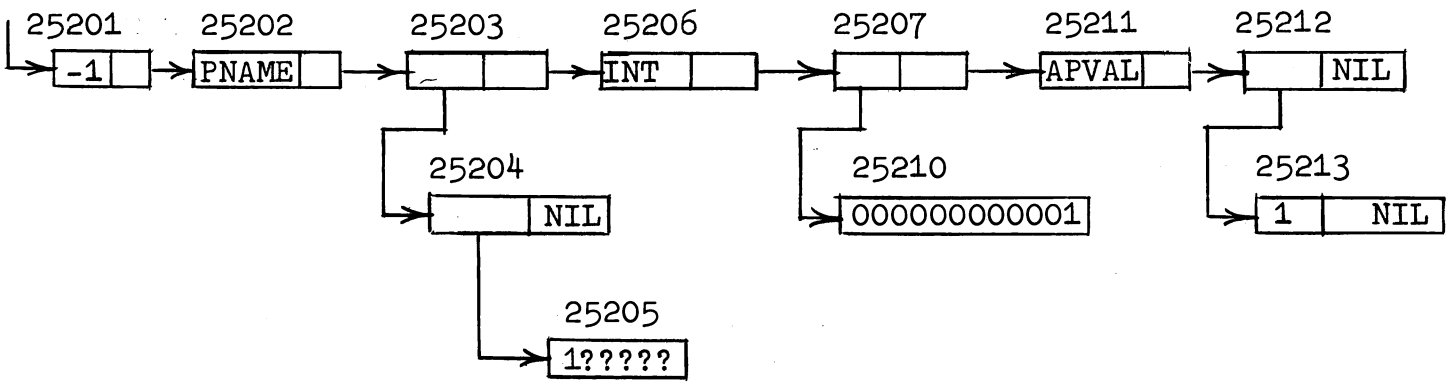


For the constant 1, the association list looks as follows:

location	entry		representing
25201	052576	077777	-1, ,$\overline{25202}$
25202	052575	051547	$\overline{26231}$, ,$\overline{25203}$
25203	052572	052574	$\overline{25204}$, ,$\overline{25206}$
25204	000000	052573	$\overline{25205}$, ,0
25205	017777	777777	BCD 1?????
25206	052571	052113	$\overline{25665}$, ,$\overline{25207}$

25207	052567	052570	$\overline{25210}$, ,$\overline{25211}$
25210	000000	000001	1, ,0
25211	052566	052644	$\overline{25134}$, ,$\overline{25212}$
25212	000000	052565	$\overline{25213}$, ,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.

6.3 The Free-Storage List and the Garbage Collector

At any given time only a part of the memory reserved for list structures will actually be in use for storing S-expressions. The remaining registers are arranged in a single list called the free-storage list. A certain register, FREE, in the program contains the location of the first register in this list. When a word is required to form some additional list structure, the first word on the free-storage list is taken and the number in register FREE is changed to become the location of the second word on the free storage list. No provision need be made for

the user to program the return of registers to the free-storage list.

This return takes place automatically whenever the free-storage list has been exhausted during the running of a LISP I program. 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. The active 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 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 is non-active; 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

---

* Special provision is made for the rare case when the sign of a register is already negative.

garbage collector does a linear sweep of the free storage area, collecting all registers with a positive sign into a new free-storage list, and restoring the original signs of the active registers. Partial experience has indicated that about a third of the running time is taken up by the garbage collector.

7. Example Exercises

Four exercises together with their solutions are given in this section. The first two exercises involve formulating LISP functions to treat list structures. In case the reader should like to try his hand at these before looking at the solution, the first two problems to be solved are:

(1) Formulate a function which will "collapse" a list structure, i.e. which will make a one-level list out of a multilevel list, so that, for example,

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

(2) Formulate a function which will reverse a list, so that, for example,

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

The third example shows how to define a length function which will operate more efficiently than the one defined in Section 4.4.

The fourth example involves applying a function to its arguments when the function is specified only at the time the APPLY operator is in control.

(1) Function to Collapse a List of Elements

The function formulated below uses the function append, which is described in Chapter 9 and also in Chapter 2. Roughly speaking append makes one list out of two, so that, for example,

append[(X,Y,Z);(P,Q)] = (X,Y,Z,P,Q)

The function collapse, is defined as follows:

$$\text{collapse}[\ell] = [\text{atom}[\ell] \rightarrow \text{cons}[\ell;\text{NIL}];\text{null}[\text{cdr}[\ell]] \rightarrow [\text{atom}[\text{car}[\ell]] \rightarrow \ell;\text{T} \rightarrow \text{collapse}[\text{car}[\ell]]];\text{T} \rightarrow \text{append}[\text{collapse}[\text{car}[\ell]];\text{collapse}[\text{cdr}[\ell]]]]$$

For this particular example a complete record of the cards punched for the run and of the computer results is given below.

In the run the APPLY operator operated on the define function to define collapse, and then applied the defined function to three test cases.

