TM-2337/010/00

TECHNICAL MEMORANDUM

(TM Series)

This document was produced by SDC in performance of contract SD-97

LISP Primer

A Self-Tutor for Q-32 LISP 1.5

Clark Weissman

14 June 1965

SYSTEM
DEVELOPMENT
CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA

LISP Primer

A Self-Tutor for Q-32 LISP 1.5

ABSTRACT

This document is a self-tutor for LISP 1.5 programming, particularly for on-line Q-32 LISP 1.5. Material is organized into chapters that, by discussion and example, progressively expand the student's understanding of the language and ability to write programs in the language. A carefully selected and graduated set of exercises for use on-line is provided as an integral part of each chapter. Computer-checked answers for each exercise are also provided as a separate appendix. The document is not an exhaustive treatise on LISP 1.5, but, rather, a practical primer that provides the serious student with a solid foundation for understanding the programming language and system. He may then easily supplement his knowledge from other sources suggested herein.

ACKNOWLEDGMENT

The author wishes to acknowledge the considerable support he received in preparing this primer from Jules Schwartz who instigated its preparation and was its first user, Dr. Danny Bobrow (SDC consultant) who reviewed the document for technical errors and suggested improvements in material presentation, Dr. Stan Kameny for suggested improvements and for contributing a number of important exercises, and Robert Saunders (I.I.I.) for checking the technical consistency in accordance with the Q-32 LISP 1.5 system.

Thanks are also extended to Miff Perstein for his technical and editorial review of the material during its preparation, and to Donna Firth and Frank Poage for their suggestions, comments, and criticisms resulting from their actual use of the primer as a teaching aid. Also, thanks to Professor Marvin Minsky (M.I.T.) for suggesting additional material that will be incorporated in the final version of this primer.

The author further wishes to acknowledge the incorporation of selected material in this document from LISP 1.5 Programmers Manual, August 1962, M.I.T. Press, Cambridge, Massachusetts, and The Programming Language LISP: Its Operation and Applications, March 1964, Information International, Inc., Cambridge, Massachusetts.

PREFACE

This primer is an experimental approach to teaching the LISP 1.5 programming language, particularly Q-32 LISP 1.5. Material is presented in a sequence that builds upon prior information by continually expanding the language's domain of functional expressions. The student is cautioned not to progress to subsequent chapters until he thoroughly understands all prior chapters. Each exercise has been carefully selected to explore his understanding. He should not treat the exercises lightly, but do as many of the problems on-line as possible to check his answers. Also, explanatory material for subtle exercises is often reviewed in the answers of Appendix A.

Further material will be included in this primer after it is "field tested" by the remote TSS user community. The author assumes full responsibility for all errors that appear and he solicits general comments on the primer and notification of any errors that exist. Such comments will be reflected in an improved final Q-32 LISP Primer.

The student who desires supplemental material should consult the following references:

1. Command Research Laboratory Users' Guide, TM-1354 S.D.C. document series.
2. LISP 1.5 Programmers Manual, August 1962, M.I.T. Press, Cambridge, Massachusetts.
3. Q-32 LISP 1.5 Reference Material, TM-2337 S.D.C. document series.
4. The Programming Language LISP: Its Operation and Applications, March 1964. Information International, Inc., Cambridge, Massachusetts.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

LISP is a relatively new and remarkable programming language for instructing large digital computers. The name comes from the contraction of the words LISt Processing, which connote the primary data structures LISP is designed to manipulate. LISP is remarkable in that it is not only a programming language for symbolic data processing but also a formal mathematical language. Ideally, with such formalism, it should be possible to "prove" a LISP program as one proves a mathematical theorem, thereby simplifying the debugging phase of program checkout. For knowledgeable LISP programmers this ideal is approached in practice; however, the mathematical formalism requires a rigorous parenthetical syntax (anathema to on-line programmers) that usually results in syntactic, rather than semantic, program errors, particularly for beginners. Careful attention to LISP syntax by the student of this primer, may ease this difficulty for the beginner.

1.1 HISTORICAL BACKGROUND

LISP is based on a paper by John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine," which was published in Communications of the ACM, April 1960. Q-32 LISP was implemented by Robert A. Saunders of Information International, Incorporated, and is based on the implementation of LISP by Timothy P. Hart for the M-460 computer at Air Force Cambridge Research Labs and the earlier IBM-790 LISP implementation at M.I.T. by a group including John McCarthy, Stephen B. Russell, Daniel J. Edwards, Paul W. Abrahams, Timothy P. Hart, Michael I. Levin, Marvin L. Minsky, and others.

These implementations of LISP have been used primarily for symbolic data processing in such areas as differential and integral calculus, electrical circuit theory, mathematical logic, game playing, linguistics, heuristic programming, and other fields of artificial intelligence. Newer implementations, such as Q-32 LISP, have improved arithmetic capability, permitting reasonably good numerical computation, which extends the domain of LISP applications to problems requiring a mix of symbolic and numerical data processing.

1.2 A SIMPLE LISP EXAMPLE

To demonstrate the power of LISP to process symbolic and numeric data, let us examine a collection of LISP programs for manipulating and evaluating algebraic equations. These are purposely simple LISP examples to keep confusion to a minimum, allowing the student to follow the subject matter easily.

LISP represents all data and programs as symbolic expressions called S-expressions. Furthermore, LISP uses function logic syntactically represented in Polish prefix notation. For simplicity, here we use only the LISP arithmetic operators and limit them to two operands, i.e.,

PLUS	for +
DIFFERENCE	for -
TIMES	for *
QUOTIENT	for /

The following tables demonstrate the equivalence between algebraic expressions and their representations as LISP S-expressions.

Algebraic Expression	S-expression Representation
a + b	(PLUS A B)
a - b	(DIFFERENCE A B)
a * b	(TIMES A B)
a / b	(QUOTIENT A B)
a*x + b*y	(PLUS (TIMES A X) (TIMES B Y))
(a*x - b*y)/c*z	(QUOTIENT (DIFFERENCE (TIMES A X)(TIMES B Y)) (TIMES C Z))
$a*x^3 + bx^2 + 2x + d$	(PLUS (TIMES A (TIMES X (TIMES X X))) (PLUS (TIMES B (TIMES X X)) (PLUS (TIMES 2 X) D)))

As a first example, let's examine a LISP program for computing points on the curve of the function

$$f(x)=x^2+2x+5$$

We can represent the algebraic expression

$$x^2+2x+5$$

as the S-expression

```
(PLUS (TIMES X X)(PLUS (TIMES 2 X) 5))
```

A point on the curve of the function f(x) would then be given as

$$(x,f(x))$$

and we can define the LISP function POINT to calculate a point on this curve.

```
DEFINE ((
(POINT (LAMBDA (X)(LIST X (PLUS (TIMES X X)(PLUS (TIMES 2 X) 5))))) ))
```

The LISP function DEFINE allows us permanently to define the function POINT in the system as compiled code on Q-32 LISP and to use it subsequently whenever desired. The function LIST is an existing LISP function that lists its arguments, in this case x and f(x). LAMBDA is part of the functional syntax of LISP.

After defining POINT, as given above, we can compute all pairs (x,f(x)) by evaluating POINT with various values of x; e.g.,

```
POINT (0) = (0 5)
POINT (1) = (1 8)
POINT (3) = (3 20)
POINT (10) = (10 125)
POINT (15) = (15 260)
```

```
DEFINE ((
(GRAPH (LAMBDA (L)(MAPCAR L (FUNCTION POINT))))
       ))
```

GRAPH is another easily defined function, one that takes a list ℓ of values of x and returns a list of points on the curve f(x). The Q-32 LISP function MAPCAR takes, sequentially, each value of x in ℓ, evaluates POINT for that value, and lists all the values computed by POINT; e.g.,

```
GRAPH ((0 1 3 10 15)) = ((0 5) (1 8) (3 20) (10 125) (15 260))
```

A more powerful and general program for computing the value of <u>any</u> algebraic expression f(x) can be easily defined in LISP. For example, COMPUTE is defined as:

```
DEFINE ((
  (COMPUTE (LAMBDA (E L)(EVAL1 (SUBLIS L E))))
       ))
```

COMPUTE is defined as a LISP function of two arguments e and ℓ. The algebraic expression to be evaluated, e, is given in the Polish prefix notation noted earlier; and ℓ is a list of pairs of variables and their numeric values as used in the algebraic expression; e.g., for the expression

$f(x)=x^2+ax+b$

e = (PLUS (TIMES X X)(PLUS (TIMES A X) B))

ℓ = ((X . 1)(A . 2)(B . 5))

COMPUTE uses two important LISP functions, SUBLIS and EVAL1. For each variable-value pair in ℓ, SUBLIS substitutes the numeric value for the variable in expression e. SUBLIS returns as its value the new expression formed by these substitutions. For the expressions e and ℓ above, SUBLIS returns the new expression

```
(PLUS (TIMES 1 1)(PLUS (TIMES 2 1) 5))
```

EVAL1 is an important Q-32 LISP function that evaluates expressions. In this example, EVAL1 evaluates the expression returned by SUBLIS and hence computes the desired value of the expression e for the values given in ℓ. Thus, COMPUTE is a general program that computes any f(x) written in Polish prefix notation; e.g.,

```
COMPUTE ((PLUS (TIMES X X)(PLUS (TIMES A X) B))
        ((X . 2)(A . 3)(B . 100))) = 110
```

1.3 A MORE INTERESTING EXAMPLE

As a more interesting example, let us consider an elementary LISP differentiation program. The LISP function DIFF (e x) is to differentiate any algebraic expression e with respect to the variable x. DIFF will use the following differentiation rules:

$$\frac{dx}{dx} = 1 \qquad \text{(rule 1)}$$

$$\frac{dy}{dx} = 0, (y \neq x) \qquad \text{(rule 2)}$$

$$\frac{d(u + v)}{dx} = \frac{du}{dx} + \frac{dv}{dx} \qquad \text{(rule 3)}$$

$$\frac{d(u*v)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx} \qquad \text{(rule 4)}$$

For example, if e = $3x^2+2x$ = (PLUS (TIMES 3 (TIMES X X)) (TIMES 2 X)) we apply rule 3 first with $u = 3x^2, v = 2x$ to yield the explicit expression

(PLUS (DIFF (TIMES 3 (TIMES X X)) X)(DIFF (TIMES 2 X) X)) (1)

If we label the elements of e, we may see how the LISP definition for DIFF operates more clearly.

$$e = (\underset{\underset{\text{1st}}{\uparrow}}{\text{PLUS}}\ \underbrace{\text{(TIMES 3 (TIMES X X))}}_{\text{2nd}}\ \underbrace{\text{(TIMES 2 X)}}_{\text{3rd}})$$

Now applying rule 3 to expression e we get expression (2)

(PLUS (DIFF(2nd of e) X)(DIFF(3rd of e) X)) (2)

which is equivalent to expression (1).

In LISP, the functions CAR, CADR, and CADDR are equivalent to the 1st, 2nd, and 3rd elements of a list, respectively, and the function QUOTE is used to name things literally. We can then write expression (2) as an implicit expression, i.e., an expression to be computed, as follows:

(LIST(QUOTE PLUS)(DIFF (CADR E) X)(DIFF (CADDR E) X)) (3)

Expression (3) is an implicit LISP form for the differentiation rule 3. Expression (4) below is the implicit LISP form for the differentiation rule 4.

```
(LIST (QUOTE PLUS)
      (LIST (QUOTE TIMES)(CADDR E)(DIFF (CADR E) X))
      (LIST (QUOTE TIMES)(CADR E)(DIFF (CADDR E) X)))                 (4)
```

Expressions (3) and (4) are general LISP forms that can differentiate any arithmetic expression satisfying rules 3 and 4. This generality is what is meant here by an implicit form.

Let us now consider the total LISP definition for DIFF for all four differentiation rules, where the algebraic expression e is given in Polish prefix notation.

```
DEFINE ((
  (DIFF (LAMBDA (E X)
     (COND ((ATOM E)(COND ((EQ E X) 1)
                          (T 0)))
           ((EQ (CAR E) (QUOTE PLUS))
              (LIST (QUOTE PLUS)(DIFF (CADR E) X)(DIFF (CADDR E) X)))
           ((EQ (CAR E) (QUOTE TIMES))
                (LIST (QUOTE PLUS)
                      (LIST (QUOTE TIMES)(CADDR E)(DIFF (CADR E) X))
                      (LIST (QUOTE TIMES)(CADR E)(DIFF (CADDR E) X))))
           (T (QUOTE UNDEFINED)))))
       ))
```

ATOM is a LISP function that has a Boolean value of true (T) or false (F); true if its S-expression argument is a simple LISP symbol, called an atom, and false if its argument is a non-atomic S-expression. Such Boolean functions are called predicates in LISP and are used for conditional branching. The function EQ is also a LISP predicate that is true if two atoms are equivalent, i.e., the same atom, and false otherwise.

COND is a special form for conditional expressions in LISP that has the format:

_If_ predicate 1 is true, _then_ the value is that of expression 1, _else_

_If_ predicate 2 is true, _then_ the value is that of expression 2, _else_, etc.
Conditional expressions may be nested as is done in the definition of DIFF.

If we examine the definition for DIFF above, we see that it is essentially a conditional expression with three if/then clauses used to segregate the four differentiation rules. (The first clause has a nested conditional with two clauses for isolating rules 1 and 2.) The first clause begins with the predicate ATOM, while clause 2 and clause 3 begin with the predicate EQ. A fourth clause is provided as an error trap if none of the prior conditions are satisfied. The definition has the structural form:

<u>If</u> e is an atom <u>then</u> (<u>if</u> e=x <u>then</u> apply rule 1 <u>else</u> apply rule 2) <u>else</u>
<u>If</u> 1<u>st</u> of e = PLUS <u>then</u> apply rule 3, <u>else</u>
<u>If</u> 1<u>st</u> of e = TIMES <u>then</u> apply rule 4, <u>else</u> DIFF is undefined.

Examination of the definition for DIFF shows the use of DIFF within its own definition. We call such practice a recursive definition and the function DIFF, when executed, will compute by recursion. The mechanisms of list processing used by LISP encourage the use of recursive definitions. In DIFF, recursion allows DIFF to be applied concurrently to the solution of the original problem and to all subproblems necessary in solving the original problem. For example, the equation

$$3x^2 + 2x$$

where $u=3x^2$, $v=2x$ is first differentiated by rule 3 to yield

$$\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx} = \frac{d(3x^2)}{dx} + \frac{d(2x)}{dx}$$

But each of these terms $\frac{du}{dx}$ and $\frac{dv}{dx}$ must be further differentiated by rule 4, i.e.,

<u>Term 1</u> = $3x^2$; $u=3$, $v=x^2$

<u>then</u> $\frac{d(u*v)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$

$= x^2 \frac{d(3)}{dx} + 3\frac{d(x^2)}{dx}$

<u>Term 2</u> = $2x$, $u=2$, $v=x$

<u>then</u> $\frac{d(u*v)}{dx} = v\frac{du}{dx} + u\frac{dv}{dx}$

$= x\frac{d(2)}{dx} + 2\frac{dx}{dx}$

Finally, by application of rules 1, 2, and 4 once again to these partial results, we get the total differential

$$f'(x) = 6x+2$$

Since we must apply differentiation rules 1, 2, 3, and 4 repeatedly to each partial result, the use of a recursive definition for DIFF "automatically" applies these rules to each partial result for us, and thereby results in a neat, tight expression for differentiating <u>any</u> polynomial expression.

Without exhausting all possible cases, let's walk through the evaluation of DIFF for one rule for the expression

e = $3x^2 + 2x$ = (PLUS (TIMES 3 (TIMES X X)) (TIMES 2 X))

First notice:

```
1st of e = (CAR E) = PLUS
2nd of e = (CADR E) = (TIMES 3 (TIMES X X))
3rd of e = (CADDR E) = (TIMES 2 X)
```

We enter COND and ask, "Is e an atom?" i.e., (ATOM E)

"No, e is a non-atomic S-expression." So we then ask, "Is PLUS the 1st of e?" i.e., (EQ (CAR E)(QUOTE PLUS))

"Yes, (CAR E) is PLUS." Therefore, we apply rule 3 and evaluate

(LIST (QUOTE PLUS)(DIFF (CADR E) X)(DIFF (CADDR E) X))

which yields, by rule 3,

```
              (PLUS(DIFF (TIMES 3 (TIMES X X)) X)
                   (DIFF (TIMES 2 X) X))
```

$$= \frac{d(u)}{dx} + \frac{d(v)}{dx}$$

To solve this expression completely, we need to evaluate DIFF of the partial results. We shall not do so here; however, the technique is exactly that of the above. The final result of DIFF yields the expression

```
(PLUS (PLUS (TIMES (TIMES X X) 0)
            (TIMES 3 (PLUS (TIMES X 1)(TIMES X 1))))
      (PLUS (TIMES X 0)(TIMES 2 1)))
```

$= 6x + 2$

1.4 ORGANIZATION OF THIS PRIMER

LISP is designed to allow expressions of increasing complexity and generality to be evaluated by the computer. This primer leads the student to an appreciation of this by building new concepts upon prior ones and thereby expanding the domain of LISP expressions step-by-step.

Starting simply, with a formal definition of LISP S-expressions, the domain is extended by the introduction of LAMBDA expressions, the fundamental functional syntax of LISP. After the introduction of numbers, composition of functions is presented as a primary capability of the language that allows concatenation of existing functions into larger functions. Conditional expressions are explained to give greater flexibility to the increasing functional domain.

During his progress through this primer, the student is exposed to a variety of basic system functions and arithmetic and predicate functions, CAR, CDR, CONS, LIST, QUOTE, and DEFINE. Arriving at this point, recursion is introduced. LISP is designed to make recursion easy, as recursive definitions are a significant addition to the domain of LISP functional expressions. Inasmuch as recursion is often costly in system operation, LISP provides the PROG feature. The PROG feature is discussed and demonstrated to show the ALGOL-like statement capability within LISP that permits iteration in lieu of recursion.

Beyond Chapter 15 the primer is devoted to further extending the functional domain by the use of macros and functionals, i.e., functions that take other functions as arguments. Also this portion of the primer gives a detailed discussion of the internal mechanics of the Q-32 LISP system concerned with program execution under the Evalquote supervisor, input output, and the binding of values to variables in the evaluation of an expression.

A word of caution before we start. LISP is not an easy language to learn for most knowledgeable algebraic language programmers because of the alien functional syntax. However, LISP is consistent in this syntax. As expressions get more complex, they still retain the same syntactic form. If the student pays careful attention to this fact, learning LISP will be a much easier task. The carefully graduated exercises should help in this respect.

CHAPTER 2. SYMBOLIC EXPRESSIONS

All programs and data in the programming language are in the form of symbolic expressions usually referred to as S-expressions. S-expressions are of indefinite length and have a branching binary tree structure, so that significant sub-expressions can be readily isolated. The bulk of available memory is used for storing S-expressions in list-structure form. This type of memory organization permits the system to free the programmer from the necessity of memory storage allocation for different sections of his program or data. It also makes LISP programs and data homogeneous; i.e., programs can be treated as data (and vice versa) by other programs.

2.1 ATOMIC SYMBOLS

The most elementary type of S-expression is called an atomic symbol, or an atom. Atoms may be numeric or non-numeric. We will discuss numbers later.

Definition: A non-numeric atomic symbol on the Q-32 is a string of capital letters and numbers of indefinite length; the first character being a letter.

Examples:
```
A
APPLE
PART2
EXTRALONGSTRINGOFLETTERS
A1B66X4ZZ
```

These symbols are called atomic because they are taken as a whole and are not viewed as individual characters. Thus A, B, and AB are three distinct and unrelated atomic symbols.

2.2 DOT NOTATION

All non-atomic S-expressions are built of atomic symbols and the punctuation marks:

```
(
)
.
```

These larger S-expressions (non-atomic S-expressions) are always parenthesized and always have two parts, a left part and a right part. A dot "." is used to delimit the two halves. For example, the S-expression

```
(A . B)
```

has atomic symbol A as its left part and atomic symbol B as its right part. Thus a non-atomic S-expression is <u>always</u> a "dotted pair."

<u>Definition:</u> An S-expression is either:

1. an atom, e.g., A1
2. a dotted pair of atoms, e.g., (A . B)
3. a dotted pair of S-expressions, e.g., ((A . B) . C)

It is composed of these elements in the following order:

a left parenthesis,
an S-expression,
a dot,
an S-expression, and
a right parenthesis.

Notice that this definition is recursive since an S-expression is defined in terms of itself.

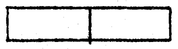
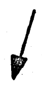
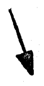
<u>Examples:</u>

```
ATOM
(A . B)
(A . ATOM)
(ATOM1 . (BETA . C))
((U . V) . X)
((U . V) . (X . (Y . Z)))
```

2.3 GRAPHICAL REPRESENTATION OF DOTTED PAIRS

All non-atomic S-expressions are internally represented as a binary tree structure, i.e., a tree structure with but two branches at each node. It is often helpful to the student to "see" the graphical representation of this tree structure.

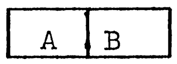
We assume the following graphical symbols and their associated meanings:

Symbol	Meaning
	A graphical node with a left and right branch.
 or 	A pointer that is the internal address of the next element of the graph.
atom names	The internal address to which the atom named is assigned.

First the graph of

(A . B)

is given by

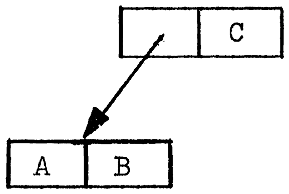


where the left part of the dotted pair, atom A, is named in the left branch of the node, and the right part of the dotted pair, atom B, is named in the right branch of the node.

The graph of

((A . B) . C)

is slightly more complicated, namely



In this case, the highest node's left branch points to the lower node, while the highest node's right branch contains the name of atom C. The lower node is exactly the graph of

(A . B)

shown above because it is the same S-expression. In this example, however, it is a <u>subexpression</u> of the S-expression

((A . B) . C)

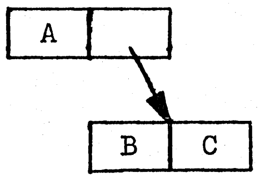
We see here graphically the meaning of "subexpression." It is an S-expression at a "lower level" and appears in dot notation as a more deeply nested S-expression.

Examples:

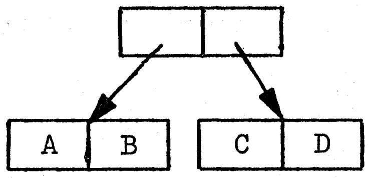
S-expression

Graph

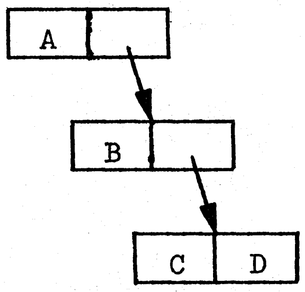
(A . (B . C))



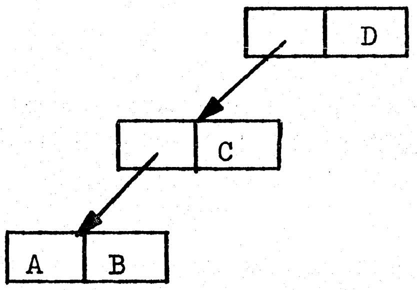
((A . B) . (C . D))



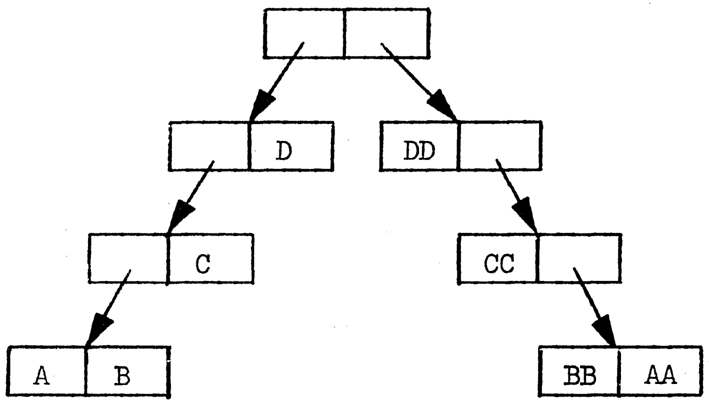
(A . (B . (C . D)))



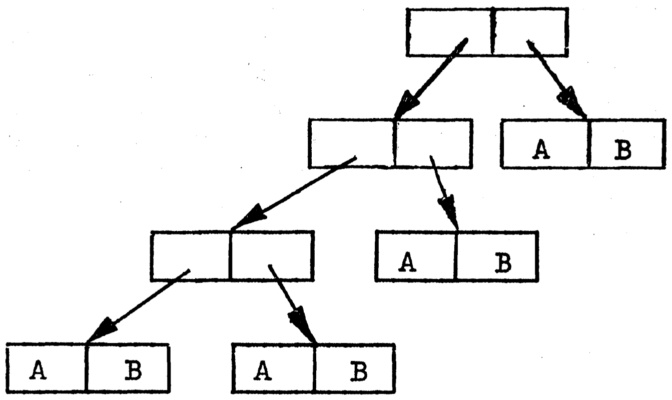
(((A . B) . C) . D)



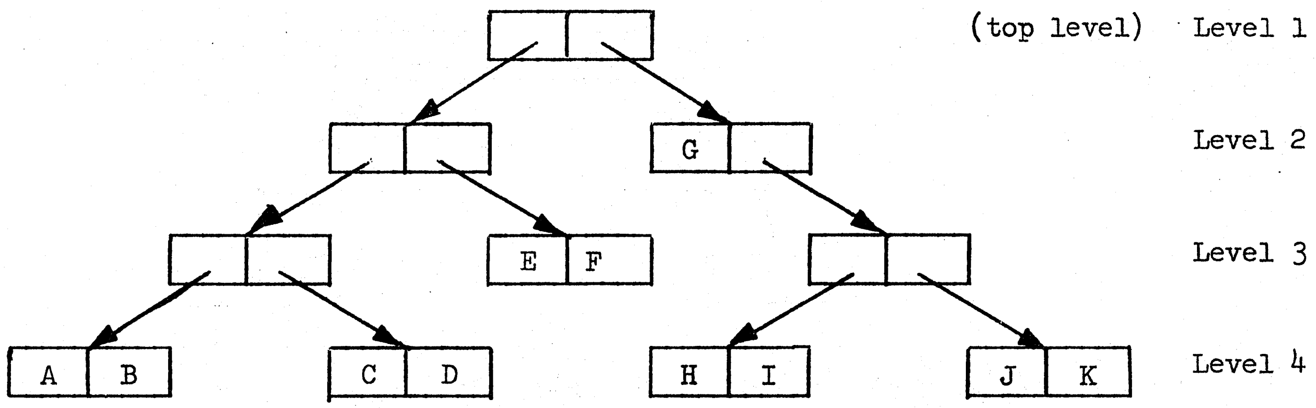
((((A . B) . C) . D) . (DD . (CC . (BB . AA))))



((((A . B) . (A . B)) . (A . B)) . (A . B)



```
((((A . B) . (C . D)) . (E . F)) . (G . ((H . I) . (J . K))))
1234    4    4     43   3     32   2   34    4   4     4321
```



In this example we have numbered the parentheses (a tutororial aid that is not a legal part of S-expressions) and labeled the graph nodes according to their subexpression depth. The correspondence between a parenthesis subscript and a graph level is one-to-one and clearly illustrates the structural meaning of the S-expression. With more complicated S-expressions we have a deeper and larger graph. Thus, we can see that S-expressions can be of unlimited size and complexity, constrained only by the physical memory capacity of the computer.

2.4 EXERCISES

Which of the following are atomic symbols?

1. ATOM
2. 1234A
3. A1B2C3
4. NIL
5. (X)
6. LISP
7. Q32
8. ONE
9. (MY . NAME)
10. 2TIMES

Identify the dotted pairs.

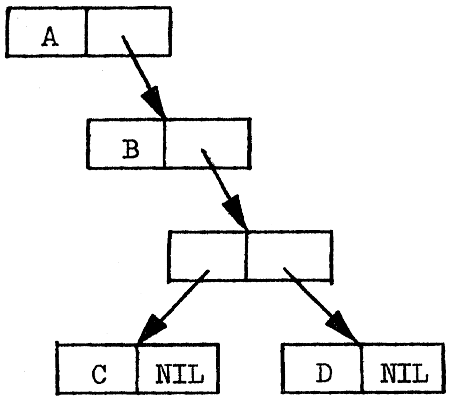
11. A . B
12. X . Y . Z
13. (YOU . AND . ME)
14. (X . Y)
15. (NIL . NIL)

Graph these dotted pairs.

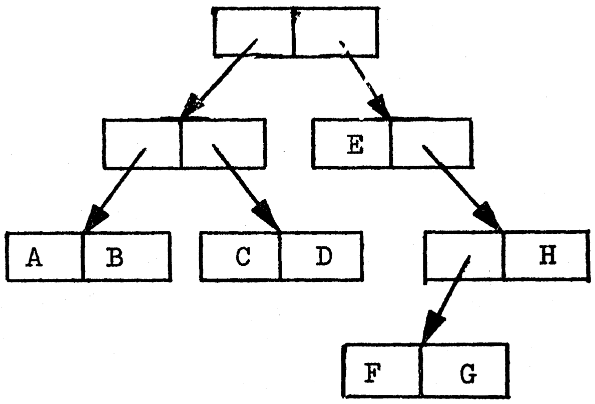
16. (ONE . (TWO . THREE))
17. (((THREE . NIL) . TWO) . ONE)
18. ((A . B) . (B . (C . D)))

What S-expressions are these structures?

19.



20.



CHAPTER 3. SYMBOLIC EXPRESSIONS IN LIST NOTATION

Dot notation is necessary and sufficient to represent <u>all</u> list structures in LISP, and, in fact, is the fundamental conception upon which the programming language is built. However, it leaves much to be desired as a convenient programming notation for S-expressions, particularly the excess of parentheses and dots. List notation was invented to improve this situation and simplify the readability and writability of S-expressions.

