

LISP 1.5

PROGRAMMER'S MANUAL

July 14, 1961

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

LISP 1.5

PROGRAMMER'S MANUAL

July 14, 1961

Artificial Intelligence Group

J. McCarthy

M. Minsky

P. Abrahams

R. Brayton

D. Edwards

L. Hodes

D. Luckham

M. Levin

D. Park

T. Hart

COMPUTATION CENTER

and

RESEARCH LABORATORY OF ELECTRONICS
Massachusetts Institute of Technology
Cambridge, Massachusetts

Preface

LISP 1.5 is a programming system for the IBM 709 and 7090. 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 1.5 as of July 1961. The LISP system is based on the theory of recursive functions of symbolic expressions. An understanding of this theory is important in order to use LISP efficiently. The necessary background is supplied in Chapter 1. If this section of the manual is studied carefully, many properties of the LISP interpreter will be seen to follow automatically.

Chapters 2 through 6 contain the basic information necessary to program in LISP. This includes the interpreter, the compiler, the program feature, the arithmetic operations, and some information on running and debugging LISP programs.

Chapter 7 is a description of the way in which list structure is stored and modified in the computer. An understanding of this will enable the user to devise more powerful programming methods.

A complete listing of LISP functions and detailed information about various aspects of the system are included in the appendices.

This manual will be supplemented at various times by memoranda.

Acknowledgements

This manual was written by M. Levin starting from the LISP I Programmer's Manual by P. Fox.

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

The interpreter was written by S. Russell starting a preliminary version by McCarthy.

The print and read programs were written by McCarthy and K. Maling.

The garbage collector and the arithmetic features were written by D. Edwards.

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

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

The secretarial work was done by E. Hottel.

Contents

1.	Recursive Functions of Symbolic Expressions	3
1.1	Functions and Function Definitions	3
1.2	Symbolic Expressions	10
2.	The LISP Interpreter System	22
2.1	Variables	24
2.2	Constants	26
2.3	Functions	26
2.4	Machine Language Functions	27
2.5	Special Forms	28
2.6	Programming for the Interpreter	28
2.7	Functions Available within the System	30
3.	Arithmetic in LISP	31
3.1	Reading and Printing Numbers	31
3.2	Arithmetic Functions and Predicates	33
3.3	Programming with Arithmetic	35
3.4	The Array Feature	36
4.	The Program Feature	38
5.	The Compiler	41
5.1	The Compiler Modes	43
5.2	Sample Compiler Program	44
6.	Running the LISP System	48
6.1	Preparing a Card Deck	48
6.2	Tracing	49
6.3	Error Diagnostics	50
6.4	The Cons Counter and Errorset	54
7.	List Structures	56
7.1	Representation of List Structure	56
7.2	Construction of List Structure	59
7.3	Property Lists	60
7.4	List Structure Operators	64
7.5	The Free Storage List and the Garbage Collector ..	66

(Contents -- cont'd)

Appendix A	Functions in the LISP System
Appendix B	The Interpreter
Appendix C	The Compiler and Assembler
Appendix D	Overlord
Appendix E	LISP 1.5 Input-Output
General Index

1. Recursive Functions of Symbolic Expressions¹

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

1.1 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 Most of this chapter is taken from "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", by John McCarthy, Communications of the Association for Computing Machinery, Vol. 3, No. 4, April, 1960

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

$$\begin{aligned}x < y \\(x < y) \wedge (b = c) \\x \text{ is prime}\end{aligned}$$

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

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

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, \dots , otherwise if p_n then e_n ," or " p_1 yields e_1, \dots, p_n yields e_n ."

We now give the rules for determining whether the value of $(p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n)$ 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.

$$(1 < 2 \rightarrow 4, 1 \geq 2 \rightarrow 3) = 4$$

$$(2 < 1 \rightarrow 4, 2 > 1 \rightarrow 3, 2 \geq 1 \rightarrow 2) = 3$$

$$(2 < 1 \rightarrow 4, T \rightarrow 3) = 3$$

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

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

$$(2 < 1 \rightarrow 3, 4 < 1 \rightarrow 4) \text{ is undefined}$$

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

$$|x| = (x < 0 \rightarrow -x, T \rightarrow x)$$

$$\delta_{ij} = (i = j \rightarrow 1, T \rightarrow 0)$$

$$\text{sgn}(x) = (x < 0 \rightarrow -1, x = 0 \rightarrow 0, T \rightarrow 1)$$

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, $\text{gcd}(m,n)$, of two positive integers m and n is computed by means of the Euclidean algorithm. This algorithm is expressed by the recursive function definition:

$$\text{gcd}(m,n) = (m \neq n \rightarrow \text{gcd}(n,m), \text{rem}(n,m)=0 \rightarrow m, T \rightarrow \text{gcd}(\text{rem}(n,m),m))$$

where $\text{rem}(n,m)$ 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

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

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 defined, the connectives \vee and \wedge 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

¹A. 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 ϕ is a form in variables x_1, \dots, x_n , then $\lambda((x_1, \dots, x_n), \phi)$ will be taken to be the function of n variables whose value is determined by substituting the arguments for the variables x_1, \dots, x_n in that order in ϕ 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

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

into

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

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, \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. Thus we can write

label(sqrt, $\lambda((a, x, \epsilon), (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow \text{sqrt}(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon)))$)
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.

1.2 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 evalquote 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 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 $(e_1 \circ e_2)$.

Examples of S-expressions are

AB
(A·B)
((AB·C)·D)

An S-expression is then either an atom, or 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, \dots, m_n)

is represented by the S-expression

$(m_1 \circ (m_2 \circ (\dots (m_n \circ \text{NIL}) \dots)))$

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 $(m \circ \text{NIL})$.
2. (m_1, \dots, m_n) stands for $(m_1 \circ (\dots (m_n \circ \text{NIL}) \dots))$.
3. $(m_1, \dots, m_n \circ x)$ stands for $(m_1 \circ (\dots (m_n \circ x) \dots))$

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

$((AB,C),D)$ for $((AB \cdot (C \cdot NIL)) \cdot (D \cdot NIL))$
 $((A,B),C,D \cdot E)$ for $((A \cdot (B \cdot NIL)) \cdot (C \cdot (D \cdot E)))$

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

$car[x]$
 $car[cons[(A \cdot B);x]]$

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[(e₁·e₂)] = e₁. 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[(e₁·e₂)] = e₂. 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[e₁;e₂] = (e₁·e₂). 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

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

We have

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

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

$$\begin{aligned} ff[(A \cdot B) \cdot C] &= [atom[(A \cdot B) \cdot C] \rightarrow ((A \cdot B) \cdot C); T \rightarrow ff[car[(A \cdot B) \cdot C]]] \\ &= [F \rightarrow ((A \cdot B) \cdot C); T \rightarrow ff[car[(A \cdot B) \cdot C]]] \\ &= [T \rightarrow ff[car[(A \cdot B) \cdot C]]] \\ &= ff[car[(A \cdot B) \cdot C]] \\ &= ff[(A \cdot B)] \\ &= [atom[(A \cdot B)] \rightarrow (A \cdot B); T \rightarrow ff[car[(A \cdot B)]]] \\ &= [F \rightarrow (A \cdot B); T \rightarrow ff[car[(A \cdot B)]]] \\ &= [T \rightarrow ff[car[(A \cdot B)]]] \end{aligned}$$

$= \text{ff}[\text{car}[(A \cdot B)]]$
 $= \text{ff}[A]$
 $= [\text{atom}[A] \rightarrow A; T \rightarrow \text{ff}[\text{car}[A]]]$
 $= [T \rightarrow A; T \rightarrow \text{ff}[\text{car}[A]]]$
 $= A$

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

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

As an example, we have

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

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

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

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

- (i) $\text{car}[(m_1, m_2, \dots, m_n)] = m_1$
- (ii) $\text{cdr}[(m_1, m_2, \dots, m_n)] = (m_2, \dots, m_n)$
- (iii) $\text{cdr}[(m)] = \text{NIL}$
- (iv) $\text{cons}[m_1; (m_2, \dots, m_n)] = (m_1, m_2, \dots, m_n)$
- (v) $\text{cons}[m; \text{NIL}] = (m)$

We define

$$\text{null}[x] = \text{eq}[x; \text{NIL}]$$

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[$e_1; e_2; \dots; e_n$] for cons[$e_1; \text{cons}[e_2; \dots; \text{cons}[e_n; \text{NIL}] \dots]$]. This function gives the list, (e_1, \dots, e_n) , as a function of its elements.

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

1. append[x;y]

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

An example is

$$\text{append}[(A,B); (C,D,E)] = (A,B,C,D,E)$$

2. among[x;y]

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

$$\text{among}[x;y] = \sim \text{null}[y] \wedge [\text{equal}[x; \text{car}[y]] \vee \text{among}[x; \text{cdr}[y]]]$$

3. pairlis[x;y;a]

This function gives the list of pairs of corresponding elements of the lists x and y, and appends this to the list a. The resultant list of pairs, which is like a table with two columns, is called an association list. We have

$$\text{pairlis}[x;y;a] = [\text{null}[x] \rightarrow a; T \rightarrow \text{cons}[\text{cons}[\text{car}[x]; \text{car}[y]]; \text{pairlis}[\text{cdr}[x]; \text{cdr}[y]; a]]]$$

An example is

$$\text{pairlis}[(A,B,C); (U,V,W); ((D.X), (E.Y))] = ((A.U), (B.V), (C.W), (D.X), (E.Y))$$

4. assoc[x;a]

If a is an association list such as the one formed by pairlis in the above example, then assoc will produce the first pair whose first term is x. Thus it is a table searching function.

We have

$$\text{assoc}[x;a] = [\text{equal}[\text{caar}[a]; x] \rightarrow \text{car}[a]; T \rightarrow \text{assoc}[x; \text{cdr}[a]]]$$

An example is

```
assoc[B; (A.(M,N)), (B.(CAR,X)), (C.(QUOTE,M)), (C.(CDR,X)))]
= (B.(CAR,X))
```

5. sublis[a;y]

Here *a* is assumed to be an association list of the form $((u_1.v_1), \dots, (u_n.v_n))$, where the *u*'s are atomic, and *y* is any S-expression. What sublis does, is to treat the *u*'s as variables when they occur in *y*, and to substitute the corresponding *v*'s from the pair list. In order to define sublis, we first define an auxiliary function. We have

```
sub2[a;z] = [null[a] → z; eq[caar[a];z] → cdar[a]; T →
             sub2[cdar[a];z]]
```

and

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

An example is

```
sublis[((X.SHAKESPEARE), (Y.HAMLET)); (X,WROTE,Y)]
= (SHAKESPEARE,WROTE,HAMLET)
```

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 ξ by ξ^* .