The following is a listing of the cards punched for the run:

```
      TST M948-371-P. FOX-EXERCISE 1
DEFINE
(((COLLAPSE,(LAMBDA,(L),(COND,
   ((ATOM,L),(CONS,L,NIL))
   ((NULL,(CDR,L)),
      (COND,((ATOM,(CAR,L)),L),(T,(COLLAPSE,(CAR,L)))))
   (T,(APPEND,(COLLAPSE,(CAR,L)),(COLLAPSE,(CDR,L))))
)))) ()
COLLAPSE ((((A,B),((C))),((D,(E,F)),(G),((H))))) ()
COLLAPSE ((A,(B,(C,(D,(E))),F,(G,(H,J))))) ()
COLLAPSE (((((A),B),C),D),E)) ()
   STOP))))))))))STOP
        FIN M948-371-P. FOX-EXERCISE 1
```

These cards were then submitted for a run, following the various directions as to deck format and so on given in Chapter 5. The results printed out during the running of the problem by the APPLY operator, were the following (where comments on the computer output have been added in square brackets):

TST M948-371-P. FOX-EXERCISE 1

```
APPLY OPERATOR AS OF SEPTEMBER 1, 1959
THE TIME IS NOW  2/16 1139.5

READ IN LISTS ...
DEFINE
(((COLLAPSE,(LAMBDA,(L),(COND,((ATOM,L),(CONS,L,NIL)),((NULL,(
CDR,X)),(COND,((ATOM,(CAR,L)),L),(T,(COLLAPSE,(CAR,L))))),(T,(
APPEND,(COLLAPSE,(CAR,X)),(COLLAPSE,(CDR,L)))))))))

COLLAPSE
((((A,B),((C))),((D,(E,F)),(G),((H)))))

COLLAPSE
((A,(B,(C,(D,(E))),F,(G,(H,J)))))

COLLAPSE
((((((A),B),C),D),E))

STOP

THE TIME IS NOW  2/16 1139.7
```

[At this point all the cards through the STOP card have been read in].

```
OBJECT LIST NOW IS ...
```

[Here the entire list of atomic symbols, including those just read in, is printed out].

```
THE TIME IS NOW  2/16 1139.7

 FUNCTION  APPLY(F,X,P) HAS BEEN ENTERED, ARGUMENTS..
DEFINE
(((COLLAPSE,(LAMBDA,(L),(COND,((ATOM,L),(CONS,L,NIL)),((NULL,(
CDR,L)),(COND,((ATOM,(CAR,L)),L),(T,(COLLAPSE,(CAR,L))))),(T,(
APPEND,(COLLAPSE,(CAR,L)),(COLLAPSE,(CDR,L)))))))))

END OF APPLY, VALUE IS ...
(COLLAPSE)
```

[The function collapse has now been defined by having EXPR followed by the defining S-expression put on its association list].

[The function collapse is now used on the three examples].

```
THE TIME IS NOW  2/16 1139.7

 FUNCTION  APPLY(F,X,P) HAS BEEN ENTERED, ARGUMENTS..
COLLAPSE
((((A,B),((C))),((D,(E,F)),(G),((H)))))

END OF APPLY, VALUE IS ...
(A,B,C,D,E,F,G,H)

THE TIME IS NOW  2/16 1139.8

 FUNCTION  APPLY(F,X,P) HAS BEEN ENTERED, ARGUMENTS..
COLLAPSE
((A,(B,(C,(D,(E))),F,(G,(H,J)))))

END OF APPLY, VALUE IS ...
(A,B,C,D,E,F,G,H,J)
```

```
THE TIME IS NOW  2/16 1139.8

 FUNCTION  APPLY(F,X,P) HAS BEEN ENTERED, ARGUMENTS..
COLLAPSE
((((((A),B),C),D),E))

END OF APPLY, VALUE IS ...
(A,B,C,D,E)
```

[The FIN card is read and the run terminates].

(2) Function to Reverse a List

Two different functions are shown which purport to reverse lists, but only the second definition is correct.

First Definition

$$rvrse[\ell] = [null[\ell] \rightarrow NIL; T \rightarrow cons[rvrse[cdr[\ell]]; cons[car[\ell]; NIL]]]$$

Second Definition

$$\begin{cases} rvde[\ell] = rev[\ell; NIL] \\ rev[j;k] = [null[j] \rightarrow k; T \rightarrow rev[cdr[j]; cons[car[j]; k]]] \end{cases}$$

To show the effect of these two definitions on a given list the following cards were punched and run:

```
DEFINE
(((RVRSE,(LAMBDA,(L),(COND,((NULL,L),NIL),
    (T,(CONS,(RVRSE,(CDR,L)),(CONS,(CAR,L),NIL))))))),
(RVDE,(LAMBDA,(L),(REV,L,NIL))),
(REV,(LAMBDA,(J,K),(COND,((NULL,J),K),
    (T,(REV,(CDR,J),(CONS,(CAR,J),K)))))))
()
RVRSE  ((A,B,C,D,E))  ()
RVDE   ((A,B,C,D,E))  ()
```

The results given for the two cases by the APPLY operator were respectively

RVRSE((A,B,C,D,E)) () = (((((NIL,E),D),C),B),A)

and

RVDE((A,B,C,D,E)) () = (E,D,C,B,A)

Thus only the second definition does the job intended; the first function, rvrse, erroneously creates a multilevel list.