For example, the list (A B C D)

is an S-expression in list notation for the same S-expression

(A . (B . (C . (D . NIL))))

written in dot notation. The atom NIL has special significance and will be discussed shortly.

3.1 LIST ELEMENTS

A list may have sublists, and these sublists may also have sublists. It is usually convenient to speak of "elements" of a list, or sublist. As used in this primer a list element is either an atom or a sublist. When a sublist has sublists and atoms, these are elements of the sublist.

For example (A B C)

is a list with three elements A, B and C.

Whereas (A (B C))

is a list of the two elements, A and (B C).

The second element (B C)

is a sublist of two elements, B and C.

Historically, the separator for elements of lists was the comma; however, one or more blanks are now generally used and either is acceptable.

Thus the two S-expressions (A B C D)

and (A,B,C,D)

are entirely equivalent lists in LISP.

The student should be cautioned that though much of the LISP programming language is written as S-expressions in list notation, the basis for these S-expressions is always dot notation. In fact all S-expressions in list notation can be transformed into their dots notation equivalents, but not all S-expressions in dot notation can be transformed into list notation. This will be evident after we examine the rules and identities required for translating between notations.

3.2 NIL

About the turn of the century theoretical physics was in a dilemma. Was light emission a wave or a particle phenomenon? Ample evidence existed to support either school of thought. Physics resolved its dilemma by considering light as a wavicle.

LISP also has a dilemma, resolved in a similar fashion. The dilemma is what to do with an empty list, i.e.,

( )

The solution is to define an atom, called NIL, that is entirely equivalent to the empty list. Like the wavicle of physics, NIL is simultaneously an atom and a list. LISP programmers can use either form when appropriate, as they are identically represented, internally.

Consistent with this definition of NIL, we use NIL as a terminator of all lists. For example, the list

(A B C)

has three elements, A, B, and C. If we walk down this list removing each element we encounter, the list gets shorter as follows:

```
(A B C)
(B C)
(C)
()
```

After we have removed the last element C, we are at the end of the list and what remains is the empty list, NIL. We consider NIL, not as an element of the list, but as the terminator of the list.

3.3 TRANSFORM: LIST NOTATION TO DOT NOTATION

All nonatomic S-expressions are defined as dotted pairs. It is therefore possible to transform an S-expression in list notation to its equivalent form in dot notation. The following rules and identities define the transformation.

Identity 1: A list of one atom is a dotted pair of the atom and NIL with NIL always the right part of the dotted pair, i.e.,

```
(atom) ≡ (atom . NIL)
```

or equivalently

```
(atom) ≡ (atom . ( ))
```

Examples:

```
               (A) ≡ (A . NIL)
   (EXTRALONGATOM) ≡ (EXTRALONGATOM . NIL)
     (NIL) )     ( (NIL . NIL)
           }  ≡  {
     (( )) )     ( (( ) . ( ))
```

When transforming a multi-element list to its equivalent form in dot notation we begin by composing the dot notation equivalent for only the top level elements of the list. We then compose the dot notation equivalent for each sublist, and so on until the list is completely transformed to dot notation. All we need then is a rule for transforming a simple list to its dot notation equivalent, and repeating that rule for all sublists. We can now state that rule.

Rule 1: The first (left-most) list element when transformed to dot notation is always the left part of a dotted pair. If the first element is also the last element of the list, by identity 1 it is dotted with NIL. If the first element is not the last element of the list, then the right part of the dotted pair is the list formed by removing the first element. Then apply rule 1 to the right part of the dotted pair.

For example, given the list

```
(A B C)
```

we apply rule 1 and get

```
(A . (B C))
```

Since (B C)

is the right part of the dotted pair and is itself a list, we apply rule 1 again to get

```
(A . (B . (C)))
```

Again, the right part is a list (C) so we apply rule 1 once more. We note, however, that the list (C) satisfies identity 1 and is equivalent to

```
(C . NIL)
```

Hence the final S-expression is given by

```
(A . (B . (C . NIL)))
```

For another example, the list

```
(A (B C) D)
```

yields these partial expansions for each application of rule 1.

```
(A . ((B C) D))
(A . ((B C) . (D)))
(A . ((B C) . (D . NIL)))
```

Now expanding the sublist (B C) we find

```
(A . ((B . (C)) . (D . NIL)))
(A . ((B . (C . NIL)) . (D . NIL)))
```

Examples:

```
(A B C)         ≡     (A . (B . (C . NIL)))
((A B)C)        ≡     ((A . (B . NIL)) . (C . NIL))
(A B (C D))     ≡     (A . (B . ((C . (D . NIL)) . NIL)))
((A))           ≡     ((A . NIL) . NIL)
((NIL))         ≡     ((NIL . NIL) . NIL)
(())            ≡     (NIL . NIL)
(A (B . C))     ≡     (A . ((B . C) . NIL))
```

From the above examples one can see that identity 1 can be stated alternatively as: When converting from list to dot notation, the only atom that appears adjacent to a right parenthesis is NIL.

3.4 TRANSFORM: DOT NOTATION TO LIST NOTATION

It is always possible to convert list notation to dot notation since S-expressions are defined by dot notation. However, we cannot always convert dot notation to list notation. For example, we cannot so

transform (A . B)

The rule that is in effect derives from identity 1.

Rule 2: Only those dotted pairs in which the only atom adjacent to a right parenthesis is NIL can be represented in list notation.

For complicated dotted pairs the following procedure can be followed starting with the most nested dotted pair:

1. If the right part of the dotted pair is an atom and not NIL, conversion to list notation is impossible.
2. If the right part of the dotted pair is non-atomic (i.e., a list or a dotted pair) or NIL (treat NIL here as ()), then
   a) delete the last right parenthesis of the dotted pair
   b) delete the dot
   c) delete the first left parenthesis of the right part; The left part thereby becomes the first element of the list
   d) repeat the procedure on the next higher level dotted pairs.

For example, given the dotted pair

(A . (B . NIL))

the most nested dotted pair is

(B . NIL)

Representing NIL by ( ) and applying the procedure above we find

(A . (B))

Applying the procedure again we get the list

(A B)

For the case

(A . ((B . C) . (D . NIL)))

repeated application of the procedure yields these expressions

(A . ((B . C) . (D)))
(A . ((B . C) D))
(A (B . C) D)

We can reduce this list no further as the second element of the list

(B . C)

is a dotted pair that cannot be represented as a list. We call the expression

(A (B . C) D)

a list, but recognize that it is in mixed notation. Mixed notation is perfectly acceptable to Q-32 LISP and is quite common in LISP S-expressions.

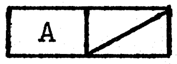
3.5 GRAPHICAL REPRESENTATION OF LISTS

S-expressions written in list notation can be transformed into identical S-expressions in dot notation; graphical representation of S-expressions is covered in section 2.3. This section will review that material but with the introduction of NIL.

Inasmuch as NIL is an atom, we need not introduce any new symbology. However, since we use NIL as a list terminator, a diagonal slash is often preferred and is adopted here. Thus the graph for

(A . NIL)

is



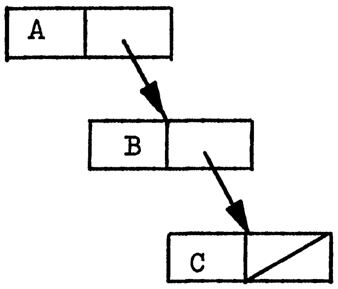
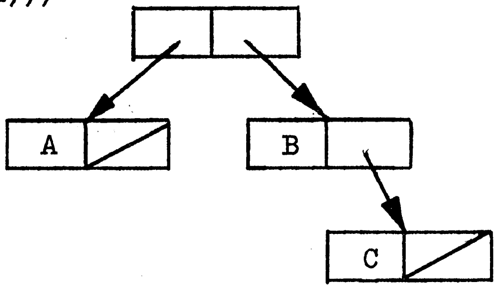
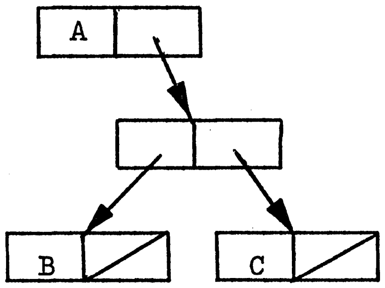
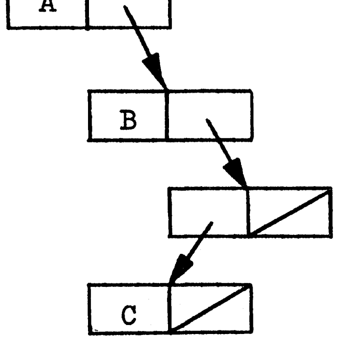
But

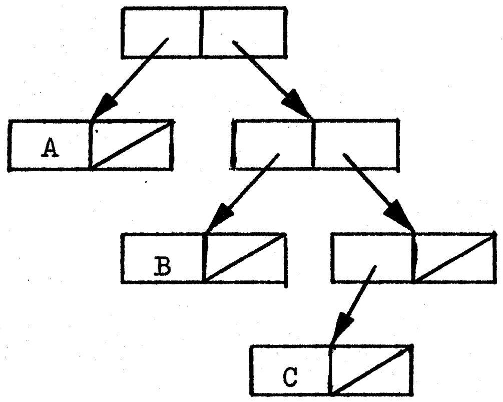
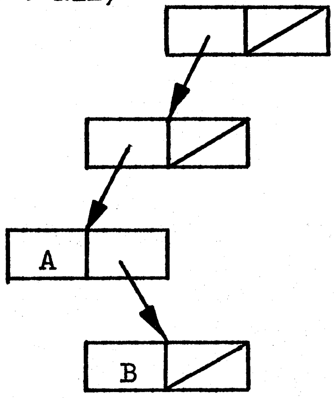
(A . NIL) ≡ (A)

so the graph also shows a single element list.

For more complicated lists we shall show the list, its dotted pair equivalence, and its graph.

Examples:

List	Dotted Pair	Graph
(A B C)	(A . (B . (C . NIL)))	
((A) B C)	((A . NIL) . (B . (C . NIL)))	
(A (B) C)	(A . ((B . NIL) . (C . NIL)))	
(A B (C))	(A . (B . ((C . NIL) . NIL)))	

List	Dotted Pair	Graph
`((A) (B) (C))`	`((A . NIL) . ((B . NIL) . ((C . NIL) . NIL)))`	
`(((A B)))`	`(((A . (B . NIL)) . NIL) . NIL)`	

3.6 EXERCISES

Transform these lists to their fully expanded dot notation equivalents.

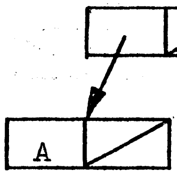
1. (ATOM)
2. ((LISP))
3. (((MORE YET)))
4. (HOW ABOUT THIS)
5. (DONT (GET (FOOLED)))

Now go the other way--dotted pairs to lists.

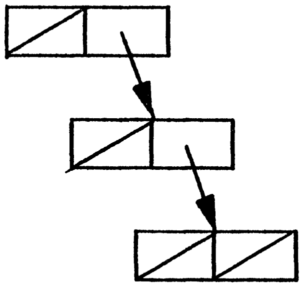
6. (X1 . NIL)
7. (NIL . (X1 . NIL))
8. (KNOW . (THY . (SELF . NIL)))
9. ((BEFORE . (AND . (AFTER . NIL))) . NIL)
10. (A . (((B . (C . NIL)) . NIL) . NIL))

To what S-expressions do these graphs correspond?

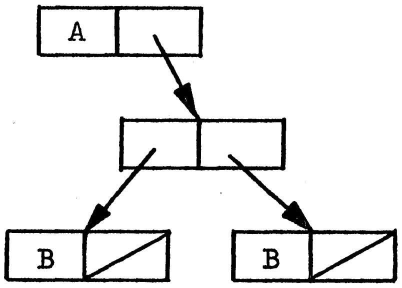
11.



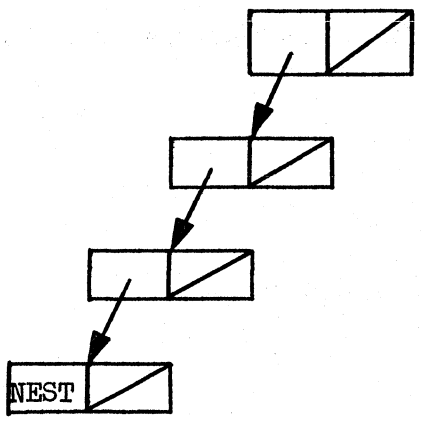
12.



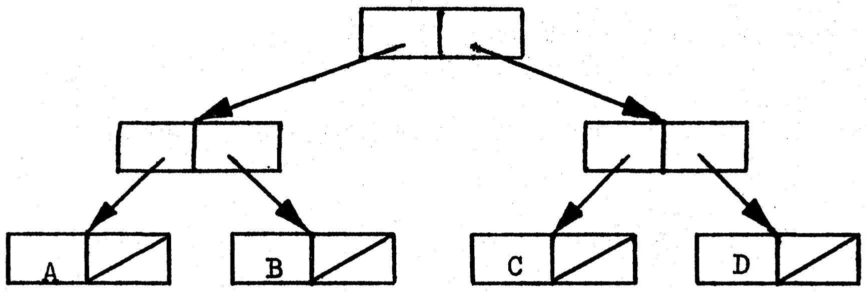
13.



14.



15.



CHAPTER 4. ON-LINE OPERATION

If you're still with me we can now try some exercises under time-sharing. Those familiar with TSS should LOAD LISP and then skip to paragraph 4.1. The uninitiated should perform steps 1 through 5, in order, as noted below.

1. "cr" is the symbol used herein to indicate you must depress the carriage return key. This transmits your input to the Time-Sharing System (TSS) Executive. Nothing happens until this key is depressed.

2. "$" is the symbol which prefixes all TSS output messages.

3. If you err (which is human) you may cancel the whole line by entering an exclamation point (!). You may also cancel the last uncancelled character for each depression of the rubout key. (n depressions cancels the n last characters including blanks.)

Enter Literally	Step	Comment
!LOGIN 1234 SDC23 LISP cr	1.	Announces your presence to TSS and requests LISP to be loaded. For 1234 use your man number, and for SDC23 use your work order number.
$LOAD xy	2.	Best of four possible TSS responses to step 1; xy is your channel number. Skip to paragraph 4.1.
$WAIT $LOAD xy	3.	As good as step 2. Skip to paragraph 4.1.
$NO LOAD DRUMS FULL	4.	Very bad! System is too busy. Type QUIT cr and try again later.
$WAIT $NO LOAD DRUMS FULL	5.	As bad as step 4. Type QUIT cr and try again later.

4.1 Q-32 LISP AMENITIES

We now have LISP loaded and available at the teletype. Type ! GO cr. LISP will "speak" to you and tell you the date, time, Q-32 LISP model number and that it is ready. But it is not ready until it has rung the bell twice. This is important since it is your only cue that LISP is waiting for your input. Q-32 LISP always rings the bell when it is ready for more input.

LISP is notoriously parenthesis sensitive so take care to parenthesize properly. You may use as many blanks or commas as you wish to delimit atoms. Left and right parentheses are always delimiters so blanks or commas before or after them are optional. Dot also acts as a delimiter between non-numeric atoms so again blanks or commas are optional. However, as we shall see later, numbers are atomic symbols and ambiguity between a real (floating point) number and a dotted pair can result if the dot is not set off by blanks. Thus, it is a good habit always to surround the dot with blanks.

4.2 EXERCISE PREAMBLE

Q-32 LISP accepts both list and dot notation, but always outputs in simplest list notation. Thus, you can use it to test your answers to the following exercises, as it will transform dot notation into list notation. For the time being, always enter your inputs in the following form:

```
PRINT (s)
```

where s is the S-expression you wish to try. LISP will print your answer twice, for reasons that will be clear later on, and then ring the bell for the next input, or it will print out an error message and ring the bell. For example, enter problem 1 of the exercises below as:

```
PRINT (  (X . Y)  )
```

where in this case s equals

```
(X . Y)
```

4.3 EXERCISES

Which of the following are S-expressions?

```
 1.  (X . Y)
 2.  SDC
 3.  (YOURFIRSTNAME . YOURLASTNAME)
 4.  (YOURFIRSTNAME YOURLASTNAME)
 5.  ((YOURFIRSTNAME) (YOURLASTNAME))
 6.  ((YOURFIRSTNAME) . (YOURLASTNAME))
 7.  (Q32 . TIME . SHARING)
 8.  ((AM . I) . (SMART . NIL))
 9.  NIL
10.  ((((( )))))
```

How about this one;

```
11.  (((Z . Z) . B)
```

LISP should ring the bell rather than print an answer since #11 is <u>not</u> an S-expression because it is short one right parenthesis. Type in this missing parenthesis and see if it prints.

Finally, try this:

12. (AN . EXTRALONG$\textcircled{cr}$
ATOMSTRING)

Q-32 LISP ignores line boundaries, thereby allowing atoms, and S-expressions to be "split" across lines. (The carriage return is ignored.) If you end an input to LISP with an atomic symbol, most frequently NIL, always insert a blank before the carriage return to delimit the atom, otherwise LISP will ring the bell on the following line believing there are more characters to be entered. If this happens, just enter a blank, return carriage, and all will be OK.

Convert the following S-expressions to list notation if possible. Put the expression in the simplest list form. Use mixed notation if necessary. Check your answers on the computer using PRINT as above.

```
13. (A . NIL)
14. (NIL . NIL)
15. (A . (B . (C . NIL)))
16. (A . (B . (C . D)))
17. ((A . (B . NIL)) . NIL)
18. ((A . NIL) . ((B . NIL) . NIL))
19. ((A . (B . NIL)) . ((C . NIL) . NIL))
20. ((X . NIL) . ((NIL . Y) . NIL))
```

CHAPTER 5. NUMBERS

In Q-32 LISP, numbers are atoms and may be used in S-expressions exactly as the previously defined atomic symbols. Thus

(1 2 3 A 4 B 5)

or

(ALPHA . 960)

are legal S-expressions.

Integer, octal, and floating point numbers are all legal LISP numerical types, and in arithmetic functions mixed data types are converted properly by the LISP system.

5.1 INTEGER NUMBERS

The only fixed-point numbers are integers, positive or negative, with or without a positive scale factor. The scale factor is denoted by the letter E followed by a blank, zero, or any positive integer. Negative scaling is illegal and not meaningful for fixed-point numbers. Thus

796E-17

is unacceptable for Q-32 LISP.

Examples:

LISP Number	Meaning
123	+123
+123E0	+123
-321E	-321
-1E3	$-1X10^3 = -1000$
53E0	+53

5.2 OCTAL NUMBERS

Integers may also be represented in octal. Octal numbers are denoted by an optional sign followed by octal digits followed by the letter Q followed by a blank, a zero, or any positive decimal integer. The Q must be present. The decimal integer following Q is a scale factor showing the power of eight. Negative scale factors are illegal and not meaningful for fixed-point numbers whether represented in octal or decimal.

Thus

75757Q-4

is unacceptable for Q-32 LISP.

The largest octal number allowed is the 16 digit

7777777777777777Q = -0

or a word of all one's on the Q-32.

Examples:

LISP Number	Meaning
123Q	$(123)_8$
123Q2	$(12300)_8$
777Q3	$(777000)_8$
-123Q	$(7777777777777654)_8$
2Q8	$(200000000)_8$
3Q10	$(30000000000)_8$

5.3 FLOATING-POINT NUMBERS

Numbers in Q-32 LISP are always integers unless they contain a decimal point. Floating-point numbers, therefore, must contain a decimal point. The decimal point must not be in the first character position. Floating-point numbers may be positive or negative, with or without a positive or negative scale factor. The scale factor is always denoted by the letter E and may be followed by a blank, zero, or any positive or negative integer.

Examples:

LISP Number	Meaning
3.14159	+3.14159
+1.0E-3	+0.001
-976.003E3	-976003.00
0.273E+2	+27.30
23.E-1	+2.3
17.	+17.0

Floating-point numbers are accurate to 10 significant figures. Remember, a floating-point number must not begin with a decimal point.

Thus

.123

or

.123E+3

are unacceptable for Q-32 LISP.

5.4 DECIMAL POINT OR DOTTED PAIR RESOLUTION

When floating-point numbers are used in S-expressions, the computer can be confused as to the meaning of the period. Is it treated as a decimal point or as the dot in a dotted pair? To eliminate confusion and avoid ambiguity always surround the dot with blanks when writing a dotted pair, and never surround the decimal point with blanks when writing a floating-point number.

The LISP system always assumes the first period embedded in a numerical field is a decimal point. A second period embedded in a numerical field will be taken as the dot for a dotted pair. For instance, if the expression

```
(1.2.3.4)
```

were given to Q-32 LISP the system would consider the expression as the dotted pair of two floating-point numbers, namely

```
(1.2 . 3.4)
```

5.5 EXERCISES (Use the computer with PRINT to check your answers.)

Which of the following are S-expressions?

```
 1.  (Q . 1Q)
 2.  (5E . (E . NIL))
 3.  (E5 . 5E)
 4.  (1.E . 1Q)
 5.  ANFSQ32
 6.  4.4
 7.  (A.9)
 8.  (B.9.9)
 9.  (9.9.9)
10.  (1.23 77Q3 27 27E5 0.321E-7 ALPHA Q.32)
```

Convert the following to list notation, if possible.

```
11.  (99.9 . NIL)
12.  (NIL . 99.9)
13.  ((PI) . 3.14159E0 . NIL)
14.  (5 . (5.5 . (5Q5 . (55.0E-1 . (5E2 . NIL)))))
15.  ((13.13 . NIL) . ((25Q2 . NIL) . NIL))
```

CHAPTER 6. ELEMENTARY FUNCTIONS

LISP is a language for manipulation of S-expressions. Fundamental to this manipulation is the ability to build S-expressions from smaller S-expressions and produce subexpressions from a given S-expression. These abilities are possible with the elementary LISP functions CONS, CAR, and CDR.

6.1 RAPPORT WITH THE SUPERVISOR

Before we examine the elementary functions we must understand a basic element of the syntax of the communication language accepted by the Q-32 LISP system. Chapter 17 covers the subject in greater detail. At this juncture we will only consider the requisite parenthesization.

When we type input to the Q-32 LISP system, we are communicating with a supervisor program that always expects <u>two</u> inputs, both S-expressions. If we call this pair of S-expressions s_1 and s_2, respectively, the first S-expression s_1 is always treated by the supervisor as:

1. the name of a function, or
2. an S-expression that behaves as a function.

(We will focus on the former case here, and examine the latter case in subsequent chapters.) Since functions have arguments, the second S-expression s_2 is <u>always a list</u> of the arguments for the function whose name is the S-expression s_1.

Consider the trigonometric function

$$\text{SIN } 90^\circ$$

If SIN were a LISP function, we would write

```
SIN (90)
```

where the first S-expression s_1 is SIN and the second S-expression s_2 is the list (90)--the list of the single argument required by SIN.

As another example, in LISP the function PLUS performs the arithmetic addition of its arguments. We can compute the sum of three numbers by giving the following pair of S-expressions to the supervisor:

$$\underbrace{\text{PLUS}}_{s_1}\ \underbrace{(1\ 2\ 3)}_{s_2}$$

The S-expression s_1 is the name of the function PLUS. The S-expression s_2 is a list with three elements (i.e., 1,2,3) each an argument for the function PLUS.

6.2 CONS

CONS refers to "the construct of" and is the function that is used to build S-expressions. It has two arguments that are both S-expressions.

Definition: The CONS of two S-expressions is the dotted pair of these arguments, with the first argument the left part and the second argument the right part of the dotted pair.

For example, given the arguments A and B we can CONS them by addressing the supervisor with

$$\underbrace{\text{CONS}}_{s_1}\ \underbrace{\text{(A B)}}_{s_2}$$

which means (A . B)

If the arguments were the lists (A) and (B) we would write

$$\underbrace{\text{CONS}}_{s_1}\ \underbrace{\text{((A) (B))}}_{s_2}$$

which is equivalent to ((A) . (B)) = ((A) B)

Examples:

CONS(M N) = (M . N)
CONS((A . B) C) = ((A . B) . C)
CONS(A (B C D)) = (A . (B C D)) = (A B C D)

6.3 CAR

CAR is a LISP function used to break S-expressions into subexpressions. Its meaning is "the first of." It has one argument, a non-atomic S-expression (i.e., a dotted pair, or a list).

Definition: The CAR of a non-atomic S-expression is the left part of the S-expression when represented in dot notation or the first element of the S-expression when represented in list notation. The CAR of an atom is undefined.

For example, the CAR of the argument (M . N) would be written

$$\underbrace{\text{CAR}}_{s_1}\ \underbrace{\text{((M . N))}}_{s_2}$$

which is equivalent to M

Examples:

```
CAR((A . B)) = A
CAR(((A . B) . C)) = (A . B)
CAR((A B C D)) = A
CAR(((A B C) D E)) = (A B C)
CAR(ATOM) = undefined for atoms
```

6.4 CDR

CDR is another LISP function used to break S-expressions into sub-expressions. Its meaning is "the rest of." It has one argument, a non-atomic S-expression similar to that accepted by CAR. CAR of a given non-atomic S-expression yields the first element as an S-expression and CDR yields the rest of that S-expression after the CAR is removed.

Definition: The CDR of a non-atomic S-expression is the right part of the S-expression when represented in dot notation or the balance of the S-expression after the first element is removed when represented in list notation. The CDR of an atom is undefined.

For example, the CDR of the argument (M . N) would be written

$$\underbrace{\text{CDR}}_{s_1}\ (\ \underbrace{(\text{M . N})}_{s_2}\ )$$

which is equivalent to N

Do not confuse list and dot notation when evaluating the CDR. If the CAR of list

(A B)

is removed, the remainder is still a list

(B)

If I remove the CAR of the dotted pair

(A . B)

I have the atom B.

Thus

```
CAR ((A . B)) = A
CDR ((A . B)) = B
CAR ((A B)) = A
CDR ((A B)) = (B)
```

Examples:

```
CDR((A . B)) = B
CDR((A . (ATOM))) = (ATOM)
CDR((A B C D)) = (B C D)
CDR(((A B C) D E)) = (D E)
CDR(NIL) = undefined for atoms
```

Note: The CDR of a list with only one element is the atomic symbol NIL. For example:

```
CDR((ATOM)) = ( ) = NIL
```

6.5 GRAPHICAL INTERPRETATION OF CAR AND CDR

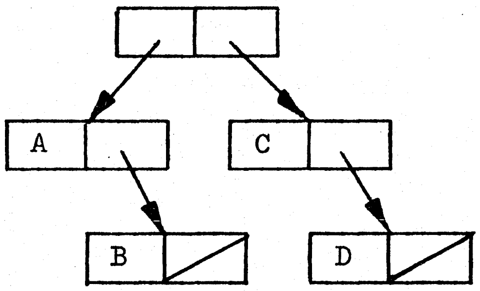
In the previous Chapters we examined the graphs of LISP S-expressions and noted the binary tree structure of these expressions. Let us now examine the meaning of the elementary functions that operate on these tree structures.

If someone asked for directions to get to your home, you would most naturally couch such directions in terms of city blocks, and house numbers. In LISP, we are faced with a similar problem: to provide the LISP system with directions for "traveling" through the binary-tree-structured representation of an S-expression. We couch such directions in terms of CAR's and CDR's, which designate the appropriate "turn" at each binary "fork" in the "road."

For example, given the S-expression

```
((A B) C D) = ((A . (B . NIL)) . (C . (D . NIL)))
```

its graph is given by



Now the CAR and CDR of this S-expression yield

```
CAR ( ((A B) C D) ) = (A B)
CDR ( ((A B) C D) ) = (C D)
```

In the graph we note that

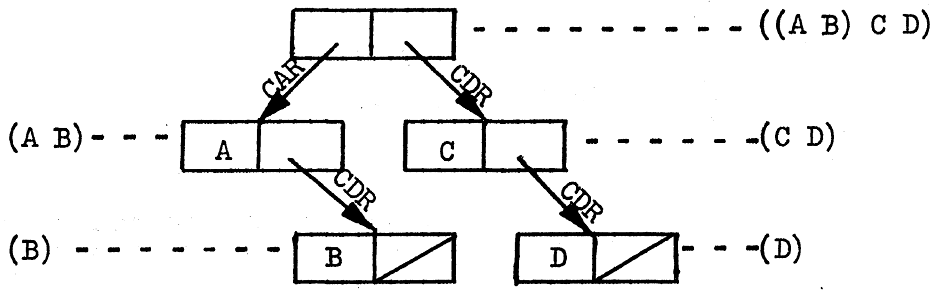
(A B)

is the left branch of the top node and

(C D)

is the right branch. The connecting arrows of this graph were called "pointers" in Chapter 2.

We see now that they are pointers to the CAR and CDR. We often say they point to the "CAR chain" or "CDR chain" of the structure. If we take the CAR and CDR repeatedly at each node, we can completely "traverse" the S-expression, and reach any sub-expression or atom of the original S-expression. The following graph is completely labeled according to these CAR and CDR directions, and the "location names" of each node.



To get from the original S-expression to the atom D we require the following directions:

1. CDR (((A B) C D)) = (C D)
2. CDR of the output of (1), i.e.,
   CDR ((C D)) = (D)
3. CAR of the output of (2), i.e.,
   CAR ((D)) = D

A shorthand for this could be

```
CADDR ( ((A B) C D) ) = D
```

and in fact is. Much of LISP programming is composing "directions" of this variety. We shall cover this fully as the subject "composition of functions" in Chapter 8. Meanwhile problems 21-31 of this chapter give us some exercise in "finding our way home."