1. If ξ is an S-expression ξ^* is (QUOTE, ξ).
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 $f[e_1; \dots; e_n]$ is translated to $(f^*, e_1^*, \dots, e_n^*)$.
Thus $\{\text{cons}[\text{car}[x]; \text{cdr}[x]]\} * \text{is } (\text{CONS}, (\text{CAR}, X), (\text{CDR}, X))$.
4. $[p_1 \rightarrow e_1; \dots; p_n \rightarrow e_n] * \text{is } (\text{COND}, (p_1^*, e_1^*), \dots, (p_n^*, e_n^*))$.
5. $\lambda[(x_1; \dots; x_n);] * \text{is } (\text{LAMBDA}, (x_1^*, \dots, x_n^*), *)$.
6. $\text{label}[a;] * \text{is } (\text{LABEL}, a^*, *)$.

f. The Universal S-Function evalquote

There is an S-expression evalquote[fn;x] with the property that if $fn = f *$ where f is a function form, and if $x = (\text{arg}_1, \dots, \text{arg}_n)$, then evalquote[fn;x] = $f[\text{arg}_1; \dots; \text{arg}_n]$ unless both are undefined. For example,

$$\begin{aligned} & \lambda[(x;y); \text{cons}[\text{car}[x]; y]][(A,B); (C,D)] \\ & = \text{evalquote}[(\text{LAMBDA}, (X,Y), (\text{CONS}, (\text{CAR}, X), Y)); ((A,B), (C,D))] \\ & = (A,C,D) \end{aligned}$$

The S-function evalquote is defined by
evalquote[fn;x] = apply[fn;x;NIL]

where

$$\begin{aligned} \text{apply}[fn;x;a] &= [\text{eq}[fn;NIL] \rightarrow \text{NIL}; \\ & \text{atom}[fn] \rightarrow [\text{eq}[fn;CAR] \rightarrow \text{caar}[x]; \\ & \quad \text{eq}[fn;CDR] \rightarrow \text{cdar}[x]; \\ & \quad \text{eq}[fn;CONS] \rightarrow \text{cons}[\text{car}[x]; \text{cadr}[x]]; \\ & \quad \text{eq}[fn;ATOM] \rightarrow \text{atom}[\text{car}[x]]; \\ & \quad \text{eq}[fn;EQ] \rightarrow \text{eq}[\text{car}[x]; \text{cadr}[x]]; \\ & \quad T \rightarrow \text{apply}[\text{eval}[fn;a]; x;a]]; \\ \text{eq}[\text{car}[fn]; \text{LAMBDA}] & \rightarrow \text{eval}[\text{caddr}[fn]; \text{pairlis}[\text{cadr}[fn]; x;a]]; \\ \text{eq}[\text{car}[fn]; \text{LABEL}] & \rightarrow \text{apply}[\text{caddr}[fn]; x; \text{cons}[\text{con}[\text{cadr}[fn]; \\ & \quad \text{caddr}[fn]]; a]] \end{aligned}$$

```
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
  [atom[car[e]→[
    [eq[car[e];QUOTE] → cadr[e];
    [eq[car[e];COND] → evcon[cdr[e];a];
    T → apply[car[e];evlis[cdr[e];a];a]]
  T → apply[car[e];evlis[cdr[e];a];]]
```

pairlis and assoc were previously defined.

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

and

```
evlis[m;a] = [cons[eval[car[m];a];evlis[cdr[m];a]]];
```

We now explain a number of points about these definitions.

1. apply labels a function by pairing its name with its definitions and adding this to the pair list. If a function begins with LAMBDA, then apply pairs the bound variables with the arguments and gives the result to eval. If a function is atomic, then apply looks for its definition on the pair list. This would be the case for successive recursions of a labeled function. In the remaining cases, the function is evaluated.

2. eval[e;a] has two arguments, an expression e to be evaluated, and an association list a. 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 a.

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,e₁,e₂) or (COND,(p₁,e₁),..., (p_n,e_n)), or (CAR,e) or (CDR,e) or (CONS,e₁,e₂) or (f,e₁,...,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, $(p_1, e_1), \dots, (p_n, e_n)$) 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, \dots, e_n) we evaluate the e_1 's and give the result to apply as a list of arguments.

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

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

$$\text{maplist}[x;f] = [\text{null}[x] \rightarrow \text{NIL}; T \rightarrow \text{cons}[f[x]; \text{maplist}[\text{cdr}[x];f]]]$$

The usefulness of `maplist` is illustrated by formulas for the partial derivative with respect to x of expressions involving sums and products or 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, \dots, e_n are allowed expressions, $(PLUS, e_1, \dots, e_n)$ and $(TIMES, e_1, \dots, e_n)$ are also, and represent the sum and product respectively, or e_1, \dots, 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 $(TIMES, X, (PLUS, X, A), Y)$, the conventional algebraic notation for which is $X(X+A)Y$.²

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

$$\begin{aligned} \text{diff}[y;x] = & [\text{atom}[y] \rightarrow [\text{eq}[y;x] \rightarrow 1; T \rightarrow 0]; \text{eq}[\text{car}[y]; \\ & \text{PLUS}] \rightarrow \text{cons}[\text{PLUS}; \text{maplist}[\text{cdr}[y]; \lambda[[z]; \text{diff}[\\ & \text{car}[z]; x]]]]; \text{eq}[\text{car}[y]; \text{TIMES}] \rightarrow \text{cons}[\text{PLUS}; \text{maplist}[\\ & \text{cdr}[y]; \lambda[[z]; \text{cons}[\text{TIMES}; \text{maplist}[\text{cdr}[y]; \lambda[[w]; \text{eq}[\\ & z; w] \rightarrow \text{car}[w]; T \rightarrow \text{diff}[\text{car}[w]; x]]]]]]]] \end{aligned}$$

The derivative of the allowed expression, as computed by this formula, is $(PLUS, (TIMES, 1, (PLUS, X, A), Y), (TIMES, X, (PLUS, 1, 0), Y), (TIMES, X, (PLUS, X, A), 0))$

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

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

The function search is used to search a list 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.

¹For more exact information on arithmetic functions see Section 2.4.

²In LISP 1.5 actual numbers can be used in these forms, and the corresponding arithmetic function will be performed. This is described in Section 2.4.

2. The LISP Interpreter System

The following example is a LISP program that defines three functions union, intersection, and member, and then applies these functions to some test cases. The functions union and intersection are to be applied to "sets", each set being represented by a list of atomic symbols. The functions are defined as follows. Note that they are all recursive, and both union and intersection make use of member.

```
member[a;x] = [null[x] → F;eq[a;car[x]] → T;T →
member[a;cdr[x]]]
```

```
union[x;y] = [null[x] → y;member[car[x];y] → union
[cdr[x];y]|T → cons[car[x];union[cdr[x];y]]]
```

```
intersection[x;y] = [null[x] → NIL;member[car[x];y]
→ cons[car[x];intersection[cdr[x];y]];T →
intersection[cdr[x];y]]
```

To define these functions, we use the pseudo-function define. The program looks like this:

```
DEFINE ((
(MEMBER (LAMBDA (A X) (COND ((NULL X) F)
((EQ A (CAR X)) T) (T (MEMBER A (CDR X))))))
(UNION (LAMBDA (X Y) (COND ((NULL X) Y) ((MEMBER
(CAR X) Y) (UNION (CDR X) Y)) (T (CONS (CAR X)
(UNION (CDR X) Y))))))
(INTERSECTION (LAMBDA (X Y) (COND ((NULL X) NIL)
((MEMBER (CAR X) Y) (CONS (CAR X) (INTERSECTION
(CDR X) Y))) (T (INTERSECTION (CDR X) Y))))))
))
INTERSECTION ((A1 A2 A3) (A1 A3 A5))
UNION ((X Y Z) (U V W X))
```

This program contains three distinct functions for the LISP interpreter. The first function is the pseudo-function define. A pseudo-function is a function that is executed for its effect on the system in core memory as well as for its value. Define causes these functions to be defined and available within the system. Its value is a list of the functions defined, in this case (MEMBER UNION INTERSECTION).

The value of the second function is (A1 A3). The value of the third function is (Y Z U V W X). An inspection of the way in which the recursion is carried out will show why the "elements" of the "set" appear in just this order.

The following are some elementary rules for writing LISP 1.5 programs.

1. A program for execution in LISP consists of a sequence of doublets. The first list or atomic symbol of each doublet is interpreted as a function. The second is a list of arguments for the function.

2. There is no particular card format for writing LISP. Columns 1-72 of any number of cards may be used. Card boundaries are ignored. The format of the above example, including indentation, was chosen merely for ease of reading.

3. A blank is the equivalent of a comma. Any number of blanks and/or commas can occur at any point in a program except in the middle of an atomic symbol.

4. Do not use the forms (QUOTE T), (QUOTE F), and (QUOTE NIL). Use T, F, and NIL instead.

5. Atomic symbols should begin with alphabetical characters to distinguish them from numbers.

6. Dot notation may be used in LISP 1.5. Any number of blanks before or after the dot will be ignored.

7. Dotted pairs may occur as elements of a list, and lists may occur as elements of dotted pairs. For example

((A.B) X (C.(E F G)))

is a valid S-expression. It could also be written

((A.B).(X.((C.(E.(F.(G.NIL))))).NIL))) or
((A.B) X (C E F G))

8. A form of the type (A B C.D) is an abbreviation for (A.(B.(C.D))). Any other mixing of commas (spaces) and dots on the same level is an error, e.g. (A.B C).

9. A selection of basic functions is provided with the LISP system. Other functions may be introduced by the programmer. The order in which functions are introduced is not significant. Any function may make use of any other function.

2.1 Variables

A variable is a symbol that is used to represent an argument of a function. Thus one might write "a + b where a = 341 and b = 216." In this situation no confusion can result and all will agree that the answer is 557. In order to arrive at this result, it is necessary to substitute the actual numbers for the variables, and then add the two numbers (on an adding machine for instance).

One reason why there is no ambiguity in this case is that "a" and "b" are not acceptable inputs for an adding machine, and it is therefore obvious that they merely represent the actual arguments. In LISP, the situation can be much more complicated. An atomic symbol may be either a variable or an actual argument. To further complicate the situation, a part of an argument may be a variable when a function inside another function is evaluated. The intuitive approach is no longer adequate. An understanding of the formalism in use is

necessary to do any effective LISP programming.

Lest the prospective LISP user be discouraged at this point, it should be pointed out that nothing new is going to be introduced here. This section is intended to reinforce the discussion of Chapter 1. Everything in this section can be derived from the rule for translating M-expressions into S-expressions, or alternately everything in this section can be inferred from the universal function evalquote of Chapter 1.

The formalism for variables in LISP is the Church-lambda notation. The part of the interpreter that binds variables is called apply. When apply encounters a function beginning with LAMBDA, the list of variables is paired with the list of arguments and added to the front of the a-list. During the evaluation of the function, variables may be encountered. They are evaluated by looking them up on the a-list. If a variable has been bound several times, the first or most recent value is used. The part of the interpreter that does this is called eval. The following example will illustrate this discussion. Suppose the interpreter is given the following doublet:

```
fn: (LAMBDA (X Y) (CONS X Y))
x:  (A B)
```

Evalquote will give these arguments to apply. (Look at the universal function of Chapter 1.)

```
apply[(LAMBDA (X Y) (CONS X Y));(A B);NIL]
```

Apply will bind the variables and give the function and a-list to eval.

```
eval[(CONS X Y);((X.A) (Y.B))]
```

Eval will evaluate the variables and give it to cons.
cons[A;B] = (A.B)

The actual interpreter skips one step required by the universal function, namely, apply[CONS;(A B)].

2.2 Constants

It is sometimes assumed that a constant stands for itself as opposed to a variable which stands for something else. This is not a very workable concept since the student learning calculus is taught to represent constants by a,b,c... and variables by x,y,z.... It seems more reasonable to say that one variable is more nearly constant than another if it is bound at a higher level and changes value less frequently.

In LISP, a variable remains bound within the scope of the LAMBDA that binds it. When a variable always has a certain value regardless of the current a-list, it will be called a constant. This is accomplished by means of the property list¹ (p-list) of the variable symbol. Every atomic symbol has a p-list. When the p-list contains the indicator APVAL or APVAL1, then the symbol is a constant and the next item on the list is the value. Eval searches p-lists before a-lists when evaluating variables, thus making it impossible to bind constants effectively.

Constants can be made by the programmer. To make the variable X always stand for (A B C D) use the pseudo-function cset.

```
CSET (X (A B C D))
```

An interesting type of constant is one that stands for itself. NIL is an example of this. It can be evaluated repeatedly and will still be NIL. T,F,NIL, and other constants cannot be used as variables.

2.3 Functions

When a symbol stands for a function, the situation is

1. Property lists are discussed in Chapter 7.

similar to that in which a symbol stands for an argument. When a function is recursive, it must be given a name. This is done by means of the form LABEL, which pairs the name with the function definition on the a-list. The name is then bound to the function definition, just as a variable is bound to its value.

In actual practice, LABEL is seldom used. It is more convenient to attach the name to the definition in a constant manner. This is done by putting on the property list of the name, the symbol EXPR followed by the function definition. The pseudo-function define used at the beginning of this chapter accomplishes this. When apply interprets a function represented by an atomic symbol, it searches the p-list of the atomic symbol before searching the current a-list. Thus a define will override a LABEL.

The fact that most functions are constants defined by the programmer, and not variables that are modified by the program is not due to any weakness of the system. On the contrary, it indicates a richness of the system that we do not know how to exploit very well.

