This document was produced by SDC and III in performance of contract AF 19(628)-5166 with the Electronic Systems Division, Air Force Systems Command, in performance of ARPA Order 773 for the Advanced Research Orijects Agency Information Processing Techniques Office, and Subcontract 65-107,

TECH MEMO

a working paper

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 200 Sixth Street / Cambridge, Massachusetts 02142

TM- 2710/101/00(DRAFT)

AUTHOR M. Levin

E. Berkeley

TECHNICAL Clark Weissman

RELEASE S. L. Kameny

L. Hawkinson

for J. I. Schwartz

DATE 15 July 1966 PAGE 1 OF 165 PAGES

(Page 2 blank)

LISP 2 PRIMER

DRAFT

Abstract

This document is an incomplete preliminary draft being circulated to provide potential users with information concerning the programming language LISP 2 currently being developed by System Development Corporation and Information International, Incorporated.

SDC and III accept no responsibility for the technical correctness of the information contained herein, and its circulation is permitted at this time for the sole purpose of meeting requests for information on LISP 2.

DRAFT

DRAFT

CONTENTS

CHAPTER 1

INTRODUCTION

The purpose of this LISP 2 Primer is to provide an understanding of the main features of the Programming language LISP 2.

The Primer is one of the two main sources of information on LISP 2; the other is the LISP 2 Reference Manual. These two books serve different purposes in making information about LISP 2 available to the interested reader and prospective programmer.

The Reference Manual is intended to be a full description of the language. It contains a complete and concise definition of each aspect of the language, and its arrangement is systematic; significant details are not omitted.

This makes the Reference Manual difficult to read through, especially for a potential user who is not familiar with other LISP systems, or with computer programming in general. Also, the Reference Manual contains many cross-references, and many explanations that seem unmotivated until some other explanation is read elsewhere. The Reference Manual is much easier to understand if one first acquires some understanding of the main features of LISP 2.

The Primer is intended to give an understanding of the main features of LISP 2. Unlike the Reference Manual, the Primer is intended to be read from beginning to end in the order in which it is written. The Primer makes only a few assumptions about what the reader already knows--mainly, a little mathematics, all of which is taught in high school. If in addition one has calculus or logic, some of the examples will appear more interesting, but neither subject is necessary.

In describing the LISP 2 source language, all non-primitive syntactic entities are written in italics. If the entity is composed of more than one word, the words are joined by italicized colons. For example, the terms *identifier* and *block:expression* are non-primitive syntactic entities, and thus are italicized.

The Primer seeks to present LISP 2 in such a way that reasons for introducing new concepts are made clear, and the programmer's knowledge of LISP 2 techniques is developed gradually. This way of explaining is inconsistent with the method of arranging subject matter into a logical classification of topics and subtopics, and then explaining each topic fully before proceeding to the next. Therefore, you, the reader, should be aware that while each explanation in the Primer is correct, it is rarely complete, and usually there are possibilities that have not been mentioned. Also, many topics have been omitted from the Primer altogether, and their explanations can be found only be consulting the Reference Manual.

For example, one of the first LISP 2 concepts discussed in the Primer is *identifier*, and examples of *identifiers* are given. But nowhere in the Primer appears any explanation which would suggest that the entities A.B. and %#(((# are acceptable *identifiers*. For a complete definition of *identifier*, therefore, see the Reference Manual.

The LISP language is founded on mathematical logic, and, in particular, on a part of logic known as recursive function theory. However, the theoretical concepts needed are not difficult or advanced, and are presented completely in the Primer. It is recommended that you understand the ideas presented in Chapter 2 before reading further in the Primer. It is also recommended that you solve the exercises in each chapter, obtaining the correct answers, before reading further in the Primer.

Finally, it is recommended that as you read this Primer, you keep in mind the types of data that will occur in the problems you want to handle, and the types of processes you wish to perform on the data. Then you should be able to decide whether a given capability in LISP 2 is relevant to your problem or not. It is hoped that some of the examples may suggest possibilities to you.

CHAPTER 2.

IDENTIFIERS, ATOMS, AND *S-EXPRESSIONS*

If you are familiar with LISP, you may skip this chapter except for noting that: (1) the definition of an *atom* is broad; (2) an *identifier* is a type of *atom* but not all *atoms* are *identifiers*; (3) the *booleans* TRUE and FALSE are *atoms* but not *identifiers*, and (4) the *predicate* ATOM is true for all types of *atoms*.

2.1 SYMBOLIC DATA PROCESSING

The data that are processed by a computer programming language can be classified into two broad divisions, numerical and symbolic. An example of a numerical (or numeric) datum is:

    2.5

An example of a symbolic datum is:

    (THIS IS A LIST)

The processing of numerical data is a well-established science. Basic operations on numbers, such as addition, multiplication, and comparison of two numbers to see which is greater, are taught in elementary school. The solving of many kinds of equations, and many useful applications of numerical processing are taught in high school. The science of dealing with numbers is presented in a logically rigorous manner in college courses.

The processing of symbolic data, however, is not a well-established science. In fact, the processing of symbolic data has only begun to be a science; and the development of this science has been called forth by the advance of computer programming. Among the computer programming languages, LISP is one of the few in which the processing of symbolic data is treated just as systematically and scientifically as the processing of numerical data is treated in all computer languages.

For symbolic data processing, just as for numeric data processing, there is a basic set of skills and a mathematical theory. These skills and theory take a particular form in the LISP system for symbolic data processing. The mathematical theory is beyond the scope of this Primer but is briefly summarized in an appendix to the Reference Manual. The basic skills of symbolic data processing could easily be taught in elementary school; but nowadays, of course, they are not. It is the purpose of this chapter of the Primer to present them.

2.2 *IDENTIFIERS*

In dealing with symbolic processing, we recognize certain sequences of characters called *identifiers*. *Identifiers* have the following properties:

- *Identifiers* are the basic units of symbolic data (i.e., *identifiers* are the words of the language).
- *Identifiers* are composed of sequences of *signs*, the elements of the LISP alphabet. *Sign* means a *letter*, a *numeral*, or a *mark*. *Letter* means one of the 26 letters of the English alphabet, written in the form of a Roman capital (A, B ... Z). *Numeral* means one of the ten Arabic numerals (0, 1 ... 9). *Mark* means one mark, each associated with a name or names in the following list:

+	*plus:sign*
-	*minus:sign*
	space, blank
.	*period, decimal:point, dot, LISP:dot, dot:operator*
,	*comma*
=	*equals:sign*
(	*left:parenthesis*
)	*right:parenthesis*
'	*quote, apostrophe*
#	*fence*

:	*colon*
;	*semi:colon*
←	*left:arrow*
↑	*up:arrow*
*	*asterisk*
<	*less:than:sign*
>	*greater:than:sign*
/	*slash*
\	*reverse:slash*

- Out of *identifiers* we may make more and more complicated units of symbolic data.
- An *identifier* is spelled in the same way (made up of the same *signs* on each occurrence.)
- *Identifiers* that are not spelled the same way have no necessary or intrinsic relation to each other. Thus, for example, as *identifiers*, ABC and ABCX are as unrelated as ABC and RQ.

There are a number of ways to compose acceptable *identifiers* in LISP, so that we can name what we want to talk about. All these ways, however, are limited by the fact that we have to use the equivalent of a typewriter key not only to compose *identifiers* but also for all other *signs* in LISP.

So there are rules for constructing *identifiers*. These are the rules (although these are not all the rules,)nevertheless, at the start they are a sufficient set.

- A sequence of *signs* that satisfies the following three rules is an *identifier*.
- The only *signs* that may be in the sequence are *letters* and Arabic *numerals*.

A *space* is not acceptable as a *sign* in an *identifier*; thus T H E is not acceptable, and the intended *identifier* must instead be written THE.

- The first *sign* of the sequence is a *letter*.
- The sequence is not broken up in any way, such as by the insertion of *spaces* or hyphens or punctuation marks or by printing or writing on two different lines.

Under these rules we can see that the following are acceptable examples of *identifiers*:

A
ITEM16
T222
XYZ
ABC
CHICAGO

The following are not acceptable *identifiers*:

LOS-ANGELES	The *minus:sign* or hyphen is not allowed in an *identifier*.
LOS ANGELES	The *space* prevents this sequence from being a single *identifier*.
5ABC	The first *sign* may not be a *numeral*.
X Y Z	This is not one *identifier*. It could be considered as three *identifiers*.

Identifiers are used in many ways in LISP. The most important use of an *identifier* is as a name for something. *Identifiers* are used to name many different types of entities; just how, is made clear in succeeding chapters.

2.3 *ATOMS*

One of the *expressions* that is acceptable in LISP 2 is called *atom*. The definition of *atom* is introduced gradually. At this point we can say:

Every *identifier* is an *atom*.

Intuitively, an *atom* in LISP is something like a word in language; an *atom* like a word, is made up of acceptable *signs* in acceptable ways, and it is treated as a basic unit of discourse. In this chapter, most examples of *atoms* are *identifiers*. In addition, any statement made in this chapter about *atoms* is true for all kinds of *atoms*.

2.4 *S-EXPRESSIONS*

The most general type of datum in LISP 2 is the *S-expression*. The term is derived from "symbolic expression", but *S-expression* has a specific technical meaning. *S-expressions* are the most important kind of datum in LISP, and they are the main subject of this chapter.

We can define *S-expression* quite simply in terms of *atom* and a *mark* which is called the *LISP:dot* and is written as a *period* with a *space* on each side. The following rules apply:

Rule 1: Every *atom* is an *S-expression*.

Rule 2: If x and y stand for *S-expressions*, then (x . y) isan *S-expression*.

In the *expression* (x . y) the *period* is called the *dot:operator* or the *LISP:dot*.

This is an example of what is known to mathematicians as an inductive definition. The way in which it works is illustrated by the following example, in which we show that (M2 . (X . M2)) is an *S-expression*.

- X is an *identifier*. Therefore, it is an *atom*. Therefore, by Rule 1 it is an *S-expression*.
- M2 is an *S-expression* by the same reasoning.
- Since both X and M2 are *S-expressions*, it follows by Rule 2 that (X . M2) is an *S-expression*.
- Since both M2 and (X . M2) are *S-expressions*, it follows by Rule 2 that (M2 . (X . M2)) is an *S-expression*

One simple detail needs to be stressed here. The *period* (.) is used in several different ways in LISP 2. When it is used as in Rule 2 above to combine *S-expressions*, it is always written with a *space* before it, and a *space* after it. Failure to do this may result in an incorrect *S-expression*.

Examples of *S-expressions*:

```
A
(A . B)
((A . B) . (C . D))
(NEWYORK . (KANSASCITY . SANFRANCISCO))
(A . (B . (C . D)))
(((A . B) . C) . D)
```

The last two examples are different *S-expressions* because the *parentheses* occur in a different pattern.

Some examples of entities that are not *S-expressions* follow, together with their explanation:

A . B	Without *parentheses*, this is not an *S-expression*.
(A . B . C)	If an *S-expression* is to contain three *S-expressions* with two dots, then two of the *S-expressions* and the *dot* between them must be enclosed in another set of *parentheses*: thus. ((A . B) . C) or (A . (B . C)) are acceptable.
(A . B))	The number of *left:parentheses* must be equal to the number of *right:parentheses*.

Problem Set 1:

Which of the following are *S-expressions*?

a. UVW
b. (A . B . C)
c. (A . BC)
d. (((A . B) . C) . E) . (F . (G . H)))
e. ((A . B) . (C . D) . (E . F))
f. ((X))))

Answers: See page 138

2.5 *FUNCTIONS*

We may have functions in algebra, so we may have *functions* in LISP. An example of a function in algebra and a *function* in LISP is subtraction. The operation of subtraction in algebra is such that given any two numbers A and B, a third number C is produced which is the result of subtracting B from A. The operation of any *function* in LISP is such that given one or more data which are called the *arguments* of the *function*, another datum is produced which is the result of the operation of the *function* on the *arguments*. This result is called the value of the *function*.

In LISP the *arguments* and *value* of a *function* may be *numbers* or *atoms* or *S-expressions*, etc., or any mixture of them, as for example a *function* which operates on an *S-expression* and tells the number of *atoms* in that *S-expression*.

It is a common convention in mathematics to write the *arguments* of a *function* with *parentheses* around the group of *arguments* and *commas* to separate them. Thus, we could write in LISP:

```
DIFFERENCE (A, B)
```

If the *identifier* DIFFERENCE had been appropriately defined, this would mean in LISP the result of A minus B.

It is possible in LISP for a *function* to have no *arguments*. Suppose FN is such a *function*. Then the fact that FN has no *arguments* may be indicated in LISP by writing nothing at all between the *left:parenthesis* and the *right:parenthesis*, thus:

```
FN ( )
```

2.6 *QUOTE*

In LISP, when an *S-expression* is used as the *argument* of a *function*, it is preceded by a *quote* (an *apostrophe*).

For example:

FN ('A)	The *S-expression* A is the *argument* of FN.
FN ('(C . R))	The *S-expression* (C . R) is the *argument* of FN.

The reasons for this procedure are given in Chapter 5; here in Chapter 2, this procedure has no consequences that create difficulties.

2.7 THE *FUNCTION* CONS

As was said earlier, if x and y stand for two *S-expressions*, then (x . y) is an *S-expression*, where the dot is the *LISP:dot*. CONS is a *function* of two *arguments* such that if its two *arguments* are x and y, then its *value* is (x . y).

For example:

CONS('A, 'B) is (A . B)

For another example:

CONS('A, '(B . C)) is (A . (B . C))

Note that the outer pair of *parentheses* following CONS delimits the *arguments* of CONS, while the inner pair of *parentheses* are essential parts of the *S-expression* (B . C), the result of CONS operating on 'B and 'C. This example may be read aloud as follows:

The value of CONS of *quote* A *comma quote* B *dot* C is A *dot* (pause) B *dot* C.