6.6 EXERCISES

Evaluate the following functions and then test your results by entering each problem, exactly as shown, to Q-32 LISP.

```
 1.  CAR((LEFT . RIGHT))
 2.  CDR((LEFT . RIGHT))
 3.  CONS(LEFT RIGHT)
 4.  CAR((A B C D))
 5.  CAR(((A) B C D))
 6.  CAR((A (B C D)))
 7.  CAR(((A . B) C D E))
 8.  CDR((THIS SENTENCE IS A LIST))
 9.  CDR((HOW (ABOUT THIS)))
10.  CDR(((DOT . PAIR1) (DOT . PAIR2)))
11.  CONS(CAR CDR)
12.  CDR((EMPTY))
13.  CDR((CAR CDR))
14.  CAR(((CAR) CDR))
15.  CONS(A ( ))
16.  CONS(75Q 100)
17.  CAR((1 . (2.0 . (30.0E-1 . 77Q))))
18.  CDR((1 . (2.0 . (30.0E-1 . 77Q))))
19.  CONS((A . B) NIL)
20.  CAR((((((ALPHA))))))
```

Note: Problems 1, 2, and 3 above demonstrate the relationship among CONS, CAR, and CDR. Can you state this relationship?

List from right to left the sequence of CAR-CDR LISP functions which, when each is applied to the value of the prior function, will find the "A" in each of the following S-expressions. For example:

CAR CDR

is the answer for the argument (C A T) by the following reasoning:

CDR ((C A T)) = (A T)

then

CAR ((A T)) = A

Q.E.D.

21. ((C A T))
22. ((A))
23. (M A R T)
24. (B . A)
25. (S M A R T)
26. (1 2Q 3E3 A)
27. ((A . B) (C . D))
28. ((B . A) (C . D))
29. (((C)) ((A)))
30. ((X . Y) (A . B))
31. ((X . Y) (B A))

Test your answers on the computer as follows:

1. Make an abbreviation for each of your lists of multiple CAR's and CDR's by forming a function name that begins with C and ends with R, and has as many A's and D's between them to correspond to each CAR and CDR, respectively, in your list.

2. Apply that function to its corresponding argument.

For the argument (C A T), above we found the answer CAR CDR.

$$\mathrm{C\underline{A}R\ C\underline{D}R = C\underline{AD}R}$$

Now we can try CADR directly on the argument with

CADR ((C A T)) = A

Note: Q-32 LISP has these abbreviations available only up to <u>four deep</u> (which is sufficient for problems 21 through 31), e.g., CAAAAR, CAAADR , , , CADDDR, CDDDDR.

CHAPTER 7. LAMBDA NOTATION

In LISP 1.5, as in other programming languages, we wish to write programs that are parameterized and can compute answers only after values have been assigned to the parameters of the program. However, in LISP 1.5, we do not use the syntax and program structure of algebraic languages. LISP programs are conceived and written with mathematical rigor based upon the formalism of function theory. As such, parameters are called variables, subroutines are analogous to "forms", procedures are functions and functional expressions, and computation is a process of "evaluation" of S-expressions.

7.1 FORMS AND FUNCTIONS

Given the algebraic expression

$$y^2 + x$$

evaluate the expression for the values 3 and 4.

For this problem statement, we immediately see a notational problem. Is x=3 and y=4, or vice versa? The value of the expression changes with our assumption. To resolve this ambiguity we need a notation that explicitly states the correspondence between variables and their values. LISP uses such a notation, the LAMBDA notation of Alonzo Church.*

Church's LAMBDA notation asserts that the expression

$$y^2 + x$$

is a form. In LISP 1.5 this form would be written as

```
(PLUS (TIMES Y Y) X)
```

Furthermore, Church's LAMBDA notation asserts that

$$f = \lambda(x,y)(y^2 + x)$$

is a function named "f," since it provides the two necessary ingredients for a function:

1. a form to be evaluated, and
2. a correspondence between the variables of the form and the arguments of the function.

* A. Church, The Calculi of LAMBDA Conversion, Princeton University Press, Princeton, New Jersey, 1941.

If we now ask the value of the function f for

$$f(3,4)$$

the previous ambiguity is resolved as Church's LAMBDA notation explicitly gives the number and order of the arguments of f and defines the correspondence of 3 with x, and 4 with y such that

$$f(3,4) = 4^2 + 3 = 19$$

In LISP, f(3,4) would be written as

$$\underbrace{\texttt{(LAMBDA } \underbrace{\texttt{(X Y)}}_{\text{list of variables}} \underbrace{\texttt{(PLUS (TIMES Y Y) X)}}_{\text{form}}\texttt{)}}_{s_1} \underbrace{\texttt{(3 4)}}_{s_2}$$

and s_1 is called a LAMBDA expression. A LAMBDA expression is our first example of a functional expression, i.e., an S-expression that acts like a function. We shall explore LAMBDA expressions more fully below.

7.2 LISP FORMS

Earlier, we noted that the syntax of input to the Q-32 LISP supervisor consists of two S-expressions s_1 and s_2. We further learned that s_1 is either a function or a functional expression and s_2 is a list of the arguments for s_1. When we type the s_1, s_2 pair, we are composing a LISP form because we have an S-expression that can be evaluated. We speak of this particular form as a top-level form. However, much of LISP programming is written as forms at other than the top level. All such lower-level forms have a different format, which we will examine here.

As a rule a LISP form is:

1. an S-expression that can be evaluated when composed as part of a functional expression, or
2. a functional expression or function applied to a list of arguments.

For example, when performing addition at the top level we write

$$\underbrace{\texttt{PLUS}}_{s_1}\ \underbrace{\texttt{(1 2 3)}}_{s_2}$$

The pair of S-expressions is a form by (2) above. If, however, we wish to compose a functional expression that, say, doubles the value of its argument, we start with a form that uses PLUS at lower than top level and write this form as

```
(PLUS X X)
```

The first thing to note is that this form is a single S-expression bounded by the left and right parentheses. These parentheses delimit the scope of the function PLUS. The form includes, in order from left to right, a function name and the variables on which to compute.

In general, a lower-level form has the following format:

```
(function-name-or-functional-expression   variables-or-other-forms)
```

The form used in the LAMBDA expression, above,

```
(PLUS (TIMES Y Y) X)
```

is a valid lower-level form that contains a nested form

```
(TIMES Y Y)
```

as one of its arguments. From all this we get a very powerful rule of thumb.

Rule: In a lower-level form, all atoms adjacent to a left parenthesis (except quoted expressions, which we shall examine subsequently) are treated as function names.

Remember that lower-level forms cannot be evaluated directly by the supervisor as are top-level forms. To evaluate lower-level forms, make them part of a functional expression, such as a LAMBDA expression.

7.3 LAMBDA EXPRESSIONS

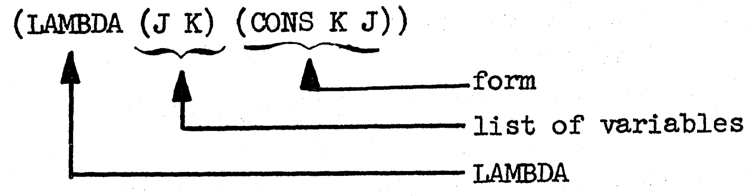
Definition: A LAMBDA expression is an S-expression. This S-expression is a list of three elements in the following order:

1. the word LAMBDA
2. a list of non-numeric atoms that may be used as variables in the form
3. the form itself.

The general format accepted by Q-32 LISP is:

```
(LAMBDA   list-of-variables   form)
```

For example:



A LAMBDA expression is a functional expression and may be used wherever functions are acceptable in top-level forms and lower-level forms. A LAMBDA expression acts like a function since it specifies the correspondence between the variables in the form and the arguments of the function. Therefore, it can be applied to arguments just as the elementary functions CONS, CAR, and CDR were used earlier.

7.4 EVALUATING LAMBDA EXPRESSIONS

When using a LAMBDA expression in a top-level form, the LAMBDA expression is the first S-expression, s_1, of the pair presented to the supervisor. Again, the second S-expression, s_2 of the pair is the list of arguments for s_1; in this case, the list of arguments for the LAMBDA expression. It is important to understand that the arguments in the list, s_2, are paired with the variables of the form in the LAMBDA expression, s_1. The arguments in the list, s_2, are matched in number and position with the variables in the list of variables following the LAMBDA. The process of pairing in this manner and then evaluating the form inside the LAMBDA expression is called LAMBDA conversion.

The doublet

$$\underbrace{\text{CONS}}_{s_1}\ \underbrace{(\text{ A B })}_{s_2}$$

is a simple top-level form. So is the doublet

$$\underbrace{(\text{LAMBDA (J K) (CONS J K)}}_{s_1}\ \underbrace{(\text{ A B })}_{s_2}$$

In fact, they yield the same value, (A . B). Note, however, that by LAMBDA conversion, the argument A is paired with the variable J, and the argument B with the variable K. Then when the form within the LAMBDA expression

(CONS J K)

is evaluated, the variables J and K are evaluated to yield A and B, respectively; and it is these arguments that are CONSed.

Three rules should be remembered concerning LAMBDA conversion.

<u>Rule 1</u>: When evaluating a form, the number and order of the arguments in the list of arguments must always match the number and order of the variables in the list of variables of a LAMBDA expression, even if the number of variables is zero; i.e., an empty list of variables; e.g.,

$$\underbrace{\text{(LAMBDA NIL 3.14159)}}_{s_1}\ \underbrace{\text{NIL}}_{s_2} = 3.14159$$

$$\underbrace{\text{(LAMBDA ( ) 3.14159)}}_{s_1}\ \underbrace{\text{( )}}_{s_2} = 3.14159$$

<u>Rule 2</u>: Only non-numeric atoms may be used as variables specified in the variables list of a LAMBDA expression. All variables specified need not be used in the form within the LAMBDA expression but must be matched in the arguments list, e.g.,

```
(LAMBDA (A B) (CDR B)) (1 (2 3)) = (3)
```

<u>Rule 3</u>: When evaluating a form

1. arguments are paired with variables
2. the lowest-level forms are evaluated first
3. variables evaluate to their paired arguments
4. atoms following a left parenthesis are evaluated as functions.

<u>Examples</u>:

```
(LAMBDA (X) X) (123Q) = 123Q
(LAMBDA (ABLE) (CAR ABLE)) ( (THIS IS A LIST) ) = THIS
(LAMBDA ( ) 77) NIL = 77
(LAMBDA (ONE TWO) (CONS TWO ONE)) (A B) = (B . A)
(LAMBDA (K) (CADAR K)) ( ((1 2 3) 4 5) ) = 2
```

7.5 <u>PARENTHESES</u>

The LAMBDA expression:

```
(LAMBDA (A B) (CONS A B))
```

uses six parentheses. They are very important. They designate scope or extent of expressions, i.e., where they begin and where they end. Parentheses have to be very precisely positioned. In order to understand them, we shall first number them in associated pairs:

```
(LAMBDA (A B) (CONS A B))
1       2  2 2        21
```

The first left parenthesis #1 tells the LISP system that this is the start of an expression. The final right parenthesis #1 tells the system that this is the end of the expression.

The first parenthesis #1 marks the beginning of the scope of the LAMBDA, the extent of the expression to which LAMBDA applies. The second parenthesis #1 marks the end of the scope of LAMBDA.

The first parenthesis #2 marks the beginning of a list, which,is ended by the second parenthesis #2.

Finally, the third parenthesis #2 marks the beginning of the scope of CONS, with the last parenthesis #2 ending that scope.

<u>Always</u>, all parentheses in the S-expressions of LISP occur in pairs of left and right parentheses; generally, each pair marks the scope of an expression, or bounds a list. The parentheses in LISP are never optional as they are sometimes in mathematics: they are required parts of expressions.

Note that in the example above, the sub-expressions

```
(A B)
```

and

```
(CONS A B)
```

are both bounded by parenthesis pairs labeled #2. If we consider the parenthesis numbers as "depth" counters or "levels," we see that these two sub-expressions are at the same depth, namely level two. Since the only occurrences of parentheses #1 completely bracket the LAMBDA expressions, we say that the LAMBDA expression is at level one, the top level.

Parenthesis counting is a good crutch in that it immediately identifies sub-expressions at the same level within a larger S-expression; a very useful debugging and formatting tool. In fact, Q-32 LISP printouts are always formatted by breaking the output, when it will not fit on one line, at the deepest possible depth (i.e., the highest numbered parenthesis) and then indenting one space for each level. This indenting device has also been found useful for input since literally numbering parentheses is prohibited.

7.6 EXERCISES

Try evaluating these LAMBDA expressions on Q-32 LISP.

```
 1.  (LAMBDA (X) X) (ATOM)
 2.  (LAMBDA (Y) Y) ((LIST))
 3.  (LAMBDA (J) (CAR J)) ((THREE ELEMENT LIST))
 4.  (LAMBDA (K) (CDR K)) ((THREE ELEMENT LIST))
 5.  (LAMBDA (U V) (CONS U V)) (VERY GOOD)
 6.  (LAMBDA (Y X) (CONS Y X)) (ONE (THEN . ANOTHER))
 7.  (LAMBDA (A) (CAADR A)) ((A (B . 77Q2)))
 8.  (LAMBDA (VARIABLE) (CDAR VARIABLE)) (((A B)))
 9.  (LAMBDA (J) 3.14159) (NIL)
10.  (LAMBDA ( ) 3.14159) ( )
```

Note: Problems 1, and 2 are the identity functions in that they always evaluate to their input. Problems 9, and 10 are constant functions which always evaluate to the constant specified, in this case 3.14159, regardless of the value of the argument. However, these arguments are necessary since Q-32 LISP always expects a doublet when a LAMBDA expression is encountered at level one. Also note that the list of variables in problem 10 is empty. In LISP, a function with an empty variable list is a function of no arguments. For proper LISP syntax, we must always include the list of variables, even when empty. In such cases NIL is as acceptable as ( ).

Evaluate:

```
11.  (LAMBDA (U V) U) (ALPHA BETA)
12.  (LAMBDA (U V) U) (BETA ALPHA)
13.  (LAMBDA (U V) V) (ALPHA BETA)
14.  (LAMBDA (V U) V) (ALPHA BETA)
15.  (LAMBDA (FIRST SECOND) (CAR FIRST)) ((FIRST) SECOND)
```

CHAPTER 8. COMPOSITION OF FUNCTIONS

When applied to argument lists, LAMBDA expressions and the function CAR, CDR, and CONS are evaluated and their results printed. These doublets--i.e., function, argument list--are programs--trivial to be sure, but programs.

To create more powerful programs we must be able to create more complex S-expressions. A first step in that direction is the ability to construct new functions by composition of functions.

Definition: Function composition is the concatenation of functions in such a fashion that an argument for a function at level n is the value resulting from the evaluation of a function at level n+1.

For example:

```
(LAMBDA (J) (CAR (CDR J)))
1        2 2 2   3     321
```

is a function constructed by composition of functions.

8.1 EVALUATING COMPOSED FUNCTIONS

The general rule for evaluating composed functions is to evaluate the innermost (deepest nested) expressions first, then the next-higher-level expressions and so on until the entire expression has been evaluated.

Example:

```
(LAMBDA (J) (CONS (CDR J) (CAR J))) ((A . B))
1        2 2 2    3     3 3     321 12     21
```

Pairing variable J with argument (A . B) we evaluate the form

```
(CONS (CDR J) (CAR J))
```

Beginning with the innermost expressions, J evaluates to (A . B) and

```
(CDR J) = B,    (CAR J) = A
```

These values are transmitted as the arguments for the next-higher-level expression

```
(CONS B A) = (B . A)
```

Since there are no further expressions to evaluate, (B . A) is the value of the entire expression.

8.2 NESTED LAMBDA EXPRESSIONS

Since a LAMBDA expression acts like a function, it, too, may be used for constructing larger expressions by composition of functions, in exactly the same manner and format as used with functions. Because LAMBDA expressions are lists with many parentheses, the student often loses sight of this fact. To show the simple mechanics of the construction, let us develop a nested form using CAR and a nested form using a LAMBDA expression, concurrently.

Step 1: S-Expressions As Functions

```
CAR                    |    (LAMBDA (X Y)(CONS X Y))
```

Step 2: S-Expressions As Forms

$$\texttt{(}\underbrace{\texttt{CAR}}_{\text{function}}\ \underbrace{\texttt{X}}_{\text{variable}}\texttt{)}$$

$$\texttt{(}\underbrace{\texttt{(LAMBDA (X Y)(CONS X Y))}}_{\text{functional expression}}\ \underbrace{\texttt{X Y}}_{\text{variables}}\texttt{)}$$

Step 3: Pairs For Supervisor

$$\underbrace{\texttt{(LAMBDA (X) }\underbrace{\texttt{(CAR X)}}_{\text{form}}\texttt{)}}_{s_1}\ \underbrace{\texttt{((A))}}_{s_2}$$

$$\underbrace{\texttt{(LAMBDA (U V)}\underbrace{\texttt{(}\underbrace{\texttt{(LAMBDA (X Y)(CONS X Y))}}_{\text{functional expression}}\ \underbrace{\texttt{U V}}\texttt{)}}_{\text{form}}\texttt{)}}_{s_1}\ \underbrace{\texttt{(A B)}}_{s_2}$$

Step 4: Evaluation of Step 3

X = (A)	1. Binding of variables	U = A, V = B

(CAR X) = A	2. Evaluation of forms	Bind 2nd LAMBDA's variables-
		X = value of U = A
		Y = value of V = B
		(CONS X Y) = (A . B)

A	3. Return value	(A . B)

In evaluating each of the Step 3 examples, first the variables are bound initially by the LAMBDA expression to the arguments in list s_2, then the nested forms are evaluated. In evaluating these forms, the variables are evaluated, yielding the values bound from the argument list s_2. For the nested-LAMBDA-expression example, we must bind variables twice; first the outermost LAMBDA's variables U and V are bound to the arguments in list s_2. Then, when evaluating the form within the outermost LAMBDA expression, that form itself is a LAMBDA expression and its variables X and Y must be bound. X and Y are bound to the values of the variables U and V after they are evaluated. Finally the form (CONS X Y) is evaluated, yielding the value for the total expression.

We can take these examples further by constructing even larger expressions by inserting forms wherever variables occur. Composition of functions allows us this freedom. As an example, let us expand Step 3 once.

Step 5: Composition of Functions

CAR Example:

$$\underbrace{\texttt{(LAMBDA (X) }\underbrace{\texttt{(CAR }\underbrace{\texttt{(CDR X)}}_{\text{form}}\texttt{)}}_{\text{form}}\texttt{)}}_{s_1}\ \underbrace{\texttt{((A B))}}_{s_2}$$

LAMBDA Example:

$$\underbrace{\texttt{(LAMBDA (U V) }\underbrace{\texttt{(}\underbrace{\texttt{(LAMBDA (X Y)(CONS X Y))}}_{\text{functional expression}}\ \underbrace{\texttt{(CAR U) (CDR V)}}_{\text{arguments}}\texttt{)}}_{\text{form}}\texttt{)}}_{s_1}\ \underbrace{\texttt{((A) (B C))}}_{s_2}$$

<u>Step 6: Evaluation of Step 5</u>

X = (A B)	1. Binding of variables	U = (A) , V = (B C)
(CDR X) = (B)	2. Evaluation of forms	Bind 2nd LAMBDA's variables

(CAR (CDR X)) = B		X = value of (CAR U) = A
		Y = value of (CDR V) = (C)
		(CONS X Y) = (A . (C)) = (A C)

B	3. Return value	(A C)

Of note here is the binding of variables X and Y. In Step 4, X and Y were bound to the value of U and V respectively. In Step 6, X and Y are bound to the values of the forms (CAR U) and (CDR V) respectively.

8.3 EXERCISES

Evaluate the following:

```
 1.  (LAMBDA (A B) (CAR (CONS A B)))  (43 NUMBER)
 2.  (LAMBDA (A) (CAR (CDR A))) ((ARG LIST))
 3.  (LAMBDA (A) (CDR (CAR A))) (((A)))
 4.  (LAMBDA (A B) (CDR (CONS A B)))  (NUMBER 43)
 5.  (LAMBDA (B A) (CDR (CONS A B)))  (NUMBER 43)
 6.  (LAMBDA (A B) (CAR (CDR (CONS A B))))  (X (Y))
 7.  (LAMBDA (J) (CONS (CONS J NIL) NIL)) ((LIST))
 8.  (LAMBDA (J) (CAR (CAR (CONS (CDR J) (CDR J))))) ((A B))
 9.  (LAMBDA (J) (CAR (CONS 123Q3 J))) (NIL)
10.  (LAMBDA (J) (CONS (CAR J) (CDR J))) ((A . B))
```

Note that problem 10 clearly demonstrates the relationship between CAR, CDR, and CONS.

11. CAR could be called FIRST since it finds the first element of a list. Write a LAMBDA expression using only CAR's and CDR's by composition of functions which finds the third element of a list. Try it with argument

(1 2 3 4)

For the argument

((A B C) D)

compose and evaluate your own LAMBDA expressions using only CAR's and CDR's that evaluate exactly as the following abbreviations:

12. CAAR
13. CADR
14. CDAR
15. CADAR

Check your answers by using CAAR, CADR, etc., directly as in problems 21-31 of Chapter 6.

Evaluate the following:

```
16. (LAMBDA (U V)((LAMBDA (X Y)(CONS (CAR X) (CDR Y))) U V)) ((A) (B C))
17. (LAMBDA (U V)((LAMBDA (X Y)(CONS (CAR X) Y)) U (CDR V))) ((A) (B C))
18. (LAMBDA (W X)(CAR ((LAMBDA (Y Z)(CONS Y Z)) W X))) ((A) (B C))
19. (LAMBDA (W X)(CDR ((LAMBDA (Y Z)(CONS Y Z)) (CAR W)(CDR X)))) ((A) (B C))
20. (LAMBDA (J)(CONS
                ((LAMBDA (X Y)(CONS Y (CONS X NIL))) (CAR J) (CADR J))
                ((LAMBDA (U V)(CONS (CONS V (CONS U NIL)) NIL)) (CADDR J)(CADDDR J))
            )) ((A B C D))
```

CHAPTER 9. QUOTE, EVALQUOTE AND LIST

In most of the simple programs we have seen so far, data for the expression to be evaluated was supplied in the argument list and paired to the variables specified in the variable list within the LAMBDA expression. For example problem 1, of Chapter 7

```
(LAMBDA (X) X) (ATOM)
1       2 2 1 1   1
```

gave the atom ATOM as data to be paired with variable X. In a sense one can think of the LAMBDA expression as the program and the argument list as the program's data. For LISP to be a more powerful programming tool we must allow data to exist within the program. That is, we must permit arguments within the LAMBDA expression. The expression QUOTE allows us to do this and thus serves to isolate a program from its data.

Definition: QUOTE takes one argument, an S-expression. The value of the expression QUOTE is its argument, taken literally.

Examples:

```
(LAMBDA NIL (QUOTE ALPHA)) ( )
1           2            21 1 1
```

evaluates to ALPHA.

```
(LAMBDA (X) (CONS (QUOTE ALPHA) X)) ( BETA )
1       2 2 2     3           3  21 1      1
```

evaluates to (ALPHA . BETA)

9.1 MEANING OF QUOTE

What does QUOTE mean? The expression QUOTE tells LISP that what follows is to be treated as itself, not as the name for something else. This meaning is like the meaning in ordinary English when we use quotation marks and say:

"Paris" has five letters.

and mean:

The particular word has five letters.

We do not say:

Paris has five letters.

because Paris is a city and it makes no sense to say that a city has five letters; what a city has is people, streets, buildings, etc.

In English one of the standard uses of quotation marks is to produce a name for an expression, instead of designating what the expression usually refers to. This is the use of QUOTE in LISP.

9.2 SPECIAL CASES

The implementation of the LISP programming system gives rise to a number of anomalies regarding QUOTE. These anomalies exist as a programming convenience for almost all LISP implementations, Q-32 LISP included.

Rule: Numbers, NIL, and the letters T and F are never quoted since they are treated by the system as follows:

1.	a number, e.g., 99	is treated as	(QUOTE 99), which evaluates to 99
2.	NIL	is treated as	(QUOTE NIL), which evaluates to NIL
3.	T	is treated as	(QUOTE T), which evaluates to T
4.	F	is treated as	(QUOTE NIL), which evaluates to NIL

Note that F is treated by the system as (QUOTE NIL) such that F evaluates to NIL. If the user wishes the value F explicitly, he must write (QUOTE F), which evaluates to F.

We have seen examples of numbers and NIL in LAMBDA expressions in exercises of Chapter 8. The use of T and F will become clear when we examine predicates and conditional expressions in the following chapters.

9.3 EVALQUOTE

When inputting a pair of S-expressions, s_1 and s_2, at the top-level to the supervisor program, we are "conversing" with the principal LISP mechanism for evaluating expressions. In many LISP systems, including Q-32 LISP, the supervisor program is referred to as a function called Evalquote. Evalquote is an illusion in Q-32, as in reality there is no such function. (Q-32 does have a callable function EVALQT that behaves like Evalquote. This is covered in greater detail in Chapter 17.) The Q-32 LISP supervisor is a program written in LISP, and historically, LISP supervisors were programs called Evalquote. Thus, the name has persisted and we speak of "pairs of S-expressions for Evalquote."

As we saw earlier, the pair of S-expressions to Evalquote which we called s_1 and s_2 previously, consist of a functional expression, or function, s_1, and a list of arguments, s_2, for s_1, e.g.,

$$\underbrace{\text{PLUS}}_{s_1}\ \underbrace{(1\ 2\ 3)}_{s_2}$$

After accepting s_1 and s_2 Evalquote quotes each argument in the list, s_2, and then applies the function or functional expression, s_1, to the quoted arguments. It is important to understand that the arguments for top-level expressions are not evaluated since they are internally transmitted by Evalquote as quoted expressions.

For clarity, examine the following examples:

1. $\underbrace{\texttt{CONS}}_{s_1}\ \texttt{(}\underbrace{\texttt{A B)}}_{s_2}$ ---------------input to Evalquote

   (QUOTE A) (QUOTE B) ---------------transmitted arguments

   (CONS (QUOTE A) (QUOTE B)) ---------------internal form evaluated

   (A . B) ---------------value

2. $\underbrace{\texttt{(LAMBDA (X Y Z) (CONS X (CONS Y (CONS Z F))))}}_{s_1}\ \underbrace{\texttt{(33 T F)}}_{s_2}$ ----input to Evalquote

   (QUOTE 33)(QUOTE T)(QUOTE F) ---------------transmitted arguments

   X = (QUOTE 33),Y = (QUOTE T), Z = (QUOTE F) ----------variables bound

   (CONS (QUOTE 33) (CONS (QUOTE T) (CONS (QUOTE F)(QUOTE NIL)))) ---internal form evaluated

   (33 . (T . (F . NIL))) = (33 T F) ---------------value

Note that the explicit argument F of CONS in s_1 is always internally represented as (QUOTE NIL), whereas the input argument F in s_2 is transmitted as a literal quote, i.e., (QUOTE F).

9.4 LIST

Since the bulk of programming in LISP is in the form of lists, we need list-processing functions that are convenient to use. One of the most useful is the function LIST.

LIST is a function of an indefinite number of arguments that forms a list of these arguments. LIST creates a list. It is a shorthand notation. The following identities, written as lower-level forms (required for composition of functions), illustrates the shorthand notation.

```
(LIST) = NIL
(LIST A) = (CONS A NIL) = (A)
(LIST A B) = (CONS A (CONS B NIL)) = (A B)
---
---
(LIST A B ... Z) = (CONS A (CONS B (CONS ... (CONS Z NIL) ... )) = (A B ... Z)
```

Students often confuse the effect of LIST and CONS. Here are some examples that demonstrate the difference.

```
CONS ( A B) = (A . B)             LIST (A B) = (A B)
CONS (A NIL) = (A)                LIST (A NIL) = (A NIL)
CONS (A (B)) = (A B)              LIST (A (B)) = (A (B))
CONS ((A) B) = ((A) . B)          LIST ((A) B) = ((A) B)
CONS ((A) (B)) = ((A) B)          LIST ((A) (B)) = ((A) (B))
```

Rule: LIST effectively wraps parentheses around its arguments.

9.5 EXERCISES

Evaluate:

```
 1.  (LAMBDA NIL (QUOTE X)) ( )
 2.  (LAMBDA (J) (QUOTE J)) (ALPHA)
 3.  (LAMBDA (J) (QUOTE (AN S EXPRESSION))) (ALPHA)
 4.  (LAMBDA (J) (CAR (QUOTE (A B C)))) (ALPHA)
 5.  (LAMBDA (J)(CDR (QUOTE (J J)))) (NOTJ)
 6.  (LAMBDA (A B) (CONS A B))  (QUOTE EXPR)
 7.  (LAMBDA (A B) (CAR (CONS (QUOTE A) B)))  (ALPHA BETA)
 8.  (LAMBDA NIL (QUOTE
       (NOW IS THE TIME FOR ALL GOOD MEN TO COME TO
        THE AID OF THE PARTY))) ( )
 9.  (LAMBDA NIL (CONS (QUOTE A) (QUOTE B))) ( )
10.  (LAMBDA NIL (QUOTE
       (LAMBDA (X) X))) ( )
11.  (LAMBDA (A B C)(LIST A B C)) (ONE TWO THREE)
12.  (LAMBDA (A B C)(CONS A (CONS B (CONS C NIL)))) (ONE TWO THREE)
13.  (LAMBDA (A B C)(LIST F A F B F C)) (F F F)
14.  (LAMBDA (A B C)(LIST (QUOTE F) A (QUOTE F) B (QUOTE F) C)) (F F F)
15.  (LAMBDA (A B C D)(LIST (LIST F (QUOTE F) A)
                            (LIST T (QUOTE T) B)
                            (LIST NIL (QUOTE NIL) C)
                            (LIST 123 (QUOTE 123) D))) (F T NIL 123)
```

CHAPTER 10. DEFINE

Evaluating LAMBDA expressions at the top level is a one-shot proposition. If we wish to evaluate the same expression for different arguments, we must type the entire doublet again. After evaluation, the state of the LISP system is as it was prior to execution. This is desirable for many situations including debugging, code execution, and program formulation. However, for the majority of cases, we would like to save the expression as part of the LISP system, give it a function name, and use it repeatedly to build larger programs. We can do this by defining new functions in LISP with the pseudo-function DEFINE.