(3) Function to Count the Number of Elements in a List

The function given in Section 4.4 for the length of a list is

$$\text{length}[\ell] = [\text{null}[\ell] \rightarrow 0.0; T \rightarrow \text{sum}[\text{length}[\text{cdr}[\ell]]; 1.0]]$$

This function will work correctly, but it is slow and inefficient in the sense that it must follow the recursion of length down to the end of the list before it starts counting. Then it counts out backwards through the list. The following pair of functions counts forwards through the list, and therefore operates faster:

```
lengthx[ℓ] = [lengthy[ℓ;0.0]]
lengthy[j;k] = [null[j] → k;T → lengthy[cdr[y];sum[
                1.0;k]]]
```

(4) Specification of a Function During Run Time

The following somewhat obscure example shows how one can leave the specification of a function until the time a program is run, and at that time specify it in a particular way. This cannot be done by the usual method of using the APPLY operation on a f;x;p triplet since the APPLY operator expects f to be already available. Instead one must use the APPLY operator on a triplet whose f is the function eval, as in

```
EVAL
((APPLY,(CAR,X),Y,NIL),
((X,(CADR,CDR)),(Y,((A,B,C)))))
()
```

Here the second argument of eval, the p-list is used to set the function (CAR,X) equal to CADR, and the result of the eval is

```
apply[cadr;(A,B,C);NIL] = B
```

8. Error Indications Given by LISP

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, 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 arguments than the APPLY operator is built to handle, (currently ten).

A2 FUNCTION OBJECT HAS NO DEFINITION -APP2-
I.e. the function represented by an atomic symbol has neither SUBR nor EXPR on its association list, nor is it paired with something on the p-list. The atomic symbol for the name of the function is printed out after the A2 print-out.

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 4.5) the program variable is not within this program, nor is it in some higher-level program in this run. The name of the program variable is printed out.

A7 SET 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 or APVAL1 (see Section 6.2) on its association list to signal its value, nor is the variable paired with something on the p-list. The name of the variable is printed out.

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 p-list. The name of the function is printed out.

A11 GO TO A POINT NOT LABELLED -INTER-
I.e. (see the program feature, Section 4.5) 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 arguments than the APPLY operator is built to handle, (currently ten).

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,NG)))

Errors due to computer inadequacies:

B1 OUT OF PUBLIC PUSH DOWN LIST -SAVE-
I.e. the program has run out of space allotted to the public push-down list.[1] Currently about 1000 registers are allotted, and, if they are used up, the recursion being done is either too "deep" for the given capacity or non-terminating.

B2 FREE STORAGE COUNTER PANIC STOP. PRESS START TO CONTINUE
There is a free storage counter, called the "CONS counter", associated with the LISP system. Every time a register is taken from free storage, the counter is increased 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 is using, and so on, or, for a program whose operation is known, how long it has been running. The counter is initially set to zero, and when it has reached 100,000, the computer stops after giving the error print out B2. When the start button has been pressed to restart the computer the counter continues counting, but no further stop occurs.

B3 DIVIDE ERROR -OCTAL TO DECIMAL CONVERTER-
This error arises only when the computer malfunctions or when some of the program has been written over.

---

1 See page 144 of the Quarterly Report referred to on the first page of Chapter 2.

Errors in list structures:

C2 OBJECT GIVEN TO DESC AS LIST
I.e. (see desc in Chapter 9) in desc[x;y], x is an atomic symbol 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; 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 list of dummy variables and their values is being appended to the p-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. This usually happens when the user gives a single dummy variable in a LAMBDA expression as an atomic symbol instead of as a list. The list is printed out following the error indication.

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 (F5), or that it is not 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 leniency allowed here has been inserted to permit a great deal of information to be gleaned from one run rather than stopping immediately 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:

01 NO INPUT DATA -OVERLORD-
Overlord is the LISP routine which is controlled by the direction cards, Error 01 arises when an end-of-file is 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 5.1, 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 8 or 9.

I_2 FIXED POINT OPERATION ATTEMPTED ON NON-FIXED-POINT-NUMBER OBJECT.

Errors during printing:

P1 PRIN1 ASKED TO PRINT NON-OBJECT
I.e. the printing program tried to print an item which was not an atomic symbol.

P2 PRIN1 ASKED TO PRINT UNPRINTABLE OBJECT
I.e. the printing program tried to print an atomic symbol which does 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.

R5 END OF FILE -RDA-
The tape or card-reader ran out of cards before the correct number of parentheses were read.