Here are more examples of the operation of CONS:

```
CONS('(A . B),'(ORANGE . VIOLET)) is ((A . B) . (ORANGE . VIOLET))
CONS('X1, CONS('X2, CONS('X3,'X4))) is (X1 . (X2 . (X3 . X4)))
CONS(CONS(CONS('X1,'X2),'X3),'X4) is (((X1 . X2) . X3) . X4)
```

Problem Set 2:

Evaluate each of these *expressions*.

a. CONS('WINE, 'CHEESE)

b. CONS('TUOLUMNE, CONS('SANJOAQUIN,'KINGS))

c. CONS('(A . B),'(C . D))

d. CONS(CONS('A,'B), CONS('C'D))

e. CONS('(A . B), CONS('C,'D))

Answers: See pages 138, 139

2.8 THE *FUNCTIONS* CAR AND CDR

Whereas CONS is a *function* that puts *S-expressions* together, CAR (pronounced "car") and CDR (pronounced "could-er") are *functions* that take apart *S-expressions* (that are not *atoms*). Any *S-expression* is either an *atom* or not an *atom*. If z is an *S-expression* that is not an *atom*, it must be of the form (x . y) where x and y are *S-expressions*.

By definition, CAR of z is x and CDR of Z is y. CAR and CDR are not defined when their *arguments* are *atoms*.

For example:

```
CAR('A) is undefined
CAR('(A . B)) is A
CAR('(A . (B . C))) is A
CDR('(A . (B . C))) is (B . C)
CDR('(A . B)) is B
CAR(CDR('(A . (B . C))) is CAR('(B . C)) which is B
CAR(CDR('A . B))) is CAR('B) which is undefined
CDR(CDR('(A . (B . C)))) is CDR('(B . C)) which is C
```

The names CAR and CDR arose as mnemonics in the early development of LISP, and have continued in use because they are short and easy to say, because they are symmetrical, and because they easily form longer names of *functions* involving several CARs and CDRs in succession: For example, CAAR is a *function* meaning CAR of CAR of, CADR is a *function* meaning CAR of CDR of , and CDADR is a *function* meaning CDR of CDR of CAR of CDR of , etc. CADR('(A . (B . C))) is CAR(CDR('(A . (B . C)))) which is CAR('(B . C)) which is B. Observe that in *expressions* using CADR or in *expressions* such as CAR(CDR('(A . (B . C)))), the CDR or D operation is done before the CAR or A operation.

For example:

```
CDAAR('(((W . X) . Y) . Z)) means CDR(CAR(CAR('(((W . X) . Y) . Z)))) which is X.
```

Problem Set 3:

Evaluate each of these *expressions*. (Some of them may be undefined.)

a. CAR('A)
b. CDR('(A . B))
c. CAR(CDR('(STRAVINSKY . (BARTOK . SIBELIUS))))
d. CDR(CAR(CAR('(((HAT . TIE) . SHIRT) . JACKET))))
e. CAR(CDR('((AQUITAINE . GASCONY) . ARAGON)))
f. CAR(CONS('A, 'B))
g. CAR(CDR(CONS('(A . B), '(C . D))))
h. CONS(CAR('(A . B)), CDR('(C . D)))
i. CONS(CAR('(A . B)),CAR('C . D)))
j. CONS('A, CAR('(C . D)))
k. CADR('(A . B))
l. CADR('(SHRIMP . (LOBSTER . CRAB)))
m. CAAR(CONS(CONS('A,'B),'C))
n. CDDR(CONS('A,'(B . C)))
o. CONS(CAAR('((A . B) . C)), CONS('D, CDDR('(E . (F . G)))))

Answers: See pages 139, 140

2.9 *BOOLEANS* AND *PREDICATES*

A *boolean* is a type of *atom*. There are exactly two *booleans*, namely TRUE and FALSE. They are very like "true" and "false" in ordinary language. Because *booleans* are *atoms*, they are also *S-expressions*. However, they are not *identifiers*.

A *function* in LISP 2 is called a *predicate* if its values are always one or the other *boolean*.

In programming, it is frequently necessary to choose between alternatives according to whether a given condition is true or false. The use of *booleans* and *predicates* in this process is illustrated further on.

The *boolean* FALSE is also expressed by either one of two other names:

NIL

( )

NIL is not an *identifier*; it is another name for the *boolean* FALSE. FALSE, NIL and ( ) are absolutely equivalent names for the same *boolean*; it is a matter of indifference which one is used at any time.

2.10 LIST:NOTATION

The notation for writing *S-expressions* that has been introduced so far is known as *dot:notation*. It is not very convenient for representing symbolic data because of the larger number of *dots* and *parentheses* required. There is another notation called *list:notation* which allows one to write many *S-expressions* more conveniently than in *dot:notation*.

It is important to understand that no new type of *S-expression* is being introduced in this way: instead we have a new way of writing *S-expressions* that have already been introduced.

Given any *S-expression* in *list:notation*, it is always possible to write the same *S-expression* in *dot:notation*. However, the converse is not always true.

Definition:
Given (x_1 x_2 ... x_n) where x_1, x_2 ... are *S-expressions*, then this by definition is the same *S-expression* as (x_1 . (x_2 . ... (x_n . NIL) ...)). The form (x_1 x_2 .... x_n) is called a *list*.

Examples:

(A M D H) is the same as (A . (M . (D . (H . NIL))))

(A B) is the same as (A . (B . NIL))

(A) is the same as (A . NIL)

( ) is the same as NIL

The *list:notation* (A B) and the *dot:notation* (A . (B . NIL) are equivalent names for exactly the same *S-expression*; either may be used.

Lists may occur within *lists* to any desired depth. For example,

```
((A B C) (D E F) (G H I))
```

is a *list* of *lists* (to depth 2.) At each depth, the *list* stands for an *expression* using *dot:notation* according to the definition given above.

For example, consider the *S-expression* ((A B C) (D E F) (G H I)). According to the rule:

(A B C) is the same *S-expression* as (A . (B . (C . NIL)))
(D E F) is the same *S-expression* as (D . (E . (F . NIL)))
(G H I) is the same *S-expression* as (G . (H . (I . NIL)))

Then

```
((A B C) (D E F) (G H I)) can be written as:
((A . (B . (C . NIL))) (D . (E . (F . NIL))) (G . (H . (I . NIL))))
```

Here *dot:notation* and *list:notation* have been mixed, and this is acceptable also. To put this into pure *dot:notation*, we observe that it is of the form (x y z) and rewrite it in the form (x . (y . (z . NIL))). This gives us:

```
((A . (B . (C . NIL))) . ((D . (E . (F . NIL))) . ((G . (H . (I . NIL))) . NIL)))
```

List:notation, where it can be used, is obviously compact and convenient.

Problem Set 4:

Rewrite each of the following *S-expressions* using only *dot:notation*.

a. (A)
b. ((A))
c. (HE MADE THE STARS ALSO)
d. (() (A) (A A))
e. (A (A) ((A)))

Rewrite each of the following *S-expressions* using *list:notation* as much as possible:

f. ((A . NIL) . (((B . NIL) . NIL) . NIL)

g. ((A . NIL). ((B . NIL) . NIL))

h. (A . B)

i. ((((A . NIL) . NIL) . NIL) . NIL)

j. ((X . NIL) . ((NIL . Y) . NIL))

Answers: See pages 140, 141

There is another mixed notation that the programmer may never use, but which from time to time appears on computer output. An *S-expression* of the form (x_1 x_2 ... x_{n-1} . x_n) is the same as the *S-expression* (x_1 . (x_2 ... (x_{n-1} . x_n) ...)).

Example:

(A B . C) is the same as (A . (B . C)).

The behavior of the *functions* CAR, CDR and CONS on *lists* can always be determined by translating the *arguments* into *dot:notation*, evaluating, and then, if desired translating back into *list:notation*.

Example:

CDR('(A B C))

CDR('(A . (B . (C . NIL)))) is (B , (C . NIL)) which can be written in *list:notation* as (B C). Therefore, CDR('(A B C)) is (B C) .

Problem Set 5:

Evaluate each of these *expressions*:

a. CAR('(A B C))

b. CADR('(A B C))

c. CADDR('(A B C))

d. CDR('(A B C))

e. CDDR('(A B C))

f. CDDDR('(A B C))

g. CAAR('(A B C))

h. CONS('A,'(B C))

i. CONS('A, CONS('B, '(C)))

j. CONS('A, CONS('B, CONS('C, NIL)))

k. CONS('(A B),'(C D))

l. CONS(CONS('A, NIL), NIL)

m. CDAR('((A B) (C D)))

Answers: See pages 141, 142

2.11 THE *PREDICATE* EQUALS

The *predicate* EQUALS or = has the same meaning in LISP as it has in ordinary mathematics. For example, it is true that 'A = 'A, but it is not true that 'A = 'B.

When evaluating an *expression* of the form x=y, the *value* is TRUE if x and y are the same *S-expression* and FALSE otherwise. Two ***S-expressions*** may be the same even if they do not look the same, because one is written in *list:notation* and the other is written in *dot:notation*. In this case, the *value* of x=y is true.

Examples:

'(A B) = '(A B) is TRUE

'(A)= '(A . NIL) is TRUE

'(A . B) = '(A B) is FALSE

CONS('A='A, 'B) is (TRUE . B)

CONS('A, 'B='C) is (A . FALSE) or (A . NIL) or (A)

Problem Set 6:

Evaluate the following *expressions*.

a. '(HELLO THERE BILL) = '(HELLO THERE JOE)

b. FALSE=( )

c. NIL=( )

d. '(A (B . C)) = '((A . B) . C)

e. CAR('(A B)) = CADR('(B A))

f. CONS(CONS('(A B), '(C D)),'A = 'B)

Answers: See pages 142, 143

2.12 The *PREDICATE* ATOM

The *predicate* ATOM has the *value* TRUE is its *argument* is an *atom*, and the *value* FALSE if its *argument* is not an *atom*. Remember that *identifiers*, *booleans*, and other things not yet defined are *atoms*.

Examples:

```
ATOM('A) is TRUE
ATOM('(A . B)) is FALSE
ATOM('(A)) is FALSE
ATOM('()) is TRUE (because () is FALSE which is a boolean)
ATOM(CAR('(A B C))) is ATOM('A) which is TRUE
```

Problem Set 7:

Evaluate the following *expressions*.

a. ATOM('TUVWXYZ)

b. ATOM('A) = ATOM('B)

c. ATOM(CDR('A B)))

d. ATOM('A = '(B C))

e. ATOM(CAR (CONS(CAR('(A B)), CDR('(C D)))))

Answers: See page 143

2.13 THE *FUNCTION* LIST

LIST is a *function* that has an indefinite number of *arguments*. It may have zero, one or more *arguments*.

LIST(x_1, ... x_n) has the same *value* as CONS(x_1, ... CONS(x_n, NIL) ...)

Examples:

```
LIST('A, 'B , 'C)has the same value as CONS('A, CONS('B, CONS('C, NIL)))
   which is (A B C)
LIST('A) has the same value as CONS('A, NIL) which is (A)
LIST( ) is ( ) or NIL
LIST(LIST (LIST('A))) is (((A)))
LIST('(A B), '(C D)) is ((A B) (C D))
```

Problem Set 8:

Evaluate the following *expressions*:

a. LIST('A, 'B, '(C D))

b. CAR(LIST('A, 'B, 'C))

c. CAR(LIST('(A B C)))

d. ATOM(LIST('A))

e. LIST('A, 'B)=CONS('A, CONS('B, NIL))

Answers: See pages 143, 144

2.14 THE *PREDICATE* NULL

The *predicate* NULL has the *value* TRUE if its *argument* is the *boolean* FALSE and has the *value* FALSE if its *argument* is anything else.

Examples:

```
NULL(FALSE) is TRUE
NULL(()) is TRUE
NULL(NIL) is TRUE
NULL(TRUE) is FALSE
NULL('A) is FALSE
NULL('(((A B C)) (D))) is FALSE
NULL(CDDR('A B))) is TRUE
```

Problem Set 9:

Evaluate the following *expressions*.

a. NULL(CADDR('(A (B C) D)))

b. CONS('A,NULL('A))

c. NULL(LIST ())

d. NULL(CDR(LIST 'A)))

Answers: See page 144

CHAPTER 3

SOME ILLUSTRATIONS OF PROGRAMMING IN LISP 2

This chapter contains several LISP 2 *programs*--miniature, but complete . The text explains how the *programs* are organized and the results they produce.

It should be possible to understand the sense of these illustrative *programs*, even though not enough information has yet been given for the reader to write a *program* himself.

3.1 A *PROGRAM* TO SOLVE QUADRATIC EQUATIONS

We shall write a *program* in LISP 2 that solves quadratic equations of the form

$$ax^2 + bx + c = 0$$

To use this *program* on any occasion, you need to type in the name of the *program* (suppose we call it QUADSOLVE) and the *numbers* a, b, and c. If the equation has real roots, the *program* replies by typing out the *numbers* that are the solutions for x; otherwise it types out the report COMPLEX.

The *program* that we shall express in LISP 2 can be summarized by the following algorithm in English:

Step 1. Compute $b^2 - 4ac$, and call it w.
Step 2. If w is negative, then type out the word COMPLEX and halt; otherwise go to step 3.
Step 3. Compute $(-b+\sqrt{w})/2a$ and print the value
Step 4. Compute $(-b-\sqrt{w})/2a$ and print the value.
Step 5. Type out the phrase PROBLEM SOLVED.
Step 6. Halt.

Let us suppose that you are working at a time-shared computer facility, and that you have just called LISP 2. The computer now waits for you to type something.