Pseudo-functions are expressions that are used like functions, but that do not behave like LISP functions in manipulating S-expressions. They are expressions that have side effects, invisible to LISP, in which we are interested. Input/output functions in LISP are good examples of other pseudo-functions.

10.1 DEFINE FORMAT

DEFINE is a pseudo-function that takes one argument, a list of functions to be defined. Like CAR, CDR, and all other single argument functions evaluated at the top level, the general format is:

$$\text{DEFINE } (_1\ e\ )_1$$

where e is the argument, a list of functions to be defined.

The format of e is:

$$(_2 f_1\ f_2\ f_3\ \dots\ f_n)_2$$

where f_1, f_2, ..., f_n are the definitions for the functions we wish to define. The formats are all the same, namely a LAMBDA expression, prefixed with a name for the expression. This name, a non-numeric atomic symbol, will become the function name. Thus

$$(_3\text{name } (_4\text{LAMBDA } (_5\text{variables})_5 \text{ form})_4)_3$$

is the general format for any of the f_i function definitions, and

```
  DEFINE ((    (name1 (LAMBDA (variables) form1 ))
         12    3      4       5         5       43

               (name2 (LAMBDA (variables) form2 ))
               3      4       5         5       43

               ---
               ---
               (namen (LAMBDA (variables) formn ))
               3      4       5         5       43

))
21
```

is the general format for the complete DEFINE expression.

Note the parentheses, their depth and meaning. The pair #1 bound the second S-expression, s_2, for Evalquote; pair #2 bound the single argument of DEFINE as a list; pairs #3 bound each of the n functional expressions to be defined.

Example:

```
DEFINE ((
       12

                (THIRD (LAMBDA (X) (CAR (CDR (CDR X)))))
                3      4       5 5 5    6    7     76543

                (IN3 (LAMBDA (X) (CAR (CAR (CAR X)))))
                3    4       5 5 5    6    7     76543

                (SECONDOF1ST (LAMBDA (X) (CAR (CDR (CAR X)))))
                3            4       5 5 5    6    7     76543

      ))
      21
```

If we wish to define only one function, the format is still the same with the argument list to DEFINE containing one functional expression.

10.2 EVALUATING DEFINE

The value of the pseudo-function DEFINE is a list of the names of the functions defined. For the example above, LISP would return

```
(THIRD IN3 SECONDOF1ST)
```

What have we really done by evaluating DEFINE? For Q-32 LISP, a compiler-based LISP system, we have compiled machine code for each of the functional expressions in the argument list. This machine code becomes a permanent part of the LISP system, which can be referenced by the name used in the functional expression, and can be used to evaluate data like all other system functions.

10.3 REDEFINING

If, after defining a function, you find the definition to be in error or you wish to change the function's definition (i.e., change the function's LAMBDA expression) for other reasons, you need only use DEFINE again with the old function name and a new LAMBDA expression. The new LAMBDA expression will be compiled and referenced under the old name. The old compiled code is lost and cannot be referenced again. Furthermore, the space occupied by the old code cannot be reclaimed and is lost to the system. Repeated redefinitions build up such "garbage" and should be avoided.

For the purposes of this primer, however, you should feel free to redefine programs at will, as you cannot, at one sitting, exhaust all the binary program space. But do not define programs whose names are system functions (e.g., CAR, CDR, LAMBDA, etc.) as you will redefine functions possibly used internally by the system.

10.4 EXERCISES

Define the following new functions and test them on list (A B C D E)

1. (FIRST (LAMBDA (X) (CAR X)))
2. (SECOND (LAMBDA (Y) (CADR Y)))
3. (THIRD (LAMBDA (Z) (CAR (CDDR Z))))
4. CADDDDR
5. Define a function, called REVDOT, that reverses the CAR and CDR of any dotted pair. Try it on the following arguments:

```
(A . B)
((A) . (B))
(((FIRST)) . (LAST))
```

CHAPTER 11. PREDICATE FUNCTIONS

To do interesting things in a programming language we must have facilities for testing data. In LISP we use predicate functions for this task. A predicate function is a function whose value is either true or false. In LISP, the values true and false are represented by the atomic symbols T and NIL, respectively.

Definition: A LISP predicate is a function whose value is either T (for true) or NIL (for false).

11.1 THE PREDICATE, ATOM

Definition: The predicate, ATOM, takes one argument. The value of ATOM is T if its argument is an atomic symbol. The value of ATOM is NIL if its argument is a non-atomic S-expression.

Examples:

```
ATOM (A) = T
ATOM (EXTRALONGSTRINGOFLETTERS) = T
ATOM ((A . B)) = NIL
ATOM ((A)) = NIL
(LAMBDA (J) (ATOM (CAR J))) ((A . B)) = T
```

11.2 THE PREDICATE, EQ

Definition: The predicate, EQ, takes two arguments. The value of EQ is T if its two arguments are the same non-numeric atomic symbol. The value of EQ is NIL if its two arguments are different non-numeric symbols. The value of EQ is undefined if either of its arguments is a non-atomic S-expression, or if either argument is a number.

Note: In pure LISP, the state "undefined" has meaning. In the programming system it means we cannot guarantee the value of the function, and quite likely it will induce a program error. For predicates, we often equate "undefined" with NIL.

Examples:

```
EQ (A B) = NIL
EQ (T T) = T
EQ ((T) ()) = undefined (NIL on Q-32 LISP)
EQ (12 12) = undefined (NIL on Q-32 LISP)
EQ (() NIL) = T
EQ (F NIL) = NIL
(LAMBDA (J) (EQ T (CAR J))) ((T . B)) = T
(LAMBDA (J) (EQ T (ATOM J))) (ANYATOM) = T
```

```
(LAMBDA (J) (EQ F J)) (NIL) = T          ) Remember the explicit F
(LAMBDA (J) (EQ F (CDR J))) ((A)) = T    } is treated by Q-32 LISP
                                         ) as (QUOTE NIL)

(LAMBDA (J) (EQ F J)) (F) = NIL    The explicit F is treated as
                                   (QUOTE NIL), whereas the
                                   argument F is transmitted
                                   as (QUOTE F) by Evalquote.
```

11.3 THE PREDICATES EQUAL AND EQUALN

Definition: The predicate EQUAL takes two arguments. The value of EQUAL is T if its two arguments are identical S-expressions. The value of EQUAL is NIL if its two arguments are not identical S-expressions.

Examples:

```
EQUAL (A B) = NIL
EQUAL (A A) = T
EQUAL ((A) (A)) = T
EQUAL ((A . B) (A . B)) = T
EQUAL ((A) A) = NIL
EQUAL (15 15) = T
EQUAL (17Q 15) = T
EQUAL (1.000000001 1) = T
EQUAL (1.000000002 1) = NIL
```

Note: For Q-32 LISP, EQUAL will accept and convert numbers of differing types before it performs the test for equality. For the case of floating-point numbers, they must agree within a specified accuracy given by the following:

EQUAL (N1 N2) = T

if N1 and N2 are floating-point numbers such that

$$\left| \frac{N1 - N2}{N1 + N2} \right| \leq 0.7 \times 10^{-9}$$

Definition: The predicate EQUALN is identical to EQUAL for non-numeric arguments. For numeric arguments, EQUALN is T if and only if the two arguments are identical S-expressions and are of the same number type. Otherwise EQUALN is NIL. EQUALN is a predicate available only on Q-32 LISP.

Examples:

```
EQUALN (A A) = T
EQUALN ((A) A) = NIL
EQUALN (1 1) = T
EQUALN (1 1.0) = NIL
EQUALN (1 1Q) = NIL
EQUALN (1.0 1Q) = NIL
EQUALN ((1.0) (1.0)) = T
```

11.4 ARITHMETIC PREDICATES

All of the following predicates end with the letter P, for predicate, as a mnemonic aid.

NUMBERP (N)	= T if N is a number of any type. = NIL if N evaluates to a non-atomic S-expression. = NIL if N evaluates to a non-numeric atomic symbol.
FIXP (N)	= T if N is an integer or octal number. = NIL if N is a floating-point number. = Undefined if N evaluates to a non-numeric S-expression.
FLOATP (N)	= T if N is a floating-point number. = NIL if N is an integer or octal number. = Undefined if N evaluates to a non-numeric S-expression.
ZEROP (N)	= T if N is a positive or negative zero of any numeric type. (1's complement arithmetic of the Q-32 can produce a binary word of all ones; i.e., a negative zero.) = NIL if N is a non-zero number. = Undefined if N evaluates to a non-numeric S-expression.
MINUSP (N)	= T if N is a negative number of any numeric type. = NIL if N is a positive number. = Undefined if N evaluates to a non-numeric S-expression.
GREATERP (N1 N2)	= T if N1 is greater than N2, where N1 and N2 may be any numeric types. = NIL if N1 is less than or equal to N2. = Undefined if N1 or N2 evaluates to a non-numeric S-expression.

LESSP (N1 N2)	= T if N is less than N2, where N1 and N2 may be any numeric types.
	= NIL if N1 is greater than or equal to N2.
	= Undefined if N1 or N2 evaluates to a non-numeric S-expression.

Note: ZEROP, GREATERP, and LESSP all use the accuracy specifications for floating point numbers noted for EQUAL.

11.5 LIST PREDICATES

NULL (L)	= T if L is the empty list () or NIL.
	= NIL if L is not NIL or ().
MEMBER (L1 L2)	= T if S-expression L1 is a top-level element of list L2.
	= NIL if L1 is not an element of L2, or is an element of a lower-level sublist of L2; e.g.,

MEMBER (A ((A))) = NIL

11.6 LOGICAL CONNECTIVES

NOT (P)	= T if P evaluates to NIL. (Remember, an explicit F evaluates to NIL.)
	= NIL if P evaluates to any non-NIL atomic symbol.
	= NIL if P evaluates to any non-atomic S-expression.

AND (x_1 x_2 ... x_n)

AND takes an indefinite number of arguments, not a list of arguments. 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 T if all arguments are true. The value of AND is NIL if any argument is false. In accordance with this definition:

AND ( ) = T.

OR (x_1 x_2 ... x_n)

OR takes an indefinite number of arguments, not a list of arguments. 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 T if any argument is true. The value of OR is NIL if all arguments are false. In accordance with this definition:

OR ( ) = NIL.

11.7 EXERCISES

Evaluate:

```
 1.  (LAMBDA (J) (CONS (EQ J J) (QUOTE (F T F)))) (X)
 2.  ATOM (NIL)
 3.  NULL (NIL)
 4.  NULL ((NIL))
 5.  NULL (( ))
 6.  EQUAL (O NIL)
 7.  NUMBERP (1965)
 8.  NUMBERP ((1965))
 9.  (LAMBDA (A B C) (OR (ZEROP A)
                         (FIXP B)
                         (FLOATP C))) (1 2 3)
10.  (LAMBDA (J) (NOT (AND (ATOM J)
                           (NUMBERP J)
                           (FLOATP J)
                           (MINUSP J)
                           (NOT (ZEROP J))))) (-1.0)
11.  GREATERP (1964 1965)
12.  GREATERP (1965 1964)
13.  LESSP (10Q 10)
14.  MEMBER (HEAR (NOW HEAR THIS))
15.  MEMBER (HEAR (NOW (HEAR THIS)))
```

16. ZEROP is true for both positive and negative zero. Define NEGZEROP which is true only for negative zero. Test it with these cases:

```
NEGZEROP (-0) = T
NEGZEROP (777777777777777Q) = T
NEGZEROP (0) = NIL
NEGZEROP (7Q15) = NIL
```

17. The propositional connective "equivalent" has the following truth table:

X	Y	X EQUIV Y
true	true	true
true	false	false
false	true	false
false	false	true

Define the LISP function EQUIV and test it on these cases:

```
EQUIV (T T) = T
EQUIV (T NIL) = NIL
EQUIV (NIL T) = NIL
EQUIV (NIL NIL) = T
```

18. The propositional connective IMPLIES has the following truth table:

X	Y	X IMPLIES Y
true	true	true
true	false	false
false	true	true
false	false	true

Define the LISP function IMPLIES and test it on these cases:

```
IMPLIES (T T) = T
IMPLIES (T NIL) = NIL
IMPLIES (NIL T) = T
IMPLIES (NIL NIL) = T
```

19. Define the predicate INSEQ that is T if a list of 5 elements are all numbers in ascending or descending order and NIL otherwise. Test it with these cases:

```
INSEQ ((1 2 3 4 5)) = T
INSEQ ((5 4 3 2 1)) = T
INSEQ ((1Q 2.0 99 1000Q 1000.0)) = T
INSEQ ((10Q 10 10.0 11.0 12Q)) = NIL
INSEQ ((10 9 8 7Q 7)) = NIL
```

20. Define the predicate EQN that is T if its two arguments are the identical atom and NIL otherwise. Test it with these cases:

```
EQN (A A) = T
EQN (1 1.0) = NIL
EQN (77Q 77Q) = T
EQN ((A) A) = NIL
```

CHAPTER 12. CONDITIONAL EXPRESSIONS

The class of functions that can be formed with what we know so far is quite limited and not very interesting. Predicates give us a mechanism for testing data. Now we need functions that branch conditionally on the value of these predicates and thereby allow a much larger class of functions to be defined.

12.1 FORMAT OF CONDITIONAL EXPRESSIONS

A conditional expression in LISP has the following form:

$$\underset{1}{(}\text{COND}\ \underset{2}{(}p_1\ e_1\underset{2}{)}\ \underset{2}{(}p_2\ e_2\underset{2}{)}\ \ldots\ \underset{2}{(}p_n\ e_n\underset{2}{)}\underset{1}{)}$$

where p_1, p_2,,, p_n are predicates or expressions that evaluate to true of false, and e_1, e_2,,, e_n are any S-expressions.

COND takes an indefinite number of arguments, called clauses, each of which is a list containing a p_i and its corresponding e_i.

12.2 MEANING OF CONDITIONAL EXPRESSIONS

LISP evaluates a conditional expression from left to right as follows:

If p_1 is true, then the value of COND is the value of e_1.

If p_1 is false, then

If p_2 is true, then the value of COND is the value of e_2.

If p_2 is false, then

If p_3 is true, etc.

The entire expression is searched by evaluating p_i of each clause, until the first p_i that is true is found, and then the corresponding e_i of that clause is evaluated. Note that e_i is never evaluated if the corresponding p_i of that clause is false.

If a true clause cannot be found (i.e., all p_i are false), then the value of the entire expression is undefined. To protect against this occurrence, LISP programmers usually set the last predicate, p_n of the last clause, equal to T and set the last expression, e_n of that clause, equal to some terminating expression. Since T is treated by the system as (QUOTE T), it always evaluates true and COND can never be undefined. If nothing else proves true then the value of e_n will be the value of the entire conditional expression.

Each p_i or e_i is itself an S-expression; possibly a function, a composition of functions, or another conditional expression. It is perfectly proper for p_i or e_i to be NIL ,T ,F , or any atom that evaluates true or false.

12.3 AN EXAMPLE

The propositional connective IMPLIES has the following Truth Table:

X	Y	X → Y
true	true	true
true	false	false
false	true	true
false	false	true

Using COND, we can define IMPLIES in many ways. Here are four alternate definitions.

```
DEFINE (( (IMPLIES (LAMBDA (X Y)(COND (X Y)(T T)))) ))
DEFINE (( (IMPLIES (LAMBDA (X Y)(COND ((EQ X Y) T)(T Y)))) ))
DEFINE (( (IMPLIES (LAMBDA (X Y)(COND (Y T)(T (NOT X))))) ))
DEFINE (( (IMPLIES (LAMBDA (X Y)(COND (X (COND (Y T)(T F)))
                                      (T T)))) ))
```

The last definition demonstrates the nesting of conditionals; however, the first definition is more elegant, since it takes full advantage of the true-false nature of the data by letting the variables act as predicates.

12.4 EXERCISES

Evaluate:

```
1.  (LAMBDA NIL (COND (F (QUOTE FALSE))
                      (T (QUOTE TRUE)))) ( )
2.  (LAMBDA NIL (COND (NIL (QUOTE FALSE))
                      (T (QUOTE TRUE)))) ( )
3.  (LAMBDA NIL (COND (NIL F) (T T))) ( )
4.  (LAMBDA (A B C) (COND (T A) (T B) (T C))) (1 2 3)
5.  (LAMBDA (A B C) (COND (NIL A) (NIL B) (NIL C))) (1 2 3)
6.  Define this expression as ABVALU:
    (ABVALU (LAMBDA (N) (COND ((GREATERP 0 N) (CONS (QUOTE MINUS) N))
                              (T N))))
    Try these:
    ABVALU (144)
    ABVALU (-14.4)
    ABVALU (-0.0)
    ABVALU (0Q)
```

7. Define a function, called SMALLER, that takes two numeric arguments and returns the smaller of the two.

   Try these:

```
SMALLER (15 7)
SMALLER (-15 7.02)
SMALLER (-15.0E-1 -7E1)
SMALLER (+0 -0.0)
SMALLER (10Q 8)
```

8. Using conditionals, define EQUIV (X Y) which is true only if
   both X is true and Y is true, or
   both X is false and Y is false;
   and is false otherwise.

   Try these:

```
EQUIV (T T) = T
EQUIV (T NIL) = NIL
EQUIV (NIL T) = NIL
EQUIV (NIL NIL) = T
(LAMBDA (X Y) (COND ((EQUIV X Y) (CONS X Y))
                    (T (QUOTE FALSE)))) (NIL NIL) = (NIL . NIL) = (NIL)
```

9. The LISP predicate OR (X Y) is the inclusive OR which is true if X or Y or both are true. Using conditionals, define a LISP predicate EXOR (X Y) of <u>two arguments</u> that is the exclusive OR and is true if and only $\overline{\text{if X}}$ or Y are true, but not both. Try it out.

10. MEMBER is a predicate in Q-32 LISP. If it were not, it could be defined as:

```
(MEMBER (LAMBDA (A X) (COND ((NULL X) F)
                            ((EQUAL A (CAR X)) T)
                            (T (MEMBER A (CDR X))))))
```

   where A is an expression to be looked for on the top level of list X.

   Study this example and the use of COND. Don't be alarmed by the use of the function MEMBER within its own definition. Treat this <u>recursive</u> function like any other function you've seen. We shall examine recursion much more fully in subsequent chapters.

   Note that we first examine the CAR of list X to see if it is equivalent to A. If yes, we return the value "true" as the value for MEMBER. If no, we apply the function MEMBER to the CDR of list X. Thus, we are applying MEMBER to a shorter and shorter list each time we recurse until A is found. If A is never found, list X is eventually reduced to NIL by the repeated CDR. This terminal condition is trapped by the NULL and the value of F, i.e., NIL, for "false" is returned as the value of MEMBER, since A is not a member of list X.

CHAPTER 13. ARITHMETIC FUNCTIONS

Chapter 5 discusses Q-32 LISP representation of numbers and it might pay to review that chapter. Let us review three important points:

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. (Never as variables in a LAMBDA expression.)

13.1 GENERAL COMMENTS

All the arithmetic functions must be given numbers as arguments, or S-expressions that evaluate to numbers; otherwise an error condition will result. For example:

(xxxx NOT A NUMBER)

where xxxx is a non-numeric argument to an arithmetic function, is the most probable error message given by the Q-32 LISP system for this error condition.

The numerical arguments to arithmetic functions may be any type of number, i.e., integer, octal, or floating point. An arithmetic function may be given some fixed-point (i.e., integer or octal) and some floating-point arguments at the same time. If all of the arguments for a function are fixed-point numbers, then the value will be a fixed-point number. (Integer and octal arguments always yield an integer value.) If at least one argument is a floating-point number, then the value of the function will be a floating-point number.

13.2 Q-32 LISP ARITHMETIC FUNCTIONS

$$\text{PLUS } (x_1\ x_2\ \ldots\ x_n) = x_1 + x_2 + \ldots + x_n$$

PLUS is a function of any number of arguments whose value is the algebraic sum of the arguments.

$$\text{DIFFERENCE } (x\ y) = x - y$$

DIFFERENCE has for its value the algebraic difference of its arguments.

$$\text{MINUS } (x) = -x$$

MINUS has for its value the one's complement of its argument.

TIMES $(x_1\ x_2\ \ldots\ x_n) = (x_1)(x_2)(\ldots)(x_n)$

TIMES is a function of any number of arguments whose value is the product (with correct sign) of its arguments.

ADD1 $(x) = x + 1$

ADD1 adds one to its argument and returns the sum as its value. The value is fixed point or floating point according to the argument type.

SUB1 $(x) = x - 1$

SUB1 subtracts one from its argument and returns the difference as its value. The value is fixed point or floating point according to the argument type.

MAX $(x_1\ x_2\ \ldots\ x_n)$

MAX chooses the largest of its arguments for its value. Note that

MAX (3 1Q 2.0) = 3.0

yields a floating-point number since at least one argument was floating-point.

MIN $(x_1\ x_2\ \ldots\ x_n)$

MIN chooses the smallest of its arguments for its value.

QUOTIENT $(x\ y) = x\ /\ y$

QUOTIENT computes the quotient of its arguments. For fixed-point arguments, the value is the number theoretic quotient, e.g., QUOTIENT (5 2) = 2. A divide-check or floating-point trap will result in a LISP error.

REMAINDER (x y)

REMAINDER computes the number theoretic remainder for fixed-point arguments, e.g., REMAINDER (5 2) = 1 and the floating-point residue for floating-point arguments.

DIVIDE (x y)

DIVIDE returns as its value a list of the QUOTIENT and the REMAINDER of its arguments. It could be defined by:

```
(DIVIDE (LAMBDA (X Y) (LIST (QUOTIENT X Y)(REMAINDER X Y))))
```

EXPT $(x\ y) = x^y$

EXPT. If both x and y are fixed-point numbers, this is computed by iterative multiplication. Otherwise, the yth power of x is computed by using logarithms. The first argument x cannot be negative if y is not an integer.

$$\text{SQRT}\ (x) = \sqrt{|x|}$$

SQRT is a LISP function unique to the Q-32. The value is the square root of the absolute value of the argument. The value is always given as a floating-point number.

$$\text{RECIP}\ (x) = 1\ /\ x$$

RECIP computes and returns as its value the reciprocal of its argument. The reciprocal of any fixed-point number is defined to be zero. (RECIP is not currently available.)

$$\text{ABSVAL}\ (x) = |\ x\ |$$

ABSVAL returns as its value the absolute value of its argument. If x is positive, it returns x. If x is negative, it returns the value of MINUS(x).

FLOAT (x)

FLOAT is a LISP function unique to the Q-32. The value is the floating-point equivalent of its argument. It could be defined by:

```
(FLOAT (LAMBDA (X) (ADD X 0.0)))
```

ENTIER (x)

ENTIER is a LISP function unique to the Q-32. The value of the function for positive numbers is the largest integer less than or equal to its argument. For negative numbers it is MINUS the ENTIER of the magnitude of the argument. For example:

```
ENTIER (93.75) = 93
ENTIER (-3.75) = -3
ENTIER (0.35) = 0
ENTIER (-0.35) = 0
```

Whereas FLOAT converts a fixed-point number to floating-point, ENTIER converts a floating-point number to fixed point.

13.3 LOGICAL ARITHMETIC FUNCTIONS

The following functions operate on 48-bit words. The only acceptable arguments are fixed-point numbers. These may be entered as octal or decimal integers, or they may be the result of a previous computation.

$$\text{LOGOR}\ (x_1\ x_2\ \ldots\ x_n)$$

LOGOR is a function of any number of arguments, whose value is the logical OR of all its arguments.

LOGXOR $(x_1\ x_2\ \ldots\ x_n)$

LOGXOR is a function of any number of arguments, whose value is the logical exclusive OR of all its arguments.

LOGAND $(x_1\ x_2\ \ldots\ x_n)$

LOGAND is a function of any number of arguments, whose value is the logical AND of all its arguments.

LEFTSHIFT $(x\ n) = (x)(2)^n$

LEFTSHIFT shifts its first argument left by the number of bits specified by its second argument. If the second argument is negative, the first argument will be shifted right.

13.4 AN EXAMPLE

The power series expression for SIN is given* by:

$$\text{SIN } x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \ldots$$

where x is in radians.

If $c_1 = 1$

$c_3 = \frac{-1}{3!} = -1.666666667\text{E-}1$

$c_5 = \frac{1}{5!} = 8.333333333\text{E-}3$

$c_7 = \frac{-1}{7!} = -1.984126984\text{E-}4$

$c_9 = \frac{1}{9!} = 2.755731922\text{E-}6$

We can approximate the power series as:

$$\text{SIN } x = c_1x + c_3x^3 + c_5x^5 + c_7x^7 + c_9x^9$$

The LISP function SIN (x) where x is in radians, can now be defined in terms of this power series approximation.

```
DEFINE ((
(SIN (LAMBDA (X)(PLUS X (TIMES -1.666666667E-1 X X X)
                        (TIMES 8.333333333E-3 X X X X X)
                        (TIMES -1.984126984E-4 X X X X X X X)
                        (TIMES 2.755731922E-6 X X X X X X X X X)))) ))
```

* Handbook of Mathematical Tables and Formulas, Burington, Handbook Publishers, Inc., Sandusky, Ohio, 1953.

If we factor out x^2 and write the power series in the form

$$\text{SIN } x = x(c_1 + x^2(c_3 + x^2(c_5 + x^2(c_7 + c_9x^2))))$$

a more computationally efficient LISP program for SIN can be defined by using a nested LAMBDA expression, as we need compute x^2 only once.

```
DEFINE ((
(SIN (LAMBDA (X) ((LAMBDA (XSQ)(TIMES X (PLUS 1 (TIMES XSQ (PLUS -1.6666667E-1
  (TIMES XSQ (PLUS 8.333333333E-3 (TIMES XSQ (PLUS -1.984126984E-4
  (TIMES XSQ 2.755731922E-6))))))))))(TIMES X X)))) ))
```

13.5 EXERCISES

Evaluate:

```
 1.  PLUS (1 2 3 4 5 6 7 8 9 10)
 2.  DIFFERENCE (99 3.14159)
 3.  TIMES (2 2 2 2 2 2 2 2 2 2)
 4.  ADD1 (77777Q)
 5.  SUB1 (1.0)
 6.  MINUS (-0)
 7.  MAX (10 12Q 10.000000001)
 8.  MIN (10 12Q 9.999999999)
 9.  QUOTIENT (55 3)
10.  QUOTIENT (55.0 3Q)
11.  REMAINDER (55 3)
12.  REMAINDER (55 3.0)
13.  DIVIDE (55 3)
14.  DIVIDE (55 3.0)
15.  DIVIDE (55 3Q)
16.  ENTIER (123.4)
17.  ENTIER (-123.4)
18.  ENTIER (0.7)
19.  ENTIER (-0.7)
20.  SQRT (25)
21.  RECIP (3.0)
22.  RECIP (3)
23.  FLOAT (123456789)
24.  ABSVAL (-3.14159)
25.  LOGOR (77777Q 12345Q)
26.  LOGOR (70707Q1 12345Q)
27.  LOGOR (77777Q 12345Q)
```

28. LOGXOR (70707Q1 12345Q)
29. LOGAND (77777Q 12345Q)
30. LOGAND (70707Q1 12345Q)
31. LEFTSHIFT (7Q1 1)
32. LEFTSHIFT (7Q1 -1)

Define the following functions and try them out with your own values of variables.

33. TRIPLE (X) = X + X + X
34. CUBE (X) = X^3
35. SIMPLEINTEREST (PRINCIPAL RATE YEARS) = P(1 + YR)
36. ANNUALCOMPOUND (P R Y) = $P(1 + R)^Y$
37. TIMECOMPOUND (P R Y T) = $P(1 + R/T)^{TY}$
38. The value of a two-by-two determinant is defined by:

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = (a_{11}\, a_{22} - a_{12}\, a_{21})$$

Define the LISP function

TWOBY (a_{11} a_{12} a_{21} a_{22})

39. The value of a three-by-three determinant is defined by:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

Define the LISP function

THREEBY (a_{11} a_{12} ... a_{32} a_{33})

40. Given three simultaneous equations

$$a_{11}\, u_1 + a_{12}\, u_2 + a_{13}\, u_3 = c_1$$

$$a_{21}\, u_1 + a_{22}\, u_2 + a_{23}\, u_3 = c_2$$

$$a_{31}\, u_1 + a_{32}\, u_2 + a_{33}\, u_3 = c_3$$

we can solve for any variable u_k by dividing two determinants. The denominator determinant, D, is as defined in problem 39. The numerator determinant is similar but with the c_k terms replacing the coefficients of the u_k variables. For example:

$$u_2 = \frac{\begin{vmatrix} a_{11} & c_1 & a_{13} \\ a_{21} & c_2 & a_{23} \\ a_{31} & c_3 & a_{33} \end{vmatrix}}{D}$$

Define the LISP function

SOLVE (a_{11} a_{12} ... a_{32} a_{33} c_1 c_2 c_3)

which computes the value of all variables u_k for three simultaneous equations in three variables. (HINT: Use your definition of THREEBY and QUOTIENT.)

Try these equation sets:

1. $2u_1 + u_2 - 2u_3 = -6$
   $u_1 + u_2 + u_3 = 2$
   $-u_1 - 2u_2 + 3u_3 = 12$

2. $2u_1 + u_2 - 2u_3 = 5$
   $2u_1 + u_2 + 3u_3 = 6$
   $-u_1 - 2u_2 + 3u_3 = 12$

3. $15u_1 + 15u_2 + 15u_3 = 15$
   $7u_1 + u_2 - 100u_3 = -100$
   $-50u_1 + u_2 + u_3 = -16$

4. $u_1 + 2u_2 - 2u_3 = -12$
   $u_1 + u_2 + u_3 = 6$
   $-2u_1 - u_2 + 3u_3 = 2$

5. $-2u_1 + 2u_2 + u_3 = -24$
   $u_1 + u_2 + u_3 = 29$
   $3u_1 - u_2 - 2u_3 = 9$

CHAPTER 14. RECURSIVE FUNCTIONS

The functions we have thus far defined have used LAMBDA expressions, composition of functions, and conditional expressions. A still wider class of functions can be defined using these methods and the method of recursion.