2.4 Machine Language Functions

Some functions instead of being defined by S-expressions are coded as closed machine language subroutines. Such a function will have the indicator SUBR on its property list followed by a pointer that allows the interpreter to link with the subroutine. There are three ways in which a subroutine can be present in the system.

1. The subroutine was coded into the LISP system.
2. The function was hand-coded by the user in the assembly type language LISP-Sap.
3. The function was first defined by an S-expression, and then compiled by the LISP compiler. Compiled functions

run about 40 times as fast as when they are interpreted.

2.5 Special Forms

Normally eval evaluates the arguments of a function before applying the function itself. Thus if eval is given (CONS X Y), it will evaluate X and Y, and then cons them. But if eval is given (QUOTE X), X should not be evaluated. QUOTE is a special form that prevents its argument from being evaluated.

A special form differs from a function in two ways. Its arguments do not get evaluated before the special form sees them. COND for example has a very special way of evaluating its arguments using evcon. The second way that special forms differ from functions is that they may have an indefinite number of arguments. Special forms have indicators on their property lists called PEXPR and FSUBR for LISP-defined forms and machine language coded forms, respectively.

2.6 Programming for the Interpreter

The purpose of this section is to help the programmer avoid certain common errors.

Example 1:

```
fn: CAR
  x: ((A B))
```

The value is A. Note that the interpreter expects a list of arguments. The one argument for car is (A B). The extra pair of parenthesis are necessary.

One could write (LAMBDA (X) (CAR X)) instead of just CAR. This is correct but unnecessary.

Example 2:

```
fn: CONS
  x: (A (B.C))
```

The value is `cons[A;(B.C)] = (A.(B.C))`.
The print program will write this as `(A B.C)`.

Example 3:

```
fn: CONS
  x: ((CAR (QUOTE (A.B))) (CDR (QUOTE (C.D))))
```

The value of this computation will be `((CAR (QUOTE (A.B))) CDR (QUOTE (C.D)))`. This is not what the programmer expected. He expected `(CAR (QUOTE (A.B)))` to evaluate to `A`, and expected `(A.D)` as the value of `cons`.

The interpreter expects a list of arguments. It does not expect a list of expressions that will evaluate to the arguments. Below are two correct ways of writing this function. The first one makes the `car` and `cdr` part of a function specified by a `LAMBDA`. The second one uses quoted arguments and gets them evaluated by `eval` with a null `a-list`.

```
fn: (LAMBDA (X Y) (CONS (CAR X) (CDR Y)))
  x: ((A.B) (C.D))
fn: EVAL
  x: ((CONS (CAR (QUOTE (A.B))) (CDR (QUOTE (C.D)))) NIL)
```

The value of both of these is `(A.D)`.

Example 4:

```
fn: (LAMBDA (A X) (MAPLIST A (FUNCTION
  (LAMBDA (J) (CONS (CAR J) X))) ))
  x: ((X Y Z) S)
```

This example contains a new special form called `FUNCTION`. `FUNCTION` is somewhat similar to `QUOTE` in its effect. It is sometimes needed when the quoted expression is a function. This is explained in detail in Appendix B. For the present we state the following rule.

When a quoted S-expression is to be used as a function, quote it with `FUNCTION`, not `QUOTE`.

The function in the example replaces each element of the list with a new element obtained by `cons`-ing it with the second argument. The value of example 4 is `((X.S) (Y.S) (Z.S))`.

2.7 Functions Available within the System

Before attempting any major programming the user should look at Appendix A so as to avoid redefining functions that are available in the system.

It is general policy not to place unnecessary definitions in the system, as this reduces the space available for computation. Extra features such as differentiation and algebra simplification will be distributed in the form of card decks with memos describing them.

The following list contains a few functions of the basic system that are of immediate interest.

- A. Basic Functions: cons, car, cdr, and all car-cdr compositions of length 2,3, and 4, e.g. cadar, cdaddr.
- B. Basic Predicates: atom, null, eq, and equal.
- C. Logical Predicates: and, or, and not.
- D. Interpreter Components: apply, eval, evlis, and pair.
- E. Defining Functions: define and cset.
- F. List Handling Functions: list, append, subst, and sublis.

3. Arithmetic in LISP

LISP 1.5 has provision for handling fixed point and floating point numbers and logical words. There are functions and predicates in the system for performing arithmetic and logical operations and making basic tests.

3.1 Reading and Printing Numbers

Numbers are stored in the computer as though they were a special type of atomic symbol. This is discussed more thoroughly in Section 7.3. The following points should be noted at this time:

1. Numbers may occur in S-expressions as though they were atomic symbols.
2. Numbers are constants that evaluate to themselves. They do not need to be quoted.
3. Numbers should not be used as variables or function names.

a. Floating Point Numbers

The rules for punching these for the read program are:

1. A decimal point must be included but not as the first or last character.

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

3. Exponent indication is optional. The letter E followed by the exponent to the base 10 is written directly after the number. The exponent consists of one or two digits which may be preceded by a plus or minus sign.

4. Absolute values must lie between 2^{128} and 2^{-128} (10^{38} and 10^{-38}).

5. Significance is limited to eight decimal digits.

6. Any possible ambiguity between the decimal point and

the point used in dot notation may be eliminated by putting spaces before and after the LISP dot. This is not required where there is no ambiguity.

The following are examples of correct floating point numbers. These are all different forms for the same number, and will have the same effect when read in.

60.0
6.E1
600.00E-1
0.6E+2

The forms .6E+2 and 60. are incorrect because the decimal point is the first or last character.

b. Fixed Point Numbers

These are written as integers with an optional sign.

Examples:

-17
32719

c. Octal Numbers or Logical Words

The correct form consists of

1. A sign (optional)
2. Up to 12 digits (0 through 7).
3. The letter Q.
4. An optional scale factor. The scale factor is a decimal integer, no sign allowed.

Examples are:

- a. 777Q
- b. 777Q4
- c. -3Q11
- d. -7Q11
- e. +7Q11

The effect of the read program on octal numbers is as follows.

1. The number is placed in the accumulator three bits per octal digit with zero's added to the left hand side, to make

twelve digits. The right most digit is placed in bits 33-35, the twelfth digit is placed in bits P, 1, and 2.

2. The accumulator is leftshifted three bits (one octal digit) times the scale factor. Thus the scale factor is an exponent to the base eight.

3. If there is a negative sign, it is OR-ed into the P bit. The number is then stored as a logical word.

The examples a through e above will be converted to the following octal words. Note that because the sign is OR-ed with the 36th numerical bit, c, d and e are equivalent.

- a. 000000000777
- b. 000007770000
- c. 700000000000
- d. 700000000000
- e. 700000000000

3.4 Arithmetic Functions and Predicates

All of these functions work on either fixed point or floating point arguments. If all of the arguments for a numerical function are fixed point numbers, then the value will be a fixed point value. If at least one argument is a floating point number, all arguments will be converted to floating point numbers, and the value will be a floating point number.

plus $[x_1, \dots, x_n]$ is a function of n arguments whose value is the algebraic sum of the arguments.

difference $[x, y]$ has as value the algebraic difference of its arguments.

minus $[x]$ has as value $-x$.

time $[x_1, \dots, x_n]$ is a function of n arguments, whose value is the product (with correct sign) of its arguments.

add1 $[x]$ has $x+1$ as its value. The value is fixed point or floating point depending on the argument.

sub1 $[x]$ has $x-1$ as its value.

max[$x_1; \dots; x_n$] chooses the largest of its arguments as its value.

min[$x_1; \dots; x_n$] chooses the smallest of its arguments as its value.

recip[x] computes $1/x$. The reciprocal of any fixed point number is defined to be zero.

quotient[$x;y$] computes the quotient of its arguments. For fixed point arguments, the value is the number theoretic quotient. A divide check or floating point trap will result in a LISP error.

remainder[$x;y$] computes the number theoretic remainder for fixed point numbers, and the floating point residue for floating point arguments.

divide[$x;y$] = list[quotient[$x;y$];remainder[$x;y$]]

expt[$x;y$] = x^y . If both x and y are fixed point numbers, this is computed by reiterative multiplication. Otherwise the power is computed using logarithms. The first argument cannot be negative.

We now list the arithmetic predicates. The rules concerning mixed expressions and evaluation of arguments are the same as for the arithmetic functions. The value of a predicate is NIL (false) or true.

lessp[$x;y$] is true if $x \leq y$, and false otherwise.

greaterp[x] is true if $x \geq y$.

zerop[x] is true if $x=0$, or if $|x| \leq 3^{-6}$.

onep[x] is true if $x=1$.

minusp[x] is true if x is negative.

"- z " is negative.

numberp[x] is true if x is a number (fixed point or floating point).

fixp[x] is true only if x is a fixed point number. If x is not a number at all, an error will result.

floatp[x] is similar to fixp[x] but for floating point numbers.

eqp[x;y] is true if $x=y$ or if $|x-y| \leq 3^{-6}$.

equal[x;y] works on any arguments including S-expressions incorporating numbers inside them. Its value is true if the arguments are identical. Floating point numbers must be exactly equal.

The logical functions operate on 36 bit words. The only acceptable arguments are fixed point numbers. These may be read in as octal or decimal integers, or they may be the result of a previous computation.

logor[$x_1; \dots; x_n$] performs a logical OR on its arguments.

logand[$x_1; \dots; x_n$] performs a logical AND on its arguments.

logxor[$x_1; \dots; x_n$] performs an exclusive OR

($0 \vee 0 = 0, 1 \vee 0 = 1, 1 \vee 1 = 0$).

leftshift[x;n] = $x \cdot 2^n$. The first argument is leftshifted by the number of bits specified by the second argument. If the second argument is negative, the first argument will be rightshifted.

3.3 Programming with Arithmetic

The arithmetic functions may be used recursively like any other functions available to the interpreter. As an example, we define factorial as it was given in Chapter 1.

$n! = [n = 0 \rightarrow 1; T \rightarrow n \cdot (n-1)!]$

```
DEFINE ((
  (FACTORIAL (LAMBDA (N) (COND
    ((ZEROP X) 1)
    (T (TIMES N (FACTORIAL (SUB1 N)))) )
  )))
```

It is sometimes convenient to refer to numerical constants by name. The pseudo-function constval allows one to do this. The argument of constval is a list of pairs, each pair consisting of a name and a number.

After executing constval[(PI 3.14) (E 2.18)] the atomic

symbols PI and E will behave as though they were actually numbers.

3.4 The Array Feature

Provision is made in LISP 1.5 for allocating blocks of storage for data. The data may consist of numbers, atomic symbols, or other S-expressions.

The pseudo-function array reserves space for arrays, and turns the name of an array into a function that can be used to fill the array or locate any element of it.

Arrays may have up to three indicies. Each element (uniquely specified by its coordinates) contains a pointer¹ to an S-expression.

Array is a function of one argument which is a list of arrays to be declared. Each item is a list containing the name of an array, its dimensions, and the word LIST. (Non-list arrays are reserved for future developments of the LISP system.)

For example, to make an array called alpha of size 7x10, and one called beta of size 3x4x5 one should execute:

```
array(((ALPHA (7 10) LIST) (BETA (3 4 5) LIST)))
```

After this has been executed, both arrays exist and their elements are all set to NIL.

Alpha and beta are now functions that can be used to set or locate elements of these respective arrays.

To set $\alpha_{i,j}$ to x, execute -

```
alpha[SET;x;i;j]
```

To set $\alpha_{3,4}$ to (A B C) execute -

```
ALPHA (SET (A B C) 3 4)
```

Inside a function or program, X might be bound to (A B C),

1. See Chapter 7.

I bound to 3, and J bound to 4, in which case the setting can be done by evaluating -

(ALPHA (QUOTE SET) X I J)

To locate an element of an array, use the array name as a function with the coordinates as axes. Thus any time after executing the previous example -

alpha[3;4] = (A B C)

Arrays use marginal indexing for maximum speed. For most efficient results, specify dimensions in increasing order. Beta[3;4;5] is better than beta[5;3;4].

4. The Program Feature

The LISP 1.5 program feature allows the user to write a Fortran-like program containing LISP statements to be executed.