First, you type the following *function:definition* of the LISP 2 *function* QUADSOLVE, which solves quadratic equations using the algorithm just stated:

```
FUNCTION QUADSOLVE(A,B,C) BEGIN REAL W;
   W←B↑2-4*A*C;
   IF W < 0 THEN RETURN #COMPLEX#;
   PRINT((-B+SQRT(W))/(2*A));
   PRINT((-B-SQRT(W))/(2*A));
   RETURN #PROBLEM SOLVED#;
END;
```

When these lines have been typed, the waiting LISP 2 computer system has absorbed the *function:definition* of the *function* QUADSOLVE. You may then call QUADSOLVE and use it.

For example, suppose you desire to solve the particular quadratic equation $3x^2+3x+4=0$. You type:

```
QUADSOLVE (3,3,4);
```

This requests the solutions of $3x^2+3x+4=0$. There are no real solutions to this equation; therefore the *program* prints out:

```
COMPLEX
```

LISP 2 is then ready for your next example, which might be:

```
QUADSOLVE(3,7,4);
```

This does have solutions, and the *program* replies:

```
-1.0
-0.75
PROBLEM SOLVED
```

Let us now comment on the components of *function:definition* and explain their meaning.

Component	Meaning
FUNCTION	This informs the LISP 2 system that a *function:definition* is being presented.
QUADSOLVE	This is the name of the *function* being defined. Any name, of course, can be chosen that has not already been given a meaning in the LISP 2 system.
(A,B,C)	This is a *list* of the names of the *arguments* or the *argument:parameters* of QUADSOLVE. It specifies that QUADSOLVE has three *arguments*, and that they are called A, B and C, respectively. They could, of course, have been called M, N, and P or any other names, but then these other names would have to be used consistently throughout the rest of the *function:definition*.
BEGIN ... END	These two words along with whatever goes between them constitute the main part of the *function:definition*. It is called the *body*. The main entities inside the *body* are either *declarations* or *statements*. They are separated by *semi:colons*.
REAL W	W is called an *internal:parameter*. REAL W is a *declaration* that says that the *values* for W are real numbers in the mathematical sense, and floating-point numbers in the computer sense.
W←B↑2-4*A*C	This is an *assignment:statement*. It says that W is assigned the *value* of B^2-4AC. The *left:arrow* means "is assigned the *value* of". The *up-arrow* means "raised to the the power....". B↑2 means B^2. The *asterisk* means "multiplied by". 4*A*C means 4AC. Although no mathematical parentheses appear around B↑2-4*A*C, the *left:arrow* implies these parentheses.

```
IF W<O THEN RETURN #COMPLEX#
```

This is a *conditional:statement*. The *less:than:sign* (<) is used to say that if W is less than 0, then the computation is complete and the *value* of the *function* QUADSOLVE is the word COMPLEX. The word RETURN means "this is the end of the computation of this *function*, and the *value* of the *function* is what follows." #COMPLEX# consisting of the word COMPLEX inside two *fences* (#) is called a *string*. A *string* is basically a sequence of characters handled as a constant unit and not having any other meaning in the LISP 2 system.

```
PRINT((-B+SQRT(W))/(2*A))
```

This is another *statement*. It says "print out the *value* of the *expression* $(-B+\sqrt{W})/2A$". SQRT is a *function* in the LISP 2 system that gives square root.

```
RETURN #PROBLEM SOLVED#
```

#PROBLEM SOLVED# is another *string* that is returned as PROBLEM SOLVED by the computation as a result.

```
END;
```

END indicates the end of the *body*. When the *semi:colon* following END is typed, the entire *function:definition* is absorbed by the LISP 2 system.

These comments are not intended as complete explanations. They serve only as a very brief illustration of a LISP 2 *program*.

This illustrative *program*, if it had been written in any of several other algebraic compiler languages, would have looked quite similar. But the next examples of *programs* illustrate programming techniques peculiar to LISP 2.

3.2 A *PROGRAM* TO COMPUTE THE FACTORIAL OF A NUMBER

Mathematically, the factorial of a positive integer is the product of all the integers starting from 1, and up to and including the given integer. The factorial of 0 is 1 by definition. The factorial of a negative integer is undefined. The factorial of n is usually written in mathematics as n!, the exclamation point being read as "factorial." For example,

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1, \text{ which is } 120.$$

The following *function:definition* expresses FACTORIAL in LISP 2:

```
FUNCTION FACTORIAL(N) BEGIN INTEGER K,L;
   K ← 0;
   L ← 1;
A: IF K = N THEN RETURN L;
   K ← K+1;
   L ← L*K;
   GO A;
   END;
```

There are some new features in this *program*, and they may be briefly explained:

Component	Meaning
INTEGER K,L	This is a *declaration*. It says that K and L are *internal:parameters* and that their *values* are *integers*.
A:	This is a *label*. It labels the *statement* following as being the *statement* named A.
GO A	This is a *go:statement*. It causes the *program* to continue by jumping to *statement* A and proceeding from there.

We shall now give an alternative definition for the *function* FACTORIAL. This alternative uses a fundamental concept of LISP 2 called recursion. Consider the following definition of factorial in English: "The factorial of 0 is 1; the factorial of any positive integer is that integer times the factorial of the next smaller integer."

This definition is not a circular definition; it is a *recursive:definition* because factorial for one *argument* is defined in terms of factorial for another *argument*, and the entire sequence of *arguments* comes to an end. For example, if we want to know what 5! is, the definition tells us that it is 5 times 4!, and additional uses of the definition tell us what other factorials are. The apparent circularity ends when the last case is resolved. For example:

```
5! = 5 x 4!
   = 5 x 4 x 3!
   = 5 x 4 x 3 x 2!
   = 5 x 4 x 3 x 2 x 1!
   = 5 x 4 x 3 x 2 x 1 x 0!
   = 5 x 4 x 3 x 2 x 1 x 1
   = 120
```

This way of defining factorial in English suggests a LISP *function:definition program* for FACTORIAL that is also recursive. It is written as follows:

```
FUNCTION FACTORIAL(N) IF N=0 THEN 1 ELSE N*FACTORIAL (N-1);
```

Like most *recursive:definitions*, it is both extremely compact and powerful. It is equivalent to the previous LISP 2 *function:definition* in the sense that it always gives the same answer.

Having given this *function:definition*, we may type:

```
FACTORIAL(7)/(FACTORIAL(5)*FACTORIAL(7-5));
```

and the *program* replies:

```
21
```

which is correct, since 7! divided by 5! times 2! equals 21

The ability to create *recursive:definitions* is a skill that can be developed by practice. *Recursive:definitions* are a very powerful feature of LISP programming; therefore,the examples in this Primer emphasizes them.

3.3 A *PROGRAM* TO DETERMINE MEMBERSHIP IN A *LIST*

The following example processes symbolic data, whereas the previous ones processed numbers. We shall use as an example a *function* related to *lists*: the *function* MEMBER. An element is a member of a *list* if and only if that element is present in the *list*. This *function* has two *arguments*, which are an element and a *list*. It is a *predicate* because its only *values* are TRUE and FALSE. If the element is a member of the *list*, then the *value* of MEMBER is TRUE, otherwise the *value* of MEMBER is FALSE.

```
FUNCTION MEMBER (E,L)
  IF NULL (L) THEN FALSE
  ELSE IF E=CAR (L) THEN TRUE
  ELSE MEMBER (E,CDR(L));
```

Let us trace through this *function:definition* step by step:

The word FUNCTION means that we are defining a *function*

The name of the *function* is MEMBER.

The two *variables* of which MEMBER is a *function* are E (which stands for an element) and L (which stands for a *list*).

IF NULL (L) means "if L is empty,"

THEN FALSE means "the *function* has the value FALSE for this case."

IF E=CAR(L) means "if the element E is the first element of the *list* L",

THEN TRUE means "the *function* has the value TRUE in this case."

ELSE MEMBER (E, CDR(L)) means "in other cases, discard the first from the *list* L and apply the same definition over again to the rest of the *list* L."

For example, consider MEMBER ('A, '(A B C)). In this case the second *if:clause* produces true, and so MEMBER has the *value* TRUE.

For another example, consider MEMBER ('B, '(A B C)). In this case, the first time through, with L set at (A B C), we obtain no decision, and so we go through a second time with L set at (B C). This time we do obtain a decision, true, because B = CAR'(B C)).

Further examples of *function definitions* and *programs* using LISP 2 are given in subsequent places in this Primer.

CHAPTER 4

ARITHMETICAL:EXPRESSIONS

An *expression* can be roughly explained by saying that it is something that can be evaluated to yield a *value*. For example, 3+4 is an *expression*; the *value* it yields is 7. However, =A(X/)) (is not an *expression* because it is simply a collection of *signs* that has not been defined to have a meaning. Also, GO A is not an *expression*, because even though it causes something to happen, it nevertheless does not yield a *value*.

In fact, GO A is called a *statement*. In a later chapter, the concepts of *expression* and *statement* are further explained and clarified. The distinction between the two concepts is essential.

Another example of an *expression* is A+3. This *expression* may be evaluated and yields a *value*; however, the *value* is dependent on the meaning given to A by some particular context. Outside of a particular context there is no reason to give any particular *value* to A. The nature of the context that gives meaning to A is discussed later, but some idea of its nature may be gained by studying the examples in this chapter.

4.1 NUMBERS

A *number* is an *expression*. It is an *expression* because it can be evaluated, and the *value* it yields is itself.

Several different types of *numbers* are used in LISP 2. The two most important types, *integers* and *real:numbers* are described here.

4.2 INTEGERS

An *integer*, sometimes called a *whole:number*, is a *number* with no fractional part. It may be positive, negative or zero.

In LISP 2 an *integer* may be:

(1) A sequence of one or more of the *numerals* 0 through 9, or

(2) A *plus:sign* (+) followed by a sequence of *numerals* as in (1) above, or

(3) A *minus:sign* (-) followed by a sequence of *numerals* as in (1) above.

(4) The same as in (1), (2) or (3) above, followed by the *letter* E followed by a sequence of one or more *numerals*.

Examples:

5
+37
-0
299
-80
007
1E9 (1×10^9)
-7E3 (-7×10^3)
3E4 (3×10^4)
30E3 (30×10^3)
+3E4 (3×10^4)
30000 (3×10^4)

The last 4 examples are all equivalent.

Examples that are incorrect in LISP 2:

E2	An *integer* must have at least one *numeral* that is not to the right of the E. (E2 is an *identifier*.)
2E+6	In the case of an *integer*, a *sign* is not permitted to the right of the *letter* E.
1E1E1	Only one E is permitted.
6E	The E must be followed by at least one *numeral*.

In LISP 2 there is a limitation on the maximum size of an *integer* (whether positive or negative). This limitation depends on the computer being used.

An *integer* with a *plus:sign* is equivalent to the same *number* without a *sign*. Thus, 3, +3, and 003 are all equivalent.

In LISP 2 an *integer* that ends in several zeros can be written using a more abbreviated notation using the *letter* E to indicate an exponent. For example, -720000000 can be more conveniently written as -72E7, meaning -72 times 10^7.

4.3 REAL:NUMBERS

In LISP 2, *real:numbers* differ from *integers* in several ways. *Real:numbers* may have fractional parts (for example, 1.75); they may often be extremely large as compared with manageable *integers* (for example, 2.5E22); they may be very small (for example, .000000098).

The definition of a *real:number* is a little more complicated than the definition of an *integer*. It is worth noting that *integers* never have *decimal:points* while *real:numbers* always have *decimal:points*.

A *real:number* has three parts of which the first and third are optional:

- Part 1 consists of a *plus:sign* (+) or a *minus:sign*(-). This part may be omitted.
- Part 2 consists of several *numerals*, followed by a *decimal:point*, followed by several *numerals*. There may be no *numerals* to the left of the *decimal:point* or there may be no *numerals* to the right of the *decimal:point*, but not both of these conditions may be true at once. In other words, there must be at least one *numeral* either to the left or the right of the *decimal:point*.

- Part 3 consists of the *letter* E followed by an *integer* that does not contain the *letter* E itself. The *integer* may have a *plus:sign* or a *minus:sign*. This part may be omitted.

The *letter* E followed by an *integer* k means that the preceding *number* is to be multiplied by 10 raised to the kth power. For example, .05E3 means .05 multiplied by 10^3, which is 50.0; and 1.E-6 means 1. times 10^{-6}, which is .000001.

Examples:

2.87
2.87E-3
.03E4
30.E4
30.+E4

Examples that are incorrect in LISP 2 are:

.E1	There must be a *numeral* on one side or the other of the *decimal:point*
3	There must be a *decimal:point*
2E3	There must be a *decimal:point* to the left of the E
3.2E1.5	No *decimal:point* is permitted to the right of E

4.4 *ARITHMETIC:OPERATORS*

Certain *marks* in LISP 2 are combined to form *arithmetic:operators* that stand for familiar operations often performed on *numbers*. Some of these *arithmetic:operators* are:

Arithmetic:operator	Meaning
+	Addition or plus
-	subtraction or minus
*	multiplication or times
/	division or divided by
-:	*integer* division (example: 14-:3 equals 4)

\	*integer* remainder (example: 14\3 equals 2)
↑	exponentiation (example: 5 ↑ 3 equals 125)

These *arithmetic:operators* permit us to form more *arithmetic:expressions.*

Rule A for Forming *Arithmetic:Expressions:*

Let x and y be *arithmetic:expressions.* Then each of the following is also an *arithmetic:expression:*

+x	plus x
x+y	x plus y
-x	minus x
x-y	x minus y
x*y	x times y
x/y	x divided by y
x-:y	the result of *integer* division of x by y
x\y	the result of *integer* remainder of x by y
x↑y	x to the power y
(x)	meaning the same as x but grouped by *parentheses*

This rule is recursive. According to this rule, each of the following examples is an *arithmetic:expression.* If you do not understand why this is so, please refer to the discussion of *recursive:definitions* in paragraph 2.4.