It takes time and practice to think recursively, particularly if you have programming experience with the linear flow of control common with algebraic languages. You cannot be taught to think recursively, but you can learn to think recursively. To help you learn, we give some helpful heuristics, and examples, and more examples.

The mechanics of defining recursive functions is like any other function composition. When we construct a form, such as

(CONS X Y)

we are making an explicit call upon the function CONS. CONS, in this case, is an already existing function. In a recursive function definition, for say function f, we likewise make explicit calls upon functions; however, one or more such calls are upon the function f itself. The only apparent difference between calls upon CONS and calls upon f, is that f is the function being defined itself. But LISP doesn't mind. In most algebraic languages, the programmer is cautioned not to write subroutines that call upon themselves, since that is recursion and most algebraic languages cannot handle recursion. In LISP we do it all the time. For example, it is syntactically proper to write

```
DEFINE ((
(EXAMPLE (LAMBDA (L)(COND ((NULL L) NIL)
                          (T (CONS (CAR L) (EXAMPLE (CDR L)))))) ))
```

We note that in this do-nothing function definition, i.e., EXAMPLE returns as its value the input list L, EXAMPLE makes an explicit call upon itself. EXAMPLE is thereby a recursive function.

Recursive definitions always define an idea in one or more special starting or finishing cases, and then define the idea in the general case in terms of a preceding or adjacent case. Let's see how this statement applies to a LISP problem.

14.1 AN EXAMPLE

Problem: given any list, such as

(A B C)

define the predicate

ATOMLIST (ℓ)

which is true if all elements of ℓ are atoms, and false otherwise. How shall we proceed? Essentially, we wish to perform the test

```
If ATOM A , then
     If ATOM B ; then
          If ATOM C , then true.
          Else false
     Else false
Else false
```

which, as a LISP function, would be defined by

```
DEFINE ((
(ATOMLIST (LAMBDA (A B C)
    (COND ((ATOM A)(COND ((ATOM B)(ATOM C))
                          (T F)))
         (T F)))) ))
```

But this is not a solution to our problem. We are not given A, B and C explicitly, but rather list ℓ, which can have any number of elements. We must do

```
                         (ATOM (CAR L))
```

to test an element of ℓ. Thus, we could write

```
DEFINE ((
(ATOMLIST (LAMBDA (L) (COND ((ATOM (CAR L))
    (COND ((ATOM (CADR L))(ATOM (CADDR L)))
         (T F))) (T F)))) ))
```

But this last definition solves the problem when we know list ℓ has exactly three elements. How about the general case where we do not know the length of list ℓ, or even when we do know, but where ℓ is very long? We don't want to write

```
                   (CADDDDDDDDDDDDDDDDDDDDR L)
```

even if we could, for a 20-element list.

The proper strategy is to test the first element of the list with the expression

```
                         (ATOM (CAR L))
```

If it is false, we exit NIL. If it is true, we need to test the second list element. If it proves true, then we test the third element, etc. But note, if after we test the first element, we remove the first element from the list, then the second element becomes the first element of the new list and we can apply the same test to the new list. The new list is

```
                            (CDR L)
```

and the test is applied recursively. Thus we can write:

```
DEFINE ((
(ATOMLIST (LAMBDA (L)
     (COND ((ATOM (CAR L)) (ATOMLIST (CDR L))) (T F)))) ))
```

which is recursive. What we have done is first examined the (CAR L). If it is an atom, we reduce the list ℓ by taking the (CDR L) to get a new list. Then test this new list with ATOMLIST. If we ever find a non-atom, the conditional will return NIL.

This last definition almost works, but not quite. It fails because we haven't set up a terminal condition. As it stands now, unless we exit NIL because some element of the list was non-atomic, we will recurse again and again, reducing ℓ each time until ℓ no longer has elements but is NIL. And then we would try recursing once more and try to take the CDR of NIL. There's the rub. (CDR NIL) is undefined. To exit properly, we must test for the terminal condition. In this case

(NULL L)

will suffice. Thus our final, correct recursive definition for ATOMLIST is:

```
DEFINE ((
(ATOMLIST (LAMBDA (L) (COND ((NULL L) T )
   ((ATOM (CAR L)) (ATOMLIST (CDR L)))
   (T F)))) ))
```

Note: if we ever encounter the null condition, ATOMLIST is true since all prior elements must have tested true. We perform the null test first to allow ℓ to be completely general including the empty list, NIL. Note then that

ATOMLIST () = T

14.2 SOME HELPFUL HEURISTICS

The following heuristics can be used to help in defining recursive functions.

1. Start with a trivial case, or a terminal case in which the rule for computation is known. Some typical trivial or terminal cases are:

for S-expression	; atoms
for Lists	; NIL
for Numbers	; 0,1

2. For the non-trivial, non-terminal case try to reduce what you are trying to compute to some function of a case "nearer" to the trivial case.

3. Combine the trivial or terminal case with the other, using the trivial or terminal case first in a conditional expression.

4. Always check your definition by trying several simple--but not all trivial--examples.

Let's try these heuristics on the recursive definition of FACTORIAL, where

$$n! = \begin{cases} \text{UNDEFINED} & \text{, for } n < 0 \\ 1 & \text{, for } n = 0 \\ (n)\ (n-1)! & \text{, for } n > 0 \end{cases}$$

1. The argument of FACTORIAL is a number. Therefore, the trivial case is for n = 0.

2. In the trivial case where n = 0, then

```
        FACTORIAL (N) = (COND ((ZEROP N) 1))
```

3. If n is not zero, then we can break n! into the product of two parts, n and (n-1)! since (n-1)! moves us nearer the trivial case (2). Thus,

```
    FACTORIAL (N) = (TIMES N (FACTORIAL (SUB1 N)))
```

4. Now combining the two cases (2) and (3) conditionally with the trivial case first, we get,

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

Let's trace through this example for n = 3.

```
Arguments of FACTORIAL = 3, descend (recursion)
   Arguments of FACTORIAL = 2, descend (recursion)
      Arguments of FACTORIAL = 1, descend (recursion)
         Arguments of FACTORIAL = 0, terminal condition
         Value of FACTORIAL = 1, ascend
      Value of FACTORIAL = 1, ascend
   Value of FACTORIAL = 2, ascend
Value of FACTORIAL = 6, complete
```

What we have effectively done in this example is to create

```
        FACTORIAL (3) = (TIMES 3 (TIMES 2 (TIMES 1 1)))
```

In general, we will descend as deep as is necessary to reach the terminal case and the effective computation would be

```
FACTORIAL (n) = (TIMES n (TIMES n-1 ... (TIMES 2 (TIMES 1 1)) ... )))
```

Examples: The following functional definitions are pedagogical devices. Although these functions are available in Q-32 LISP, these definitions may not exactly replicate those in the system.

1. The function EQUAL (x y) as we have seen in Chapter 11 can be defined by:

```
DEFINE (( (EQUAL (LAMBDA (X Y)
             (COND ((ATOM X) (EQ X Y))
                   ((ATOM Y) NIL)
                   ((EQUAL (CAR X) (CAR Y)) (EQUAL (CDR X) (CDR Y)))
                   (T NIL)))) ))
```

2. The value of the function APPEND of two arguments, both lists, is a list formed by appending the second list to the first. For example:

```
APPEND ((A B) (D E F)) = (A B D E F)
```

```
DEFINE (( (APPEND (LAMBDA (X Y)
             (COND ((NULL X) Y)
                   (T (CONS (CAR X) (APPEND (CDR X) Y)))))) ))
```

3. The function LAST of one argument, a list, returns the last top level element of the list.

```
DEFINE (( (LAST (LAMBDA (L)
             (COND ((NULL L) NIL)
                   ((NULL (CDR L)) (CAR L))
                   (T (LAST (CDR L)))))) ))
```

14.3 LABEL NOTATION

Earlier we saw that we could compose and evaluate an expression as a temporary LAMBDA expression, or as a permanent function defined by a LAMBDA expression. Recursive expressions point up an inadequacy in LAMBDA notation that requires us to define as permanent, recursive expressions that which we wish to consider temporary expressions. This difficulty stems from the inability to use the expression within itself, since the LAMBDA expression is not named; and when a function is recursive, it must be given a name. To resolve this difficulty and thereby allow composition and evaluation of temporary recursive expressions, we use the LABEL feature of LISP.

In order to be able to write expressions that bear their own name, we write,

```
(LABEL name LAMBDA-expression)
```

where name is any non-numeric atomic symbol you choose as the name for the given LAMBDA expression.

Example:

```
          (LABEL DUMMY (LAMBDA (X)
          1            2       3 3
             (COND ((ATOM X) X)
             3     45      5 4
                   (T (DUMMY (CAR X)))))) (argument list)
                   4  5      6    654321 1              1
```

LABEL notation, as this is called, creates temporary expressions that, like the temporary LAMBDA expressions seen earlier, must be provided immediately with a list of arguments to be associated with the LAMBDA variables during evaluation. Also, like temporary LAMBDA expressions, the expression must be entered again each time it is applied to a different argument list. In fact, that is the meaning of "temporary expression" as used here. Of course, we can always use DEFINE to create permanent functions rather than repeatedly use LAMBDA or LABEL. In practice, temporary LAMBDA expressions are used frequently, but LABEL is seldom used, the preference being to attach the name by use of DEFINE.

14.4 EXERCISES

1. Define FACTORIAL as given in the previous examples. Try it for a few values of n ≤ 10.

2. To see the recursion dramatically, do the following:

   enter,

   ```
   TRACE ((FACTORIAL))
   ```

   After the system responds

   ```
   (FACTORIAL)
   ```

   enter,

   ```
   FACTORIAL (5)
   ```

   The system will print the argument and value of FACTORIAL each time it is entered so that you may see the recursion as the function first descends, and then ascends in its computation. When 5! has been computed, enter,

   ```
   UNTRACE ((FACTORIAL))
   ```

   to remove the tracing action from the function FACTORIAL.

3. Evaluate

   ```
   (LABEL NAME (LAMBDA (X) (COND ((ATOM X) X) (T (NAME (CDR X))))))
   ```

   for the following arguments; (Remember, LABEL takes a list of arguments):

```
A
(A . B)
((X . Y) . (X . Z))
(A B C)
(A (C . E))
```

4. Evaluate

```
(LABEL MATCH (LAMBDA (X Y)
   (COND ((NULL X) (QUOTE NO))
         ((NULL Y) (QUOTE NO))
         ((EQ (CAR X) (CAR Y)) (CAR X))
         (T (MATCH (CDR X) (CDR Y))))))
```

for the following arguments:

```
(X) (X)
(A B E) (J O E)
(K A Y) (E V E)
( E L L I N) (H E L E N)
```

5. Define

```
(TWIST (LAMBDA (S)
  (COND ((ATOM S) S)
        (T (CONS (TWIST (CDR S))
                 (TWIST (CAR S)))))))
```

Evaluate

```
TWIST (A)
TWIST ((A . B))
TWIST (((A . B) . C))
TWIST ((A B C))
TWIST (((A . B)))
```

6. Let us plan how to define, recursively, the function

SUM (x y) = x + y.

using only the functions ADD1 and SUB1, and the predicate ZEROP.

- The trivial case is if y = 0. Then the value of SUM would be the value of x.
- Now if we try to reduce the general case to this trivial one, we see that if y ≠ 0, then reduce y by 1, and increase x by 1 and SUM these two numbers. Then recurse. We can write

```
(SUM (LAMBDA (X Y)
  (COND ((ZEROP Y) X)
       (T (SUM (ADD1 X) (SUB1 Y))))))
```

1. Using this definition, show the arguments and values of SUM each time it is entered for

   SUM (1 2)

2. Check your answers to (1) by defining SUM on the computer and tracing its evaluation for x=1, y=2.

3. Don't forget to UNTRACE ((SUM)).

7. Define, recursively, using only the functions ADD1, SUB1, and ZEROP

   PROD (x y) = (x)(y)

   HINT: If y = 0, then the product is trivially zero. If not then the product is the SUM of x and the PROD of x and y-1.

8. We know that division is essentially repeated subtraction, and that the remainder in division is the residue when subtraction is no longer possible. Therefore,
   Define recursively

   REMXY (x y)

   which yields the remainder resulting from the division of x by y.

9. The greatest common divisor (G.C.D) of two whole numbers is the largest number that will exactly divide both of them. Euclid gave an algorithm, which can be stated in English as:
   The G.C.D of x and y is:

   . If x is greater than y, the G.C.D. of y and x. Else,

   . If the remainder of y divided by x is zero, x. Else,

   . The G.C.D. of x and the remainder of y divided by x.

   Use this algorithm to define

   GCD (x y)

   GCD (7 7) = 7
   GCD (19 7) = 1
   GCD (28 35) = 7

10. Define

```
AMONG (a ℓ)
```

which is a predicate that is true if and only if atom a is among the top level elements of list ℓ.

```
AMONG (X (A B X)) = T
AMONG (X (A B (X))) = NIL
```

11. Define

```
INSIDE (a e)
```

which is a predicate that is true if and only if atom a appears anywhere at any level in the S-expression e.

```
INSIDE (X (A B X)) = T
INSIDE (X (A (X) B)) = T
INSIDE (X (A . (B . X))) = T
```

12. Define

```
COPYN (x n)
```

which will put n copies of x on a list; e.g.,

```
COPYN ((A B) 3) = ((A B) (A B) (A B))
```

13. Define

```
LENGTHS (ℓ)
```

which counts the number of top level elements of a list; e.g.,

```
LENGTHS ((A B (C D) E)) = 4
```

14. Define

```
UNIONS (x y)
```

which returns a list that contains every element that is in one list or the other or both. The order in which the elements are presented is first, all the elements that are in the first list x and not in the second list y, and second, all elements in the second list y whether or not they are in list x.
HINT: Use the function MEMBER as given in problem 10, Chapter 12.

```
UNIONS ((U V W) (W X Y)) = (U V W X Y)
UNIONS ((A B C) (B C D)) = (A B C D)
```

15. Define

```
INTERSECT (x y)
```

which returns a list of elements common to both list x and list y.

```
INTERSECT ((A B C) (B C D)) = (B C)
INTERSECT ((A B C) (D E F)) = NIL
```

16. Define

REVERSAL (ℓ)

which reverses the order of top level elements of the list ℓ; e.g.,

REVERSAL (((A B) D (D E) G)) = (G (D E) D (A B))

HINT: Use APPEND as given in the earlier examples.

17. Define

PAIRS (ℓ1 ℓ2)

which produces a table (list of dotted pairs) of the elements of two lists of equal length; e.g.,

PAIRS ((ONE TWO THREE) (1 2 3)) = ((ONE . 1) (TWO . 2) (THREE . 3))

18. Define

DELETE (a ℓ)

which produces a new list in which all references to atom a have been deleted from the top level of list ℓ; e.g.,

DELETE (Y (X Y Z)) = (X Z)

19. Define the predicate

INSEQ (ℓ)

which is true if list ℓ contains a numerical sequence in proper ascending or descending order and false otherwise.
HINT: Use an auxiliary function INSEQA that tests ascending order only. Use INSEQA with REVERSE (a system function analogous to REVERSAL above) to test descending order.

INSEQ (1 2 3 4) = T
INSEQ (40 30 2 1) = T
INSEQ (1 23 24 27 26 30) = NIL
INSEQ (10.0 9 8 7.4 2.3) = T
INSEQ (A B C D E) = NIL

20. Define

REPLACE (a b x)

a function that replaces atom b by atom a for every occurrence of a in S-expression x.

REPLACE (A B (A B C D)) = (A A C D)
REPLACE (TWO TO (WE TO HAVE TO CATS)) = (WE TWO HAVE TWO CATS)

CHAPTER 15. THE PROGRAM FEATURE

The LISP 1.5 program feature, which is called by the LISP expression PROG, allows us to write an ALGOL-like program containing LISP statements to be executed. For JOVIAL programmers, its greatest attribute appears to be the ability to perform iteration by allowing looping and the use of temporary variables.

15.1 PROG FORMAT

The PROG format is embedded within a LAMBDA expression and so may be used in the same way LAMBDA expressions are used: for temporary evaluation of expressions; for permanent definition of expressions with DEFINE; in recursive expressions; and with LABEL.

Recall, a LAMBDA expressions has the following format:

```
(LAMBDA list-of-variables form)
```

The PROG format becomes the "form" in a LAMBDA expression. Like all forms, it is an S-expression; it has the structure:

```
(PROG list-of-variables sequence-of-statements)
```

The list of variables comprise the temporary variables required by the sequence of statements, which are themselves S-expressions.

Thus the complete LAMBDA expression with the PROG form has the structure:

```
(LAMBDA (lambda-variables) (PROG (program-variables) statements ))
```

15.2 PROGRAM VARIABLES

We usually call the variables associated with the LAMBDA expression "LAMBDA variables," and those associated with the PROG, "program or PROG variables." The list of PROG variables, just like the list of LAMBDA variables, must always be present in the structure of the expression. If we have none, then the list is entered as NIL or ( ).

Unlike LAMBDA variables, which have no value until an argument list is provided, PROG variables always have value NIL until they are changed or set by statements within the PROG structure. Two forms are used to set program variables, SET and SETQ.

SET acts like a function of two variables, and has the structure:

(SET v_1 v_2)

which can be read as "set value of v_1 equal to value of v_2." Both variables v_1 and v_2 can be and usually are S-expressions themselves. They are evaluated and the value of v_1 is set equal to the value of v_2. If we wish to set the name of something rather than set its value, we must always use QUOTE, e.g.,

```
(SET (QUOTE PI) 3.14159)
```

SET is not available on Q-32 LISP; however, this is not a serious loss as SETQ is available.

SETQ is like SET, but for convenience, SETQ always quotes its first argument. The Q in the name SETQ is to remind us of this fact, e.g.,

```
(SETQ PI 3.14159)
```

SETQ returns as its value the value of its second argument v_2.

15.3 FLOW OF CONTROL

Each program statement is an S-expression, and the sequence of statements is a sequence of S-expressions. The simplest S-expressions are atomic symbols, and these are used as location tags or markers for the statements that follows. For example,

```
     (SETQ PI 3.14159)
LOC1 (SETQ R N)
     (SETQ AREA (TIMES 2 PI R))
```

has atomic symbol LOC1 as a location tag for the statement

```
(SETQ R N).
```

Statements are normally executed in sequence. Executing a statement means evaluate the S-expression. Program statements are often executed for their effect rather than their value, as with SETQ above. GO is a perfect example of execution for effect rather than value. It is a form used to cause a transfer to a tagged statement. It acts like a function of one argument that is not evaluated; that argument being a location tag, e.g.,

```
(GO LOC1)
```

To exit from a PROG, we use RETURN. It acts like a function of one argument, and the argument is evaluated. The value is returned as the value of the PROG. No further statements are executed.

We can also exit from a PROG without the RETURN statement by just "running out" of statements. In that case, the value of the PROG is always NIL.

Statements can be constructed of any of the expressions available in LISP. They may be conditional or recursive expressions. They may even be LAMBDA or PROG expressions, thereby allowing nesting of PROG expressions.

15.4 SOME CAUTIONS

Conditional expressions as PROG statements have a useful peculiarity. If there is no true clause, instead of an error indication, which would otherwise occur, the program continues with the next statement. In other words, you "fall through" the conditional if there are no true conditions. This peculiarity is true only for conditional expressions that are on the top level of a PROG. The top level is the statement level, and, except for nested conditionals, conditionals are usually at the top level.

This attention to the top level is also required for the GO statement. It also must be used on the top level of a PROG or immediately inside a COND that is on the top level of a PROG.

If we nest a PROG within a PROG, within a PROG, etc., the GO, RETURN, SETQ, etc., will have a scope local to the most recent PROG. For example, GO cannot transfer to a statement tag within another higher or lower level PROG. Similarly, RETURN takes you "up" one level to the next higher PROG. In certain special cases, SETQ may be used on variables defined at a higher level PROG. These variables are then called "free" variables and require special attention. We will discuss variables and their "bindings" to values in the next chapter.

15.5 EXAMPLES

In the last chapter we saw the recursive definition of FACTORIAL. Let's contrast that expression with one using the PROG feature.

FACTORIAL--Recursive definition

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

FACTORIAL--PROG feature

```
DEFINE ((
(FACTORIAL (LAMBDA (N) (PROG (Y)
                             (SETQ Y 1)
                       TAG1  (COND ((ZEROP N) (RETURN Y)))
                             (SETQ Y (TIMES N Y))
                             (SETQ N (SUB1 N))
                             (GO TAG1))))
))
```

In this example, the recursive definition appears to be simpler than the one using the program feature. In other problems it may be otherwise. The choice of whether to use the PROG feature or to use "pure LISP" in programming in LISP depends in large measure on the problem. Style in programming is often, however, the stronger influence. See "Styles of Programming in LISP," by Fisher Black, in The Programming Language LISP: Its Operation and Application, March 1964, Information International Inc., Cambridge, Massachusetts. In that article, Black discusses this subject in depth.

15.6 PROG2

The function PROG2 has nothing to do with PROG. It is a function of two arguments that evaluates both its arguments in order, i.e., argument one first, argument two second. PROG2 has as its value, the value of its second argument. Thus

```
          PROG2 (1 2) = 2
(LAMBDA (X Y) (PROG2 (CONS X Y) Y)) (A B) = B
```

The utility of PROG2 can be seen in the following example:

Problem: Given a list of numbers, define the function SORT, which sorts these numbers into odd or even and returns a list of two sublists of the form;

((odd-count list-of-odd-numbers) (even-count list-of-even-numbers))

```
e.g., SORT ((1 2 3 4 5)) = ((3 (5 3 1)) (2 (4 2)))
      SORT (( 1 3 5 7 9)) = ((5 (9 7 5 3 1)) (0 NIL))
      SORT ((2 4 6 8 10)) = ((0 NIL) (5 (10 8 6 4 2)))
```

```
DEFINE ((
(SORT (LAMBDA (X) (PROG (ODD EVEN ODDCNT EVENCNT L)
      (SETQ L X) (SETQ ODDCNT 0) (SETQ EVENCNT 0)
LOOP  (COND ((NULL L) (RETURN (LIST (LIST ODDCNT ODD)
                              (LIST EVENCNT EVEN))))
           ((ZEROP (REMAINDER (CAR L) 2))
            (SETQ EVEN (PROG2 (SETQ EVENCNT (ADD1 EVENCNT))
                              (CONS (CAR L) EVEN))))
            (T (SETQ ODD (PROG2 (SETQ ODDCNT (ADD1 ODDCNT))
                                (CONS (CAR L) ODD))))
      (SETQ L (CDR L))
      (GO LOOP) ))) ))
```

15.7 EXERCISES

1. Using PROG, define the function

   NEGCNT (ℓ)

   which counts the number of negative numbers at the top level of list ℓ.

2. The discriminant b^2-4ac of a second degree equation of the form

   $$ax^2+bxy+cy^2+dx+ey+f=0$$

   can be used to determine the type of curve for the plot of the equation according to the following schedule:

   1. a parabola if discriminant = 0
   2. an ellipse if discriminant <0
   3. a hyperbola if discriminant >0

   Define

   CURVE (a b c)

   which evaluates to PARABOLA, ELLIPSE, OR HYPERBOLA as a function of the numerical values of arguments a, b, and c.

3. The recursive definition for LENGTHS is:

```
DEFINE ((
(LENGTHS (LAMBDA (M)
    (COND ((NULL M) 0)
         (T (ADD1 (LENGTHS (CDR M))))))) ))
```

   Define LENGTHS using PROG.

4. The recursive definition for LAST is:

```
DEFINE ((
(LAST (LAMBDA (L)
    (COND ((NULL L) NIL)
          ((NULL (CDR L)) (CAR L))
          (T (LAST (CDR L)))))) ))
```

   Define LAST using PROG.

5-7. Define the following functions using PROG. (See Chapter 14, problems 16, 17, 18.)

5. REVERSAL

6. PAIRS

7. DELETE

8. Each different arrangement of all or a part of a set of things is called a "permutation." The number of permutations of n different things taken r at a time is

P(n r) = n! / (n-r)!

Define, with and without PROG, (define FRACTORIAL first)

PERMUT (n r) = n! / (n-r)!

9. Each of the groups or relations which can be made by taking part or all of a set of things, without regard to the arrangement of the things in a group, is called a "combination." The number of combinations of n different things taken r at a time is

C(n r) = n! / r! (n-r)!

Define, with and without PROG,

COMBIN (n r) = n! / r! (n-r)!

10. A convenient way to obtain the combinations of n different things taken r at a time is to construct Pascal's triangle. The triangle looks like

```
                          r=0
                         ↙
n=0 →                  1   r=1
                          ↙
n=1 →               1   1   r=2
                           ↙
n=2 →             1   2   1    r=3
                              ↙
n=3 →          1   3   3   1    r=4
                               ↙
n=4 →       1   4   6   4   1    r=5
                                ↙
n=5 →    1   5  10  10   5   1
```

Given the pseudo function PRINT, which takes one S-expression as its argument and prints the value of that argument, e.g.,

.... (PRINT (LIST (QUOTE A) (QUOTE B) 3 (QUOTE C))) = (A B 3 C)

and ignoring the triangular format, use your definition for COMBIN to define

PASCAL (n)

which prints Pascal's triangle, e.g.,

PASCAL (5) =

```
(1)
(1 1)
(1 2 1)
(1 3 3 1)
(1 4 6 4 1)
(1 5 10 10 5 1)
NIL
```

CHAPTER 16. VARIABLES AND THEIR BINDING

So far, as a teaching convenience, I have been intentionally vague and loose in the mechanisms used by LISP in evaluating S-expressions. This chapter attempts to formalize what we have been doing regarding 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, an argument may be a variable when a functional expression inside another functional expression is evaluated, as we have seen with nested LAMBDA expressions. The intuitive approach is no longer adequate. In the examples so far, we have seen functions applied to specific arguments to get specific results. We have also provided for arbitrary arguments by means of bound variables with LAMBDA and PROG.

16.1 BOUND VARIABLES

RULE: An atom never stands for itself unless it is part of a quoted expression. (Note that T, F, NIL, and numbers are here considered quoted expressions.)

The implication of this rule is that all non-quoted atoms are bound variables. This is true. But what does this mean? In the simplest form, it means that all non-quoted atoms have values other than their names, and we say that a value is "bound to the atom." This value can be a number, or an S-expression. The binding is in actuality an association of data internal to the LISP system that is recognized and manipulated by the LISP system when an expression is evaluated. To understand bound variables adequately, we must examine more closely how data associations are constructed internal to Q-32 LISP.

Variables may be bound in one of two places, in a Special Cell associated with an atomic symbol, or on an internal binary last-in-first-out (LIFO) stack or table referred to as the pushdown list. We'll look at atomic bindings first.

16.2 ATOMIC BINDINGS

Atomic symbols are themselves lists internally. Unless you know what you're doing, it is fruitless to take the CAR, or CDR of an atom, but it can be done, and is frequently done by "knowledgeable" internal LISP functions. The atomic structure of an atom is the repository for many things. It contains the Print Name of the atom in BCD. It contains a Property List, which is a reserved list for storing useful collections of properties a programmer may wish to attach to an atom. (There are numerous functions available for manipulating property lists; however, they are not covered in this Primer and you should refer to the LISP 1.5 Programmer's Manual* for a complete discussion of property list functions.) Lastly, for Q-32 LISP, the atom structure includes a Special Cell that is reserved for binding values to the atom.

A value is an address where the number or the beginning of an S-expression is located in memory. We therefore think of the address as a pointer. To bind a value to an atom, we store the pointer in the atom's Special Cell by using one of two functions, CSET and CSETQ; functions that are analogous to the PROG function SETQ.

CSET acts like a function of two variables, and has the form

(CSET e_1 e_2)

e_1 and e_2 are S-expressions, and each is evaluated. e_1 must evaluate to an atomic symbol; e_2 may evaluate to a number, or S-expression. CSET then binds the value of e_2 to the atomic symbol that is evaluated for e_1, and returns the value of e_1. For example,

```
(LAMBDA ( ) (CSET (QUOTE PI) 3.14159)) ( ) = PI
```

binds the value 3.14159 to the atom PI. If we use CSET at the top level to Evalquote, e_1 and e_2 are of course quoted and to get the same binding we write

```
CSET (PI 3.14159)
```

When we use CSET within a LAMBDA or PROG expression, it is annoying to frequently have to quote the first argument. For convenience, we use CSETQ. CSETQ is like CSET, except CSETQ always quotes its first argument, and the Q in the name CSETQ reminds us of this fact. Effectively, CSETQ is defined as:

(CSETQ e_1 e_2) $\equiv$ (CSET (QUOTE e_1) e_2)

for the problem above, but using CSETQ, we get:

```
(LAMBDA ( ) (CSETQ PI 3.14159)) ( ) = PI
```

* LISP 1.5 Programmer's Manual, M.I.T. Technology Press, Cambridge, Massachusetts.

Unlike CSET, we never use CSETQ at the top level since Evalquote quotes the arguments e_1 and e_2. For e_1 CSETQ would receive (QUOTE e_1) which is not an atom. It may be clearer with examples. Let's bind list (A) to atom LIS1 at the top level using CSET and CSETQ:

CSET (LIS1 (A))	Input expressions to Evalquote	CSETQ (LIS1 (A))
(QUOTE LIS1)(QUOTE(A))	Transmitted arguments	(QUOTE LIS1)(QUOTE(A))
(CSET (QUOTE LIS1)(QUOTE (A)))	Form evaluated	(CSET(QUOTE (QUOTE LIS1)) (QUOTE(A)))
(QUOTE LIS1) = LIS1 (QUOTE (A)) = (A)	Evaluated arguments	(QUOTE(QUOTE LIS1)) = (QUOTE LIS1) (QUOTE (A)) = (A)
LIS1	Value	Error, because the first argument (QUOTE LIS1) is non-atomic.