R6 NUMBER TOO LARGE IN CONVERSION
The conversion program can convert a floating-point number x of up to nine digits and an exponent, provided

$$|x| < 2^{128} \sim 10^{38}$$

9. Functions Available in LISP

In this section, a brief description is given of all the functions available in the LISP system as of March 1, 1960. A brief account is given for each function explaining the form of its argument, 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, and in other cases a program written in the notation of the program feature (Section 4.5) is given. The user of course need only give the function name and arguments--the M-expression or program are just included here as amplified description.

Since there are about ninety functions in the LISP system, they have been grouped into subgroups for easier reading. The function index of Section 9.5 provides an alphabetic index for looking up a function. Sections 9.1 and 9.2 include all the functions which are generally used. Section 9.1 discusses predicates, apply and eval, simple functions, pseudo-functions for defining other functions, functions to operate on lists, simple arithmetic functions, input-output functions, and the compiler functions. Section 9.2 consists of some special forms, such as cond and quote. Section 9.3 discusses some of the less frequently used functions and could be skipped at first reading. Section 9.4 contains rather specialized functions which have been relegated to the Supplementary LISP System. To find out how to use a function from this last category one must consult the local experts on LISP at M.I.T.

9.1 General Functions

Predicates:

atom[x] : machine language

The argument of atom is evaluated and the value of atom is true or false depending on whether the argument is or is not an atomic symbol. In list terminology (see Chapter 6) the argument is an atomic symbol if and only if car[x] = -1.

null[x] : machine language

The value of null is true if its argument is zero, and false otherwise.

and[x_1;x_2;...;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 or true respectively.

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 or false respectively.

not[x] : machine language

The value of not is true if its argument is false, and false if its argument is true.

eq[x;y] : machine language

The value of eq is true if x = y and false if x ≠ y.

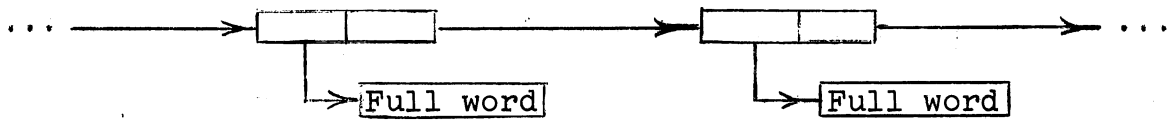
equal[x;y] : machine language

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

```
equal[x;y] = [eq[x;y] → T;null[x]∨null[y] → F;atom[x]∨atom[
             y] → F;T → equal[car[x];car[y]]∧equal[cdr[x];
             cdr[y]]]
```

eq1[x;y] : machine language

The function eq1 compares lists x and y of the following special two-level form, where each element of the top level points to a full word:



The value of eq1[x;y] is true if the two lists are identical, and false otherwise.

```
eq[x;y] = [eq[x;y] → T;null[x]∨null[y] → F;T → eq[cwr[
          car[x]];cwr[car[y]]∧eq1[cdr[x];cdr[y]]]
```

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

Apply and Eval:

apply[f;x;p] : machine language

The reader usually will not have occasion to use this function because essentially it gets done for him, but it operates as follows,

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

The forms involving LAMBDA, LABEL, and FUNARG are discussed in Section 4.3. The satellite functions for apply are the following:

app2[f;x;p] : machine language

The function app2 is the apply operator (see apply) in the case when f is atomic.

```
app2[f;x;p] = select[f;[CAR;caar[x]];[CDR;cdar[x]];[CONS;
              cons[car[x];cadr[x]]];[LIST;x];search[f;
              λ[[j];[eq[car[j];SUBR]∨eq[car[j];EXPR]]];
              λ[[j];[eq[car[j];SUBR] → app3[cadr[j];x];
              T → apply[cadr[j];x;p]];
              λ[[j];apply[car[sassoc[f;p;error]];x;p]]]]
```

See the function select.

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.

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, (b is the list of pairs of bound variables):

```
eval[e;b] = [atom[e] → search[e;λ[[j];eq[car[j];APVAL]∨
                 eq[car[j];APVAL1]];cadr;λ[[j];search[b;
                 λ[[j];eq[caar[j];e]];λ[[j];cadar[j]];
                 λ[[j];error]]]];

  atom[car[e]] →
    search[cdar[e];λ[[j];eq[car[j];FSUBR]∨eq[car[j];
    SUBR]∨eq[car[j];FEXPR]∨eq[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];eq[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 function evlis is used by the function eval above. 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[x;b] = maplist[x;λ[[j];eval[car[j];b]]]

Simple Functions:

car[x] : machine language

See Chapter 2.

Examples:

car[(A,B)] = A

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

cdr[x] : machine language

See Chapter 2.

Examples:

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

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

The APPLY operator can perform some multiple car's and cdr's, e.g.

cadddr[x] ≡ car[cdr[cdr[cdr[x]]]].

A depth of up to (and including) four a's or d's between the c and the r is permissible.

cons[x;y] : machine language

cons[x;y] = (x.y)

See Chapter 2.

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

Defining Functions:

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 def1 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[x] = deflist[x;EXPR]
```

deflist[x;PRO]

The function deflist is usually used to tie the EXPR-search required by define to the PRO-search in the program for def1.

```
deflist[x;PRO] = deflis1[x]
```

deflis1[x]

The argument of deflis1 is a list of pairs. The function deflis1 does a def1 of each pair and has as value the list of the first element of each pair.