An example for the program feature is the function length, which examines a list and decides how many elements there are in the top level of the list. The value of length is an integer.

Length is a function of one argument l. The program uses two program variables u and v, which can be regarded as storage locations to be changed by the program. In English the program is written.

This is a function of one argument l.

It is a program with two program variables u and v.

Store \emptyset in v.

Store the argument l in u.

A If u contains NIL, then the program is finished,
and the value is whatever is now in v.

Store in u, cdr of what is now in u.

Store in v, one more than what is now in v.

Go to A.

We now write this program as an M-expression, with a few new notations. This corresponds line for line with the program written above.

```
length[2] = prog([u;v];
```

```
  v = 0;
```

```
  u = l;
```

```
  A [null[u] → return[v]]
```

```
    u = cdr[u];
```

```
    v = v+1;
```

```
    go [A]
```

Rewriting this as an S-expression, we get the following program.

```
DEFINE ((
  (LENGTH (LAMBDA (L)
    (PROG (U V)
      (SETQ V  $\emptyset$ )
      (SETQ U L)
      A (COND ((NULL U) (RETURN V)))
      (SETQ U (CDR U))
      (SETQ V (ADD1 V))
      (GO A) ))) )
  LENGTH (A B C D)
  LENGTH ((X.Y) A CAR (N B) (X Y Z))
```

The values of the test cases are four and five, respectively.

The program form has the structure -

```
(PROG, list of program variables, sequence of statements and atomic symbols...)
```

An atomic symbol in the list is the location marker for the statement that follows. In the above example, A is a location marker for the statement beginning with COND.

The first list after the symbol PROG is a list of program variables. If there are none, then this should be written NIL or (). Program variables are treated much like bound variables, but they are not bound by LAMBDA. The value of each program variable is NIL until it has been set to something else.

To set a program variable, use the form SET. To set variable P1 to 3.14 write (SET (QUOTE P1) 3.14). SETQ is like SET except that it quotes its first argument. Thus (SETQ P1 3.14). SETQ is usually more convenient. SET and SETQ can change variables that are on the a-list from higher level functions.

Statements are normally executed in sequence. Executing a statement means evaluating it with the current a-list and ignoring its value. Program statements are often executed for their effect rather than their value.

GO is a form used to cause a transfer. (GO A) will cause the program to continue at statement A.

Conditional expressions as program statements have a useful peculiarity. If none of the propositions are true, instead of an error indication which would otherwise occur, the program continues with the next statement.

RETURN is the normal end of a program. The argument of RETURN is evaluated, and this is the value of the program. No further statements are executed.

If a program runs out of statements, it returns with the value NIL.

The program feature, like other LISP functions, can be used recursively. The function rev, which reverses a list and all its sublists is an example of this.

```
rev[x] = prog[[y;z];
A  [null[x] → return[y];
    z = car[x];
    [atom[z] → go[B]];
    z = rev[z];
B  y = cons[z;y];
    x = cdr[x];
    go[A]]
```

The function rev will reverse a list on all levels so that

$$\text{rev}[(A ((B C) D))] = ((D (C B)) A)$$

5. The Compiler

The LISP compiler writes machine language subroutines from S-expressions defining functions. Compiled functions run up to 60 times as fast as interpreted functions.

The 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

```
((NIL LID 0 4) (NIL TXI G00007 4 -1)
(NIL TRA *+5) (G00008 BSS 0) ...)
```

In this example, the objects beginning with G are atomic symbols generated for use within the compiler. The BSS 0 in the last element above is used as it is in SAP to tag symbols which need to have a memory location assigned to them, but no actual space reserved for them, i.e. the usual location-field SAP symbol.

After the compiler has created the LISP-SAP program for a function, the binary program is generated from LISP-SAP in two passes. 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 function's 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.

f₁ uses f₂
f₂ uses f₃

.

.

.

f_n uses f₁,

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.

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 compiler will accept most function definitions acceptable to the interpreter, including the program feature. In fact, the program feature will compile more efficiently than functions using recursive definition, and should be used whenever possible when compiling.

SET is not allowed in compiled programs. SETQ must be used instead.

The compiler will give trouble in the following situations.

1. If any of the functions to be compiled call on FEXPR defined functions, the compiler will fail.

2. If a function uses free variables not bound by its own LAMBDA, this will cause trouble. The one way to make such functions compile is as follows: Suppose *f*₁, ..., *f*_{*n*} is a sequence of functions each of which may use free variables bound by the functions before it. They will compile correctly if the order of compilation is to start with *f*_{*n*} and end with *f*₁. This may be done in several comdef's or in one comdef. Functions within a comdef are compiled in the order listed.

5.1 The Compiler Modes

The compiler modes are switches set by the function compilemode. They control the manner in which compiling is done, and the compiler output.

To set a particular mode, execute compilemode of the mode name, e.g.

COMPILEMODE (OPENCONS)
COMPILEMODE (NOPUNCH)

The modes are:

PRINT: The LISP-Sap program will appear in the printed output.

NOPRINT: No printed LISP-Sap output.

PUNCH: The LISP-Sap program will be punched on cards.

NOPUNCH: No punched LISP-Sap output.

TRACE: }
NOTRACE: } See Section 6.2

OPENCONS: When cons occurs in functions to be compiled, it will be compiled as an open subroutine or macro.

CLOSECONS: Cons will be compiled as a TSX to the closed subroutine cons.

Opencons is slightly faster than closecons in execution but takes more binary program space. Opencons cannot affect the cons counter. (See Section 6.4).

Before they have been set otherwise, the modes are UNPRINT, UNPUNCH, UNTRACE, and CLOSECONS.

5.2 Sample Compiler Program

The function member defined in Section 2.1 has been compiled as an example of the way in which the compiler works. The printed output is listed below with comments to the right of the assembly listing.

Because the function member is recursive, the subroutine can be entered several times during a single computation. This requires that certain partial results be saved at each entry. Saving is done on the push down list which is simply a large block of storage. A pointer to the head of the list is stored in \$CPPI. When the push down list is used, this pointer must be updated.

Conventions for LISP subroutines are as follows:

- 1 Arguments are put in AC, MQ, \$ARG3,...,\$ARG10.
- 2 Answer is returned in AC.
- 3 Calling sequence is TSX SUBR,4.
- 4 All index registers must be saved.
- 5 Exit is TRA 1,4

The program for member contains the following locations for temporary storage

G00576 IX4
G00577 A
G00578 X
G00579 Answer
G00583 car[X]
G00585 cdr[X]
\$ARG2 temporary saving of MQ

\$ENPDL is a test for out of push down list.

The printed output is as follows:

```
FUNCTION EVALQUOTE HAS BEEN ENTERED, ARGUMENTS..  
DEFINE  
(((MEMBER (LAMBDA (A X) (COND ((NULL X) F) ((EQ A (CAR X)) T)  
      (T (MEMBER A (CDR X)))))))  
  
END OF EVALQUOTE, VALUE IS ..  
(MEMBER)
```

```
FUNCTION EVALQUOTE HAS BEEN ENTERED, ARGUMENTS..  
COMDEF  
((MEMBER))
```

```
(MEMBER BSS $ZERO) entry point to function member  
(NIL SXD G00576 4) save IX 4  
(NIL STQ $ARG2) store MQ
```

(NIL LXD \$CPPI 4)	pick up the location of the next available word on the push down list
(NIL XEC \$ENPDL)	test to see if out of push down list
(NIL LDQ G00576)	save IX 4 on the push down list
(NIL STQ 0 4)	
(NIL TIX *+1 4 1)	update \$CPPI
(NIL SXD \$CPPI 4)	
(NIL LDQ \$ARG2)	restore MQ
(NIL STO G00577)	store A
(NIL STQ G00578)	store X
(NIL CLA G00578)	put X in AC
(NIL TNZ G00581)	if X is not null, go to next condition
(NIL CLA \$ZERO)	X is null, pick up <u>false</u>
(NIL STO G00579)	store false in answer
(NIL TRA G00580)	go to end of program
(G00581 BSS \$ZERO)	start of second condition
(NIL LXD G00578 4)	get pointer to X
(NIL CLA 0 4)	pick up X
(NIL PAX 0 4)	get pointer to car[X]
(NIL SXD G00583 4)	store pointer to car[X]
(NIL CLA G00577)	pick up pointer to A
(NIL SUB G00583)	eq test for car[X] and A
(NIL TNZ G00582)	if not eq, go to next condition
(NIL CLA \$ONE)	pick up <u>true</u>
(NIL STO G00579)	store <u>true</u> in answer
(NIL TRA G00580)	go to end of program
(G00582 BSS \$ZERO)	start of last condition
(NIL LXD G00578 4)	get pointer to X
(NIL CLA 0 4)	pick up X
(NIL STD G00585)	store pointer to cdr[X]
(NIL LDQ G00585)	cdr[X] in MQ
(NIL CLA G00577)	A in AC
(NIL TSX MEMBER 4)	recursive entry to member
(NIL STO G00579)	store answer
(G00580 BSS \$ZERO)	

(NIL LXD \$CPPI 4) restore \$CPPI
(NIL TXI *+1 4 1)
(NIL SXD \$CPPI 4)
(NIL LDQ 0 4) unsave IX 4
(NIL STQ G00576)
(NIL CLA G00579) pick up answer
(NIL LXD G00576 4) restore IX 4
(NIL TRA 1 4) exit

MEMBER

BINARY PROGRAM OCCUPIES 23407 TO 23471 OCTAL

END OF EVALQUOTE, VALUE IS ..

(MEMBER)

6. Running the LISP System

6.1 Preparing a Card Deck

A LISP program consists of several sections called packets. Each packet starts with an overlord direction card, followed by a set of doublets for evalquote, and ending with the word STOP.

Overlord directions control tape movement, restoration of the system memory between packets, and core dumps. A complete listing of overlord directions is given in Appendix D.

Overlord direction cards are punched in Share symbolic format; the direction starts in column 8, and the comments field starts in column 16. Some overlord cards are described here:

TEST: Subsequent doublets are read in until the word STOP is encountered, or until a read error occurs. The doublets are then evaluated and each doublet with its value is written on the output tape. If an error occurs, a diagnostic will be written and the program will continue with the next doublet. When evalquote is finished, control is returned to overlord which restores the core memory to what it was before the TEST by reading in a core memory image from the temporary tape.

SET: The doublets are read and interpreted in the same manner as a TEST. However when evalquote is finished, the core memory is not restored. Instead, the core memory is read out onto the temporary tape over-writing the previous core image, and becomes the base memory for all remaining packets. Definitions and other memory changes made during a SET will affect all remaining packets.

Several SET's during a LISP run will set on top of each other.

A SET will not set if it contains an error. The memory will be restored from the temporary tape instead.

SETSET: This direction is like SET, except that it will set even if there is an error.

FIN: End of LISP run.

The reading of doublets is normally terminated by the word STOP. If parentheses do not count out, STOP will appear to be inside an S-expression and will not be recognized as such. To prevent reading from continuing indefinitely, each packet should end with STOP followed by a large number of right parentheses. An unpaired right parenthesis will cause a read error and terminate reading.

A complete card deck for a LISP run might consist of:

- a: LISP loader
- b: ID card (Optional)
- c: Several Packets
- d: FIN card
- e: Two blank cards to prevent card reader from hanging up

The ID card may have any information desired by the computation center. It will be printed at the head of the output.

6.2 Tracing

Tracing is a technique used to debug recursive functions. The tracer prints the name of a function and its arguments when it is entered, and its value when it is finished. By tracing certain critical subfunctions, the user can often locate a fault in a large program.

Tracing is controlled by the pseudo-function traclis whose argument is a list of functions to be traced. After traclis has been executed, tracing will occur whenever these functions are entered.

When tracing of certain functions is no longer desired, it can be terminated by the pseudo-function untraclis whose

argument is a list of functions that are no longer to be traced.

Tracis can trace any type of function which is indicated by EXPR, SUBR, PEXPR, or FSUBR. However, it can only trace a function when it has been entered from the interpreter. Thus when a compiled function uses certain subfunctions, these subfunctions will not be traced when entered from the higher level compiled function.

The compiler tracer called track is entirely independent of tracis. It can trace compiled functions regardless of how they are entered.