Examples of *arithmetic:expressions:*

35
2.7E4
---2
X*Y
12-:A
A+B*C
(A+B)*C
(A+B*C)

```
A/A/A/3
A
A+5
U-V
5/3
A↑2.0
A+(B*C)
((A+B))*(((C)))
(((2)))
```

The meaning of some of these *expressions* may not be clear until the end of this chapter.

In LISP 2, *arithmetic:expressions* may contain a mixture of *integers* and *real:numbers*. It is not necessary to keep them separated in any way. The following rules determine what happens in various cases.

Rule 1: When the operations of addition (+), subtraction (-), negation (also -), and multiplication (*) are performed, the value is an *integer* if all of the arguments are *integers*. The value is a *real:number*, if at least one argument is a *real:number*.

Examples:

```
2+3 is 5
2+3.0 is 5.0
1E2 + 3 is 103
1.5*1.5 is 2.25
1.E2-2.E-2 is 99.98
```

Rule 2: When the operation of division (/) is performed, the value is always *real*. The division is carried out to the limitation of the accuracy of the computer on which it is performed.

Examples:

1/3 is .3333333333 (exactly how many 3's occur depends on the capacity of the computer)

6/3 is 2.0

5.0/2 is 2.5

2.5/2.5 is 1.0

Rule 3: When *integer* division (-:) and *integer* remainder (\) are performed, the result is always an *integer*.

The *integer* quotient is defined as being the integral number of times that the divisor goes into the dividend. This may be a positive or negative *integer* or zero.

The remainder is what is left over after this process has been performed. The remainder always has the same sign as the dividend.

These definitions have been chosen so that the following identity holds exactly:

dividend = (divisor * quotient) + remainder

If either *argument* of an *expression* containing an *integer* division or *integer* remainder operator is a *real:number*, the *argument* is converted to an *integer* by the process of rounding to the nearest *integer* (see below). The rounding happens before the operation -: or \ is performed. This procedure sometimes has peculiar consequences. For example, 3.4-:1.7 is the same as 3-:2 which is 1, while of course 3.4/1.7 is 2.0

Examples:

5-:2 is 2

5\2 is 1

-5-:-2 is 2

-5\-2 is -1

-5-: 2 is -2

5\2 is -1

5-: -2 is -2

5\-2 is 1

5.0-:2.0 is 2

5.0\2.0 is 1

3.4-: 1.7 is 1

3.4\1.7 is 1

Rule 4: If x and y are two *expressions*, then x ↑ y is x raised to the exponent y.

Examples:

2 ↑ 3 is 8

3 ↑ 2 is 9

If x, y, and z are three *expressions*, then x ↑ y ↑ z is x ↑ (y ↑ z).

Examples:

2 ↑ 3 ↑ 2 is 2 ↑ (3 ↑ 2), which is 2 ↑ 9, which is 512

(2 ↑ 3) ↑ 2 is 8 ↑ 2, which is 64

What about the type of the result, and special cases involving zero? The specifications are shown in Table 1. Here a is any *number*, i is an *integer*, and r is a *real:number*.

Table 1

Case	Subcase	Type and Remarks
a ↑ i	i > 0	same type as a; if the result is too big (or small), it is expressed as a *real:number*
a ↑ i	i = 0, a ≠ 0	1, of the same type as a
a ↑ i	i = 0, a = 0	undefined
a ↑ i	i < 0, a ≠ 0	of type *real*
a ↑ i	i < 0, a = 0	undefined
a ↑ r	a > 0	exp (r $\log_e$a), of type *real*
a ↑ r	a = 0, r > 0	0.0, of type *real*
a ↑ r	a = 0, r ≤ 0	undefined
a ↑ r	a < 0	always undefined

Examples:

10 ↑ 7 is 10000000

10 ↑ 30 is 1E30

0 ↑ 0 is undefined

0 ↑ 1.37 is 0

13.76 ↑ 2.5 is $e^{2.5 \log_e 13.76}$

-4 ↑ 2 is 16

14 ↑ 2.0 is undefined

8 ↑ 0 is 1

8.0 ↑ 0 is e^0, is 1.0

Problem Set 10:

Evaluate each of these *arithmetic:expressions* using the following table to determine the *values* of the *variables* occurring in the *expressions*

Variable	*Value*
A	2
B	-3.0
C	-5
D	7.5

a. A-1

b. A+B

c. B↑A

d. C-:D

e. C/D

f. A*C

g. D-:1.0

Answers: See page 145

4.5 PRECEDENCE

The fact that many *arithmetic:expressions* are recursive (see Section 4.4) sometimes makes their meaning ambiguous. For example, consider A + B * C. How is this to be evaluated? Suppose that A is 2, B is 3 and C is 4. If we take the *expression* to mean (A + B) * C, then the *expression* becomes (2+3) * 4, which equals 20. If we take the *expression* to mean A + (B * C), then the *expression* becomes 2 + (3 * 4), which equals 14. In a programming language this kind of ambiguity is intolerable; to remove it we use a set of conventions called the rules of precedence.

Precedence rules are dependent upon the *operators* used in the *expression.* If an *operator* appears interspersed between its operands, it is called an *infix:operator.* If the *operator* precedes its operands, it is called a *prefix:operator.*

We can state many of the rules of precedence quite simply using Table 2 and some additional statements.

Table 2

Rank or Precedence	*Prefix* and *Infix:Operators*	For More Details, See
6	CAR, CDR	Section 5.3
5	*arithmetic:operators* within *expressions*, ↑, *, +, -	Section 4.4
4	*equals* (=), *less:than* (<), *greater:than* (>), *not:equal* (/=), *less:than:or:equal* (<=), *greater:than:or:equal* (>=)	Section 5.4
3	ATOM, NULL	Section 5.1
2	the *boolean:operators*, AND, OR, NOT, etc.	Chapter 8
1	the *infix:operator* for CONS which is *space dot space*	Section 5.3

All *operators* of higher rank according to this table take precedence over *operators* of lower rank. For example, CAR A + B means (CAR A) + B since CAR (rank 6) has higher rank than plus (rank 5). But A . B + C means CONS (A, B+C) since plus (rank 5) takes precedence over the *dot* for CONS (rank 1).

Within rank 5, the rules of precedence are as follows:

Table 3

Rank or Precedence	*Functions* and *Operators*
3	↑ (raising to an exponent)
2	* (times), / (divided by), -: (integer-divide), \ (remainder)
1	+ (plus), - (minus)

In a case of equal rank, operations are regularly grouped in sequence from left to right:

For example:

(1) A + B - C + D means ((A + B) - C) + D (and does not mean (A + B) - (C+D), for example)

(2) A/B/C/D means ((A/B)/C)/D

The one exception is that raising to an exponent (↑) is grouped from right to left. Thus A↑B↑C↑D means A↑(B↑(C↑D)).

More information on precedence is explained in later chapters, but there is a simple and universal rule that can always be followed: When in doubt, put in enough *parentheses* to be unambiguous.

Problem Set 11:

Examine each *expression*. (1) Insert *parentheses* and produce an equivalent *expression* which if there were no precedence rules would be completely unambiguous. (2) Evaluate this *expression* using the table to determine the *values* of the *variables* occurring within the *expression*.

Variable	*Value*
A	5
B	2.5
C	1
D	-6

a. A-3*C

b. (A-3)*C

c. A-(3*C)

d. D↑C↑A

e. A+B*C+D

f. A*B+C*D

g. -D+A

h. -(D+A)

i. -D-A

j. 6/3/2

k. 6/(3/2)

l. 6/(3*2)

m. 6/3*2

Answers: See pages 146, 147

4.6 *ARITHMETIC FUNCTIONS*

Certain operations on *numbers* are written in the form "*function* (*argument*, *argument*, ...,*argument*)" rather than expressing the *function* as an *infix:operator* or *prefix:operator*. Note that the *arguments* are grouped using *parentheses* and *commas*

If there are no *arguments*, then it is correct to write fn( ). If there are one or more *arguments*, then there will be one less *comma*(,), than there are *arguments*. The ellipsis (...) is not part of the LISP 2 language. It is merely a device used in this text for designating a *list* of indefinite length.

Examples:

```
COS(A-3)
MAX(A,B,C)
ABS(X)*W
ROUND(M)
```

The following is a partial catalogue of *arithmetic:functions* available in LISP 2:

function	Number of Arguments	Description
ABS(X)	1	The absolute value of X is -X if X is negative, and X otherwise. The type (*integer* or *real*) of ABS(X) is the same as the type of X.
SIGN(X)	1	The arithmetic sign of X is 1 if X is positive, 0 if X is zero (any zero including -0), and -1 if X is negative.
MAX(X_1,...,X_n)	indefinite	The maximum of the X_i is the largest (most positive) value. If at least one argument of MAX is *real*, then the value is *real*. (e.g., MAX(2.0,5) is 5.0)

MIN(X_1, X_2, ...,X_n)	indefinite	The minimum of the X_i is the smallest (most negative) value. If at least one *argument* of MIN is *real*, the *value* is *real*.
ROUND(X)	1	X is rounded to the nearest *integer* by the formula: ROUND(X) = ENTIER (X + .5).
ENTIER(X)	1	The entier of X is the largest *integer* that is not greater than X. For example, ENTIER (2.7) is 2. ENTIER(-2.7) is -3.
SQRT(X)	1	If X is not negative, then the square root of X is its non-negative root. If X is negative, then SQRT(X) is not defined. The value of SQRT is always *real*.

Other *arithmetic:functions* are EXP, LOG, SIN, COS, and ARCTAN.

Problem Set 12:

Evaluate the following *expressions* using the table to determine the *values* of the *variables*.

Variable	*Value*
A	2
B	3.0
C	4
D	-0.0E6
E	-1
F	2.5

a. ABS(A)

b. ABS(E)

c. SIGN(-B)

d. SIGN(D)

e. MAX(A,-B)

f. MAX(A,-C)

g. MIN(A,E)

h. ROUND(F)

i. ENTIER(F)

j. ROUND(-F)

k. ENTIER(-F)

l. SQRT(C)

m. SQRT(E)

n. ABS(A)+ABS(B)*ABS(C)

o. -ROUND(E)-ROUND(D)

p. ROUND (-F + .3)

Answers: See pages 147, 148 149

CHAPTER 5. *SIMPLE:EXPRESSIONS*

5.1 NUMBERS AS ATOMS

In Chapter 2 we stated the rule that *identifiers* and *booleans* are *atoms*. We now wish to extend this rule by stating that *integers* and *real:numbers* are also *atoms*. As a result, *numbers* may occur within *S-expressions* in various ways.

Examples of *Atoms*:

ABC	(an *identifier*)
TRUE	(a *boolean*)
2.5E6	(a *real:number*)
-50	(an *integer*)

Examples of *S-expressions*:

```
2.5
(A (6 TRUE) 7.2)
(A 6 B)
(Y . 2.6)
(3.4)
(3 . 4)
```

The last two examples are not equivalent. The *S-expression* (3.4) is a *list* of one element consisting of the *real:number* 3.4 (three, *decimal:point*, four); whereas (3 . 4) is the CONS value of 3 and 4.

The *predicate* ATOM is TRUE if its *argument* is any type of *atom*. There are other *predicates* that can be used to distinguish the different types of *atoms*.

IDP(X) is TRUE if and only if X is an *identifier*.

BOOLP(X) is TRUE if and only if X if a *boolean*.

NUMBP(X) is TRUE if and only if X is a *number*; *integers* and *real:numbers* are both *numbers*.

INTP(X) if TRUE if and only if X is an *integer*.

REALP(X) is TRUE if and only if X is a *real:number*.

Problem Set 13. Evaluate the following *expressions*.

a. CAR('(A B C))

b. CADR('(4 5 6))

c. CDR('(1 2))

d. ATOM(500)

e. REALP(7)

f. REALP(CAR('(3.5 4.5)))

g. CAR('(1.1))

h. CAR('(1 . 1))

i. ATOM('(7))

j. NUMBP(CAR('(7)))

k. CONS('(1 2),'(3 4))

Answers: See page 150

5.2 CONSTANTS AND VARIABLES

A datum is an *S-expression*. Thus a *number* is a datum, because *numbers* are *atoms*, which are in turn *S-expressions*. We refer to a *program* in a computer language such as LISP 2 as "data processor." A LISP 2 *program* performs various operations (processes) on its data, which are *numbers*, *identifiers*, composite *S-expressions*, etc.

A *constant* is a datum occuring within a *program*. It stands for itself as distinct from a *variable*, which stands for something else. For example, in the *expression* X+3, X is a *variable* which must stand for some *number* in order for addition to be performed, but 3 is a *constant*. It only means the *number* 3, because *numbers* are never used in LISP 2 as *variables*; instead, *identifiers* are used as *variables*.

Now what do we do if an *identifier* is to be used as a *constant*? To overcome this problem, we use a convention: we put an *apostrophe* (or single quote mark) in front of the *identifier*, and then the *identifier* refers to itself and not to something else. This mark is called *quote*, and the operation is called quoting. For example, the *identifier* ANSWER refers to some *variable* which supposedly is the answer to some problem; but if we want the word ANSWER itself written in part of the printout of a solution, then when we issue that instruction, we put a *quote* mark in front, and write 'ANSWER. Then this actual word itself is printed where instructed. In the same way, 'A means the *atom* A itself; but A with no *quote* mark is an *identifier* which is a *variable* referring to something else.

For another example, in the *expression*:

```
CONS (A,'A)
```

the first A is a *variable* that may stand for any *S-expression*, while the second A is a *constant*, and means A itself.

The following rule specifies when an *apostrophe* (') should be used to make a *constant*.

Definition: A *constant* is either

(1) an *apostrophe* (') followed by any *S-expression*, or

(2) a *boolean*, or

(3) a *number*.