Thus the functions deflist and deflis1 carry out the purpose of define by tying the PRO in def1 to EXPR and by carrying out the entire list of definitions (by the recursive function deflis1).

```
deflis1[x] = [null[x] → NIL;T → cons[def1[caar[x];
                                     cadar[x]];deflis1[cdr[x]]]]
```

def1[x;e]

The function def1 puts a pointer to the expression e on the rest of the list x from the point where it finds the attribute PRO. If no such attribute is found, one is created. The value of def1 is x. In practice, PRO is usually paired with EXPR on the p-list.

def1 is equivalent to the following (where the program feature has been used)

λ[[x;e];Ξ] where Ξ is the following program with no program variables.

rplaca[prop[x;PRO;λ[[()];cdr[attrib[x;list[pro;NIL]]]]];e]
return[x]

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

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 onto the end of an 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.

nconc[x;y] : machine language

The function nconc 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 nconc[x;y] has the program variable m and is as follows:

```
nconc[x;y] ≡ prog[[m];
             go[null[x] → RETU;T → CONT]
CONT         m = x
A1           go[null[cdr[x]] → A2;T → MORE]
MORE         m = cdr[m]
             go[A1]
A2           cdr[m] = y
RETN         return[x]
RETU         return[y]
```

Operations on Lists:

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

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

For the APPLY operator, if p is the associated p-list, and x represents the string of x_i's above,

list[x] = maplist[x;λ[[j];eval[car[j];p]]]

append[x;y] : machine language

The function append combines its two arguments into one new 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]]]
```

Note that append copies the top level of the first list; append is like nconc except that nconc does not copy its first argument.

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

maplist[x;f]

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

```
maplist[x;f] = [null[x] → NIL;T → cons[f[x];maplist[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[x;f] = [null[x] → NIL;T → append[f[x];mapcon[cdr[
              x];f]]]
```

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 f(x) is important.

The program for map[x;f] has the program variable m and is the following:

```
map[x;f] ≡ prog[[m];
           m = x
   LOOP    go[null[m] → END;T → CONT]
   CONT    f[m]
           m = cdr[m]
           go[LOOP]
   END     return[NIL]
```

conc[x;y] : special form

The function conc evaluates the items on the list x, and concatenates the values. The value of conc is the final concatenated list. The items on the list x must either be bound on the list of pairs, 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 $\widetilde{Y}$, then
conc[x;y] = (W,$\widetilde{Y}$,R)

```
conc[x;y] = mapcon[x;λ[[j];eval[car[j];y]]]
```

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 should not

be atomic symbols.

search[x;p;f;u] :

The function search looks through a 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 of one argument, x, is taken as the value of search (in this case x is, of course, NIL).

```
search[x;p;f;u] = [null[x] → u[x];p[x] → f[x];T → search[
                   cdr[x];p;f;u]]
```

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.

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

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.

Note that the following M-expression is different from that given in Chapter 2, though the result is the same.

```
sublis[x;y] = [null[x] → y;null[y] → NIL;T → search[x;
              λ[[j];equal[y;caar[j]]];λ[[j];cadar[j]];
              [atom[y] → y;T → cons[sublis[x;car[y]];
              sublis[x;cdr[y]]]]]]
```

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 false if z cannot be obtained as sublis[$\tilde{x}$;y] where $\tilde{x}$ is a completion of x obtained by substituting appropriate pairs (u,v) 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.

```
inst[x;y;z] = [null[x] → F;null[y]∨null[z] → F;atom[y] →
              search[x;λ[[j];eq[caar[j];y]];λ[[j];null[
              cdar[j]] → maplist[x;λ[[k];[not[eq[k;j]] →
              car[k];T → cons[y;cons[z;NIL]]]]];eq[cadar[
              j];z] → x;T → F];[eq[y;z] → x;T → F]];T →
              inst[inst[x;car[y];car[z]];cdr[y];cdr[z]]]
```

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 function u of no arguments is taken as the value of sassoc.