In order to track a function, it is necessary to compile the function while the compiler is in a special mode. The compiler is put in the tracing mode by -

COMPILEMODE (TRACE)

and is restored by -

COMPILEMODE (NOTRACE)

If a function has been compiled in the tracing mode, then it can be traced at any time by track whose argument is a list of functions to be traced. Untrack turns off the tracing for a list of functions.

6.3 Error Diagnostics

When an error occurs in a LISP 1.5 program, a diagnostic giving the nature of the error is printed out. The diagnostic gives the type of error, and the location in the machine where it occurred. In some cases a back-trace is also printed. This is a list of functions that were entered recursively but not completed at the time of the error.

In most cases, the program continues with the next doublet. However certain errors are fatal and in this case control is given to the monitor overlord. Errors during overlord also continue with overlord.

A complete list of error diagnostics is given below, with comments.

Interpreter Errors:

- A 1 **APPLIED FUNCTION CALLED ERROR**
The function error will cause an error diagnostic to occur. The argument (if any) of error will be printed. Error is of some use as a debugging aid.
- A 2 **FUNCTION OBJECT HAS NO DEFINITION - APPLY**
This occurs when an atomic symbol is given as the first argument of apply, and it does not have a definition either on its property list or on the a-list of apply.
- A 3 **CONDITIONAL UNSATISFIED - EVCON**
None of the propositions following COND are true.
- A 4 **SETQ GIVEN ON NON-EXISTENT PROGRAM VARIABLE - APPLY**
- A 5 **SET GIVEN ON NON-EXISTENT PROGRAM VARIABLE - APPLY**
- A 6 **GO REFERS TO A POINT NOT LABELLED - INTER**
- A 7 **TOO MANY ARGUMENTS - SPREAD**
The interpreter can handle only 10 arguments for a function.
- A 8 **UNBOUND VARIABLE - EVAL**
The atomic symbol in question is not bound on the a-list for eval nor does it have an APVAL or APVAL1.
- A 9 **FUNCTION OBJECT HAS NO DEFINITION - EVAL**
Eval expects the first object on a list to be evaluated to be an atomic symbol. A 8 and A 9 frequently occur when a parenthesis miscount causes the wrong phrase to be evaluated.

Compiler Errors:

- C 1 **NOT ENOUGH NUMBERS FOR SAVING - COMPILER**
The function cannot be compiled because there are too many variables that need to be saved. The limit is currently 15.

- C 2 NOT ENOUGH NUMBERS FOR UNSAVING - COMPILER
- C 3 NOT ENOUGH BINARY PROGRAM SPACE - COMPILER
Lack of room to store compiled program.

Character Handling Functions:

- CH 1 TOO MANY CHARACTERS IN PRINT NAME - PACK
 - CH 2 FLOATING POINT NUMBER OUT OF RANGE - NUMOB
 - CH 3 TAPE READING ERROR - ADVANCE
- The character handling functions are described in Appendix E.

Miscellaneous Errors:

- F 1 CONS COUNTER TRAP
The cons counter is described in the next section.
- F 2 FIRST ARGUMENT LIST TOO SHORT - PAIR
- F 3 SECOND ARGUMENT LIST TOO SHORT - PAIR
Pair is used by the interpreter to bind variables to arguments. If a function is given the wrong number of arguments, these errors may occur.
- F 4 OBJECT GIVEN AS INPUT - DESC
- F 5 STR TRAP - CONTINUING WITH NEXT EVALQUOTE
When the instruction STR is executed, this error occurs. If sense switch 6 is down when an STR is executed, control goes to overlord instead.
- G 1 FLOATING POINT TRAP OR DIVIDE CHECK
- G 2 OUT OF PUSH DOWN LIST
The push down list is the memory device that keeps track of the level of recursion. When recursion gets very deep, this error will occur. Non-terminating recursion will cause this error.

Garbage Collector Errors:

- GC 1 FATAL ERROR - RECLAIMER
This error only occurs when the system is so checked that it cannot be restored. Control goes to overlord.

GC 2 NOT ENOUGH WORDS COLLECTED - RECLAIMER

This error restores free storage as best it can and continues with the next doublet.

Number Errors:

I 1 NOT ENOUGH ROOM FOR ARRAY

Arrays are stored in binary program space.

I 2 FIRST ARGUMENT NEGATIVE - EXPT

I 3 BAD ARGUMENT - NUMVAL

I 4 BAD ARGUMENT - FIXVAL

Errors I 3 and I 4 will occur when numerical functions are given wrong arguments.

Overlord Errors:

O 1 ERROR IN SIZE CARD - OVERLORD

O 2 INVALID TAPE DESIGNATION - OVERLORD

O 3 NO SIZE CARD - OVERLORD

O 4 BAD DUMP ARGUMENTS - OVERLORD

O 5 BAD INPUT BUT GOING ON ANYHOW - OVERLORD

O 6 END OF FILE ON INPUT - OVERLORD

O 7 OVERLAPPING PARAMETERS - SETUP

Overlord is discussed in Appendix D.

Input - Output Errors:

P 1 PRIN1 ASKED TO PRINT NON-OBJECT

R 1 FIRST OBJECT ON INPUT LIST IS ILLEGAL - RDA

This error occurs when the read program encounters a character such as ")" or "." out of context. This occurs frequently when there is a parenthesis miscount.

R 2 CONTEXT ERROR WITH DOT NOTATION - RDA

R 3 ILLEGAL CHARACTER - RDA

R 4 END OF FILE ON READ-IN - RDA

R 5 PRINT NAME TOO LONG - RDA

Print names may contain up to 30 BCD characters.

R 6 NUMBER TOO LARGE IN CONVERSION - RDA

6.4 The Cons Counter and Errorset

The cons counter is a useful device for breaking out of program loops. It automatically causes a trap when a certain number of cons's have been performed.

The counter is turned on by executing `count [n]`, where `n` is an integer. If `n` cons's are performed before the counter is turned off a trap will occur and an error diagnostic will be given. The counter is turned off by `uncount [NIL]`. The counter is turned on and reset each time `count [n]` is executed. The counter can be turned on so as to continue counting from the state it was in when last turned off by executing `count [NIL]`.

The function `speak [NIL]` gives the number of cons's counted since the counter was last reset.

Open cons's in compiled functions will not be counted.

Errorset is a function available to the interpreter and compiler for making a graceful retreat from an error condition encountered during a subroutine.

`errorset[e;n;m]` is a function of three arguments. `e` is a form to be evaluated. `n` is the number of conses to be permitted before an error should be indicated. `m` is the mode of operation; T if error diagnostics are to be printed, F if they are to be suppressed.

Errorset computes `eval[e;NIL]`. Since `e` is evaluated once before `errorset` sees it, it may need to be quoted. The value of `errorset` is a list of the value of `e`, that is, `list[eval[e;NIL]]`.

If an error condition occurs during the `errorset` evaluation of `e`, `errorset` will return the value `NIL`. Too many conses will be treated as an error.

During an `errorset`, the cons counter will be decremented whether it is turned on or off. An error condition will result when either the original value of the cons counter is reduced to zero, or the number specified in the `errorset` has been reduced to zero. Whether an error was encountered or not, the cons

counter will be left at its original value minus the number of cones actually used, when errorset is completed. The on-off status of the cons counter will be as it was before entering the errorset.

Errorset may be used recursively.

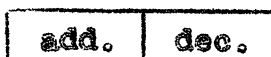
7. List Structures

In previous sections of this manual, lists have been discussed using the LISP input-output language. In this section, we discuss the representation of lists inside the computer, the nature of property lists of atomic symbols, representation of numbers, and the garbage collector.

7.1 Representation of List Structure

Lists are not stored in the computer as sequences of BCD characters, but as structural forms using computer words as parts of trees.

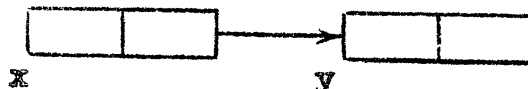
In representing list structure, a computer word will be depicted as a rectangle divided into two sections, the address and decrement.



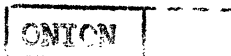
Each of these is a fifteen bit field of the word.

We define a pointer to a computer word as the fifteen bit quantity that is the complement of the address of the word. Thus a pointer to location 77777 would be 00001.



Suppose the decrement of word x contains a pointer to word y. We diagram this as-

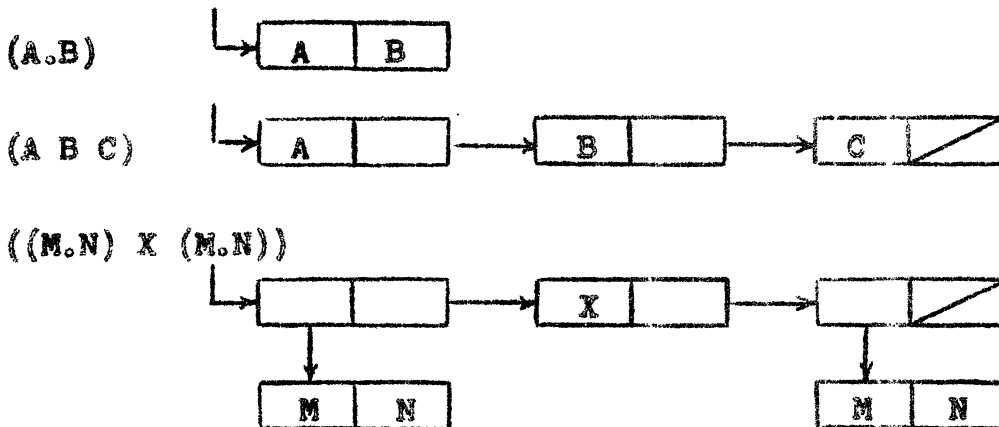


We can now give a rule for representing S-expressions in the computer. The representation of atomic symbols will be explained in section 7.3. When a computer word contains a pointer to an atomic symbol in the address or decrement, the atomic symbol will be written there:

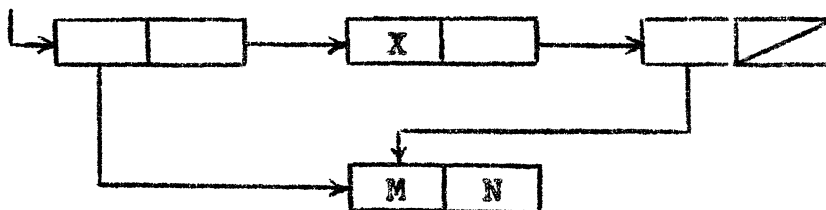


The rule for representing non-atomic S-expressions is to start with a word containing a pointer to car of the expression in the address, and a pointer to cdr in the decrement.

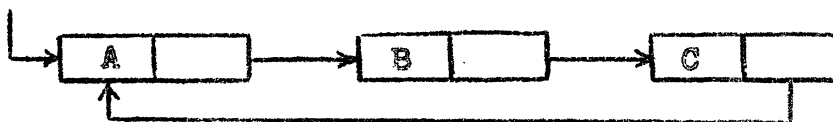
Following are some diagrammed S-expressions as they would appear in the computer. It is convenient to indicate NIL by  instead of .



It is possible for lists to make use of common sub-expressions. ((M.N) X (M.N)) could also be represented as-

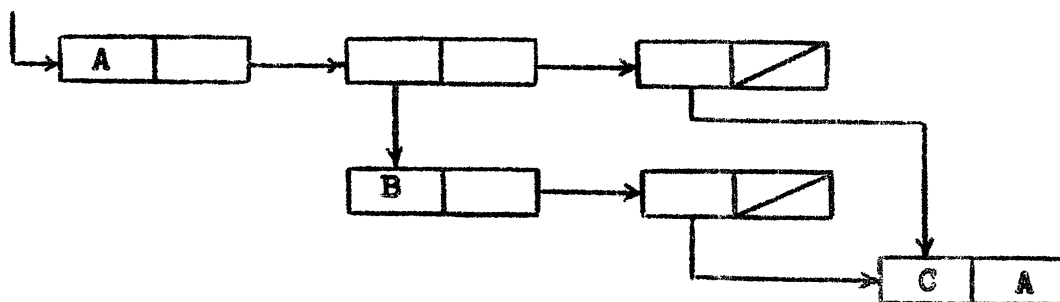


Circular lists are ordinarily not permitted. They may not be read in; however they can occur inside the computer as the result of computations involving certain functions. Their printed representation is infinite in length. For example, the structure



will print as (A B C A B C A...)