Since a *number* is an *S-expression*, this rule tells us that '3 is a *constant*. But 3 is also a *constant* (without the *apostrophe*). Thus, the *apostrophe* is permitted but not required for *numbers* and *booleans* and it is generally omitted. The *apostrophe* is required whenever an *identifier* or a non-atomic *S-expression* is used as a *constant*.

Problem Set 14

Evaluate each of the following *expressions*, using the table to determine the *values* of the *variables* occurring in the *expressions*.

Variable	*Value*
A	X
B	NIL
C	3.5
D	(A 4)
E	A

a. CONS(A,B)

b. CONS('A,B)

c. CONS(E,'B)

d. CDR(D)

e. C+CADR(D)

f. SQRT(CADR(D))

g. CONS(E,C)

h. CONS(C,B)

i. C+2

Answers: See pages 151, 152

5.3 LISP *OPERATORS*

CAR, CDR and their compositions (such as CDAR, CADADR, etc.) may be used as *prefix:operators* without the need to enclose their *arguments* in parentheses. Their precedence is highest. So the following examples should be clear.

Examples:

CAR A	means	CAR(A)
CADR B + C	means	CADR(B) + C
	and not	CADR(B + C)
CONS(CAR A, CAR B)	means	CONS(CAR(A), CAR(B))

The *infix:operator space dot space* means CONS. It has a precedence which is lower than the precedence of any other *operator*; and if two or more CONS *dots* occur together, they are grouped from right to left.

Examples:

A . B	means	CONS(A, B)
A . B . C	means	CONS(A, CONS(B, C))
	and not	CONS(CONS(A, B), C)
CAR A . CDR B	means	CONS(CAR(A), CDR(B))
	and not	CAR(CONS(A, CDR(B)))
A+B . C	means	CONS(A+B, C)
	and not	A+CONS(B, C)

Problem Set 15.

Rewrite each *expression* adding enough parentheses to determine the correct grouping. Then evaluate them using the table to determine the *values* of the *variables*.

Variable	*Value*
W	4
X	(A B)
Y	C
Z	(2)

a. W . NIL

b. Y . X

c. W*3 . CAR Z

d. CAR Z+2

e. CAR X . CDR Z

f. Y . NIL

g. 'Y . NIL

Answers: See page 152

5.4 *BOOLEAN:EXPRESSIONS*

As stated earlier, a *predicate* is a *function* whose value is TRUE or FALSE. Using *predicates* we can form an *expression* whose *value* is TRUE or FALSE. These are called *boolean:expressions*.

The *predicates* introduced in Chapter 2 were ATOM and = (meaning equal). Also, the *predicates* IDP, BOOLP, NUMBP, INTP, and REALP have also been defined. There is another set of basic *predicates* known as the *arithmetic:relation:operators*. Each of these is an *infix:operator*.

Operator	Meaning
=	is equal to
/=	is not equal to
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to

The reason that the *operator* = is here listed again is that when it was first mentioned, it was defined only for *atoms*, not *arithmetic:expressions*.

Equality (=) may be used to test any two data and is TRUE if they are equal; and FALSE, otherwise. If a *real:number* and an *integer* are numerically equal, then the value of = is TRUE; for example, 3.0=3 is TRUE.

Inequality (/=) is TRUE when = is FALSE, and FALSE when = is TRUE.

The other four relations are defined only when their *arguments* are *numbers*, since it is not meaningful to ask if one *S-expression* is greater than another.

Problem Set 16.

Evaluate these *expressions* using the table to determine the *values* of the *variables*.

Variable	*Value*
A	3
B	2.4
C	3.0
D	A
E	(X Y)

a. A=3

b. A=C

c. D=A

d. B>=C

e. E='X . 'Y . NIL

f. 'A=D

g. CAR E='X

h. 0<B<=3

i. 2<C+3<7

j. 2<A<3

Answers: See pages 153, 154

5.5 THE GRAMMAR AND SYNTAX OF *SIMPLE:EXPRESSIONS*

The purpose of this section is to describe part of the grammar and syntax of LISP 2 accurately.

The terms *arithmetic:expression, boolean expression,* etc., classify *expressions* according to the type of datum they have as *values*. From a broader point of view, however, all *expressions*can be classified as being *simple:expressions*, *conditional:expressions*, or *block:expressions*, and every *expression* belongs in exactly one of these three classes. *Conditional:expressions* and *block:expressions* are not discussed in this chapter but are discussed later. What is a *simple:expression?*

In order to define *simple:expression*, we shall make use of the concept of a *primary*. The definitions of *primary* and *simple:expression* are interdependent.

Definition 1: Each of the following is a *primary*:

(1) A *constant*;

(2) A *variable*;

(3) A *form*. A *form* is a *function* name followed by a *left:parenthesis*, followed by the *arguments* of the *function* separated from each other by commas, and followed by a *right:parenthesis*. For example, FN('A, B*C) is a *form*;

(4) A *conditional:expression* (see Chapter 6) enclosed in a pair of *parentheses* ;

(5) A *simple:expression* (let's take this on faith for a few more paragraphs) enclosed in a pair of *parentheses*. For example, (A+B) or (G-SQRT(M)).

It follows from this definition that a *primary* is an *expression* which, whenever it occurs, is unambiguous. The *simple:expression* A+B is not a *primary*, because in some contexts it is ambiguous. For example, placed in the context A+B*C, the symbols A+B no longer mean the *expression* A+B, because A+B*C means A+(B*C).

Definition 2: Each of the following is a *simple:expression*:

(1) A *primary*;

(2) A *prefix:operator* followed by a *simple:expression*;

(3) A *simple:expression* followed by an *infix:operator* followed by a *simple:expression*.

These rules simply generalize the rule for forming *arithmetic:expressions* in Chapter 4.

The *simple:expressions* that result from these rules may be ambiguous. To prevent ambiguity, it is necessary to consider the rules of precedence to determine how *simple:expressions* are to be grouped. These rules are summarized below:

Rule 1: CAR, CDR, and their compositions have the highest precedence. They capture the smallest possible *expression* to the right of them. For example, CAR A↑B means (CAR A)↑B.

Rule 2: *Arithmetic:operators* are next in the hierarchy of precedence. Within the class of *arithmetic:operators*, there is a subhierarchy:

Rule 2a: ↑ has the highest precedence, and a↑b↑c is grouped as a↑(b↑c).

Rule 2b: *,/,-:, and \ are next. When these occur together, they are grouped from left to right. a/b*c is grouped as (a/b)*c. a*b/c*d is grouped as ((a*b)/c)*d and not as (a*b)/(c*d).

Rule 2c: + and - have the lowest precedence among the *arithmetic:operators* When these occur together, they are grouped from left to right. a-b+c is grouped as (a-b)+c. a+b-c+d is grouped as ((a+b)-c)+d and not as (a+b)-(c+d).

Rule 3: The *relation:operators* =, /=, $<$, $>$, $<=$ and $>=$ are lower in precedence then *arithmetic:operator*. These *relation:operators* are all on the same level of precedence and may be so arranged; for example, $a \leq b=c<d$ means that $a \leq b$, b=c, and $c<d$.

Rule 4: ATOM and NULL may be used as *prefix:operators*, that is, without always putting *parentheses* around their *arguments*. The precedence of ATOM and NULL is lower than the *relation:operators*.

Rule 5: The *logical:operators* NOT, AND, OR, XOR, IMPLIES and EQUIV as a group have next lower precedence. Their relative precedence is explained in a later chapter.

Rule 6: The *infix:operator* for CONS which is . (*space, dot, space*) has the lowest precedence of all. In other words, group everything else first. Finally, a . b . c is grouped from right to left as a . (b . c) and not from left to right as (a . b) . c.

Problem Set 17

Examine each *simple:expression* below. Then rewrite it adding sufficient *parentheses* to make it unambiguous assuming no rules of precedence.

a. CAR A+B

b. CAR A+CDR B*C

c. A-B/C/D+E

d. A-B/C*D↑E

e. CAR X='A

f. 0<CAR A=B+SIN(Y)<5

g. A+B↑C↑CADR D

h. X . 'A . FN(X,Y,CDR Z*W)

i. ATOM X=Y

j. NULL U . NULL CAR X+Y

Answers: See page 154

CHAPTER 6. *CONDITIONAL:EXPRESSIONS*

Consider the problem of describing the *function* which is Y of X shown in the graph of Figure 1.

It is not natural to write a *simple:expression* that gives the *value* of Y as a *function* of X. However, the following *conditional:expression* describes it precisely:

```
IF X<0 THEN X↑2 ELSE IF X<1 THEN 2*X ELSE 2
```

The *conditional:expression* is a means by which a computer program can make a choice between several alternatives depending upon conditions that are determined at the time in the program's execution when the choice is to be made.

6.1 THE ACCEPTED FORM OF *CONDITIONAL:EXPRESSIONS*

A *conditional:expression* is written either in the form

IF p_1 THEN e_1

or in the form

IF p_1 THEN e_1 ELSE e_2

where p_1 is any *expression* (including a *conditional:expression*), e_1 is an *unconditional:expression* (that is, it must be either a *simple:expression* or a *block:expression*), and e_2 is any *expression* (and therefore may be another *conditional:expression*).

The *expression* between the IF and the THEN is called an *antecedent*; the *expression* between the THEN and the ELSE, or following the ELSE, is called a *consequent*. Examples of *conditional:expressions*:

```
IF ATOM X THEN X ELSE CAR X

IF X=Y THEN 5
```

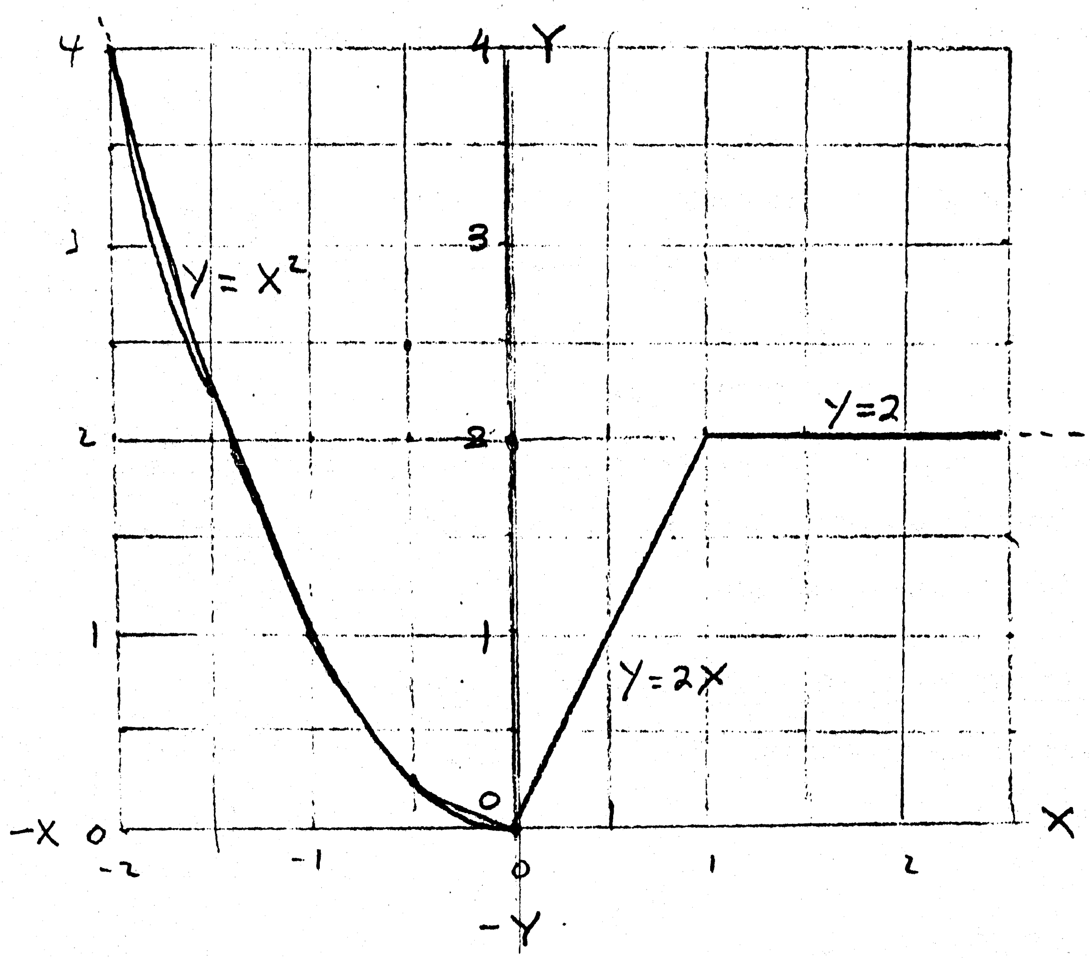


Figure 1. Example of the use of a *conditional:expression* for describing precisely the graph y = f(x), where the body of f(x) is:

IF X <0 THEN X↑2 ELSE IF X <1 THEN 2*X ELSE 2

The provision that e_1 cannot be a *conditional:expression* is a trivial restriction; its purpose is to make *expressions* unambiguous. If a *conditional:expression* is enclosed by a pair of *parentheses,* then it becomes a *simple:expression* and this *simple:expression* may be used as *consequent* e_1.

Example:

```
IF X>0 THEN (IF Y>0 THEN FN (X,Y) ELSE FN(X,-Y)) ELSE Z
```

Since e_2 may be any kind of *expression* including a *conditional:expression,* we are permitted to write *conditional:expressions* with many *antecedents* and *consequents*.

Examples:

```
IF A THEN B ELSE IF C THEN D ELSE IF E THEN F ELSE G
IF A THEN B ELSE IF C THEN D ELSE IF E THEN F
```

Since p_1 may be a *conditional:expression, conditional:expressions* may be nested within each other.