```
sassoc[x;y;u] = [null[y] → u[];eq[caar[y];x] → cdar[y];
                 T → sassoc[x;cdr[y];u]]
```

Arithmetic Functions:

sum[x;y] : machine language

The function sum computes the sum of two floating-point numbers. The arguments x and y can be quoted numbers, or can represent numbers by being paired with numbers on the p-list. Thus the following two cases are equivalent:

(1) (SUM,(QUOTE,3.0),(QUOTE,21.4))
(2) (SUM,(QUOTE,3.0),V) with (V,21.4) on the p-list.

prdct[x;y] : machine language

The function prdct computes the floating-point product of two floating-point numbers. The arguments x and y must be expressed in one of the forms given under the function sum.

expt[x,n] : machine language

The value of expt is the floating-point number x^n where x is a floating-point number and n is a floating-point positive or negative integer. The arguments must follow one of the forms described under the function sum.

Input-Output Functions

read : machine language

The function read of no arguments reads one list from cards or tape (depending on the sense-switch settings). The value of read is the list it has read.

print[x] :

The function print prints out (on-line or off-line depending on sense-switch settings) its argument x if x is a legal list structure, and malfunctions if it is not. If x is NIL a blank line is printed. The value of print is always zero.

In the following explanation of print, prin2 is a routine which places in the line to be printed up to six BCD characters when the characters are pushed to the left with the rest of the register filled in by the illegal character 77. The locations α, β, γ, and δ mentioned below contain respectively BCD representations of left-parenthesis, right-parenthesis, comma, and blank (space) in the form appropriate for prin2.

```
print[x] = prog[[];printa[x];terpri]
```

Here terpri (terminate print) is a machine language function which prints out the line of characters which have been placed there by the function prin2.

```
printa[x] ≡ [null[x] → prin2[δ];atom[x] → prin1[x];
             T → prog[[ℓ1];(printa subprogram)]]
```

where the (printa subprogram) is the following:

```
        ℓ1 = x
        prin2[α]
PL      printa[car[ℓ1]]
        null[cdr[ℓ1]] → go [END]
        prin2[γ]
        ℓ1 = cdr[ℓ1]
        go[PL]
END     prin2[β]
        return
```

prin1[x] : machine language

The routine prin1 prints the print name of the atomic symbol x. The value of prin1 is NIL. If x is not an atom, an error occurs.

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]];
                     printp1[cdr[x]]]
```

prints out

(PROPERTIES,OF,X)

and then goes to printp1 below.

printp1[x] (see printprop[x])

The function printp1, which does the printing of the properties is

```
printp1[x] = [null[x] → NIL;T → prog2[print[car[x]];
                search[(FLO,SUBR,FSUBR,PNAME,APVAL,INT);
                λ[[j];eq[car[j];car[x]]];
                λ[[k];printp1[cdr[cdr[x]]]];
                λ[[m];printp1[cdr[x]]]]]
```

punch*ℓ*[x] : machine language

The function punch*ℓ* writes the list x out onto tape 7 in BCD form for off-line punching. The resultant cards have the list information in consolidated form, i.e. extra blanks have been left out and commas inserted in the correct locations. All 72 columns of the card are used.

punchs[x] : machine language

The function punchs writes the list x out onto tape 7 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.

punchdef[x] :

The function punchdef writes the definition of the functions named in the list x out on tape 7 in BCD for off-line punching. If any item of the list x does not have EXPR or its association list, punchdef gives an error indication.

Error Function:

error[x] : machine language

The function error of one argument (the list x) causes an error print-out followed by a print-out of x. x can be given as NIL.

Compiler Functions:

comdef[x] :

The pseudo-function comdef compiles all the functions

named on the list x provided that the functions have EXPR and a definition on their association lists. See Section 4.6.

The function comdef uses the function compile (see below) as shown in the following:

```
comdef[x] = compile[mapcon[x;λ[[j];"program"]]]
```

where the "program" involved is as follows with program variables k and p:

```
          p = car[j]
          k = get[p;EXPR]
          return[null[k] → PROGA;T → PROGB]
PROGA     [null[get[p;SUBR]] → print[cons[p;(IS,NOT,DEFINED)]];
             T → print[cons[p;(HAS,ALREADY,BEEN,COMPILED)]]
          return[NIL]
PROGB     remprop[p;EXPR]
          return[eq[car[k];LABEL] → list[k];T → list[
             list[LABEL;p;k]]]
```

get[x;y] :

The function get is used by comdef above. It searches the list x for an item identical with y. When such an element is found the value of get is car of the rest of the list beginning immediately after the element.

compile[x] :

The pseudo-function compile can be used to compile functions not previously defined. The argument of compile is a list of function definitions, each one of which must be of the form

(LABEL,NAME,(LAMBDA,(list of free variables),Expression))

9.2 Special Forms

quote : machine language; special form

The value of a list beginning with QUOTE is always the rest of the list. Note that QUOTE must be used in expressions being evaluated whenever one wishes to avoid evaluating an item--for example (QUOTE,X) yields X itself rather than a value assigned to X by some other means.

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

The function cond has a variable number of arguments, each one of which is a pair of expressions of the form

(conditional,expression)

The propositions are evaluated until one is found that is true. The expression corresponding to this proposition is evaluated and taken as the value of the entire conditional.