The following is an actual assembly listing of the list (A (B (C.A)) (C.A)) which is diagrammed-



The atoms A, B, and C are represented by pointers to locations 12327, 12330, and 12331 respectively. NIL is represented by a pointer to location 00000.

10425	0	65451	0	67352	-A, -*-1
10426	0	67350	0	67351	-*-2, -*-1
10427	0	67346	0	00000	-*-3
10430	0	65450	0	67347	-B, -*-1
10431	0	67346	0	00000	-*-1
10432	0	65447	0	65451	-C, -A

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.

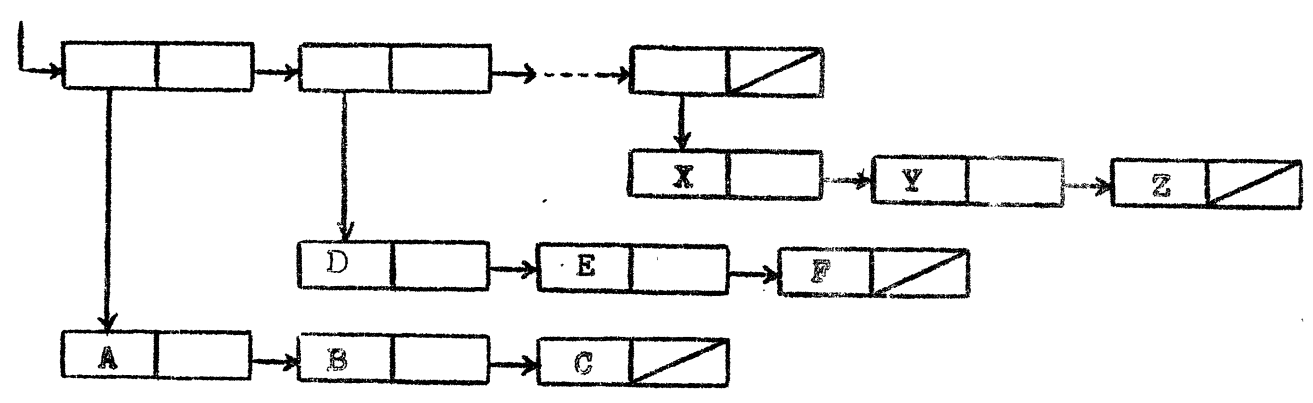
7.2 Construction of List Structure

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

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

$$l_1 = ((A B C) (D E F), \dots, (X Y Z)),$$

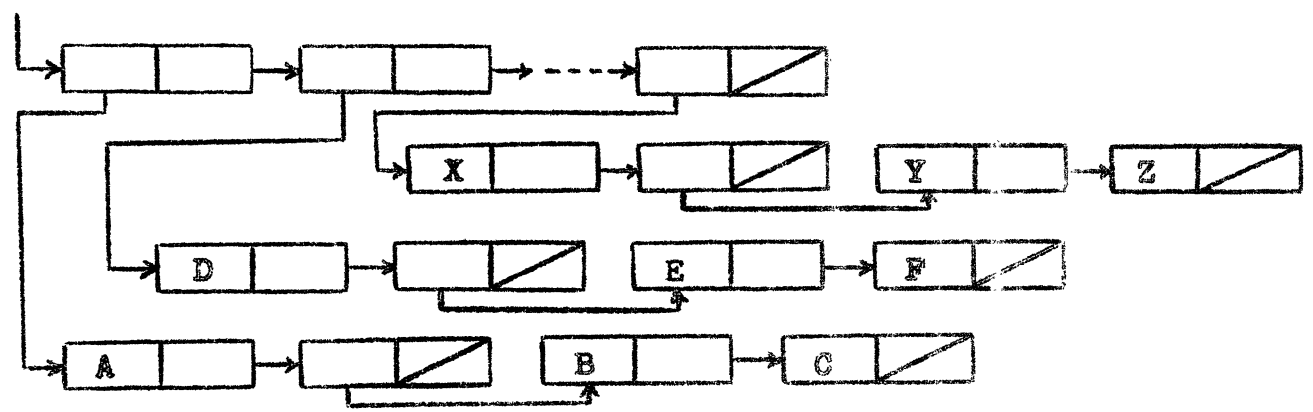
which is represented as



and that we wish to construct a list of the form

$$l_2 = ((A (B C)) (D (E F)), \dots, (X (Y Z))),$$

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 function, we can write the same thing as

```
list[A;list[B;C]]
```

In any case, given a list, x, of three atomic symbols, $x = (A B C)$, the arguments A, B, and C to be used in the previous construction are found from

```
A = car[x]
B = cadr[x]
C = caddr[x]
```

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).

```
grp[x] = list[car[x];list[cadr[x];caddr[x]]]
```

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 as

```
mltgrp[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.

7.3 Property Lists

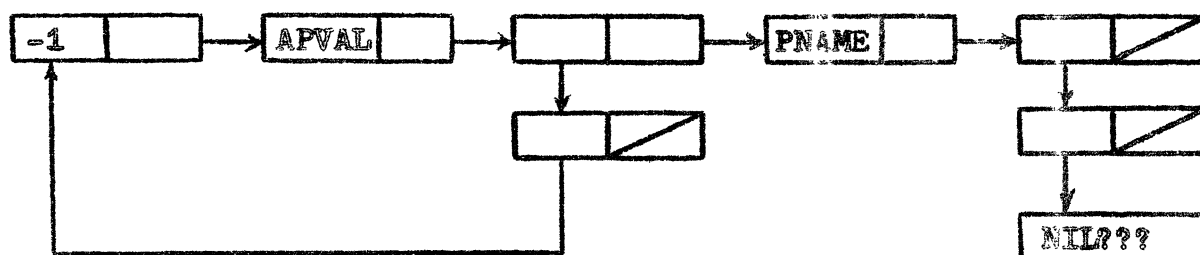
In the previous sections, atomic symbols were considered only as pointers. This section describes the property lists of atomic symbols which begin at the appointed locations.

Every atomic symbol has a property list. When an atomic symbol is read in for the first time, a property list is created for it.

A property list is characterized by having the special constant 77777g (i.e. minus 1) as the first element of the list. The rest of the list contains various properties of the atomic symbol. Each property is preceded by an atomic symbol which is called its indicator. Some of the indicators are:

- PNAME - the BCD print name of the atomic symbol for input-output use.
- EXPR - S-expression defining a function whose name is the atomic symbol on whose property list the EXPR appears.
- SUBR - Function defined by a machine language subroutine.
- APVAL1 - Permanent value for the atomic symbol considered as a variable.
- APVAL - Similar to APVAL1 except that it points to a full word instead of list structure.
- FLOAT - Indicates a floating point number.
- FIX - Indicates a fixed point number

The atomic symbol NIL has two things on its property list--its PNAME, and an APVAL which gives it a value of NIL. Its property list looks like this:

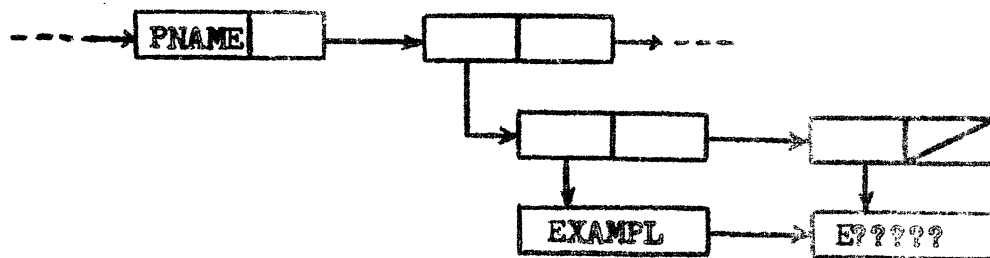


00000	0	00134	0	77777	-1, , -NIL
77644	0	00133	0	11741	-APVAL, , -* -1
77645	0	00131	0	00132	-*-1, , -* -2
77646	0	00000	0	00000	0
77647	0	00130	0	10236	-PNAME, , -* -1

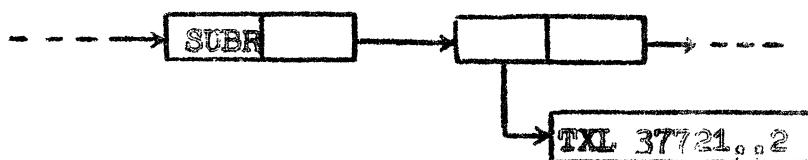
```

77650  0 00000 0 00127    -* -1
77651  0 00000 0 00126    -* -1
77652  453143777777       BCD  NIL???
```

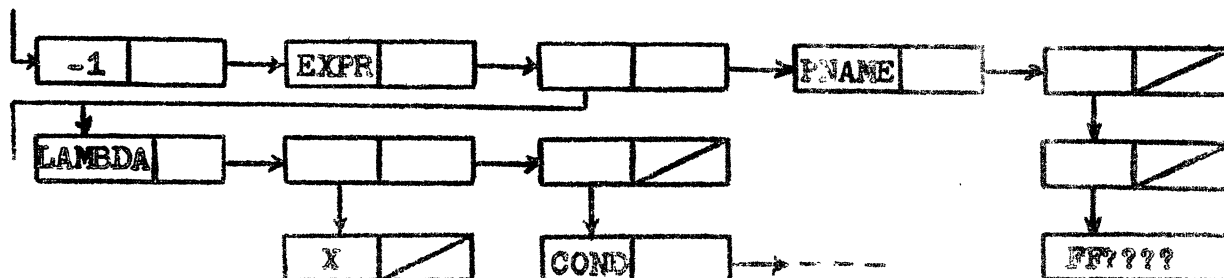
The print name (PNAME) is depressed two levels to allow for names of more than six BCD characters. The last word of the print name is filled out with the illegal BCD character 77g (?). The print name of EXAMPLE would look like:



The property list of a machine language function contains the indicator SUBR followed by a TXL instruction giving the location of the subroutine and the number of arguments. For example -



The indicator EXPR points to an S-expression defining a function. The function define puts EXPR's on property lists. After defining ff, its property list would look like -



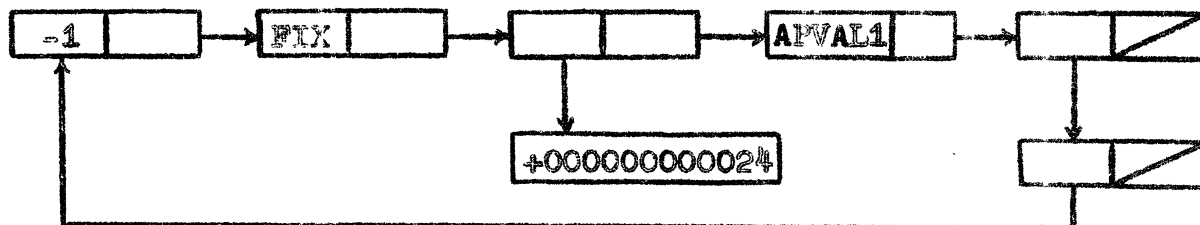
The function get[x;i] can be used to find a property of x whose indicator is i. The value of get[IF;EXPR] would be (LAMBDA (X) (COND...

A property with its indicator can be removed by remprop[x;i].

Compile, comdef, and compsap are functions that put SUBR's on property lists.

The function deflist[x;i] can be used to put any indicator on a property list. The first argument is a list of pairs as for define, the second argument is the indicator to be used. deflist[x;EXPR] = define[x].

Numbers have property lists similar to those for atomic symbols. The actual number is in a full word with the appropriate indicator, FLOAT or FIX. The property list for the integer 20 (decimal) is-



Because numbers have APVAL1's pointing to themselves, they are constants and do not need to be quoted.

Numbers do not have PNAME indicators. Conversion to or from BCI is made during reading or writing.

Unlike atomic symbols, numbers are not stored uniquely. If a number is read in twice, there will be two separate property lists created. When a number is the result of a computation, a property list for it is automatically created.

If a number is read in as an octal, it has an indicator OCT in addition to the indicator FIX. This signals the print program to print the number as BCI octal rather than decimal.

7.4 List Structure Operators

The theory of recursive functions developed in Chapter 1 will be referred to as pure LISP. Although this language is universal in terms of computable functions of symbolic expressions, it is not convenient as a programming system without additional tools to increase its power.

In particular, pure LISP has no ability to modify list structure. The only basic function that affects list structure is cons, and this does not change existing lists, but creates new lists. Functions written in pure LISP such as subst do not actually modify their arguments, but make the modifications while copying the original.

LISP is made general in terms of list structure by means of the basic list operators rplaca, and rplacd. These operators can be used to replace the address or decrement or any word in a list. They are used for their effect as well as for their value, and are sometimes called pseudo-functions.

rplaca[x;y] replaces the address of x with y. Its value is x, but x is something different from what it was before. In terms of value, rplaca can be described by the equation:

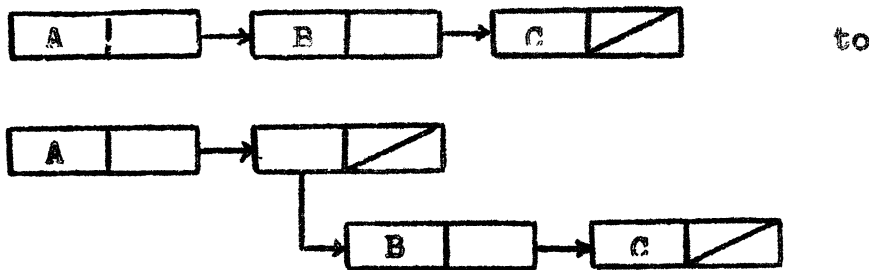
$$\text{rplaca}[x;y] = \text{cons}[y;\text{cdr}[x]]$$

But the effect is quite different. There is no cons involved, and a new word is not created.

rplacd[x;y] replaces the decrement of x with y.

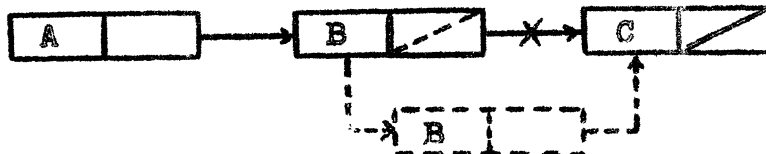
These operators must be used with caution. They can permanently alter existing definitions and other basic memory. They can be used to create circular lists, which can cause infinite printing, and look infinite to functions that search such as equal and subst.

As an example, consider the function mltgrp of section 7.2. This is a list altering function that alters a copy of its argument. The subfunction grp rearranges a subgroup:



The original function does this by creating new list structure, and uses four cons's. Because there are only three words in the original, at least one cons is necessary, but grp can be rewritten using rplaca and rplacd.

The modification is as follows:



The new word is created by `cons[cadr[x];cddr[x]]`. A pointer to it is provided by `rplaca[cdr[x];cons[cadr[x];cddr[x]]]`.

The other modification is to break the pointer from the second to the third word. This is done by `rplacd[cdr[x];NIL]`.

pgrp is now defined as