Examples:

```
IF IF X=5 THEN Y=3 ELSE Y<X THEN FN(X,Y)
IF IF IF A THEN B ELSE C THEN D ELSE E THEN F
```

6.2 THE EVALUATION OF *CONDITIONAL: EXPRESSIONS*

The following rules apply to the evaluation of *conditional:expressions*.

1. The parts of the *expression* are evaluated in order from left to right.
2. Only those parts of the *conditional:expression* that are needed to determine a *value* are evaluated.
3. Each *antecedent* is evaluated in succession until one is found that evaluates to be true. For this purpose, the *boolean* FALSE (for which NIL and () are equivalents) is considered to be false, while

any other datum is considered to be true. Usually the *antecedents* are chosen to be *boolean:expressions* so that their *value* are TRUE or FALSE.

4. If an *antecedent* evaluates to FALSE, then the corresponding *consequent* is skipped over and is not evaluated. If an *antecedent* evaluates to TRUE, then the corresponding *consequent* is evaluated, and this *value* is the *value* of the *conditional:expression.* The remaining *antecedents* and *consequents* in the same *conditional:expression* , if any, are not evaluated.
5. If a *conditional:expression* ends with ELSE e_n and if all of the preceding *antecedents* are false, then e_n is evaluated, and its *value* is the *value* of the *conditional:expression.*
6. If a *conditional:expression* ends with ELSE IF p_n THEN e_n and if all of the *antecedents* including p_n are false, then the *value* of the *conditional:expression* is undefined, and an error condition results.

Examples:

In the following examples, suppose the *variables* are bound by the following table:

W	5
X	A
Y	(A . B)
Z	(3 4 5)

Example 1:

```
IF W<4 THEN X ELSE IF CADR Z<W THEN Y ELSE NIL
```

Steps in evaluation:

1. W<4 is FALSE: therefore skip over X.
2. CADR Z<W is TRUE because 4<5; therefore the *value* of Y which is (A . B) is the *value* of the ***conditional:expression***.
3. The part ELSE NIL is not considered.

Example 2:

IF W<4 THEN 'B

The *value* is B.

Example 3:

IF W<4 THEN 'B

The *value* is undefined.

Example 4:

IF X THEN W

The *value* of X is 5 which is not FALSE, and is taken as true; the *value* of the ***conditional:expression*** is A.

Example 5:

IF W=CADDR Z THEN (IF X=CAR Y THEN W↑2 ELSE 10) ELSE 20

Steps in evaluation:

1. CADDR Z is 5 and this =W. Take the ***consequent***.
2. CAR Y is A and this =X. Take the ***consequent***.
3. W↑2 is W squared, which is 5 squared, which is 25.

Example 6:

IF IF X=Y THEN W>4 ELSE W<4 THEN 'B ELSE 'C

Steps in evaluation: Think of IF (......) THEN 'B ELSE 'C

1. X is not equal to Y. Take what follows the first ELSE.
2. W is not less than 4. Therefore (......) evaluates to FALSE.

   Take what follows the second ELSE.
3. 'C evaluates to C.

6.3 OMISSION OF ELSE

If the *reserved:word* ELSE is immediately followed by the *reserved:word* IF, then that ELSE may be omitted because there is no ambiguity.

Example:

IF A THEN B ELSE IF C THEN D ELSE E

may be shortened to

IF A THEN B IF C THEN D ELSE E

The second ELSE may not be omitted because it is not followed by an IF.

Problem Set 18

Evaluate the following *expressions* using the list of *values* for *variables*. REALP means "is a *real:number*"; SQRT means "the square root of"; SIGN means "the sign of."

Variable	*Value*
A	5
B	2.0
C	(7 14)
X	(3 . 9)
Y	(A B C)
Z	(A C)

a. IF A=5.0 THEN B

b. IF REALP(Z) THEN C ELSE IF REALP(B) THEN (IF CAR A↑2=CDR A THEN Y ELSE Z) ELSE X

c. IF IF CAR C=7 THEN FALSE ELSE TRUE THEN Z

d. IF A=B THEN A=B ELSE A=B

e. IF C THEN A

f. IF SIGN(B)=SIGN(A) THEN (IF SQRT(CDR X)=CAR(X) THEN 'A ELSE A) ELSE 'B

g. IF CAR Y=CAR Z THEN 'ELSE ELSE 'IF

h. IF TRUE THEN 'IF IF 'IF THEN 'THEN

Answers: See pages 155, 156

CHAPTER 7.

FUNCTION:DEFINITIONS AND RECURSION

After you understand this chapter, you should be able to write simple LISP *programs*. This chapter explains how to make a series of related *function:definitions*, how the process called recursion works, and how to define some *functions* and how to use these *functions* to operate on some simple data. There is more to these topics than is explained here.

7.1 *FUNCTION:DEFINITIONS*

A *function:definition* is a *declaration* that the programmer makes to the LISP system. The *declaration* names a *function*, specifies its *arguments*, specifies what computation is to be performed, and what is to be the *value* of the *function*.

Each *function:definition* has two parts, the heading and the body. Each *function:definition* includes a *semi:colon* which terminates the definition. The system then holds this definition in memory, and at the appropriate time compiles it into efficient machine code so that it can be executed.

7.2 THE HEADING

The heading of a *function* has the form:

FUNCTION name (a_1, ..., a_n);

This consists of several parts; the first part of the heading is the *constant: identifier* FUNCTION, which is a *reserved:word*. Then comes the particular name of the *function* that is being defined. The name of the *function* is an *identifier*. Then comes the *argument:parameter:list*. If there are no *argument:parameters*, one must still write ( ). The *argument:parameters* are *identifiers*. If there are two or more *arguments*, they must be separated from each other by *commas*. The last part of the heading is a *semi:colon*. It is optional.

Examples of headings of *function:definitions*:

```
FUNCTION READ( );
FUNCTION CUBE (X)
FUNCTION SUBST (X1, X2, X3);
```

7.3 THE BODY

The body of a *function:definition* is always an *expression*. It may be any kind of *expression*. *Simple:expressions* and *conditional:expressions* are defined in Chapters 4 through 6. The third kind, *block:expressions*, are defined in Chapter 9. Examples of *function:definitions*:

Each definition has a heading and a body, and is followed by a *semi:colon:*

```
FUNCTION CUBE(X)  X↑3 ;
FUNCTION HYPOTENUSE(SIDE1,↑SIDE2); SQRT(SIDE1↑2+SIDE2↑2);
FUNCTION PUT(X,Y,L) CAR X . CDR Y . L;
```

7.4 EVALUATION OF *FUNCTIONS*

A *function* is called, or invoked, by the evaluation of a *form* which begins with the *function* name. For example, suppose that the *form* HYPOTENUSE(3,4) is to be evaluated. The *numbers* 3 and 4 are the *argument* of HYPOTENUSE. A *form* that calls a *function* must have as many *arguments* as the *function* has *argument:parameters*. The *arguments* are paired with the *argument:parameters* in the order in which they are written. Thus, the *argument* 3 is paired with the *argument:parameter* SIDE1, and the *argument* 4 is paired with the *argument:parameter* SIDE2.

The evaluation of a *function* consists of evaluating the *expression* which is its body. This *expression* usually contains *variables* which are *argument:parameters* of the *function*. The values associated with these *variables* are the *arguments* that are paired with them. We speak of this association as *bindings*. This is an incomplete explanation of *bindings*, which is covered more fully in section 10.2, but it is sufficient for the present.

To continue the preceding example, the *function* HYPOTENUSE is evaluated by evaluating the *expression* SQRT(SIDE1↑2+SIDE2↑2). The current *bindings* of SIDE1 and SIDE2 are 3 and 4 respectively; therefore the *value* of the *expression* and the *value* of HYPOTENUSE is 5.0.

The body of one *function* may contain *forms* that call or invoke other *functions*. These in turn may call other *functions*. This may occur to any depth. Sometimes a *function* calls itself, either directly or by means of several *function* calls that eventually call the first *function*. This process is known as *recursive: definition* or recursion and is not only permitted, but is encouraged as a standard technique in LISP. It was illustrated earlier (Chapter 3.) by the definition of FACTORIAL, and is discussed below.

It is important to distinguish an *argument* from an *argument:parameter*. It is also important to distinguish an *argument* from the *expression* which is used to compute the *argument*. This *expression* is the one that occurs in the *argument:position* of the *form* that calls the *function*, not in the *function* itself and is called an *argument:expression*. The following example should make this clear.

Consider for example the *function* DIAG which is defined to compute the diagonal of a rectangular prism given the three dimensions of the prism.

```
FUNCTION DIAG(X,Y,Z) HYPOTENUSE(HYPOTENUSE(X,Y),Z);
```

Now suppose that we evaluate the *expression* DIAG(3,4,12). The *arguments* of DIAG are 3, 4 and 12, and these correspond to the *argument:parameters* X, Y and Z, respectively. The inner call to HYPOTENUSE must be performed first in order to obtain a necessary *argument* for the outer call. The *argument:expressions* are X and Y; these are evaluated to obtain the *arguments* , which are 3 and 4. The *arguments* are what are transmitted to HYPOTENUSE. Once HYPOTENUSE has been called, the *variables* X and Y are no longer relevant--only the *values* 3 and 4 obtained from this evaluation are relevant.

Within the body of HYPOTENUSE, the *arguments* are available as the *values* of the *argument:parameters* SIDE1 and SIDE2. The *argument:parameters* X and Y of DIAG have no meaning within the body of HYPOTENUSE. They are bound to the *values* 3 and 4 only within the body of DIAG.

The *value* of the inner call to HYPOTENUSE is 5.0. So the *arguments* for the second call to HYPOTENUSE are 5.0 and 12 respectively. The first *argument* 5.0 was obtained by the evaluation of the *expression* HYPOTENUSE(X,Y). The second *argument* was obtained from DIAG as the *value* of the *variable* Z.

Similar remarks apply to the second call to HYPOTENUSE. The *bindings* of SIDE1 and SIDE2 this time are 5.0 and 12 respectively.

The *value* of DIAG(3,4,12) is 13.0

Note: This description of *argument* evaluation and transmission applies to *arguments* transmitted by *value* only. The other alternative in LISP known as transmission by location is treated in a later chapter. *Arguments* are always transmitted by *value* unless specified otherwise. You may ignore this distinction for the present.

Problem Set 19.

In this problem set, several *function:definitions* are given, and a table of *bindings* for *free:variables* is given. The problem is to evaluate the *expressions* that follow, using the *function:definitions* and the table of *variable:bindings* when necessary.

When a *variable* occurs within the body of a *function*, and this *variable* is an *argument:parameter* of the *function*, the proper *binding* for the *variable* is the *argument* corresponding to its use as an *argument:parameter*. Only if you cannot obtain a *binding* for a *variable* in this way, make use of the table of *variable: bindings*.

function:definitions:

```
FUNCTION POLY(X); 2*X↑2+3*X-5;
FUNCTION CHOOSE(X,Y) IF X=0 THEN Y ELSE Y-X;
FUNCTION TAKE(X,Y) IF ATOM X THEN Y ELSE IF ATOM Y THEN NIL ELSE CAR
  X . CDR Y;
```

```
FUNCTION MAKE(X) ; X . Z;
```

Table of *bindings:*

Variable	*Binding*
U	'A
X	3
Z	7

Expressions to be evaluated:

a. POLY(3)

b. POLY(Z)

c. CHOOSE(1,-4)

d. CHOOSE(POLY(Z)-114,X)

e. MAKE(U)

f. TAKE(U,Z)

g. LIST(U, TAKE(X . Z, IF POLY(1)<1 THEN '(D E) ELSE '(F G))

Answers: See pages 157, 158

7.5 RECURSION

We shall give three examples of definition by recursion; the first is numerical, the second is symbolic, and the third has an *argument* which is a *list*, and gives an *integer:value*.

The important thing to keep in mind is that the *argument:parameters* of a *function* generally have different *bindings* each time that the *function* is called.

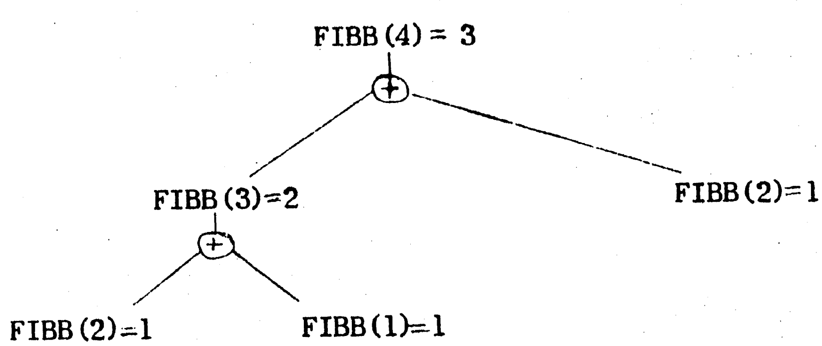
7.5.1 EXAMPLE 1: THE FIBONACCI SERIES

The Fibonacci series is a sequence of integers. The first two terms are 1 and 1, respectively. After that, each term of the series is the sum of the preceding two terms. The Fibonacci sequence begins therefore 1, 1, 2, 3, 5, 8, 13, 21, ...

The *function* FIBB defined here gives the nth term of the sequence.

```
FUNCTION FIBB(N) IF N=1 THEN 1 ELSE IF N=2 THEN 1 ELSE FIBB(N-1)+FIBB(N-2);
```

Suppose we evaluate FIBB(4). The definition tells us that FIBB(4) is FIBB(3)+ FIBB(2). FIBB(3) is defined to be FIBB(2)+FIBB(1). The computations of FIBB(1) and FIBB(2) are not recursive and yield the *values* 1 and 1 immediately. The evaluation of FIBB(4) is shown schematically in the following diagram:



Recursive definitions do not always terminate. For example, the computation of FIBB(0) according to the above definition will never terminate. The computation continues with the depth of recursion getting deeper and deeper until lack of computer memory or lack of time causes an error condition in the computer.