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

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

The effect of label as the first element of an expression is described in Section 4.3 of this manual.

label[a;b] = eval[cadr[a];append[list[a];b]]

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];t][attrib[n;sublis[[[ϒ;v];
                [F;f];[P;formatp[v]]];(EXPR,(LAMBDA,V,(
                SUBLIS,(LIST,P),(CONST,F)))]]formatq[n;f;v]]]]]

   formatp[v] = [null[v] → Λ;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] : 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]

prog[pv;e_1;e_2;e_3;...;e_n] : machine language; special form

The program feature is discussed in Section 4.5. For interest we include here the program which the interpreter uses to work out a program given by the list pv of program variables, and the sequence, $e_1,e_2,\ldots,e_n$ of program statements.

In the following p is the usual p-list of pairs, and e stands for the entire list

$$(pv,e_1,e_2,e_3,\ldots,e_n)$$

so that car[e] = pv.

The program variables used below for the program for prog are t2, p2,pr, and r.

```
        t2 = maplist[car[e];λ[[j];list[car[j];NIL]]]
        p2 = append[t2;p]
        pr = cdr[e]
         r = pr
ML      go[atom[car[r]] → ADV;eq[caar[r];GO] → GO;
           eq[caar[r];RETURN] → RET;T → NORM]
```

```
NORM    eval[car[r];p2]
ADV     r = cdr[r]
        go[ML]
RET     return[eval[cadar[r];p2]]
GO      t2 = eval[cadar[r];p2]
        search[pr;λ[[j];eq[car[j];t2];λ[[j];setq[r;cdr[j]];
          λ[[j];error]]
        go[ML]
```

select[q;(q_1,e_1);(q_2,e_2);...;(q_n,e_n);e] : special form

The q_i's in select are evaluated in sequence from left to right until one is found such that

$$q_i = q,$$

and the value of select is the value of the corresponding e_i. If no such q_i is found the value of select is that of e.

9.3 Further Functions

rplaca[x;y] : machine language

This pseudo-function replaces the address part of the first argument x by the second argument y. I.e. y is stored in the address part of the location pointed to by the first argument. The value of rplaca is NIL.

rplacd[x;y] : machine language

This pseudo-function replaces the decrement part of the first argument x by the second argument y. I.e. y is stored in the decrement part of the location pointed to by the first argument. The value of rplacd is NIL.

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

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[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]]]]
```

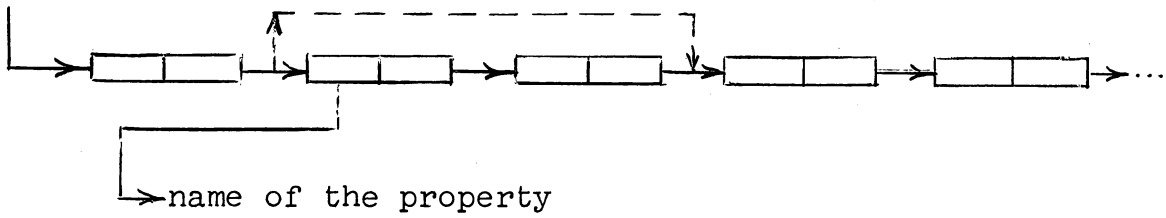
prop[x;y;u] : machine language

The function prop searches the list 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, where u is a function of no arguments.

```
prop[x;y;u] = [null[x] → u[];eq[car[x];y] → cdr[x];
               T → prop[cdr[x];y;u]]
```

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[x;p] = remp1[x],

where,

```
remp1[x] = [null[x]∧null[cdr[x]] → NIL;cadr[x] = p →
            prog2[rplacd[x;[null[cddr[x]] → NIL;T →
            cdddr[x]]];remp1[x]];T → remp1[cdr[x]]] ,
```

where prog2[u;v] has the value v, and uses u to effect an action.

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 4.5). The program variables are initially bound to null lists.

Note that (in S-expression form),

(SET,(QUOTE,V),e) ≡ (SETQ,V,e)

intv[x] : machine language; special function

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

```
intv[x] = caar[prop[cdr[x];INT;error]]
```

flval[x] : machine language

The function flval finds the address of the floating-point number on the association list x. The value of flval is the address of the floating-point number.

The program for flval is

```
flval[x] ≡ prog[[];
        not[atom[x]] → return[error]
B1      null[cdr[x]] → return[error]
        x = cdr[x]
        car[x] ≠ FLO → go[B1]
        return[cadr[x]]
```

tsflot[x] : machine language

The proposition tsflot is true if the association list for x contains FLO pointing to a floating-point number. Otherwise the proposition is false.

The program for tsflot is

```
tsflot ≡ prog[[];
        not[atom[x]] → return[F]
B1      null[x] → return[F]
        car[x] = FLO → return[T]
        x = cdr[x]
        go[B1]
```

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 resets the counter to zero.

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.

uncount : machine language

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

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.

compsrch[x;d;f;u] :

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

The program for compsrch has the program variable v and is the following:

```
compsrch[x;d;f;u] ≡ prog[[v];
        v = gensym              (see the function gensym)
        return[def1[x;[searchf[x;v;subst[v;car[bndv[d]];
          form[d]];subst[v;car[bndv[f]];form[f]];form[u]]]]]
```

In this program the PRO of def1 has been tied to EXPR on the p-list.

Note that compsrch avoids the rebinding of d and f by substituting the uniquely generated atomic symbol for v into d and f.

The functions searchf, bndv, and form are described below.

As an example of the use of compsrch, assume that a function, called FINDNAME, 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, on the other hand, by using COMPSRCH.