The only way to set or remove an atomic binding is by evaluation of a CSET, CSETQ, or in special cases discussed below, a SETQ. As such, atomic bindings are the most permanent bindings available in LISP, and when atoms with atomic bindings are used as variables, they are called "global variables" or "constants."

16.3 PUSHDOWN LIST BINDINGS

The majority of bound variables in LISP 1.5 are variables bound on the pushdown list, and the principle way of binding variables on the pushdown list is through the use of LAMBDA expressions and LAMBDA conversion. Bindings on the pushdown list can also be established with PROG expressions and with SETQ.

When we create a LAMBDA expression, we state all the LAMBDA variables in the list of variables following the word LAMBDA. In so doing, we say that "the variables are bound by the LAMBDA." For example, in the following expression

```
(LAMBDA (A B C) (LIST A B C)) (1 2 3)          (1)
```

the variables A, B, and C are bound by the LAMBDA to the values 1, 2, and 3, respectively. Internally at run time (when LAMBDA conversion takes place), a block of cells on the pushdown list is reserved for LAMBDA variables, and the pointers to the values 1, 2, and 3 are placed in these reserved cells in their proper places. After evaluation, the reserved block of pushdown storage is released and the values bound to the variables A, B, and C are lost. Clearly then, LAMBDA conversion creates temporary bindings so that we may evaluate a defined function repeatedly with different arguments.

Before we leave LAMBDA variables, note the term dummy variables. If we write

(LAMBDA (X Y Z) (LIST X Y Z)) (1 2 3) (2)

the expression (2) would evaluate exactly as (1) above. More importantly, the systematic substitution of X, Y, and Z for A, B, and C, respectively, did nothing to change the form or meaning of the expression. This is the whole point of Church's LAMBDA notation. Thus, any atomic symbol (except T, F, numbers, and NIL) will suffice as a LAMBDA variable and so they are called dummy variables.

The second most important way of putting variable bindings on the pushdown list is by use of the PROG feature. For Q-32 LISP, PROG variables are treated exactly like LAMBDA variables with the exception that PROG variables are always initially bound to NIL on the pushdown list. Whereas LAMBDA variables are bound implicitly through the mechanism of lambda-conversion and evaluation, PROG variables must be explicitly set by the programmer; the form SETQ being used for that purpose. SETQ may also be used on LAMBDA variables. PROG variables are also dummy variables.

CSET or CSETQ may also be used with dummy variables. The bindings so created are not permanent, but temporary, analogous to SETQ bindings. However, CSET or CSETQ also set such dummy variables into a special state discussed further below. Suffice to say, use SETQ on dummy variables to produce temporary bindings. Use CSET or CSETQ on constants to produce permanent bindings.

16.4 MULTI-LEVEL BINDINGS

A good question about this time is, "What value is bound to a variable during recursion?" The answer is simple, but the mechanism is not so simple. The values bound to variables during recursion (or during any multi-level nesting of the variables in a hierarchy of precedents) are the last computed values for these variables. The mechanisms of list processing permit the manipulation of LIFO storage, and values are effectively piled one atop the other. Thus, when a function is evaluated at a particular level, the current binding of a variable is the value at the "top of the heap," which is the most recent value computed. When returning from a function, this value is discarded and the previous value is exposed at the "top of the heap." For example, in

FACTORIAL (3)

(see the definition for FACTORIAL in the previous chapter) the dummy variable N is bound by the LAMBDA on the pushdown list many times; once for each evaluation of FACTORIAL. As we descend deeper in the recursion, we use more pushdown storage to bind each current value of N.

"Top of the heap" binding of N	Value of Factorial	Level
3	descending	top level = 1
2	descending	2
1	descending	3
0	1 ascending	4
1	1 ascending	3
2	2 ascending	2
3	6	1

When we trace FACTORIAL, we are essentially printing the current binding of N at each level of the recursion.

Of course, this only applies to variables bound on the pushdown list. The value of a variable bound in the Special Cell on its atom structure is always the same regardless of the level at which the variable is evaluated. That is why we call them global variables, or constants.

16.5 FREE VARIABLES

The next interesting question is, "can a variable be used in a function in which it is not bound as a LAMBDA variable?" Yes. A variable used but not bound within the scope of the current function is said to be a free variable. This is shown in the two expressions below.

```
(LAMBDA NIL (CSETQ PI 3.14159)) NIL        (1)
(LAMBDA (R) (TIMES 2 PI R)) (2)            (2)
```

Expression (1) sets PI as a global variable. Expression (2) uses PI as a constant. Both expressions use PI as a free variable, since it is not a bound LAMBDA variable.

Free variables must be bound in the Special Cell attached to their atom name if the proper binding is to be retrieved during evaluation. To convey this information to the Q-32 LISP compiler, you must declare free variables as special cases before they are used. If you use CSET or CSETQ on free variables before they are referenced in functional expressions, you needn't take explicit action as CSET or CSETQ automatically make this special declaration for you. Otherwise, you must use the function SPECIAL.

SPECIAL is a function of one argument, a list of all the variables used free in the functions being defined. Thus,

```
SPECIAL ((PI DOG))
```

would declare the atoms PI and DOG as special cases and any time PI or DOG is used as a free variable, values will be bound and retrieved from each atom's Special Cell.

Quite frequently, you may wish to use an atom name as a free variable in one expression, and as a bound LAMBDA variable elsewhere. You may, therefore, un-special any SPECIAL variables with the function UNSPECIAL. UNSPECIAL is the inverse in function but the same in form as SPECIAL. Thus,

```
UNSPECIAL ((PI DOG))
```

removes the special status from the atoms PI and DOG.

Free, SPECIAL variables are generally used as constants and used as read only data in LAMBDA or PROG expressions. If you try to set such variables with CSET, CSETQ, or SETQ, you permanently change the constant, since you change the binding on the Special Cell. Be careful when doing any evaluation that you don't unintentionally reset your constants.

16.6 SPECIAL VARIABLES AS LAMBDA OR PROG VARIABLES

With conscious forethought, you may use SPECIAL variables (i.e., variables explicitly declared with SPECIAL, or implicitly declared with CSET or CSETQ) as bound LAMBDA or PROG variables without recourse to UNSPECIAL. When SPECIAL variables are used this way, they will act as dummy variables and their prior value will be unchanged afterward, i.e., after the expression in which they are bound is evaluated. For example, if we evaluate these expressions in order, PI remains unchanged after the last expression is evaluated even though we use SETQ to temporarily bind PI to 5.

```
CSET (PI 3.14159) = PI
(LAMBDA (R)(PROG (PI)
   (SETQ PI 5)(RETURN (TIMES 2 PI R)))) (6) = 60
(LAMBDA () PI)() = 3.14159
```

The slight-of-hand that goes on internally to achieve this effect may be of interest to you. When an expression is first entered during evaluation, all dummy variables (i.e., LAMBDA and PROG variables) are assigned locations on the pushdown list, including any SPECIAL variables that are being used as dummy variables. During this assignment phase, the values of all dummy variables that are also SPECIAL variables are moved from their Special Cells to their assigned pushdown list locations, and the old values in these pushdown list locations are placed in the Special Cells. Thus, the contents of the Special Cell and the pushdown list location for each SPECIAL variable are interchanged. (Note that for PROG variables, which are initialized to NIL, NIL winds up in the Special Cell.) During evaluation of the expression, all references to SPECIAL variables are references to the Special Cell, whereas all other dummy variable references are to the pushdown list. If recursion takes place, new pushdown list locations are allocated to all dummy variables, and swapping of pointers between Special Cells and these new pushdown list locations for SPECIAL variables is repeated. Finally, upon exiting the expression that was evaluated, the pointers are swapped back between Special Cells and pushdown list locations, with the end result leaving the original values of the SPECIAL variables in their Special Cells where they belong.

16.7 CAUTIONS

1. Never use T, F, numbers, or NIL as bound variables.
2. Never use the same atom as both a LAMBDA and PROG variable within the same expression.
3. Declare variables SPECIAL when they are used free and they are not bound by a prior CSET or CSETQ.
4. UNSPECIAL free variables when they are no longer needed as free variables, as it will increase evaluation speed by eliminating unnecessary swapping of SPECIAL variable values.
5. Always use QUOTE when you want the literal name of something, rather than its value, except where quote is performed for you automatically, e.g., SETQ, CSETQ, GO, and top level arguments.
6. Except that CSETQ returns as its value the value of its first argument, and SETQ returns as its value the value of its second argument, CSETQ and SETQ bind variables exactly the same way; however, CSETQ also makes its variables SPECIAL. On dummy variables they make temporary bindings, and on free variables, they make permanent bindings. CSET is like CSETQ in this regard; however, CSET evaluates its first argument, whereas CSETQ quotes its first argument.
7. Never try to SETQ an unbound free variable (i.e., a variable that was never used previously with CSET, CSETQ, or as a dummy variable) it will cause an error.

16.8 EXERCISES

1. Identify the dummy variables, the LAMBDA variables, the PROG variables, the bound variables, SPECIAL variables, and the free variables in the following expressions:

```
CSET (PI 3.14159)
UNSPECIAL ((X Y Z A B N M))
DEFINE (( (TEST (LAMBDA (X Y Z)(PROG (A B)
          (RETURN (LIST A B Y N Z PI
            ((LAMBDA (Z)(CONS Z (LIST X M ))) Y) ))))) ))
```

2. Evaluate CSET and UNSPECIAL above and then define the function TEST above on the computer and see the Q-32 error message:

"____ NOT DECLARED"

returned for each occurrence of a non-SPECIAL free variable. Does the error diagnostic(s) agree with your answers in problem 1?

3. Evaluate

```
                    TEST (1 2 3)
```

4. Declare all free variables SPECIAL and redefine TEST. Any error messages?
   Evaluate

```
                    TEST (1 2 3)
```

   Is there any difference between the evaluation of TEST here and in problem 3?

5. Evaluate the following in order:

```
1.  CSET (K 1965)
2.  (LAMBDA () K)()
3.  (LAMBDA (X) X)(K)
4.  (LAMBDA () (QUOTE K))()
5.  (LAMBDA () (ADD1 K))()
```

6. Evaluate the following in order:

```
1.  SPECIAL ((V1 V2))
2.  (LAMBDA ()(CSETQ V1 (QUOTE V2)))()
3.  (LAMBDA ()(CSETQ V2 (QUOTE V1)))()
4.  LIST (V1 V2)
5.  (LAMBDA ()(LIST V1 V2))()
```

7. Evaluate the following in order:

```
1.  (LAMBDA NIL (CSETQ PI 3.14159)) NIL
2.  (LAMBDA NIL PI) NIL
3.  (LAMBDA NIL (PROG (PI)(SETQ PI 5)(RETURN PI))) NIL
4.  (LAMBDA NIL PI) NIL
```

8. Evaluate the following in order:

```
1.  (LAMBDA NIL (CSETQ PI 3.14159)) NIL
2.  (LAMBDA NIL PI) NIL
3.  (LAMBDA NIL (PROG (PI) (CSETQ PI 5)(RETURN PI))) NIL
4.  (LAMBDA NIL PI) NIL
```

9. Evaluate the following expressions and see the bindings of PI at various levels of evaluation.

```
CSET (PI 3.14159)
(LAMBDA ()(PROG2
   ((LAMBDA ()(PROG (PI)
      (PRINT (LIST (QUOTE BEFORESETQ) PI))
      (SETQ PI 1234)
      (PRINT (LIST (QUOTE AFTERSETQ) PI)))))
   (LIST (QUOTE PERMANENTVALUE) PI)))()
```

10. Evaluate the following in order:

```
1. CSET (PI 3.14159)
2. (LAMBDA ()(SETQ PI 54321))()
3. (LAMBDA () PI)()
```

11. Evaluate the following in order:

```
1. CSET (ABE LINCOLN)
2. (LAMBDA (X)(SETQ X (QUOTE FREEDOM)))(ABE)
3. (LAMBDA () ABE)()
4. LAMBDA (ABE)(SETQ ABE (QUOTE CIVILWAR)))(BOOTH)
5. (LAMBDA () ABE)()
```

12. Evaluate the following in order:

```
1. CSET (PI 5)
2. (LAMBDA (R)(TIMES 2 PI R)))(5)
3. (LAMBDA (R)(PROG ()
      (SETQ PI 3.14159)
      (RETURN (TIMES 2 PI R))))(5)
4. (LAMBDA NIL PI)()
```

13. Evaluate the following in order:

```
1. UNSPECIAL ((PI ABE))
2. (LAMBDA NIL (SPECIAL (LIST (QUOTE PI)(QUOTE ABE))))()
```

14. Evaluate the following in order:

```
1. CSET(ABE LINCOLN)
2. CSET(JOHN BOOTH)
3. (LAMBDA NIL (SPECIAL (LIST ABE JOHN)))()
```

15.
```
1. CSET (ABE LINCOLN)
2. CSET (JOHN BOOTH)
3. (LAMBDA (X Y)(LIST
   (LIST X (CSET ABE (QUOTE PRESIDENT)) LINCOLN)
   (LIST Y (CSET JOHN (QUOTE ACTOR)) BOOTH)))(ABE JOHN)
4. (LAMBDA (A B)(LIST (LIST A ABE LINCOLN)
                (LIST B JOHN BOOTH)))(ABE JOHN)
```

CHAPTER 17. INPUT-OUTPUT AND THE SUPERVISOR

Input and output in LISP are handled principally by the two pseudo-functions READ and PRINT, which read and print one S-expression, respectively. Since input-output is extremely machine dependent, we shall, here, only concentrate on those machine independent primitives available on Q-32 LISP.

READ NIL

READ is a pseudo-function of no arguments. Its evaluation causes one S-expression to be read from the user's Teletype, which is returned as the value of READ. READ signals for more input by ringing two bells.

PRINT (s)

PRINT is a pseudo-function of one argument s. Its evaluation causes the one S-expression s to be printed on the user's Teletype. The S-expression s is also returned as the value of PRINT.

PRINO (s)

PRINO (sounds like PRIN-zero) is a pseudo-function of one argument an S-expression, s. Its evaluation causes the Print Name of all atoms in S-expression s to be entered into the print line, including characters for left and right parentheses and dot, wherever necessary, without terminating the print line. It is like PRINT in all respects, except it does not evaluate TERPRI as its last internal function. PRINO uses PRIN1 below, and is used by PRINT. Its argument s is returned as its value.

PRIN1 (a)

PRIN1 is a pseudo-function of one argument, an atom, a. Its evaluation causes the Print Name on the property list of the atomic symbol a to be entered into the print line without terminating the print line. The argument of PRIN1 must be an atomic symbol, which is returned as its value.

TERPRI NIL

TERPRI, for TERminate PRInt line, is a pseudo-function of no arguments. Its evaluation terminates the print line and outputs the line on the user's Teletype. If the line is already terminated, say, by an immediately preceding TERPRI or PRINT function, a blank line is printed. The value of TERPRI is NIL. PRINT uses both PRIN1 and TERPRI.

TEREAD NIL

TEREAD, for TERminate READ, is a pseudo-function of no arguments. Its evaluation clears the read line and terminates the print line by executing TERPRI. The value of TEREAD is NIL.

BLANKS (n)

BLANKS is a pseudo-function of one argument, a number, n. Its evaluation causes n blanks to be entered into the print line without terminating the print line. The argument of BLANKS must be a number. BLANKS returns NIL as its value.

17.1 READING AND PRINTING

The LISP READ program consists of two basic parts. There is a machine language routine to convert character strings into atoms. Its output is an atom read, with special atoms used for parentheses, period and other punctuation characters. A recursive subroutine CONS's these into list structure. When a character string is read, it must be compared with the character representation of all atoms seen so far, to determine whether this string is a new atom or a reference to one seen before. Therefore, there must be a means of rapid access to all the atoms in the system. There exists, therefore, a list called the object list, or OBLIST of all atoms. To speed up the search for comparisons, the OBLIST is usually organized as a list of a hundred or so sublists or "buckets." The atoms are distributed among the buckets by a computation upon their BCD representations (hash coding), which yields a reasonably uniform and random distribution of atoms among the buckets.

Manipulation of the OBLIST is the exclusive responsibility of the internal system, in much the same way as the JOVIAL dictionary is the JOVIAL system's responsibility. Though long, about fifty Teletype lines, it is sometimes useful to see the current list of system atoms. The Q-32 LISP OBLIST can be printed by entering:

```
CAR (OBLIST)
```

Punctuation characters can be read and printed with Q-32 LISP. For printing, the following atoms are permanently bound to the Print Names as listed and will print as such; e.g.,

```
PRINT (SLASH) = /
```

Atom	Print Name
LPAR	(
RPAR	)
BLANK	space
PERIOD	.
SLASH	/
EQSIGN	=
DOLLAR	$
STAR	*

Since the current READ function uses many of these characters for syntactic analysis, they cannot be read directly. The "$$ artifact," however, circumvents this difficulty. Any character string preceded by $$ will be bracketed by the character following the second dollar sign and that character's next occurrence. The character cannot be a blank. All characters between these "ad hoc" brackets will be taken as the Print Name for an atomic symbol. That atom is a literal atom, i.e., no conversions or translations take place, and can be used like all other non-numeric LISP atoms. Some examples are shown below.

	$$Artifact	Atom Formed	Bracket Character
1.	$$*NOW IS THE TIME*	NOW IS THE TIME	*
2.	$$$123$	123 (in BCD not binary)	$
3.	$$AATOMA	ATOM	A
4.	$$B B	space	B
5.	$$.((.	((	.
6.	$$(..(	..	(

In example 3 above, $$AATOMA is internally equivalent to the atom ATOM. Thus, bindings for $$AATOMA are bindings for ATOM; e.g.,

CSETQ ($$AATOMA 123) = ATOM

(LAMBDA NIL ATOM) = 123

17.2 EVALQT

Earlier, we learned of the existance of the Q-32 LISP supervisory program called Evalquote. Evalquote is the principle interface between a LISP user and the LISP system (PRINT and READ also permit direct user interaction with the system), and is the mechanism that permits top level expression evaluation. Evalquote is in reality a LISP function, though not externally available. However, a similar function is available to Q-32 LISP users, appropriately named EVALQT. We introduce EVALQT at this time to demonstrate the power and flexibility of LISP to create other supervisors that emulate Evalquote. Before proceeding in that direction, let us define EVALQT and, thereby, review Evalquote.

EVALQT is a function of two arguments, both S-expressions, of the form

EVALQT (s_1 s_2)

The first argument, s_1, is a functional expression or a function. The second argument, s_2, is a list of arguments required by the function or functional expression, s_1. The arguments, s_1 and s_2, are of the same form as input to Evalquote. If s_1 is an atom carrying a functional definition, that definition, in the form of compiled code, is applied to the arguments in the list of arguments, s_2. If s_1 is a functional expression, the expression is compiled and then applied to the arguments in the list of arguments, s_2. The value of EVALQT is the value of s_1 applied to the arguments in the list of arguments, s_2.

Since s_1 and s_2 are identical inputs for Evalquote and EVALQT, at the top level we can evaluate s_1 applied to s_2 with either, e.g., if s_1 and s_2 stand for Evalquote inputs, let s_1' and s_2' be their EVALQT counterparts.

Then,

Evalquote

$$\underbrace{\texttt{CAR}}_{s_1}\ \underbrace{\texttt{((A B))}}_{s_2} = \texttt{A}$$

$$\underbrace{\texttt{(LAMBDA (X) (CAR X))}}_{s_1}\ \underbrace{\texttt{((A B))}}_{s_2} = \texttt{A}$$

EVALQT

$$\underbrace{\texttt{EVALQT}}_{s_1}\ \underbrace{\texttt{(}\ \underbrace{\texttt{CAR}}_{s'_1}\ \underbrace{\texttt{((A B))}}_{s'_2}\ \texttt{)}}_{s_2} = \texttt{A}$$

$$\underbrace{\texttt{EVALQT}}_{s_1}\ \underbrace{\texttt{(}\ \underbrace{\texttt{(LAMBDA (X)(CAR X))}}_{s'_1}\ \underbrace{\texttt{((A B))}}_{s'_2}\ \texttt{)}}_{s_2} = \texttt{A}$$

EVALQT may also be used at other than the top level, e.g.,

$$\underbrace{\texttt{(LAMBDA (X Y)(EVALQT X Y))}}_{s_1}\ \underbrace{\texttt{(}\underbrace{\texttt{CAR}}_{s'_1}\ \underbrace{\texttt{((A B))}}_{s'_2}\texttt{)}}_{s_2} = \texttt{A}$$

In this example, CAR is bound to X and ((A B)) is bound to Y by LAMBDA conversion. During evaluation of s_1, X and Y are evaluated to CAR and ((A B)), respectively, as the arguments of EVALQT. Like Evalquote, EVALQT then quotes each argument in the list ((A B)) and applies CAR to ((A B)).

17.3 SUPERVISORS

We can now examine a few examples that duplicate, in format but not in detail, the mechanism called Evalquote. Examine the following expression:

```
DEFINE ((
(SUP1 (LAMBDA ()(PROG ()
    TAG1 (PRINT (EVALQT (READ)(READ)))
         (GO TAG1)))) ))
```

SUP1 is a function of no arguments, yet it will evaluate expressions exactly as does Evalquote. SUP1 evaluates two explicit READ functions. The first READ reads s_1, the function to be evaluated by SUP1. The second READ reads s_2, the list of arguments for s_1. The values of these two READ expressions are the arguments for EVALQT exactly as above. The value of EVALQT is the value of s_1 applied to s_2, and that value is the argument of PRINT, which prints the value.

The program then loops for another evaluation. This program looks as if it will loop continuously, and it will; but then it will evaluate a new pair of S-expressions, s_1 s_2, each loop and that is exactly what a supervisory program is supposed to do. (We can always return to Evalquote in Q-32 LISP by (1) entering an escape character, the percent sign (%), after the bells in READ; (2) typing !STOP and after the error messages and bells, entering the quote mark ("), or; (3) depressing the "Break Key" a few times and after the error messages and bells, entering the quote mark (").)

The supervisor, SUP1 above, needs improvement because it is quite sensitive to input errors and extra parentheses. Good housekeeping requires that we re-initialize our read and print line each loop. Examine SUP2, below, which performs similarly to SUP1, above, but with TEREAD used to re-initialize I/O buffers.

```
DEFINE ((
(SUP2 (LAMBDA ()(PROG (X Y)
  TAG1 (TEREAD)
       (SETQ X (READ))
       (SETQ Y (READ))
       (PRINT (EVALQT X Y))
       (GO TAG1)))) ))
```

17.4 EVAL1

EVAL1 is another interesting Q-32 LISP function that may be used to construct various supervisory programs. It is a function of one argument, an S-expression that has the format of a lower-level form. EVAL1 will simply evaluate that form. No LAMBDA conversion or variable bindings take place; therefore, the arguments of the form to be evaluated by EVAL1 are not quoted, but must themselves be computed. For example, the form

(CAR (QUOTE (A)))

can be evaluated by:

$$\underbrace{\text{EVAL1}}_{s_1}\ \underbrace{\text{((CAR (QUOTE (A))))}}_{s_2}$$

The form (CAR (A))

is not acceptable, since the argument of CAR, (A), is an explicit value, not one that is computed. (QUOTE (A)) performs properly since it evaluates to a legal argument of CAR. One could also use (LIST (QUOTE A)) in this example.

The need to compute arguments for forms during evaluation is a responsibility common to all lower-level forms, not only for the argument of EVAL1. The following rule is of value for understanding when such computation is necessary.

RULE: Any atomic symbol immediately following a left parenthesis must be a function name that can be evaluated, with the following exceptions:

1. Atoms in a quoted list, e.g., (QUOTE (A B)).
2. LAMBDA and PROG variables, e.g., (LAMBDA (X Y).
3. Variables used as predicates in conditional clauses of COND, e.g., (COND (T F)).

We can now write a supervisor using EVAL1 for evaluating lower level forms.

```
DEFINE ((
  (SUP3 (LAMBDA ()(PROG ()
   TAG1 (TEREAD)
        (PRINT (EVAL1 (READ)))
        (GO TAG1)))) ))
```

17.5 TOP LEVEL ANOMALIES

Some straightforward, but unexpected top-level phenomena derive directly from the operation of Evalquote. We list them here with a brief explanation.

1. Composition of functions cannot be used directly at the top level except within a LAMBDA expression. For example, if we write

   (CAR (QUOTE (A B C)))

   we do not have a pair of S-expressions for Evalquote. We could evaluate this form with our SUP3 supervisor, however.

2. Bound variables are never evaluated at the top level except within a LAMBDA expression because all arguments are quoted, e.g.,

   CSET (PI 3.14159) = PI
   CAR ((PI)) = PI,

   not 3.14159 since what CAR really sees is (QUOTE (PI)). But

   (LAMBDA NIL PI) NIL = 3.14159

   since here PI is used free and will be evaluated. This is why top level LAMBDA expressions are so important.

3. If more than one pair of S-expressions is presented to Evalquote, only the first pair of S-expressions will be evaluated, since Evalquote only takes two arguments, e.g.,

   CAR ((A B C)) CDR ((A B C)) $\bigcirc\!\!\!\!\text{cr}$

   yields A. The CDR expression is never seen by Evalquote.

   This feature allows the user to end a top-level expression with more right parentheses than are necessary as Evalquote stops reading as soon as the parentheses count out correctly in the second argument. For example, the expression

   CAR ((A B C))))))))))))) = A

4. If less than one pair of S-expressions is given to Evalquote, it will demand more input by ringing the bell. This is a useful debugging tool and usually means one or more parentheses are missing in the entered pair of expressions.

5. Expressions evaluated at the top level that explicitly PRINT their values may have the values of the expressions output twice. Once by the explicit call to PRINT, and once by Evalquote, which always prints the value of the expression, e.g.,

```
              PRINT (ABCD) yields
ABCD
ABCD
```

17.6 EXERCISES

Evaluate the following in order:

1. PRINT ((LIST))
2. TERPRI NIL
   TERPRI NIL
3. (LAMBDA(X Y)(PROG()(PRIN1 X)(BLANKS 3)(PRIN1 Y)(TERPRI)))(ATOM1 ATOM2)
4. READ NIL
   after two bells enter:
   (NOW HEAR THIS)
5. (LAMBDA (J) (CONS (READ) J)) ((ANYTHING))
   after two bells enter:
   (INPUT)
6. (LAMBDA NIL (PROG (PI R)
   (SETQ PI 3.14159)
   TAG (SETQ R (READ))
   (COND ((EQUAL (QUOTE END) R) (RETURN R)))
   (PRINT (TIMES 2 PI R))
   (GO TAG))) NIL
   after two bells enter a number for R. (Remember to insert a space or comma before cr to delimit atom.) Program returns computation of (2)(PI)(R) and then reads again.
   You can stop the loop by entering:

   END, cr
7. (LAMBDA ()(LIST LPAR RPAR BLANK PERIOD SLASH EQSIGN DOLLAR STAR
   (QUOTE $$* NOW HEAR THIS *)(QUOTE $$+ -533.17+)))()
8. CDR ((A B C)) CDR ((1 2 3)) entered on one line.
9. 1. CSET(PERCENT $$*%*)
   2. (LAMBDA () PERCENT)()
   3. (LAMBDA (J) J)(%)

10. CAR ((A B C)))))))))))))

11. Define SUP2 as given in the examples above. Try SUP2 with these cases:
SUP2() - - - - - - - - - - - - to start SUP2 looping
    1. CAR ((A B C))
    2. CDR ((A B C))
    3. CONS (A B)
    4. CSET (PI 3.14159)
    5. (LAMBDA () PI)()
    6. % - - - - - - - - - - - - to exit SUP2

12. Define SUP3 as given earlier and try it with these cases:
SUP3()

```
1.  (CAR (LIST (QUOTE (A))))
2.  (CONS (QUOTE A)(QUOTE B))
3.  (CSETQ K 3.14159)
4.  (CONS K NIL)
5.  (PROG (X)(PRIN1 (QUOTE X)(BLANKS 5)(PRIN1 (QUOTE SQUARE))(TERPRI)(TERPRI)
          (SETQ X 0)
     TAG1 (COND ((EQUAL X 10)(RETURN (QUOTE END))))
          (PRIN1 X)(BLANKS 7)(PRIN1 (TIMES X X))(TERPRI)
          (SETQ X (ADD1 X))
          (GO TAG1))
6.  %
```

13. Define SUP4, a supervisor that reads S-expression pairs in reverse order from that accepted by Evalquote, i.e., s_2 followed by s_1.
Try these pairs:
    1. ((A B C)) CAR,
    2. ((A B C)) CDR,
    3. (A B) EQ,
    4. (1 2 3 4) PLUS,
    5. (K 3.14159) CSET
    6. NIL (LAMBDA () K)
    7. %

14. Define SUP5, a supervisor that evaluates pairs like SUP2, but also:
    1. Saves the symbolic pairs.
    2. Prints the pair for inspection after input, like an echo.
    3. Queries your acceptance or rejection of the printed pair.
    4. If you answer NO, it loops for another pair.
    5. If you answer YES, SUP5
    prints the pair, followed by an equal sign, followed by the value of the pair and then loops for another pair.

15. Write a program that prints a table of the following values for a range of X specified at program run time.
X XSQUARE SQRTX RECIPX FACTORIALX

CHAPTER 18. MACROS

In a compiler-based LISP system such as Q-32 LISP, we must be concerned with both compile time and run time activities of the system. When Evalquote evaluates DEFINE, we are talking about run time for the pseudo-function DEFINE. If we are defining a function, for example LAST, we are talking about compile time for LAST. In other words, one function's run time is another function's compile time.