There is no general rule possible for determining whether a recursive computation will terminate or not. Therefore, the programmer must understand the particular type of recursion he is using and why he expects the recursive computation to terminate on the type of data being operated on. This understanding can be acquired with practice. The exercises in this Primer provide a start in this direction.

The left-to-right sequence for evaluating *conditional:expressions* is essential for the *recursive:definition* to operate properly. For example, consider the evaluation of FIBB(1). Substituting 1 for N in the body of the definition gives:

```
IF 1=1 THEN 1 ELSE IF 1=2 THEN 1 ELSE FIBB(0)+FIBB(-1)
```

If all the parts of the *conditional:expression* had to be evaluated first, before a choice between the parts was made, then the computation would not terminate, and so no *value* could be obtained for it.

7.5.2 EXAMPLE 2: SUBSTITUTION

Suppose we want to substitute a given *S-expression* for each instance of a given *identifier* in another *S-expression*. The *function* SUBST does this. We define SUBST(X,Y,Z) as the result of "Substitute the ***S-expression*** x for all occurrences of the *identifier* y in the *S-expression* z." An example is:

```
SUBST('(THE TREE),'OBJECT,'((THE MAN) SAW OBJECT)) is ((THE MAN) SAW (THE TREE))
```

The definition of SUBST in LISP 2 is:

```
FUNCTION SUBST (X,Y,Z) IF ATOM Z THEN (IF Z=Y THEN X ELSE Z) ELSE
            SUBST(X,Y,CAR Z) . SUBST(X,Y,CDR Z);
```

Another example is SUBST('Q,'B,'((A B) B C)). The *value* is ((A Q) Q C). This is demonstrated in painstaking detail by the following account of the 11 calls to SUBST necessary to complete this computation.

```
                                (2)                         (7)
 1.  SUBST('Q,'B,'((A B) B C)=SUBST('Q,'B,'(A B)) . SUBST('Q,'B,'(B C))='((A Q) Q C)
                         (3)                (4)
 2.  SUBST('Q,'B'(A B))=SUBST('Q,'B,'A) . SUBST('Q,'B,'(B))='(A Q)

 3.  SUBST('Q,'B,'A)='A
                       (5)               (6)
 4.  SUBST('Q,'B,'(B))=SUBST('Q,'B,'B) . SUBST('Q,'B,NIL)='(Q)

 5.  SUBST('Q,'B,'B)='Q

 6.  SUBST('Q,'B,NIL)=NIL
                         (8)               (9)
 7.  SUBST('Q,'B,'(B C))=SUBST('Q,'B,'B) . SUBST('Q,'B,'(C))='(Q C)

 8.  SUBST('Q,'B,'B)='Q
                        (10)              (11)
 9.  SUBST('Q,'B,'(C))= SUBST('Q,'B,'C) . SUBST('Q,'C,NIL)='(C)

10.  SUBST('Q,'B,'C)='C

11.  SUBST('Q,'B,NIL)=NIL
```

It is interesting to note that the *argument:parameter* Z is bound to many different *arguments* in the 11 calls to SUBST, but that the *argument:parameters* X and Y do not change. This is a fairly common occurrence.

7.5.3 EXAMPLE 3: LENGTH OF A *LIST*

The length of a *list* is equal to the number of elements in the *list*. For example, the length of the *list* (A 4 (B C)) is 3 because there are 3 elements in the *list* (the substructure of the element (B C) is irrelevant). The length of the empty *list* ( ) is 0. The definition of LENGTH is:

```
FUNCTION LENGTH(L) IF NULL L THEN 0 ELSE LENGTH(CDR L)+1;
```

The evaluation of LENGTH('(A 4 (B C))) proceeds as follows:

```
LENGTH('(A 4 (B C)))=LENGTH ('(4 (B C)))+1
=LENGTH('((B C)))+1+1=LENGTH ('())+1+1+1
=0 + 1 + 1 + 1=3
```

Problem Set 20

a. The following definition of FIBB uses an auxiliary *function* FIBB1. It gives the same answers as the definition in Example 1. Why does this definition lead to more efficient computation of FIBB for large *arguments*?

```
FUNCTION FIBB(N); FIBB1(N,1,2);
FUNCTION FIBB1(X,Y,Z) IF X=1 THEN Y ELSE FIBB1(X-1,Z,Y+Z);
```

b. Is there any set of *arguments* for which SUBST, as defined in Example 2, does not converge? Why or why not?

c. Define the recursive *function* COUNT having one *argument*. The *argument* may be any *S-expression*. The *value* of COUNT is the number of *atoms* (not just *identifiers*) in the *argument*.

Answers: See pages 158, 159

CHAPTER 8

THE *LOGICAL:OPERATORS*

The six *logical:operators* of LISP 2 are AND, OR, NOT, IMPLIES, XOR, and EQUIV. They may be regarded as *functions* whose *arguments* are *boolean* and whose *value* is also *boolean*. But some of them (AND, OR, IMPLIES) differ in an important way from *functions*. These three operators have the property that their *arguments* are evaluated from left to right, and that only as many *arguments* as are necessary to determine the *value* of the *boolean* are evaluated. In this respect, they are more like *conditional:expressions* than *functions*.

8.1 NOT

The *boolean* NOT has one *argument*. The *value* of NOT is TRUE if its *argument* is FALSE (or NIL or ()), and FALSE (or NIL or ()) if its *argument* is anything else. As with *conditional:expressions*, any *argument* except FALSE is regarded as equivalent to TRUE.

The *expression*

```
NOT e
```

is equivalent in meaning to the *conditional:expression*

```
IF e THEN FALSE ELSE TRUE
```

NOT is a *prefix:operator*; therefore it is permissible to write either

```
NOT (e)
```

or

```
NOT e
```

The precedence of NOT is highest of the *logical:operators*.

The *operator* NULL is identical with NOT both in meaning and in precedence.

8.2 AND

The *operator* AND has an indefinite number of *arguments*. It is either a *prefix:operator* or an *infix:operator*: one may write either

AND(e_1,...,e_n)

or

e_1 AND...AND e_n

The precedence of AND is below that of NOT but higher than that of the other four *logical:operators*.

The

e_1 and e_2 .... AND e_n

is equivalent in meaning to the *expression*

IF NOT e_1 THEN FALSE IF NOT e_2 THEN FALSE ....ELSE e_n

In other words, the

e_1 AND ... AND e_n

has the *value* TRUE if each e_1 is evaluated and the *values* are all true (not FALSE), but if the evaluation of any e_1 is FALSE, then the *value* of the entire *expression* is FALSE, and the remaining e_i to the right of this one are not evaluated.

AND ( ) (meaning AND of no *arguments*) has by convention the *value* TRUE.

8.3 OR

The *operator* OR has an indefinite number of *arguments* and it is either an *infix* or *prefix:operator*. One may write either

OR(e_1,...,e_n)

or

e_1 OR ... OR e_n

The precedence of OR is fourth of the *logical:operators:* below NOT, AND, XOR; above IMPLIES and EQUIV.

The *expression*

e_1 OR ... OR e_n

is equivalent in meaning to the *expression*

IF e_1 THEN TRUE ELSE IF e_2 THEN TRUE ... ELSE e_n

In other words, the *expression*

e_1 OR ... OR e_n

has the *value* TRUE if at least one e_i has a true *value*. In this case, the remaining e_i to the right of this one are not evaluated. If all of the e_i evaluate to FALSE, then the *value* of the entire *expression* is FALSE.

OR ( ) (meaning OR of no *arguments* has by convention the *value* FALSE.)

8.4 EXAMPLE

As an example of the use of the *logical:connectives,* we shall give another definition of MEMBER:

```
FUNCTION MEMBER(X,L) NOT NULL L AND (X=CAR L OR MEMBER(X,CDR L));
```

The recursion in this definition terminates only because AND and OR have the property of not evaluating *arguments* further to the right of the one that determines their *value*.

The *parentheses* around the OR *expression* are necessary because AND has a higher precedence than OR, and if the *parentheses* were missing, then AND would capture X=CAR L as its *argument* on the right.

Problem Set 21.

(1) Insert *parentheses* in the following LISP 2 *expressions* in such a way that they are unambiguous assuming no rules of precedence.

(2) Evaluate the following *expressions* using the table:

Variable	*Value*
A	TRUE
B	( )
C	7.0
X	A
Y	(3 4)
Z	(A B)

a. CAR Y + CADR Y=C AND A

b. B AND 2+2=4

c. A OR 2+2=5

d. NOT A OR B OR X=Y

e. IF A OR B THEN C

f. IF C THEN C ELSE 'C

g. NOT (A AND B)

h. NOT A AND B

Answers: See pages 160, 161

8.5 IMPLIES

IMPLIES is a *binary:operator*. It may be written either as a *prefix:operator* as in

IMPLIES(e_1, e_2)

or as an *infix:operator* as in

e_1 IMPLIES e_2

For those who are logicians, the meaning of IMPLIES is almost "material implication."

For those who are not logicians, the meaning of IMPLIES is almost the meaning according to the following table of cases:

e_1	e_2	e_1 IMPLIES e_2
False	False	True
False	True	True
True	False	False
True	True	True

We say "almost" because in LISP 2 the evaluation procedure does not evaluate e_2 unless e_1 is true. This evaluation procedure is different from the evaluation procedure in logic.

The evaluation procedure for IMPLIES is the following:

e_1 is evaluated. If its *value* is FALSE, then the *value* of

e_1 IMPLIES e_2

is TRUE. Otherwise, e_2 is evaluated, and its *value* is the *value* of the entire *expression*.

e_1 IMPLIES e_2

is thus equivalent in meaning to the *conditional:expression*

IF e_1 THEN e_2 ELSE TRUE

IMPLIES has next to the bottom precedence of the *logical:operators*.

8.6 XOR

XOR has an indefinite number of *arguments*. It may be written as

XOR(e_1,e_2... ,e_n)

or as

e_1 XOR e_2 ... XOR e_n

Unlike AND, OR, and IMPLIES, XOR evaluates all of its *arguments* in no specified order. If the number of *arguments* that are true is odd, then the *value* of XOR is TRUE; otherwise the *value* of XOR if FALSE. XOR has third rank in the precedence of the *logical:operators*.

8.7 EQUIV

EQUIV has an indefinite number of *arguments*. It may be written as

$$\mathrm{EQUIV}(e_1 \ldots e_n)$$

or as

$$e_1 \text{ EQUIV } \ldots \text{ EQUIV } e_n$$

It has lowest precedence of the *logical:operators*.

All of the *arguments* of EQUIV are evaluated in no specified order. The *value* of EQUIV is TRUE if all of its *arguments* are true, or if all of its *arguments* are FALSE. In any other case, the *value* of EQUIV is FALSE.

CHAPTER 9

BLOCK:EXPRESSIONS AND *STATEMENTS*

So far we have described how to write LISP *programs* using recursive *function: definitions*. It can be proved that any computation can be described by recursive *function:definitions*; however, often it is easier to describe a computation in some other way. We need, in addition to recursion, a way of writing a series of *statements* that perform certain operations, and a way of controlling the order in which those *statements* are executed.

For a concrete example of this point, see the two different ways given in Chapter 3, Section 3.2 for defining the *function* FACTORIAL. The first definition uses *statements;* the second definition uses recursion. The first method, although longer to write, compiles into a smaller and faster-running program. Most old-time LISP programmers however prefer the second method, recursion, which is mathematically more elegant, and is an important distinguishing feature of all LISP systems.

9.1 *BLOCK:EXPRESSIONS*

For developing the second method, two new kinds of entities that are not *expressions* are needed--*declarations* and *statements*. *Statements* are described fully in this chapter, but *declarations* are described only briefly here; they are described more fully later.

A context is needed in which *statements* and *declarations* can occur. The *block:expression* provides such a context. It is a special kind of *expression* that contains *declarations* and *statements* inside it.

Definition:

A *block:expression* has the form

BEGIN d_1; ... ; d_m;s_1; ... ;s_n END

In this form each d_1 is a *declaration,* and each s_1 is a *statement*. Either m or n may be 0; that is, there may be no *declarations* or no *statements* or both. All the *declarations* must precede all the *statements* in a *block:expression*. The *declarations* and *statements* are separated from each other by *semi:colons;* there is one less *semi:colon* than the total number of *declarations* and *statements*.

9.2 DECLARATIONS

There are several kinds of *declarations;* one kind of *declaration* that is suitable in this context is known as the *internal:parameter:declaration* .

Definition:

An *internal:parameter:declaration* may have one of the following forms (there are others):

SYMBOL v_1, ... , v_n

or

INTEGER v_1, ... , v_n

or REAL v_1, ... , v_n

or

BOOLEAN v_1, ... , v_n

where each v_1 is a *variable*. The four words in capital letters denote the *data:type* of the *variable*.

If there are two or more *variables* following the word SYMBOL (or INTEGER or REAL or BOOLEAN), then they are separated from each other by *commas*. An *internal:parameter:declaration* is almost always followed by a *semi:colon* since another *declaration* or a *statement* is to follow; however, the *semi:colon* is not regarded as being part of the *declaration*.

Example of a *block:expression* with *internal:parameter:declarations:*

```
BEGIN REAL X,Y; INTEGER Z; SYMBOL A1, A2; ... END
```

where ... represents some *statements* .

The *internal:parameter:declaration* has the following effects on the *program*:

(1) The *variables* mentioned in the *declaration* are declared to be *internal:parameters* which can be referenced throughout the *block:expression* (or *block*) in which the *declaration* occurs. One may refer to a *variable* either to obtain its *value* or to change its *value*. Thus the *internal:parameters* may be used as storage places for data.

(2) If an *internal:parameter* is declared to be of type SYMBOL, then its *value* may be any type of datum. (That is, any type of datum may be stored in it.) If the *internal:parameter* is of type INTEGER, REAL or BOOLEAN, then its *value* may be only a datum of the specified type.