```
DEFINE ((
(FINDNAME,(LAMBDA,(L),(SEARCH,L,
     (FUNCTION,(LAMBDA,(J),(EQ,(CAR,J),(QUOTE,PNAME)))),
     (FUNCTION,(LAMBDA,(J),(CADR,J))),
     (FUNCTION,ERROR)
     )))  ))
or
COMPSRCH
(FINDNAME,
(LAMBDA,(J),(EQ,(CAR,J),(QUOTE,PNAME))),
(LAMBDA,(J),(CADR,J)),
(LAMBDA,(),ERROR),
()
```

bndv[x] :

This function has been introduced into the function compsrch as a function whose form later may change.

bndv = cadr[x]

form[x] :

This function has been introduced into the function compsrch as a function whose form later may change.

form[x] = caddr[x]

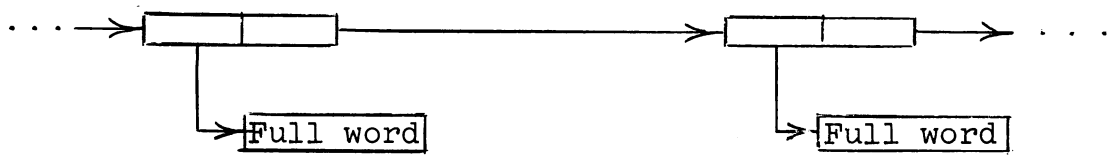
searchf[x;v;p;f;u] :

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

```
searchf[x;v;p;f;u] = 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)))))]]]
```

cp1 : machine language

The function cp1 copies into free storage a list of the following special two-level form (see the function eq1), where each element of the top level points to a full word.



The value of cp1 is the location of the copied list.

```
cp1[x] = [null[x] → NIL;T → cons[consw[cwr[car[x]]];
          cp1[cdr[x]]]]
```

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

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.

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.

tracklist[x] : See Section 5.2

The function tracklist 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. The value of tracklist is a list of the routines it will be able to trace.

makcblr[x] :

This function ("make car or cdr abler") uses the function desc 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 makcblr takes this list and on the association list of the first element of each pair puts EXPR followed by the lambda expression,

λ[[j];desc[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 def1 to EXPR.

```
makcblr[x] = funarg[λ[j];maplist[j;λ[[k];def1[car[car[k]];
             subst[car[cdr[car[k]]];WAY;
             (LAMBDA,(J),(DESC,(QUOTE,WAY),K))]]]]];
             ((PRO,EXPR))]
```

prog2[x;y] :

The value of prog2 is always the value of the second argument y, but the expression for the first argument x is evaluated first. x generally is used to effect some action. Cf. the use of prog2 in the function printprop.

9.4 Functions on the Supplementary LISP System

compab[x;y;z] : machine language

The function compab is a predicate whose value is true if the absolute value of the difference between x and y is less than z, and whose value is false otherwise. x, y, and z are all floating-point numbers.

greater[p;q] : machine language

The function greater is a predicate whose value is true if the fifteen-bit quantity p is greater than the fifteen-bit quantity q. Otherwise, the value of greater is false. 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.

larger[x;y] : machine language

The predicate larger is true if list x is larger than list y, and false 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[x;y] = [null[x] → F;null[y] → T;atom[x]∧atom[y] →
               greater[x;y];atom[x] → F;atom[y] → T;
               larger[car[x];car[y]] → T;larger[car[y];
               car[x]] → F;T → larger[cdr[x];cdr[y]]]
```

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)
(PLUS,A,B,C); (any number (>1) of arguments)
(MINUS,X)
(POWER,X,Y) ≡ X^Y
(RECIP,X) ≡ $\frac{1}{X}$

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, M.I.T., August, 1959.

distrb[e;p] : machine language

The function dstrb 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+2) distributes to
(xw)(y+z+b)+(2x)(y+z+b)+(aw)(y+z+b)+(2a)(y+z+b)

For the correct notation for sums and products in LISP see above under the function smplfy.

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.

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 p-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,A12,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 to 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, M.I.T., August, 1959.

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×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, Daniel J., "Symbolic Circuit Analysis with the 704 Electronic Calculator", S.B. Thesis, Course VI, M.I.T., June, 1959.

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

9.5 Alphabetic Index to Functions

GENERAL INDEX

GENERAL INDEX

GENERAL INDEX

GENERAL INDEX

GENERAL INDEX

GENERAL INDEX