As we have already seen, functions can be compiled by DEFINE or by top level evaluation of a LAMBDA expression. In the latter case, evaluation means first compile and then run the compiled code with the supplied arguments. This is often called "compiling at run time." This distinction is significant because it enables compiled code to operate where previously an interpreter was necessary. In particular, it affects the code that is compiled for a function that enables that function to retrieve the correct binding for variables at run time.

A classic problem for compilers is, how do you define a function of an indefinite number of arguments, such as PLUS? The key to the answer is that the arguments are only indefinite when you define the function, not when you run it. If you could delay compilation until run time, at which time the number of arguments is definite, you can resolve this dilemma. In essence, this is what an interpreter does. To resolve this problem in Q-32 LISP, we make use of macros via the pseudo-function MACRO.*

18.1 MACRO EXPANSION

The function MACRO takes an argument list in the same format as DEFINE, e.g.,

MACRO (((name$_1$ (LAMBDA (variables) form$_1$))
(name$_2$ (LAMBDA ...
...
(name$_n$ (LAMBDA (variables) form$_n$))))

As DEFINE, MACRO compiles each of these definitions. Now watch closely, for here comes the difference. When a macro function (defined by MACRO) is used in a LAMBDA expression, either at the top level or within a DEFINE, the macro function is executed before the LAMBDA expression, of which it is part, is compiled. The argument for the macro is the S-expression in which it is used. In other words, the macro function is run before compile time for the new LAMBDA expression. What does this buy us? That depends on the macro, but essentially it allows us to expand the LAMBDA expression before it is compiled, by substituting, for all occurrences of

* The Q-32 Macro system is based upon an idea suggested by Tim Hart. See Hart, T. P., "MACRO Definitions for LISP," Artificial Intelligence Project, MIT Computation Center Memo 57, October 1963.

the macro function and its arguments, other expressions tailored to the particular use of the macro in the LAMBDA expression. We call this macro expansion. For example, it permits us to define a "special form" of an indefinite number of arguments by converting that special form to a composition of nested function of just two arguments--the nesting being determined by examination of the particular use of the special form in the given LAMBDA expression.

Take for example

$$\text{PLUS } (x_1\ x_2\ \ldots\ x_n)$$

Here we have a special form of an indefinite number of arguments. But when we use PLUS, we always have a fixed number of arguments. Given a function *PLUS, which takes the sum of its two arguments, we can expand

$$\text{PLUS } (x_1\ x_2\ x_3\ x_4) = (\text{*PLUS } x_1\ (\text{*PLUS } x_2\ (\text{*PLUS } x_3\ x_4)))$$

Thus the macro definition of PLUS involves a body of code whose sole purpose is to substitute an appropriate number of *PLUS's in the proper places wherever PLUS appears in a LAMBDA expression being compiled. Then after compilation, there is no trace of PLUS, but many *PLUS's. The operating code, however, works exactly as desired. Let's examine the macro definition for PLUS to see how this works.

```
MACRO ((
(PLUS (LAMBDA (L) (*EXPAND L (QUOTE *PLUS))))
      ))
```

where L is the form

$$(\text{PLUS } x_1\ x_2\ x_3\ x_4)$$

*EXPAND is a system function used exclusively for expanding macros. It has the format

*EXPAND (form fn)

where form is the expression to be expanded, as L above, and fn is the system function, a function of two variables, to be used in the expansion. For

$$\text{form} = (\text{PLUS } x_1\ x_2\ x_3\ x_4),\quad \text{fn} = \text{*PLUS}$$

we get

$$\text{*EXPAND } ((\text{PLUS } x_1\ x_2\ x_3\ x_4)\ \text{*PLUS}) = (\text{*PLUS } x_1\ (\text{PLUS } x_2\ x_3\ x_4)).$$

Note that *EXPAND just expands the form by one *PLUS when it is executed. When we then attempt to compile the new form

$$(\text{*PLUS } x_1\ (\text{PLUS } x_2\ x_3\ x_4))$$

*EXPAND is called again to expand the inner PLUS, yielding

$$(\text{*PLUS } x_1\ (\text{*PLUS } x_2\ (\text{PLUS } x_3\ x_4)))$$

By repeated application of *EXPAND each time the macro PLUS is encountered, we eventually arrive at the complete expanded form for PLUS regardless of the number of arguments x_n.

The definition for *EXPAND is straightforward and noted here for reference.

```
DEFINE ((
(*EXPAND (LAMBDA (FORM FN)
  (COND ((NULL (CDDR FORM)) (CADR FORM))
        (T (CONS FN
                (CONS (CADR FORM)
                (CONS (CONS (CAR FORM)(CDDR FORM)) NIL))))))) ))
```

Note how nicely *EXPAND works for the last term of the expansion, of say

(*PLUS x_1 (*PLUS x_2 (*PLUS x_3 (PLUS x_4))))

When entered because macro PLUS was encountered,

form = (PLUS x_4), fn = *PLUS

the

(CDDR form) = NIL, (CADR form) = x_4

Thus the form

(PLUS x_4)

gets replaced by just x_4, yielding the final expanded expression

(*PLUS x_1 (*PLUS x_2 (*PLUS x_3 x_4)))

It should be clear now how elegant this macro system is. To solve the knotty problem of special forms of an indefinite number of arguments, all we need are three things:

1. *EXPAND--a single LISP function available on Q-32 LISP.
2. A function like the macro to be defined, but of just two arguments. Such a two-argument function is easily defined in LISP.
3. A macro definition of the special form.

Macros must be defined before they are used. Once defined, macros may be used within other macro definitions, thereby providing complete generality of MACRO.

18.2 MACRO DEFINITIONS OF NEW FUNCTIONS

MACRO has utility in areas other than expansion of special forms. It can be used to define functions not already in the system. Take, for example, the pseudo-function CSETQ. Assuming we have CSET, we can define CSETQ by:

```
MACRO ((
 (CSETQ (LAMBDA (FORM) (LIST (QUOTE CSET)
                            (LIST (QUOTE QUOTE) (CADR FORM)) (CADDR FORM))))
       ))
```

Then whenever the form

(CSETQ A B)

is encountered

(CSET (QUOTE A) B)

will be substituted and compiled.

18.3 EXERCISES

1. *TIMES exists as a function of two arguments whose value is the product of its arguments. Define a macro function PROD, using *TIMES and *EXPAND such that

   PROD (x_1 x_2 ... x_n) = (*TIMES x_1 (*TIMES x_2 ... (TIMES x_{n-1} x_n)...))

2. *MAX and *MIN exist as functional counterparts of MAX and MIN, but only having two arguments. Define the macros MAXIMUM and MINIMUM.

3. In the next chapter, dealing with functional arguments, we will see that we must always use the special form FUNCTION, when we wish to quote a functional expression appearing as an argument of another functional expression; e.g.,

   (LAMBDA (L) (MAPLIST L (FUNCTION (LAMBDA (J) (LIST J))))) (1)

   Define the macro FLAMBDA, which when used as in form (2),

   (LAMBDA (L) (MAPLIST L (FLAMBDA (J) (LIST J)))) (2)

   will expand form (2) to form (1).

4. If you define LIST as a macro and it's wrong, you can wreck the system. Therefore, define LIST1 as a macro that does exactly what LIST does.
   HINT: Remember that

   (CONS x_{n-1} x_n) = (x_{n-1} . x_n)

   so the macro must produce

   (CONS x_n NIL)

   as its last expansion. In other words, we want

   LIST1 (A B C) = (A B C)

   and <u>not</u> (A B . C)

5. When printing multi-word messages in LISP, we always print the message as a parenthetical expression, i.e., a list; e.g.,

   (NOW HEAR THIS)

Define a macro PRINTQ that is a special form of an arbitrary number of arguments that quotes its arguments and prints them (on one line if they will fit) without parenthesization; e.g.,

```
(LAMBDA ()(PRINTQ NOW HEAR THIS)) () = NOW HEAR THIS
```

HINT: Define PRINTQ as a macro that uses an auxiliary function PRINTQ1 which in turn uses PRINO on each argument of PRINTQ.

CHAPTER 19. FUNCTIONAL ARGUMENTS

Mathematically, it is possible to have functions as arguments of other functions. For example, in arithmetic one could define a function

OPERATE (op a b)

where op is a functional argument that specifies which arithmetic operation is to be performed on a and b. Thus

```
OPERATE (+ 3 4) = 7
OPERATE (* 3 4) = 12
```

In LISP, functional arguments are extremely useful and further expand the class of LISP functions. We call the class of functions that take functional arguments, functionals.

19.1 SPECIAL FORM FUNCTION

When arguments of a function are transmitted to the function they are always evaluated, except when they are quoted arguments. Quoted arguments are transmitted unevaluated as "literals." When we use functionals, we use functions and functional expressions as their arguments. At such times, we wish to transmit these arguments unevaluated. The special form FUNCTION is used for this purpose on Q-32 LISP. FUNCTION acts very much like QUOTE, and in fact on other LISP systems, FUNCTION and QUOTE are often interchangeable. Not so on Q-32 LISP, as FUNCTION must be used with functional arguments to signal the LISP system that a function is being transmitted.

FUNCTION is a special form that takes one argument, a function name or a LAMBDA expression. It has the format

FUNCTION (fn)

We can see the application of FUNCTION and functionals by examining a particularly powerful class of functionals prefixed with the name MAP. These functionals are generally alike, in that they all apply a transmitted functional argument to a transmitted list.

19.2 MAP

MAP is a function of two arguments of the form

MAP (x fn)

where the first argument x must be a list, and the second argument fn must be a function of one argument. MAP applies the function fn to list x and to successive CDR's of list x until x is reduced to a single atom (usually NIL) which is returned as the value of MAP. MAP is defined by

```
DEFINE ((
 (MAP (LAMBDA (X FN)(PROG (M)
    (SETQ M X)
TAG1 (COND ((ATOM M)(RETURN M)))
     (FN M)
     (SETQ M (CDR M))
     (GO TAG1)))) ))
```

Examples:

```
1.   (LAMBDA (L)(MAP L(FUNCTION PRINT)))((THIS IS (A LIST))) =
     (THIS IS (A LIST))
     (IS (A LIST))
     ((A LIST))
     NIL

2.   CSET (K ((CDR K)(CAR K)(CONS (QUOTE A)(QUOTE B))))
     (LAMBDA ()(MAP K (FUNCTION (LAMBDA (J)(PRINT (EVAL1 (CAR J))))))) () =
     ((CAR K)(CONS (QUOTE A)(QUOTE B)))
     (CDR K)
     (A . B)
     NIL
```

In example (1), PRINT is the functional argument. Each line of output represents the application of PRINT to successive CDR's of the list (THIS IS (A LIST)). The final NIL is the value of MAP. In example (2), the LAMBDA expression

(LAMBDA (J)(PRINT (EVAL1 (CAR J))))

is the functional argument. The dummy variable J is bound to the successive CDR's of the list bound to K, i.e.,

((CDR K)(CAR K)(CONS (QUOTE A)(QUOTE B)))

This is a list of forms evaluated by EVAL1 in the functional argument. The result of each such evaluation is returned by PRINT. The final NIL is the value of MAP.

19.3 MAPLIST

MAPLIST is a function that performs almost exactly as does MAP, except MAPLIST returns as its value a LIST of the values of the repeated evaluation of fn applied to x.

MAPLIST is a function of two arguments of the form

MAPLIST (x fn)

where the first argument, x, must be a list, and the second argument, fn, must be a function of one argument. The value of MAPLIST is a new list of the evaluation of each of the listed forms below:

```
    ((fn x) (fn (CDR x)) (fn (CDDR x)) ... (fn (CD...DR x)))
```

A definition for MAPLIST (though not the one used by Q-32 LISP) can be given as

```
DEFINE ((
(MAPLIST (LAMBDA (X FN)
  (COND ((NULL X) NIL)
        (T (CONS (FN X) (MAPLIST (CDR X) FN)))))) ))
```

Examples:

```
DEFINE ((
(SQUARECAR (LAMBDA (X) (TIMES (CAR X) (CAR X)))) ))


1.  (LAMBDA (J)(MAPLIST J(FUNCTION SQUARECAR)))((1 2 3 4 5)) = (1 4 9 16 25)
2.  (LAMBDA (J)(MAPLIST J(FUNCTION CDR)))((A B C)) = ((B C) (C) NIL)
```

In these examples, SQUARECAR, and CDR are functional arguments.

19.4 MAPCAR

The function MAPCAR is a function peculiar to Q-32 LISP. It is a function like MAPLIST, in that it lists the values of functional argument fn successively applied to the CADR's of list x. It differs from MAPLIST, in that it applies fn to each element of the list x; i.e., the CAR of what fn is applied to in MAPLIST. MAPCAR could be defined by:

```
DEFINE ((
 (MAPCAR (LAMBDA (X FN)
   (COND ((NULL X) NIL)
         (T (CONS (FN (CAR X))(MAPCAR (CDR X) FN)))))) ))
```

Examples:

```
1.  (LAMBDA (J)(MAPCAR J (FUNCTION ADD1)))((0 1 2 3)) = (1 2 3 4)
2.  (LAMBDA (J)(MAPCAR J (FUNCTION (LAMBDA (L)
        (COND ((NUMBERP L)(TIMES L L))
              (T L)))))) ((A 1 B 2 C 3)) = (A 1 B 4 C 9)
```

In example (1) ADD1 is the functional argument, and the total expression adds one to each element in a list of numbers. In example (2), we have a LAMBDA expression as the functional argument. The complete expression returns its input list with each numerical element replaced by its square.

19.5 CAUTIONS

Most functionals (i.e., functions that take functional arguments) cannot be used at the top level to Evalquote, since the functional arguments must be evaluated, or compiled and evaluated. As we know, Evalquote quotes the arguments when transmitting them to the function and thus the functional arguments for functionals would not be evaluated. Therefore, use functionals only in LAMBDA expressions.

19.6 EXERCISES

Evaluate the following:

```
1.  (LAMBDA (L)(MAP L (FUNCTION PRINT))) ((TRY THIS SIMPLE CASE FIRST))
2.  (LAMBDA (L)(MAPLIST L (FUNCTION PRINT))) ((NOW THIS ONE))
3.  (LAMBDA (L)(MAPCAR L (FUNCTION PRINT))) ((AND LASTLY THIS ONE))
4.  (LAMBDA (J)(MAPLIST J(FUNCTION
        (LAMBDA (K)(SUBST (QUOTE ONE) 1 K)))))((1 2 3 1 4 1 5))
5.  (LAMBDA (J)(MAPLIST J (FUNCTION
        (LAMBDA (K)(MAPCAR K (FUNCTION LENGTH))))))
         (((A)(1 2)(A B C)(1 2 3 4)))
6.  (LAMBDA (L) (MAPLIST L (FUNCTION (LAMBDA (J) (CONS (CAR J)
         (CAR J))))))((A B C D E))
7.  (LAMBDA (L) (MAPCAR L (FUNCTION (LAMBDA (J)
        (CONS J (QUOTE X))))))((A B C D E))
8.  SPECIAL ((Y))
    DEFINE ((
      (YDOT (LAMBDA (L Y) (MAPCAR L (FUNCTION (LAMBDA (J)
            (CONS J Y)))))) ))
    UNSPECIAL ((Y))
```

Note: If we consider the functional argument here as a separate function, it is evident that it contains a bound variable J, and a free variable Y. This free variable requires a SPECIAL declaration, even though it is bound in YDOT.

```
Try YDOT ((A B C D E) Z)
    YDOT ((A B C D E) (1 2 3 4 5))
```

9. MAPCAR is a function of two arguments in which the second argument is a function that takes one argument. Define functional MAPCAR2 as a function of three arguments in which the first two arguments are lists of equal length and the last argument is a function that takes two arguments; e.g.,

```
(LAMBDA (A B)(MAPCAR2 A B (FUNCTION DIFFERENCE)))((5 6 7 8)(1 2 3 4))
    = (4 4 4 4)
(LAMBDA (A B)(MAPCAR2 A B (FUNCTION CONS)))((ONE TWO THREE)(1 2 3))
    = ((ONE . 1)(TWO . 2)(THREE . 3))
```

10. Define a function using functionals called

TYPE (x)

where x is a list of items. The value of TYPE is a list of type-descriptors of each top level element of x according to the following schedule:

if fixed point number--FIX
if floating point number--FLT
if non-numeric atom--ATOM
if dotted pair of atoms--DOTPAIR
if none of the above--LIST

e.g.,

```
TYPE ((1.0 (A . B) (1 2 3) A12 46)) = (FLT DOTPAIR LIST ATOM FIX)
```

CHAPTER 20. META-LANGUAGE

The preceding chapters have all dealt with the LISP programming language, or S-expressions. Outside computers, however, a short-hand language is often used. This language, often called "source language" or "M-language," consists of meta-expressions or M-expressions, the counterpart of S-expressions. The M-language is the one most frequently used for publication and program specification.

In this chapter we shall present the M-language and some straightforward rules for conversion between M- and S-expressions.

20.1 META-LANGUAGE FEATURES

- Function names and variable names use lower case letters.
- Quoted atoms use capital letters.
- The arguments of a function are bound by square brackets and separated from each other by semicolons.
- Compositions of functions may be written by using nested sets of brackets.
- A conditional expression has the form:

  $[p_1 \rightarrow e_1;\ p_2 \rightarrow e_2;\ \ldots;\ p_n \rightarrow e_n]$

- A LAMBDA expression has the form:

  λ [[args]]; form]

- A PROG form is given by:

  prog[[args]; statement_1; statement_2; ... statement_n]

- Setting a PROG variable has the form:

  u:=ℓ; v:=v+1 ; w:=cdr[w]

- A quoted list is parenthesized.

Examples:

```
1.  equal [x;y] = [atom[x] → [atom[y] → eq[x;y]]; T → F];
                   equal[car[x]; car[y]] → equal[cdr[x]; cdr[y]] ; T → F]

2.  member [x;y] = [null[y] → F;
                    equal[x;car[y]] → T;
                    T → member[x;cdr[y]]]

3.  maplist [x;fn] = [null[x] → NIL;
                      T → cons [fn[x];maplist[cdr[x];fn]]]
```

4.
```
length [ℓ] = prog[[u;v];
  v:=0;
  u:=ℓ;
A [null[u] → return [v]];
  u:=cdr [u];   v:=v+1 ; go[A]]
```

20.2 TRANSLATION: M-EXPRESSIONS TO S-EXPRESSIONS

The following rules define a method of translating functions written in the meta-language into S-expressions.

1. If the function is represented by its name, it is translated by changing all of the letters to upper case, making it an atomic symbol; e.g., "car" is translated to CAR.

2. If the function uses LAMBDA notation, then the expression

   λ[[args]; form]

   is translated into

   (LAMBDA (args) form)

3. If the function begins with "label," then

   label [name; form]

   is translated into

   (LABEL name form)

4. A variable, like a function name, is translated by using upper case letters; e.g., "var1" is translated to VAR1.

5. Constants, represented as capital letters, translate into quoted expressions, except numbers, NIL, T, and F, e.g.,

   X to (QUOTE X)
   NIL to NIL
   F to NIL

6. The conditional expression

   $[p_1 \rightarrow e_1; \ldots p_n \rightarrow e_n]$

   is translated into

   $(\mathrm{COND}\ (p_1\ e_1)\ \ldots\ (p_n\ e_n))$

Examples:

M-expressions	S-expressions
x	X
X	(QUOTE X)
(X)	(QUOTE (X))
car	CAR
car[x]	(CAR X)
T	T
F	NIL
NIL	NIL
1234	1234
car[cdr[y]]	(CAR (CDR Y))
[atom[x] → x;T → car[x]]	(COND ((ATOM X) X) (T (CAR X)))
label[ff; λ[[x]; [atom[x] → x; T → ff[car[x]]]]]	(LABEL FF (LAMBDA (X)(COND ((ATOM X) X) (T (FF (CAR X))))))

20.3 EXERCISES

Evaluate the following M-expressions. To check your answers on Q-32 LISP, convert problems to S-expressions and evaluate them with various arguments.

1. [T → A;T → B]
2. [F → A;T → B]
3. [eq[A;A] → car [(A)]; T → cdr [(B)]
4. [null[X] → Y; null[( )] → NIL;T → atom [A]]
5. [atom[x] → atom [x]; T → eq[X;X]]

In the following problems, variables f,m,n,t,x,y, and z are bound as follows:

f=F	t=T	z=AB
m=AB	x=((AB))	
n=(AB . C)	y=((AB . C))	

all other variables are unbound and therefore undefined.

Evaluate:

6. [eq[m;z] → n;T → x]
7. [f → A;T → B;T → c]
8. [eq[AB;m] → A;T → B]
9. [atom[m] → A;T → B]
10. [atom[n] → A;T → B]
11. [eq[m;caar[x]] → y;T → w]
12. [[T → F; F → T] → F; T → [F → T; T → T]]
13. [[eq[cdr[n]; cdar[y]] → F; T → T] → y; T → z]
14. [[eq[m;z] → eq[f;t]; T → null[cdr[cdr[x]]] → B; T → C]
15. cons[cons[cons[f;t];z];x]

Evaluate by λ-conversion:

16. λ[[x];x][A]
17. λ[[x];car[(A)]][(B)]
18. λ[[u;v];v][A;B]
19. λ[[x;y];cons[car[y];cdr[x]]][(A . B);(C . D)]
20. cons[A;[λ[[x;y];y][T;F] → B; T → C]]

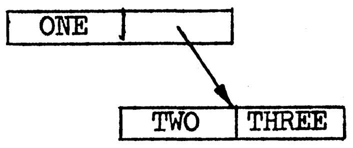
APPENDIX A

EXERCISE ANSWERS

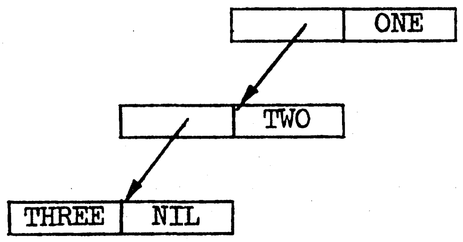
CHAPTER 2.

1. Yes
2. No. First character not a letter.
3. Yes
4. Yes
5. No. There are no parentheses in an atomic symbol.
6. Yes
7. Yes
8. Yes
9. No. This is a dotted pair.
10. No. First character not a letter.
11. No. Parentheses missing.
12. No. Parentheses missing.
13. No. Too many dots without proper parenthesization.
14. Yes
15. Yes

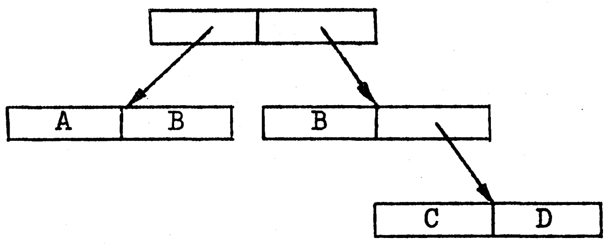
16.



17.



18.



19. (A . (B . ((C . NIL) . (D . NIL))))

20. (((A . B) . (C . D)) . (E . ((F . G) . H)))

CHAPTER 3.

```
 1.  (ATOM . NIL)
 2.  ((LISP . NIL) . NIL)
 3.  (((MORE . (YET . NIL)) . NIL) . NIL)
 4.  (HOW . (ABOUT . (THIS . NIL)))
 5.  (DONT . ((GET . ((FOOLED . NIL) . NIL)) . NIL))
 6.  (X1)
 7.  (NIL (X1))
 8.  (KNOW THY SELF)
 9.  ((BEFORE AND AFTER))
10.  (A ((B C)))
11.  ((A)) = ((A . NIL) . NIL)
12.  (NIL NIL NIL) = (NIL . (NIL . (NIL . NIL)))
13.  (A ((B) B)) = (A . ((B . NIL) . (B . NIL)))
14.  ((((NEST)))) = ((((NEST . NIL) . NIL) . NIL) . NIL)
15.  ( ((A) B) (C) D) = (((A . NIL) . (B . NIL)) . ((C . NIL) . (D . NIL)))
```

CHAPTER 4.

1. Yes
2. Yes
3. Yes
4. Yes
5. Yes
6. Yes
7. No, yields error messages:

```
        (RPAR RQD AFTER .)                                       (1)
        PRINT READ1 READ1 PRINO READ1 READ                       (2)
```

   Message (1) means right parenthesis required after dot.
   Message (2) is an automatic backtrace of the internal functions executed (from right to left) leading up to the error. The Q-32 always provides a diagnostic backtrace at an error condition.

8. Yes
9. Yes
10. Yes ((((NIL))))
11. Yes, after the missing right parenthesis is supplied.
12. Yes (AN . EXTRALONGATOMSTRING)
13. (A)
14. (NIL)
15. (A B C)
16. (A B C . D)
17. ((A B))
18. ((A) (B))
19. ((A B) (C))
20. ((X) (NIL . Y))

CHAPTER 5.

1. Yes
2. Yes (5 E)
3. Yes (E5 . 5)
4. Yes (1.0 . 1Q)
5. Yes
6. Yes 4.4000000000
7. Yes (A . 9)
8. Yes (B . 9.8999999999)
9. Yes (9.8999999999 . 9)

   Note that the use of blanks to delimit the dot would remove user confusion here. Q-32 LISP always assumes the second dot terminates a numerical field and treats the second dot as the dot for dot-notation.

10. Yes (1.2300000000 77Q3 27 2700000 3.2099999999E-8 ALPHA Q . 32)
11. (99.9000000000)
12. (NIL . 99.9000000000)
13. Not a legal S-expression as there are too many dots.
    Yields error message:

    ```
    (RPAR RQD AFTER .)
    ```

    plus a backtrace.
14. (5 5.5 5Q5 5.5 500)
15. ((13.1299999999) (25Q2))

CHAPTER 6.

1. LEFT
2. RIGHT
3. (LEFT . RIGHT)
4. A
5. (A)
6. A
7. (A . B)
8. (SENTENCE IS A LIST)
9. ((ABOUT THIS))
10. ((DOT . PAIR2))
11. (CAR . CDR)
12. NIL
13. (CDR)
14. (CAR)
15. (A)
16. (75Q . 100)
17. 1
18. (2.0 3.0 . 77Q)

19. ((A . B))
20. ((((ALPHA))))

The relationship among CONS, CAR, and CDR is that CONS puts together that which CAR and CDR tear apart. More exactly, if the argument to CAR and to CDR is the same S-expression, e.g., (LEFT . RIGHT) and the two arguments to CONS are the values of the CAR and CDR of this S-expression, then the value of CONS is the original S-expression, i.e., (CONS (CAR S)(CDR S)) = S.

21. CAR CDR CAR = CADAR
22. CAR CAR = CAAR
23. CAR CDR = CADR
24. CDR = CDR
25. CAR CDR CDR = CADDR
26. CAR CDR CDR CDR = CADDDR
27. CAR CAR = CAAR
28. CDR CAR = CDAR
29. CAR CAR CAR CDR = CAAADR
30. CAR CAR CDR = CAADR
31. CAR CDR CAR CDR = CADADR

CHAPTER 7.

1. ATOM
2. (LIST)
3. THREE
4. (ELEMENT LIST)
5. (VERY . GOOD)
6. (ONE THEN . ANOTHER)
7. B
8. (B)
9. 3.1415900000
10. 3.1415900000
11. ALPHA
12. BETA
13. BETA
14. ALPHA
15. FIRST

CHAPTER 8.

1. 43
2. LIST
3. NIL
4. 43
5. NUMBER
6. Y
7. (((LIST)))

```
  8.  B
  9.  123Q3
 10.  (A . B)
 11.  (LAMBDA (J) (CAR (CDR (CDR J)))) ( (1 2 3 4) ) = 3
 12.  (LAMBDA (X) (CAR (CAR X))) ( ((A B C) D) ) = A
 13.  (LAMBDA (Y) (CAR (CDR Y))) ( ((A B C) D) ) = D
 14.  (LAMBDA (Z) (CDR (CAR Z))) ( ((A B C) D) ) = (B C)
 15.  (LAMBDA (VARIABLE) (CAR (CDR (CAR VARIABLE)))) ( ((A B C) D) ) = B
 16.  (A C)
 17.  (A C)
 18.  (A)
 19.  (C)
 20.  ((B A) (D C))
```

CHAPTER 9.

```
  1.  X
  2.  J
  3.  (AN S EXPRESSION)
  4.  A
  5.  (J)
  6.  (QUOTE . EXPR)
  7.  (CAR (A . BETA)) = A
  8.  (NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THE PARTY)
  9.  (A . B)
 10.  (LAMBDA (X) X)
 11.  (ONE TWO THREE)
 12.  (ONE TWO THREE)
 13.  (NIL F NIL F NIL F)
 14.  (F F F F F F)
 15.  ((NIL F F) (T T T) (NIL NIL NIL) (123 123 123))
```

CHAPTER 10.

```
  1.  DEFINE (( (FIRST (LAMBDA (X) (CAR X))) ))
                FIRST ( (A B C D E) ) = A

2-4.  DEFINE ((
                (SECOND (LAMBDA (Y) (CADR Y)))
                (THIRD (LAMBDA (Z) (CAR (CDDR Z))))
                (CADDDDR (LAMBDA (J) (CAR (CDDDDR J))))
             ))
      SECOND ( (A B C D E) ) = B
      THIRD ( (A B C D E) ) = C
      CADDDDR ( (A B C D E) ) = E
```

5.
```
DEFINE (( (REVDOT (LAMBDA (J) (CONS (CDR J) (CAR J)))) ))
REVDOT ( (A . B) ) = (B . A)
REVDOT ( ((A) . (B)) ) = ((B) . (A)) = ((B) A)
REVDOT ( (((FIRST)) . (LAST)) ) = ((LAST . ((FIRST))) = ((LAST) (FIRST))
```

CHAPTER 11.