```
pgrp[x] = rplacd[rplaca[cdr[x];cons[cadr[x];cddr[x]]];NIL]
```

The function pgrp is used entirely for its effect. Its value is not useful, being the substructure ((B C)). Therefore a new mitgrp is needed that executes pgrp and ignores its value. Since the top level is not to be copied, mitgrp should do no consing.

```
pmitgrp[l] = [null[l] → NIL;
```

```
T → prog2[grp[car[l]];mitgrp[cdr[l]]]]
```

Prog2 is a function that evaluates its two arguments. Its value is the second argument.

The value of pmitgrp is NIL. It is a pure pseudo-function.

7.5 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 program. The program which retrieves the storage is 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 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.

Sometimes list structure points to full words such as BCD print names and numbers. The garbage collector cannot mark these words because the sign bit may be in use. The garbage collector must also stop tracing because the pointers in the address and decrement of a full word are not meaningful.

These problems are solved by putting full words in a reserved section of memory called full word space. The garbage collector stops tracing as soon as it leaves the free storage space. Marking in full word space is accomplished by a bit table.

Appendix A

This appendix contains functions available in the LISP System as of July 1961. Supplementary functions and programs such as integrate and simplify are not included.

The description of each function contains the type (i.e. EXPR, FEXPR, SUBR, FSUBR), the nature of the arguments, a LISP definition when applicable, and an explanation of its purpose.

Keeping track of the contents of a particular system can be simplified by the following tricks:

1. The entire set of objects (atomic symbols) existing in the system can be printed out by the doublet

EVAL (OBLIST NIL)

2. The properties of an atomic symbol can be printed by executing printprop[x].

Functions are arranged in groups in the descriptive listing. An alphabetical index follows.

Elementary Functions

car[x] : SUBR

cdr[x] : SUBR

All multiple car's and cdr's of length two, three, and four are available in the system, e.g. cadr, caddar, etc.

cons[x;y] : SUBR

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

rplaca[x;y] : SUBR pseudo-function

This function replaces the address of x with y. The value of rplaca is the new x.

rplacd[x;y] : SUBR pseudo-function

This function replaces the decrement of x with y. The value of rplacd is the new x.

See Section 7.3 of this manual.

Elementary Predicates

atom[x] : SUBR

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 7) the argument is an atomic symbol if and only if car[x] = -1.

null[x] : SUBR

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

and[x₁;x₂;...;x_n] : FSUBR

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[x₁;x₂;...;x_n] : FSUBR

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] : SUBR

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

eq[x;y] : SUBR

If x and y are atomic symbols, then eq[x;y] is true if they are identical and false if they are different atomic symbols. If

the arguments are not atomic symbols but S-expressions or numbers, then the value eq will certainly be false if they are different, and may be true or false if they are identical, depending on how they happen to be represented in the machine.

equal[x;y] : SUBR

Equal is true if its arguments are identical and false otherwise. The arguments may be any type of legal list structure such as numbers, atomic symbols, or S-expressions which may contain numbers.

$$\begin{aligned} \text{equal}[m;n] = & [\text{eq}[m;n] \rightarrow T; \\ & \text{atom}[m] \rightarrow m = n; \\ & \text{atom}[n] \rightarrow F; \\ & \text{equal}[\text{car}[m];\text{car}[n]] \rightarrow \text{equal}[\text{cdr}[m];\text{cdr}[n]]; \\ & T \rightarrow F] \end{aligned}$$

Interpreter Components

Information on these functions is contained in Chapters 1, 2, and in Appendix B. evalquote[fn;args] operates automatically on the input doublets; however, it can also be called explicitly.

apply[fn;args;a-list] applies its first (functional) argument to its second argument, thereby computing fn[arg1;arg2;...;argn]. The a-list is used to bind variables and function names. Apply uses eval.

eval[expression;a-list] evaluates its first argument, and in particular evaluates variables using the a-list. Eval and apply are interdependent.

* test for numerical equality

Defining Functions and Functions Useful for Property Lists

define[x] : EXPR pseudo-function

The argument of define, x, is a list of pairs

$((u_1 v_1) (u_2 v_2) \dots (u_n v_n))$

where each u is a name and each v is a λ -expression for a function.

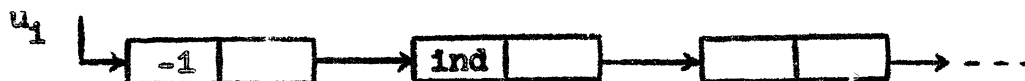
For each pair, define puts an EXPR on the property list for u pointing to v. The function define puts things on at the front of the property list. The value of define is the list of u's.

define[x] = deflist[x;EXPR]

deflist[x;ind] : EXPR pseudo-function

The function deflist is a more general defining function.

Its first argument is a list of pairs as for define. Its second argument is the indicator that is to be used. After deflist has been executed with $(u_1 v_1)$ among its first argument, the property list of u_1 will begin -



If deflist or define is used twice on the same indicator, the old value will be replaced by the new one.

attrib[x;e] : SUBR pseudo-function

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 a property 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 property list for FF.

prop[x;y;u] : SUBR

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.

$prop[x;y;u] = [null[x] \rightarrow u[]; eq[car[x];y] \rightarrow cdr[x];$
 $T \rightarrow prop[cdr[x];y;u]$

get[x;y] : SUBR

Get is somewhat like prop; however its value is car of the rest of the list if the indicator is found, and NIL otherwise.

$get[x;y] = [null[x] \rightarrow NIL; eq[car[x];y] \rightarrow cadr[x];$
 $T \rightarrow get[cdr[x];y]$

cset[ob;val] : EXPR pseudo-function

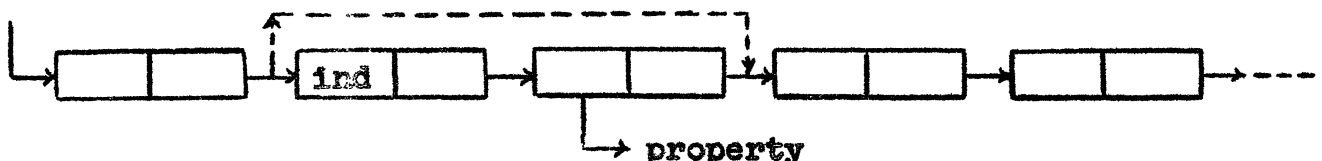
This pseudo-function is used to create a constant by putting the indicator APVAL1 and a value on the property list of an atomic symbol. The first argument should be an atomic symbol; the second argument is the value.

csetq[ob;val] : FEXPR pseudo-function

Csetq is like cset except that it quotes its first argument instead of evaluating it.

remprop[x;ind] : SUBR pseudo-function

The function remprop searches the list, x, looking for all occurrences of the indicator ind. When such an indicator is found, its name and the succeeding property 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.

List Handling Functions

list[$x_1; x_2; \dots; x_n$] : FSUBR

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

append[$x; y$] : SUBR

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] \rightarrow y; T \rightarrow 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.

conc[$x_1; x_2; \dots; x_n$] : FEXPR pseudo-function

Conc concatenates its arguments by stringing them all together on the top level. For example -

conc[(A (B.C) D); (F); (G H)] = (A (B.C) D F G H).

Conc concatenates its arguments without copying them. Thus it changes existing list structure and is a pseudo-function.

nconc[$x; y$] : SUBR pseudo-function

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[ $m$ ];  
  null[x]  $\rightarrow$  return[y];  
   $m = x$ 
```

```

A  null [cdr m] ] → go[B]
   m = cdr[m]
   go [A]
B  rplacd[m;y]
   return[x]]

```

copy[x] : SUBR

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]]]]

```

pair[x;y] : SUBR

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.

```

pair[x;y] = [prog [u;v;m]
  u = x
  v = y
A  null[u] → [null[v] → return[m]; T → error[F2]] ;
   null[v] → error[F3];
   m = cons [cons [car [u]; car [v]] ; m];
   u = cdr[u];
   v = cdr[v];
   go[A]

```

sassoc[x;y;u] : SUBR

The function sassoc searches y, which is a list of dotted pairs, for a pair whose first element is identical with x. If such a pair is found, the value of sassoc is this pair. 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] → car[y];
  T → sassoc[x;cdr[y];u]

```

subst[x;y;z] : SUBR

The function subst has as value the result of substituting x for all occurrences of the S-expression y in the S-expression z.

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

sublis[x;y] : SUBR

Here x is a list of pairs,

((u₁.v₁) (u₂.v₂) ... (u_n.v_n))

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 1, 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];cdar[j]];
λ[[]];[atom[y] → y;
T → cons[sublis[x;car[y]];sublis[x;
cdr[y]]]]]]
]]
```

reverse[l] : SUBR

This is a function to reverse the top level of a list. Thus reverse[(A B (C.D))] = ((C.D) B A)

```
reverse[l] := prog[v];
```

```
u = l;
```

```
A null[u] → return[v];
```

```
v = cons[car[u];v]
```

```
u = cdr[u]
```

```
go[A]
```

Functionals or Functions with Functions as Arguments

maplist[x;f] : SUBR functional

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] : SUBR functional

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 → conc[f[x];mapcon[cdr[x];f]]]

map[x;f] : SUBR functional

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 null[m] → return[NIL];  
  f[m];  
  m = cdr[m];  
  go[LOOP]
```

search[x;p;f;u] : SUBR functional

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]]

Arithmetic Functions

These are discussed at length in Chapter 3.