1. (T F T F)
2. T
3. T
4. NIL
5. T
6. NIL
7. T
8. NIL
9. T
10. NIL
11. NIL
12. T
13. T
14. T
15. NIL, since HEAR is not a member of list (NOW (HEAR THIS)). HEAR is a member of the sublist (HEAR THIS), but MEMBER tests only for elements at the top level of a list.
16. DEFINE (((NEGZEROP (LAMBDA (J) (AND (MINUSP J)(ZEROP J))))))
17. DEFINE (((EQUIV (LAMBDA (X Y) (EQ X Y)))))
18. DEFINE (((IMPLIES (LAMBDA (X Y)(OR (EQ X Y) Y)))))
19. This program can be easily written with conditionals and recursion. However, since the student has not learned these techniques, the following expression is required.

```
DEFINE ((
(INSEQ (LAMBDA (J)
  ((LAMBDA (V W X Y Z) (AND
    (AND (NUMBERP V)(NUMBERP W)(NUMBERP X)(NUMBERP Y)(NUMBERP Z))
    (OR (AND (LESSP V W)(LESSP W X)(LESSP X Y)(LESSP Y Z))
        (AND (GREATERP V W)(GREATERP W X)(GREATERP X Y)(GREATERP Y Z)))))
        (CAR J)(CADR J)(CADDR J)(CADDDR J)(CAR (CDDDDR J)) )))
))
```

Note the use of nested LAMBDA expressions, here, permits us to bind V to the first list element, W to the second, X to the third, etc. This practice creates temporary storage for these partial results and simplifies the total expression as well as reducing the total computation, since we need compute these repeatedly-used arguments only once.

```
20. DEFINE (( (EQN (LAMBDA (X Y)(OR (EQ X Y)
                                    (AND (NUMBERP X)
                                    (EQUALN X Y))))) ))
```

CHAPTER 12.

```
 1. True  }  Constant functions that always evaluate true regardless of input.
 2. True  }
 3. T
 4. 1
 5. (COND ERROR A3)}  Error A3 means COND is undefined, since there is no
    Plus backtrace.}  "true" clause for the conditional expression.
 6. ABVALU (144) = 144
    ABVALU (-14.4) = (MINUS . -14.399999999)
    ABVALU (-0.0) = (MINUS . -0.0)
    ABVALU (0Q) = 0Q
 7. DEFINE (( (SMALLER (LAMBDA (A B)(COND ((LESSP A B) A)(T B)))) ))
    SMALLER (15 7) = 7
    SMALLER (-15 7.02) = -15
    SMALLER (+0 -0.0) = -0.0
    SMALLER (10Q 8) = 8
 8. DEFINE (( (EQUIV (LAMBDA (X Y)(COND ((EQUAL X Y) T)(T F)))) ))
 9. DEFINE (( (EXOR (LAMBDA (X Y)(COND ((EQUAL X Y) F)(X T)(Y T)))) ))
```

CHAPTER 13.

```
 1. 55
 2. 95.858410000
 3. 1024
 4. 32768
 5. 0.0
 6. 0
 7. 10.000000000
 8. 9.9999999990
 9. 18
10. 18.333333333
11. 1 i.e., number-theoretic remainder for fixed-point arguments.
12. 9.3132257461E-10 i.e., floating-point residue for floating arguments.
13. (18 1)
14. (18.333333333 9.3132257461E-10)
15. (18 1)
16. 123
17. -123
18. 0
19. 0
20. 5.0
```

```
21. }   RECIP not available at this time.  It may be defined, however, by
22. }   DEFINE (( (RECIP (LAMBDA (X) (QUOTIENT 1.0 X))) ))
23.  1.2345678900E+8
24.  3.1415900000
25.  77777Q
26.  717375Q
27.  765435Q
28.  715335Q
29.  12345Q
30.  204Q1
31.  16Q1
32.  34Q
33.  DEFINE (( (TRIPLE (LAMBDA (X)(PLUS X X X))) ))
34.  DEFINE (( (CUBE (LAMBDA (X)(TIMES X X X))) ))
35.  DEFINE (( (SIMPLEINTEREST (LAMBDA (PRINCIPAL RATE YEARS)
       (TIMES PRINCIPAL (ADD1 (TIMES YEARS RATE))))) ))
36.  DEFINE (( (ANNUALCOMPOUND (LAMBDA (P R Y)
       (TIMES P (EXPT (ADD1 R) Y)))) ))
37.  DEFINE (( (TIMECOMPOUND (LAMBDA (P R Y T)
       (TIMES P (EXPT (ADD1 (QUOTIENT R T)) (TIMES T Y))))) ))
38.  DEFINE (( (TWOBY (LAMBDA (A11 A12 A21 A22)
       (DIFFERENCE (TIMES A11 A22) (TIMES A12 A21)))) ))
39.  DEFINE (( (THREEBY (LAMBDA (A11 A12 A13 A21 A22 A23 A31 A32 A33)
       (PLUS (TIMES A11 (TWOBY A22 A23 A32 A33))
             (MINUS (TIMES A12 (TWOBY A21 A23 A31 A33)))
             (TIMES A13 (TWOBY A21 A22 A31 A32))))) ))
40.  DEFINE (( (SOLVE (LAMBDA (A11 A12 A13 A21 A22 A23 A31 A32 A33 C1 C2 C3)
      ((LAMBDA (U1 U2 U3 D)(LIST (CONS (QUOTE U1)(QUOTIENT U1 D))
                                 (CONS (QUOTE U2)(QUOTIENT U2 D))
                                 (CONS (QUOTE U3)(QUOTIENT U3 D))))
     (THREEBY C1 A12 A13 C2 A22 A23 C3 A32 A33)
     (THREEBY A11 C1 A13 A21 C2 A23 A31 C3 A33)
     (THREEBY A11 A12 C1 A21 A22 C2 A31 A32 C3)
     (THREEBY A11 A12 A13 A21 A22 A23 A31 A32 A33)))) ))

 1.  SOLVE (2 1 -2 1 1 1 -1 -2 3 -6 2 12) = ((U1 . 1)(U2 . -2)(U3 . 3))
 2.  SOLVE (2 1 -2 2 1 3 -1 -2 3 5 6 12) = (U1 . 7)(U2 . -9)(U3 . 0))
 3.  SOLVE (15 15 15 7 1 -100 -50 1 1 15 -100 -16) = ((U1 . 0)(U2 . 0)(U3 . 1))
 4.  SOLVE (1 2 -2 1 1 1 -2 -1 3 -12 6 2) = ((U1 . 8)(U2 . -6)(U3 . 4))
 5.  SOLVE (-2 2 1 1 1 1 3 -1 -2 -24 49 9) = ((U1 . 22)(U2 . -5)(U3 . 32))
```

CHAPTER 14.

```
 3.  A
     B
     Z
     NIL
     NIL
 4.  X
     E
     NO
     L
 5.  DEFINE (( (TWIST (LAMBDA (S)(COND ((ATOM S) S)
                                       (T (CONS (TWIST (CDR S))
                                                (TWIST (CAR S)))))) ))
     A
     (B . A)
     (C . (B . A)) = (C B . A)
     (((NIL . C) . B) . A)
     (NIL . (B . A)) = (NIL B . A)
 6.  DEFINE (( (SUM (LAMBDA (X Y)(COND ((ZEROP Y) X)(T (SUM (ADD1 X)(SUB1 Y)))))) ))
     ARGS OF SUM
     1
     2
     VALUE OF SUM
     3
 7.  DEFINE (( (PROD (LAMBDA (X Y)(COND ((ZEROP Y) 0)
                                        (T (SUM X (PROD X (SUB1 Y))))))) ))
 8.  DEFINE (( (REMXY (LAMBDA (X Y)(COND ((LESSP X Y) X)
                                         ((EQUAL X Y) 0)
                                         (T (REMXY (DIFFERENCE X Y) Y))))) ))
 9.  DEFINE (( (GCD (LAMBDA (X Y)(COND ((GREATERP X Y) (GCD Y X))
                                       ((ZEROP (REMAINDER Y X)) X)
                                       (T (GCD X (REMAINDER Y X)))))) ))
10.  DEFINE (( (AMONG (LAMBDA (A L)(COND ((NULL L) NIL)
                                         ((EQ A (CAR L)) T)
                                         (T (AMONG A (CDR L)))))) ))
11.  DEFINE (( (INSIDE (LAMBDA (A E)(COND ((ATOM E) (EQ A E))
                                          ((INSIDE A (CAR E)) T)
                                          (T (INSIDE A (CDR E)))))) ))
12.  DEFINE (( (COPYN (LAMBDA (X N)(COND ((ZEROP N) NIL)
                                         (T (CONS X (COPYN X (SUB1 N))))))) ))
13.  DEFINE (( (LENGTHS (LAMBDA (L)(COND ((NULL L) 0)
                                         (T (ADD1 (LENGTHS (CDR L))))))) ))
14.  DEFINE (( (UNIONS (LAMBDA (X Y)(COND ((NULL X) Y)
                                          ((MEMBER (CAR X) Y)(UNIONS (CDR X) Y))
                                          (T (CONS (CAR X)(UNIONS (CDR X) Y)))))) ))
15.  DEFINE (( (INTERSECT (LAMBDA (X Y)(COND ((NULL X) NIL)
                                             ((MEMBER (CAR X) Y)(CONS (CAR X)
                                                      (INTERSECT (CDR X) Y)))
                                             (T (INTERSECT (CDR X) Y))))) ))
```

```
16.  DEFINE (( (REVERSAL (LAMBDA (L)(COND ((NULL L) NIL)
                                         (T (APPEND (REVERSAL (CDR L))
                                                    (LIST (CAR L))))))) ))
17.  DEFINE (( (PAIRS (LAMBDA (L1 L2)(COND ((NULL L1) NIL)
                                          (T (CONS (CONS (CAR L1)(CAR L2))
                                                   (PAIRS (CDR L1)(CDR L2))))))) ))
18.  DEFINE (( (DELETE (LAMBDA (A L)(COND ((NULL L) NIL)
                                         ((EQ A (CAR L)) (CDR L))
                                         (T (CONS (CAR L) (DELETE A (CDR L))))))) ))
19.  DEFINE ((
          (INSEQ (LAMBDA (L) (OR (INSEQA L)(INSEQA (REVERSE L)))))
          (INSEQA(LAMBDA (L)(COND ((NULL L) T)
                                  ((NULL (CDR L)) T)
                                  ((NOT (NUMBERP (CAR L))) NIL)
                                  ((NOT (NUMBERP (CADR L))) NIL)
                                  ((LESSP (CAR L)(CADR L)) (INSEQA (CDR L)))
                                  (T NIL))))
     ))
20.  DEFINE (( (REPLACE (LAMBDA (A B X)(COND ((ATOM X) (COND ((EQ B X) A)(T X)))
                                           (T (CONS (REPLACE A B (CAR X))
                                                    (REPLACE A B (CDR X))))))) ))
```

CHAPTER 15.

```
 1.  DEFINE (( (NEGCNT (LAMBDA (L)(PROG (X)
               (SETQ X 0)
       TAG1  (COND ((NULL L) (RETURN X))
                   ((MINUSP (CAR L))(SETQ X (ADD1 X))))
                   (SETQ L (CDR L))
                   (GO TAG1)))) ))
 2.  DEFINE (( (CURVE (LAMBDA (A B C)(PROG (X)
               (SETQ X (PLUS (TIMES B B) (TIMES -4 A C)))
               (COND ((ZEROP X) (RETURN (QUOTE PARABOLA)))
                     ((LESSP X 0) (RETURN (QUOTE ELLIPSE))))
               (RETURN (QUOTE HYPERBOLA))))) ))
 3.  DEFINE (( (LENGTHS (LAMBDA (M)(PROG (X)
               (SETQ X 0)
         LOC1  (COND ((NULL M)(RETURN X)))
               (SETQ X (ADD1 X))
               (SETQ M (CDR M))
               (GO LOC1)))) ))
 4.  DEFINE (( (LAST (LAMBDA (L)(PROG (U)
           T1  (COND ((NULL L) (RETURN U)))
                     (SETQ U (CAR L))
                     (SETQ L (CDR L))
                     (GO T1)))) ))
```

```
5-7.  DEFINE ((
                     (REVERSAL (LAMBDA (L)(PROG (Y)
                T2   (COND ((NULL L)(RETURN Y)))
                     (SETQ Y (CONS (CAR L) Y))
                     (SETQ L (CDR L))
                     (GO T2))))

                     (PAIRS (LAMBDA (L1 L2)(PROG (X)
                T3   (COND ((NULL L1)(RETURN (REVERSE X))))
                     (SETQ X (CONS (CONS (CAR L1)(CAR L2)) X))
                     (SETQ L1 (CDR L1))
                     (SETQ L2 (CDR L2))
                     (GO T3))))

                     (DELETE (LAMBDA (A L)(PROG (Z)
                T4   (COND ((NULL L)(RETURN (REVERSE Z)))
                           ((EQ A (CAR L))(GO T5)))
                     (SETQ Z (CONS (CAR L) Z))
                T5   (SETQ L (CDR L))
                     (GO T4))))
                  ))
  8.  DEFINE (( (PERMUT (LAMBDA (N R)
                (QUOTIENT (FACTORIAL N)(FACTORIAL (DIFFERENCE N R))))) ))

      DEFINE (( (PERMUT (LAMBDA (N R)(PROG ()
                (RETURN (QUOTIENT (FACTORIAL N)(FACTORIAL (DIFFERENCE N R))))))) ))
  9.  DEFINE (( (COMBIN (LAMBDA (N R)
                (QUOTIENT (FACTORIAL N)(TIMES (FACTORIAL R)
                                              (FACTORIAL (DIFFERENCE N R)))))) ))

      DEFINE (( (COMBIN (LAMBDA (N R)(PROG ()
                (RETURN (QUOTIENT (FACTORIAL N)
                                  (TIMES (FACTORIAL R)
                                  (FACTORIAL (DIFFERENCE N R)))))))) ))
```

```
10.  DEFINE (( (PASCAL (LAMBDA (N)(PROG (X R LINE)
                (SETQ X 0)
      OUTLOOP   (SETQ R 0)
                (COND ((LESSP N X)(RETURN NIL)))
       INLOOP   (COND ((LESSP X R)(GO BUMPX)))
                (SETQ LINE (CONS (COMBIN X R) LINE))
                (SETQ R (ADD1 R))
                (GO INLOOP)
        BUMPX   (PRINT LINE)
                (SETQ LINE NIL)
                (SETQ X (ADD1 X))
                (GO OUTLOOP)))) ))
```

```
 PASCAL(15)
  (1)
  (1 1)
  (1 2 1)
  (1 3 3 1)
  (1 4 6 4 1)
  (1 5 10 10 5 1)
  (1 6 15 20 15 6 1)
  (1 7 21 35 35 21 7 1)
  (1 8 28 56 70 56 28 8 1 )
  (1 9 36 84 126 126 84 36 9 1)
  (1 10 45 120 210 252 210 120 45 10 1)
  (1 11 55 165 330 462 462 330 165 55 11 1)
  (1 12 66 220 495 792 924 792 495 220 66 12 1)
  (1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1)
  (1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1)
  (1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1)
  NIL
```

NOTE: PASCAL (16) is the largest triangle possible with this definition since 16! is maximum fixed-point accuracy of Q-32.

CHAPTER 16.

1.

Bound Variables			
Dummy Variables			
LAMBDA Variables	PROG Variables	SPECIAL Variables	Free Variables
X	A	PI	PI
Y	B		N
Z			M

2. DEFINE gives the following output:

```
(N NOT DECLARED)
(M NOT DECLARED)
(TEST)
```

Though PI is also a free variable, CSET automatically declared PI SPECIAL.

3. TEST (1 2 3) = (NIL NIL 2 LA777777 3 3.1415900000 (2 1 LA777777))

- The PROG variables A and B are always initialized to NIL, and, not being changed, evaluate to NIL.
- The top-level LAMBDA variables, X, Y, and Z, are bound to 1, 2, and 3, respectively.
- The nested LAMBDA variable, Z, does not conflict with the top-level LAMBDA variable, Z, as the nested LAMBDA variable, Z, has its own pushdown list location allocated. This Z is bound to the value of Y, the argument of the nested LAMBDA expression, which is 2.
- PI evaluates to 3.14159.
- M and N are unbound free variables, a potential error source. (Never try to SETQ an unbound free variable.) Q-32 LISP evaluates M and N, but finding no binding and hence no Print Name or S-expression, prints an LA symbol. An LA symbol is an error message which freely translated means, "I do not recognize this entity as a meaningful LISP expression so I will print its address prefixed by the letters LA, for LISP address."

4. There are no error messages and

    TEST (1 2 3)

here evaluates exactly as in problem 3. It would appear to the student that, except for error messages, one needn't declare free variables SPECIAL. In subsequent chapters we examine functions which take other functions, including LAMBDA expressions, as arguments. Expressions that use such functional arguments will evaluate differently when free variables are SPECIAL than when free variables are not SPECIAL. Therefore, unless you understand what you are doing, all free variables should be declared SPECIAL before they are used.

```
 5. 1. K
    2. 1965
    3. K
    4. K
    5. 1966
 6. 1. (V1 V2)
    2. V1
    3. V2
    4. (V1 V2)
    5. (V2 V1)
 7. 1. PI
    2. 3.1415900000
    3. 5 PI is bound temporarily.
    4. 3.1415900000 PI is restored to its permanent binding.
 8. 1. PI
    2. 3.1415900000
    3. 5 PI is bound temporarily.
    4. 3.1415900000
 9. PI
    (BEFORESETQ NIL) All PROG variables are initially bound to NIL.
    (AFTERSETQ 1234) PI is bound temporarily.
    (PERMANENTVALUE 3.1415900000) PI is restored to its permanent binding.
10. 1. PI
    2. 54321
    3. 54321 SETQ makes permanent bindings as CSETQ if the first argument
             of SETQ is a free and also a SPECIAL variable.
11. 1. ABE
    2. FREEDOM Though X is bound to ABE by LAMBDA conversion, SETQ quotes
               its first argument X and temporarily binds FREEDOM to X.
    3. LINCOLN The permanent binding of ABE remains unchanged.
    4. CIVILWAR ABE is temporarily bound to BOOTH by LAMBDA conversion;
                however, as in (2) SETQ quotes its first argument ABE and
                temporarily binds CIVILWAR to ABE.
    5. LINCOLN The permanent binding of ABE remains unchanged.
12. 1. PI PI is permanently bound to 5.
    2. 50
    3. 31.415900000 PI is permanently bound to 3.14159 by SETQ since SETQ
                    makes permanent bindings to SPECIAL variables used free.
    4. 3.1415900000 This evaluation verifies that permanent binding of PI
                    is 3.14159.
13. 1. (PI ABE)
    2. (PI ABE) SPECIAL and UNSPECIAL may be used at other than the top level
                as all other LISP functions.
```

14. 1. ABE
    2. JOHN ABE and JOHN made SPECIAL and permanently bound to LINCOLN and BOOTH, respectively.
    3. (LINCOLN BOOTH) Carrying problem 13 one step further, ABE and JOHN are evaluated to LINCOLN and BOOTH, respectively, and these values are made SPECIAL.

15. 1. ABE
    2. JOHN ABE and JOHN made SPECIAL and permanently bound to LINCOLN and BOOTH, respectively.
    3. ((ABE LINCOLN PRESIDENT)(JOHN BOOTH ACTOR))
       ABE and JOHN temporarily bound to LAMBDA variables X and Y, respectively, by LAMBDA conversion CSET evaluates its arguments and permanently binds the value of its second argument to the value of its first argument. Hence, PRESIDENT is permanently bound to the value of ABE, namely LINCOLN, and ACTOR is permanently bound to the value of JOHN, namely BOOTH. The answer is the list of the values of each of these bindings.
    4. ((ABE LINCOLN PRESIDENT)(JOHN BOOTH ACTOR)) This evaluation verifies these permanent bindings.

CHAPTER 17.

```
 1.  (LIST)
     (LIST)
 2.    (cr)
     NIL
       (cr)
     NIL
 3.  ATOM1   ATOM2
     NIL
 4.  (NOW HEAR THIS)
 5.  ((INPUT) ANYTHING)
 6.  For R=5   , 31.415900000
     For R=50  , 314.15900000
     For R=10  , 62.83180000
     For END   , END
 7.  (( )   . / = $ * NOW HEAR THIS  -533.17 )
 8.  (B C)
 9.  1.  PERCENT  This expression binds the BCD for % to the name PERCENT.
                  The $$ artifact is the only way to enter illegal read
                  characters.  All Teletype characters but line feed and
                  carriage return can be entered this way.
     2.  %
     3.  Yields error message  IMPROPER CHARACTER READ  as % is still an
         illegal read character.
10.  A
```

```
11.  1.  A
     2.  (B C)
     3.  (A . B)
     4.  PI
     5.  3.1415900000
12.  1.  (A)
     2.  (A . B)
     3.  K
     4.  (3.1415900000)
     5.  X      SQUARE

         0      0
         1      1
         2      4
         3      9
         4      16
         5      25
         6      36
         7      49
         8      64
         9      81
         END
13.  DEFINE ((
     (SUP4 (LAMBDA () (PROG (X Y)
      TAG1 (TEREAD)
           (SETQ X (READ))
           (SETQ Y (READ))
           (PRINT (EVALQT Y X))
           (GO TAG1)))) ))
     1.  A
     2.  (B C)
     3.  NIL
     4.  10
     5.  K
     6.  3.1415900000
14.  DEFINE ((
       (SUP5 (LAMBDA () (PROG (X Y)
        TAG1 (TEREAD)
             (SETQ X (READ))
             (SETQ Y (READ))
             (PRINO X)(PRINO Y)(TERPRI)
             (COND ((EQ (QUOTE NO)(READ))(GO TAG1)))
             (PRINO X)(PRINO Y)(BLANKS 1)(PRIN1 EQSIGN)(BLANKS 1)
             (PRINO (EVALQT X Y))
             (TERPRI)
             (GO TAG1)))) ))
```

```
15.  DEFINE (( (PI (LAMBDA (X)(PROG (HI)
                    (PRINT (QUOTE (ENTER MAX X)))
                    (SETQ HI (READ))
                    (SETQ X (TIMES X 1.0))
                    (PRINT (QUOTE $$$
        X           XSQUARE          SQRTX          RECIPX           FACTORIALX
     $))
     TAG1 (COND ((LESSP HI X)(RETURN (QUOTE $$$LIMIT REACHED$))))
          (PRIN1 X)(BLANKS 10)(PRIN1 (TIMES X X))(BLANKS 10)(PRIN1 (SQRT X))(BLANKS 3)
          (PRIN1 (QUOTIENT 1.0 X)) (BLANKS 3)(PRIN1 (FACTORIAL X))(TERPRI)
          (SETQ X (ADD1 X))
          (GO TAG1)))) ))
```

CHAPTER 18.

```
 1.  MACRO (( (PROD (LAMBDA (J)(*EXPAND J (QUPTE *TIMES)))) ))
 2.  MACRO ((
     (MAXIMUM (LAMBDA (J)(*EXPAND J (QUOTE *MAX))))
     (MINIMUM (LAMBDA (J)(*EXPAND J (QUOTE *MIN))))
             ))
 3.  MACRO (( (FLAMBDA (LAMBDA (J)
              (LIST (QUOTE FUNCTION)
                    (CONS (QUOTE LAMBDA)(CDR J))))) ))
 4.  MACRO ((
     (LIST2 (LAMBDA (J)(*EXPAND J(QUOTE CONS))))
     (LIST1 (LAMBDA (J)(APPEND (CONS (QUOTE LIST2)
                                     (CDR J))
                               (QUOTE (NIL)))))
           ))
```

We note here that given a form

$$(\text{LIST1 } x_1\ x_2\ x_3) \qquad (1)$$

the macro LIST2 expands form (1) to

$$(\text{LIST2 } x_1\ x_2\ x_3 \text{ NIL}) \qquad (2)$$

and with repeated application to

$$(\text{CONS } x_1\ (\text{CONS } x_2\ (\text{CONS } x_3 \text{ NIL})))$$

Thus the sole purpose of macro LIST1 is to insert NIL as the last argument of the form. If we used the macro definition for LIST2 only, we would get a value of

$$(x_1\ x_2\ .\ x_3)$$

rather than

$$(x_1\ x_2\ x_3)$$

the list we desire.

5.
```
DEFINE ((
  (PRINTQ1 (LAMBDA (J)(PROG (X Y)
     (SETQ X J)
  T1 (COND ((NULL X)(RETURN (TERPRI))))
     (PRINO (CAR X))(BLANKS 1)
     (SETQ X (CDR X))
     (GO T1)))) ))

MACRO ((
  (PRINTQ (LAMBDA (J)(LIST(QUOTE PRINTQ1)
                        (LIST (QUOTE QUOTE)
                              (CDR J))))) ))
```

The form

(PRINTQ NOW HEAR THIS)

after the macro PRINTQ has been executed, will be replaced by the form

(PRINTQ1 (QUOTE (NOW HEAR THIS)))

The function PRINTQ1 enters each element of its argument list into the print line with PRINO, and executes a final TERPRI when the list is empty.

CHAPTER 19.

```
1. (TRY THIS SIMPLE CASE FIRST)
   (THIS SIMPLE CASE FIRST)
   (SIMPLE CASE FIRST)
   (CASE FIRST)
   (FIRST)
   NIL
2. (NOW THIS ONE)
   (THIS ONE)
   (ONE)
   ((NOW THIS ONE)(THIS ONE)(ONE))
3. AND
   LASTLY
   THIS
   ONE
   (AND LASTLY THIS ONE)
4. ((ONE 2 3 ONE 4 ONE 5)(2 3 ONE 4 ONE 5)(3 ONE 4 ONE 5)(ONE 4 ONE 5)
    (4 ONE 5)(ONE 5)(5))
5. ((1 2 3 4)(2 3 4)(3 4)(4))
6. ((A . A)(B . B)(C . C)(D . D)(E . E))
7. ((A . X)(B . X)(C . X)(D . X)(E . X))
8. ((A . Z)(B . Z)(C . Z)(D . Z)(E . Z))
   ((A 1 2 3 4 5)(B 1 2 3 4 5)(C 1 2 3 4 5)(D 1 2 3 4 5)(E 1 2 3 4 5))
```

```
 9.  DEFINE ((
       (MAPCAR2 (LAMBDA (X Y FN)
           (COND ((NULL X) NIL)
                 (T (CONS (FN (CAR X)(CAR Y))(MAPCAR2 (CDR X)(CDR Y) FN)))))) ))
10.  DEFINE ((
       (TYPE (LAMBDA (J)(MAPCAR J (FUNCTION
         (LAMBDA (K)(COND ((NUMBERP K)(RETURN (COND ((FIXP K)(QUOTE FIX))
                                                    (T (QUOTE FLT)))))
                          ((ATOM K)(QUOTE ATOM))
                          ((EQ (ATOM (CAR K))(ATOM (CDR K)))(QUOTE DOTPAIR))
                          (T (QUOTE LIST)))))))) ))
```

CHAPTER 20.

```
 1.  A
 2.  B
 3.  A
 4.  NIL
 5.  T
 6.  (AB . C)
 7.  B
 8.  A
 9.  A
10.  B
11.  ((AB . C))
12.  T
13.  AB
14.  C
15.  (((F . T) . AB) . ((AB))) = (((F . T) . AB) (AB))
16.  A
17.  A
18.  B
19.  (C . B)
20.  (A . C)
```

APPENDIX B

INDEX*

*In almost all cases, this index notes in order (most important references first) the principle pages in which the subject is discussed. No reference is listed for the first mention of a subject merely as an introduction to its later principle presentation.

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* System Development Corporation, Santa Monica, California	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP

3. REPORT TITLE

LISP PRINTER - A SELF-TUTOR FOR Q-32 LISP 1.5.

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(Last name, first name, initial)*

Weissman, C.

6. REPORT DATE 14 June 1965	7a. TOTAL NO. OF PAGES 166	7b. NO. OF REFS
8a. CONTRACT OR GRANT NO. SD-97	9a. ORIGINATOR'S REPORT NUMBER(S) TM-2337/010/00	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	
d.		

10. AVAILABILITY/LIMITATION NOTICES

This document has been cleared for open publication and may be disseminated by the Clearing House for Federal Scientific & Technical Information.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

This document is a self-tutor for LISP 1.5 programming, particularly for on-line Q-32 LISP 1.5. Material is organized into chapters that, by discussion and examples, progressively expand the student's understanding of the language and ability to write programs in the language. A carefully selected and graduated set of exercises for use on-line is provided as an integral part of each chapter. Computer-checked answers for each exercise are also provided as a separate appendix. The document is not an exhaustive treatise on LISP 1.5, but, rather, a practical primer that provides the serious student with a solid foundation for understanding the programming language and system. He may then easily supplement his knowledge from other sources suggested herein. (author)

DD FORM 1 JAN 64 1473

Unclassified
Security Classification

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
LISP Self-Tutor Programming Language Computers AN/FSQ-32						

Unclassified
Security Classification

Corrections for TM-2337/010/00

LISP Primer

A Self-Tutor for Q-32 LISP 1.5

Page	Line	Correction
9	19	...by Timothy P. Hart and Thomas Evans for the M-460
21	6	((((A . B) . (A . B)) . (A . B)) . (A . B))
100	Example 20, line 4	...of b in S-expression x.
117	Example 11, line 4	4. (LAMBDA (ABE)(SETQ ABE (QUOTE CIVILWAR)))(BOOTH)
120	25	(LAMBDA () SLASH) = /
121	6	$$*ATOM* ATOM *
	10	In example 3 above, $$*ATOM* is internally...
	11	Thus, bindings for $$*ATOM* are bindings...
	12	CSETQ (($$*ATOM* 123) = ATOM
139	Insert after line 22	. PROG statement labels are capital letters
140	23	expression, except numbers, statement labels, NIL, T, and F, e.g.,
144	7	7. (NIL X1)
	13	13. (A (B) B) = (A . ((B . NIL) . (B . NIL)))
152	8	((EQ A (CAR L))(DELETE A (CDR L)))