<u>function</u>	<u>type</u>	<u>number of args</u>	<u>value</u>
plus	FSUBR	indef.	$x_1 + x_2 + \dots + x_n$
minus	SUBR	1	$-x$
difference	SUBR	2	$x - y$
times	FSUBR	indef.	$x_1 \cdot x_2 \cdot \dots \cdot x_n$
divide	SUBR	2	list[x/y; remainder]
quotient	SUBR	2	x/y
remainder	SUBR	2	$x - \langle x/y \rangle$
add1	SUBR	1	$x+1$
sub1	SUBR	1	$x-1$
max	FSUBR	indef.	largest of x_1
min	FSUBR	indef.	smallest of x_1
recip	SUBR	1	$1/x$; fixp[x] $\rightarrow 0$
expt	SUBR	2	x^y
lessp	SUBR	2	$x \leq y$
greaterp	SUBR	2	$x \geq y$
zerop	SUBR	1	$ x \leq 3 \times 10^{-6}$
onep	SUBR	1	$x = 1$
minusp	SUBR	1	x is negative
numberp	SUBR	1	x is a number
fixp	SUBR	1	x is a fixed point number
floatp	SUBR	1	x is a floating point no.
eqp	SUBR	2	$ x-y \leq 3 \times 10^{-6}$
logor	FSUBR	indef.	$x_1 \vee x_2 \vee \dots \vee x_n$ ORA
logand	FSUBR	indef.	$x_1 \wedge x_2 \wedge \dots \wedge x_n$ ANA
logxor	FSUBR	indef.	$x_1 \oplus x_2 \oplus \dots \oplus x_n$ ERA
leftshift	SUBR	2	$x \cdot 2^y$

constval[p] : EXPR

Constval makes certain atomic symbols appear as numbers, thus

track - untrack : for compiler tracing. See Section 6.2

count - uncount - speak : for the cons counter. See Section 6.4

Compiler Pseudo-Functions

comdef[*l*] : EXPR pseudo-function

The argument of comdef is a list of the names of functions to be compiled. They must already have EXPR definitions at compiling time. Subfunctions used by the functions to be compiled will also be compiled automatically if they are defined by EXPR's. The order in which functions should be compiled is described in Chapter 5.

compile[*l*] : SUBR pseudo-function

The argument of compile is a list of functions each of the form (LABEL FN (LAMBDA...

sap[*s;org*] : SUBR pseudo-function

This pseudo-function calls the LISP-Sap assembler. The first argument is a symbolic listing such as ((SYM CLA 0 4) (NIL TLX 0 4 1) ...). The second argument is the location for the first assembled word. If org is NIL, the program will be placed in the first free space available in binary program space. LISP-Sap is discussed in the compiler-assembler appendix.

comsap[*l*] : SUBR pseudo-function

This is an assembly pseudo-function that assembles at org = NIL, and links with the interpreter by putting SUBR on the property list of the object which is the location name of the first assembly instruction.

opdefine[*l*] : SUBR pseudo-function

This pseudo-function defines new machine instructions. For

example -

```
opdefine(((CAL 45Q10) (TIX 2Q11)))
```

Input-Output Functions

read[] : SUBR

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] : SUBR pseudo-function

This function prints one S-expression (on the on-line printer and/or output tape depending on the sense switch settings). The value of print is its argument.

Print always starts printing at the beginning of a line or record.

punch[x] : SUBR pseudo-function

This function is exactly like print except that the output appears on the punched output tape.

printprop[x] : EXPR pseudo-function

The argument of printprop is an atomic symbol. Printprop will print the name of the atomic symbol, all indicators on its property list, and the properties associated with certain indicators that point to list structure.

punchdef[x] : EXPR pseudo-function

If x is a list of atomic symbols each of which has EXPR on its property list, then punchdef will write the EXPR definitions on the punched output tape.

Character Handling Functions

The following functions are described in detail in the appendix in LISP input-output, reading, and printing operations (and in Memo 22A until it is available).

advance	liter
charcount	mknam
clearbuff	numob
curchar	opchar
dash	pack
digit	startread
endread	terpri
intern	unpack

Interpreter Special Forms

(QUOTE α)

The unevaluated argument α is the value of QUOTE.

(COND ($p_1 e_1$) ($p_2 e_2$) ... ($p_n e_n$))

COND evaluates each p_i in order until one is found whose value is not NIL (0). Then the corresponding e_i is evaluated, and this is the value of COND.

(FUNCTION fn)

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 fn , and third element equal to the current list of bound variables. I.e.

eval[(FUNCTION fn);b] = (FUNARG fn b)

Having such a list of bound variables carried along with the function insures that the proper values of the bound varia-

bles are used when the function is evaluated. Thus
 $\text{apply}[(\text{FUNARG } f \text{ } b);x;a] = \text{apply}[f;x;b]$

Program Feature

prog[...] : FSUBR

Prog signals the program feature. The first argument is a list of program variables. Each non-atomic argument after the first is a statement. Each atomic symbol is a location symbol for the next statement.

go[x] : FSUBR pseudo-function

Go causes a transfer to the location x. Its argument is not evaluated. The argument must be a location marking symbol.

set[x;y] : SUBR pseudo-function

Set can be used to change any type of variable on the current a-list. The first argument is the variable to be set. The second argument is the value.

setq[x;y] : FSUBR pseudo-function

Setq is like set except that it quotes its first argument instead of evaluating it.

return[x]

The argument of return is evaluated, and the program is terminated with this value.

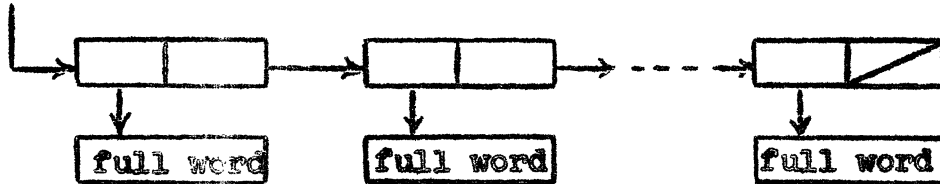
Miscellaneous Functions

prog2[x;y] : SUBR

Prog2 is defined by $\lambda[[x;y];y]$. It is an identity in its second argument. The first argument is evaluated for its effect rather than its value.

cp1[x] : SUBR

Cp1 copies its argument which must be a list of a very special type.



The copied list is the value of cp1.

gensym[] : SUBR

The function gensym has no arguments. Its value is a new, distinct, and freshly-created atomic symbol with a print name of the form G00001, G00002, ..., G99999.

This function is useful for creating atomic symbols when one is needed; each one is guaranteed unique. Gensym names are not permanent and will not be recognized if read back in.

select[q; (q₁ e₁); (q₂ e₂); ...; (q_n e_n); e] : FEXPR

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.

makeblr[x] : EXPR pseudo-function

This function speeds up the execution of multiple cars and cdrs ("makes car or cdr abler"). The argument is a list of pairs such as

((CAAR (A A)) (CADR (A D)) (CADDR (A D D)) ...).

Makeblr turns each form such as CADR into a compiled function with the appropriate definition as indicated by the string of A's and D's.

format[name;form;varlis] : EXPR pseudo-function

Arguments:

- name: an atomic symbol
- form: any list structure
- varlis: a list of variables each of which occurs once and only once in form.

Effect:

Format creates several compiled functions.

Name becomes a function which has as many arguments as there are variables in varlis. It is a substitution function that substitutes its arguments for the variables of varlis in the S-expression form.

Each variable name becomes a function of one argument. The function is a chain of car's and cdr's that is sufficient to obtain the variable in its proper location in format.

Consider the following example:

```
format[VICO; (HE IS HCE WITH ALP SHEM SHAUN AND ISEULT);  
        (HCE ALP SHEM SHAUN ISEULT)]
```

After executing the pseudo-function format, six functions will have been defined and compiled.

Vico is now a substitution function. For example -

```
vico[(SUCH A GRANDFALLER);(A POCKED WIFE IN PICKLE);TWO;  
      (TWILLING BUGS);(ONE MIDGIT PUCELLE)]
```

```
= (HE IS (SUCH A GRANDFALLER) WITH (A POCKED WIFE IN PICKLE)  
  TWO (TWILLING BUGS) AND (ONE MIDGIT PUCELLE))1
```

1 James Joyce, Finnegan's Wake, p. 29.

hee, alp, shem, shaun, and iseult are now functions each of which selects from its argument the substructure determined by the position of the function name in form.

For example -

```
shem[(BUT THE (DOVLIN SULPH) WAS IN GLOGGER THAT LOST-TO-  
LEARNING)]1  
= GLOGGER
```

and

```
hee[((HOHOHOMO MISTER FINN) (YOUR GOING TO BE) (MISTER  
FINNAGAIN))]2  
= (MISTER FINNAGAIN)
```

tempus-fugit[] : SUBR pseudo-function

Executing this will cause a time statement to appear in the output. The value is NIL. (Tempus-fugit is for MIT users only.)

load[] : SUBR pseudo-function

Program control is given to the LISP loader which expects octal correction cards, 704 row binary cards, and a transfer card.

plb [] : SUBR pseudo-function

This is equivalent to pushing "LOAD CARDS" in the console in the middle of a LISP program.

reclaim[] : SUBR pseudo-function

Executing this will cause a garbage collection to occur. The value is NIL.

pause[] : SUBR pseudo-function

Executing this will cause a program halt. Pushing START will cause the program to continue by returning the value NIL.

¹ Ibid., p. 222.

² Ibid., p. 5.

Alphabetical Index of Functions

ADD1	77
ADVANCE	81
AND	69
APPEND	73
APPLY	70
ARRAY	78
ATOM	69
ATTRIB	71
CAR	68
CDR	68
CHARCOUNT	81
CLEARBUFF	81
COMDEF	79
COMPILE	79
COMPSAP	79
CONC	75
COND	81
CONS	68
CONSTVAL	77
COPY	74
COUNT	79
CPI	83
CSET	72
CSETQ	72
CURCHAR	81
DASH	81
DEFINE	71
DEFLIST	71
DIFFERENCE	77
DIGIT	81
DIVIDE	77
ENDREAD	81
EQ	69
EQP	77
EQUAL	70
ERROR	78
ERRORSET	78
EVAL	70
EVALQUOTE	70
EXPT	77
FIXP	77
FLOATP	77
FORMAT	84
FUNCTION	81
GENSYM	83
GET	72
GO	82
GREATERP	77
INTERN	81
LEFTSHIFT	77
LESSP	77
LIST	77
LITER	81

LOAD	85
LOGAND	77
LOGOR	77
LOGXOR	77
MAKCBLR	83
MAP	76
MAPCON	76
MAPLIST	*76
MAX	77
MIN	77
MINUS	77
MINUSP	77
MKNAM	81
NCONC	73
NOT	69
NULL	61
NUMBERP	77
NUMOB	81
ONEP	77
OPCHAR	81
OPDEFINE	79
OR	69
PACK	81
PAIR	74
PAUSE	85
PLB	85
PLUS	77
PRINT	80
PRINTPROP	80
PROG	82
PROG2	82
PROP	71
PUNCH	80
PUNCHDEF	80
QUOTE	81
QUOTIENT	77
READ	80
RECIP	77
RECLAIM	85
REMAINDER	77
REMPROP	72
RETURN	82
REVERSE	75
RPLACA	68
RPLACD	69
SAP	79
SASSOC	74
SEARCH	76
SELECT	83
SET	82
SETQ	82
SPEAK	79

STARTREAD	81
SUBLIS	75
SUBST	75
SUBI	77
TEMPUS-FUGIT	85
TERPRI	81
TIMES	77
TRACK	79
TRACLIS	78
UNCOUNT	79
UNPACK	81
UNTRACK	79
UNTRACLIS	78
ZEROP	77