(3) As soon as the *block* is entered, the *internal:parameter* is assigned an initial *value*. Of course, this initial *value* may be changed almost immediately by what the programmer writes, and it may be ignored entirely. The initial *value* depends upon the type of the *variable* as follows:

Type	Initial *Value*
SYMBOL	NIL
INTEGER	0
REAL	0.0
BOOLEAN	FALSE

9.3 *STATEMENTS*

The *statements* within a *block* are normally executed in sequence starting with the first one. The sequence in which *statements* are executed may be controlled by several means; the simplest of these is the

go:statement

The kinds of statements which will be described in this chapter are:

assignment:statements

conditional:statements

go:statements

empty:statements

return:statements

simple:statements

Some more kinds of *statements* are described later.

9.4 *ASSIGNMENT:STATEMENTS*

The *assignment:statement* is a *statement* that causes a *value* to be assigned to a *parameter*. The *assignment:statement* has the form

v←e

where v is a *variable* and e is an *expression*.

The *expression* e is evaluated first; then its *value* is stored in the *variable* v. The previous *value* of v is lost at that point.

For example, suppose A has the *value* 5, and one executes the *assignment:statement* A←A↑2. The *expression* on the right is evaluated with A having the *value* 5. The *value* of the *expression* is 25. This is now assigned as the new *value* of A. The old *value* of A is lost.

An *assignment:statement* occurring inside the body of a *function:definition* may change the *value* of an *argument:parameter* (see below) instead of changing the *value* of an *internal:parameter*. This change remains in effect throughout the evaluation of the *function*.

An *assignment:statement* may be used as an *expression*, in which case it is called an *assignment:expression*. The *assignment:expression* has the same effect as the *assignment:statement*, but the *assignment:expression* also has a *value*. The *value* of an *assignment:expression* is the *value* of its right half.

Example (of an *assignment:statement*):

A←B←X↑2+3

The portion of this *assignment:statement* to the right of the first *left:arrow* is an *assignment:expression* B←X↑2+3. The effect of this *statement* is to assign the *value* x^2+3 to both A and B.

The *left:arrow* behaves somewhat as if it were an *infix:operator*, but a rather peculiar one. On the left, it has high precedence. It grasps the smallest possible *expression* it can find. On the right, it has very low precedence, lower even than the LISP *dot*. It grasps as much as possible.

Example:

```
A←CAR C←D . E
```

means the same as:

```
A←(CAR(C←(D . E)))
```

In other words, this *expression* CONSes D and E and puts the result in C. It then takes CAR of this which is D again, and puts this in A.

9.5 THE *CONDITIONAL:STATEMENT*

A *conditional:statement* is like a *conditional:expression*; the only difference is that its *consequents* are *statements* rather than *expressions*.

Definition:

A *conditional:statement* has one of the following forms:

IF p_1 THEN e_1

or

IF p_1 THEN e_1 ELSE e_2

where p is an *expression*, e_1 is a *basic:statement* (see below), and e_2 is any *statement*.

A *basic:statement* is any kind of *statement* except a *conditional:statement* or a *for:statement* (which is explained later). The restriction that a *statement* must be *basic* is trivial and intended only to avoid certain kinds of ambiguity. A *conditional:statement* enclosed by BEGIN ... END, is changed into a *basic:statement*. Examples of *conditional:statements*:

```
IF A=0 THEN GO L
IF P THEN A←A+1 ELSE A←A-1
IF A<B THEN GO M ELSE IF A>B GO N ELSE IF B=0 GO L
IF A THEN BEGIN IF B THEN X←1 ELSE X←2 END ELSE GO L
```

The following rules apply to the execution of *conditional:statements*.

(1) The *antecedents* are evaluated from left to right until one is found whose *value* is TRUE (or in fact, any datum other than FALSE).

(2) When an *antecedent* is found that is true, the corresponding *consequent* is executed. The rest of the *conditional:statement* is ignored.

(3) If a *conditional:statement* ends in ELSE s_n, and if all the preceding *antecedents* are false, then s_n is executed.

(4) If a *conditional:statement* ends in IF p_n THEN s_n and if all the *antecedents* including p_n are false, then nothing is executed, and the program proceeds in the normal manner. This <u>is not</u> an error condition in contrast to the analoguous situation for *conditional:expressions*.

(5) *Conditional:statements* are not *expressions*; therefore they <u>never</u> have *values*.

9.6 *LABELS*

A *label* is a means of giving a name to a *statement*. *Identifiers* are used as *labels*.

Definition: A *labeled:statement* has the form

```
lb:s
```

where lb is a *label* and s is a *statement*.

Examples of *labeled:statements*:

```
A:  IF X=Y THEN GO A

B:  X←X+1

C:  GO A
```

The kind of a *statement* is not changed by labeling the *statement*. Thus the first *statement* above is a *conditional:statement,* whether labeled or not.

9.7 *GO:STATEMENTS*

The *go:statement* has the form

```
GO lb
```

where lb is a *label*.

The effect of a *go:statement*, GO lb, is to cause execution of the *program* to continue at the *statement* labeled lb; the *program* proceeds from there in the normal way.

There are certain restrictions as to where in a *program* it is possible to go from a given location. These restrictions follow common sense and exclude cases where the execution of a *go:statement* could be poorly defined. They will be discussed later. The following interesting example is quite permissable however:

```
GO A;

.
.
.

IF X=0 THEN Y←2*Y ELSE A; IF X<=0 THEN Y←Z;
```

If the *go:statement* is executed and if X=0 at the time, then Y←Z will be executed. If one started at the beginning of the *conditional:statement* with X=0, X←2*Y would be executed.

9.8 THE *VALUE* OF A *BLOCK:EXPRESSION; RETURN:STATEMENTS*

A *block:expression* must have a *value* because it is an *expression*. *Block:expressions* may obtain *values* in two different ways.

The first way occurs when the *block:expression* ends because it has run out of *statements* to execute. This happens when the last *statement* has been executed and is not a *go:statement*. The word END follows, but is not a *statement*. In this case, the evaluation of the *block:expression* is terminated and the *value* is NIL. This is the usual way of ending a *block:expression* when the *value* is not being used for any purpose.

Sometimes,however,the last *statement* in a *block:expression* is a *go:statement*. To get out of the *block*, one needs to branch to some point after this *statement*. The *empty:statement* is useful for this purpose. For example, here is a *block:expression* with an *empty:statement* used as a way out:

```
BEGIN ...   IF TERMINALCONDITION THEN GO B; ... GO A; B:; END
```

An *empty:statement* is specified by two consecutive *semi:colons* with no *statements* between them. Since a *label* is not a *statement* it may intervene as in the above example. The *empty:statement* is here represented by:

; B: ;

The second way to obtain a *value* for a *block:expression* is to use a *return::statement*.

Definition: A *return:statement* has the form

RETURN e

where e is an *expression*.

A *return:statement* may occur in any *statement* context within a *block:expression*; for example, it may appear as one of the *consequents* of a *conditional:statement*. Also there may be several *return:statement* within one *block:expression*. As soon as one of them is executed, the following happens:

(1) The *expression* e is evaluated

(2) The *block:expression* is terminated. No further *statements* are executed no matter where one is in the *block*.

(3) The *value* of e is the *value* of the entire *block:expression*.

9.9 SIMPLE:STATEMENTS

A *simple:expression* may be used as a *statement*, in which case it is called a *simple:statement*. The only way to tell that it is a *statement* is the context in which it appears. A *simple:statement* always occurs in a context which has the property that even if the *simple:expression* were to produce a *value*, the *value* would be ignored.

Since the *value* of a *simple:statement* is ignored, the only reason for executing it is to produce an effect, for example, PRINT(X).

9.9.1 EXAMPLE AND PROBLEMS

Example: Define REV which is a *function* that reverses a *list* and all its *sublists* . Thus,

```
REV ('((A B C) (D E)))
```

is

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

Here is a definition of REV:

```
FUNCTION REV(X) BEGIN SYMBOL Y;
    A:  IF NULL X THEN RETURN Y ELSE IF ATOM X THEN RETURN X;
        Y← REV(CAR X) . Y ;
        X←CDR X;
    END ;
```

This example has several interesting features:

(1) This definition uses both recursion and iteration of a loop of *statements*--the two most important means of controlling a repetitive process. Recursion is used to apply the *function* REV to *sublists* at all levels. But the job of reversing any one level is done by means of an open loop of *statements*.

(2) It illustrates the use of an *assignment:statement* to set the *argument:parameter* X, and another *assignment:statement* to set the *internal::parameter* Y.

(3) Each time the *function* REV is entered recursively, a new *argument: :parameter* X and a new *internal:parameter* Y are created. The different copies of X and Y have independent *value* and do not interfere with each other. Only the innermost X and Y are available at any given time, but when a particular recursion is terminated, the immediately previous X and Y are accessible once more, unchanged from when they were last accessible.

Problem Set 22.

a. Define REV using recursion and without using *block:expressions* and *statements*.

b. Define REV by means of a single non-recursive *function:definition* using *block:expressions* and *statements*.

c. Define the LISP *function* SINE(X,N) that computes an approximation to the sin of X by summing the first N terms of the sequence

$$\sin(x)=x/1!-x^3/3!+x^5/5!-x^7/7!....$$

(Do not use the LISP system *function* SIN.)

Answers: See pages 162, 163

CHAPTER 10.

BLOCKS

Many entities in LISP 2 can be classified into three kinds; *expressions*, *statements*, or *declarations*. This distinction is important and needs to be mastered by a user of the language. To some extent, these kinds of entities are like interrogative, imperative, and declarative sentences, respectively, in English. However, this analogy cannot be carried too far.

An *expression* in LISP can be evaluated; that is, it has a *value* which can be computed. For example, the *value* of 3+4 is 7. In the same way, an interrogative sentence in English can be answered; that is, it has an answer or calls for an answer.

A *statement* in LISP is a request or command that some process be performed. For example GO J is a *statement* requesting execution of the process beginning at J. In the same way, an imperative sentence in English is a request or command that some action be performed or that some state exist; that is, it calls for some action to be performed or for some state to exist. For example, "Give me that list" or "Be careful."

A *declaration* in LISP informs the computer of some fact or condition. For example, REAL M, says that there will be an *internal:parameter* in the *program* and that it will have *real:values*. In the same way, in English, a declarative sentence (also called an indicative sentence) tells or provides information. For example, "M will be a variable in this program, with real values."

Both in LISP and in English the classification is sometimes more nominal than actual, and is determined more by the way in which an entity occurs in its surroundings (by the grammar and syntax) than by meaning (the semantic context). For example, in LISP the evaluation of an *expression* may not only yield a *value* but cause certain other things to happen. These are called *side:effects*. Similarly in English, a sentence which is interrogative in form may be declarative in substance. For example, the interrogative "Why isn't the butter on the table?" may mean the imperative "Please put the butter on the table"; the speaker is not really interested in knowing why the butter is not on the table. Another example is the interrogative: "How much more of this nonsense do I have to listen to?" This means the declarative "I don't want to listen to any more of this because I consider it to be nonsense." The speaker does not want to be answered "About 15 minutes more nonsense."

Both in LISP and in English one can argue in favor of linguistic purity. But impure use of the language will remain and spread because it is often convenient and direct, and often economical.

In LISP, it is always possible to classify entities into *expressions*, *statements*, and *declarations*, by analyzing the syntax. But it is not always possible to do this by examining a single entity. Usually one must consider the context in which it appears. Thus we shall be referring to a *statement:context* and an *expression:context*. (In LISP, *declarations* present no problem. They can always be distinguished by their first words regardless of context.)

What contexts have been encountered so far? One is the context of the body of a *function:definition*. This is always an *expression*. Therefore, whatever appears after the heading of a *function:definition* is in an *expression:context*. At the top level of a LISP *program*, one may write *expressions* but not *statements*. Therefore, this is an *expression:context*.

10.1 BLOCKS

A *block* has precisely the following syntax:

BEGIN d_1; d_2; ... d_m; s_1; s_2; ... s_n END

where each d_1 is a *declaration* and each s_1 is a *statement*. Either m or n or both could be 0.

A *block* consists of the *reserved:word* BEGIN, followed by some *declarations*, followed by some *statements*, followed by the *reserved:word* END. All the *declarations* in a *block* come before any of the *statements*. The *declarations* and *statements* are separated from each other by *semi:colons*.

How is a *block* to be classified? If a *block* appears in an *expression:context* then it is an *expression*, and specifically it is called a *block:expression*. If a *block* appears in a *statement:context*, then there are two possibilities. If it has no *declarations*, then it is called a *compound:statement*; if it has one or more *declarations*, then it is called a *block:statement*. This classification is summed up in the following table:

CLASSIFICATION OF BLOCKS

Context:	*expression:context*	*statement:context*
Declarations:		
none	*block:expression*	*compound:statement*
at least one	*block:expression*	*block:statement*

A *block:expression* may be used on any level as the body of a *function:definition*, or it may be used on the top level as an *expression*. Within the *block:expression* there may be *statements* (including *block:statements* and *compound:statements*). From this specification, it follows that when *blocks* appear nested one within the other as in:

```
BEGIN ...                                      ... END
           BEGIN ...                  ... END
                      BEGIN ... END
```

the outermost one, at least, must be a *block:expression*.

10.2 *VARIABLES, BINDINGS,* AND *SCOPES*

A *variable* (to repeat what was said earlier) is an *identifier* used within a *program* to denote some *value*. For example, the *variable* M may turn out to have the *value* 4.

A *variable* may be mentioned in any one of four ways. It may be mentioned in order to *bind* it either as an *argument:parameter* (see Chapter 7) or as an *internal: parameter* (see Chapter 9). It may be mentioned for the purpose of changing it. It may be mentioned for the purpose of making use of its *value*. This is summarized in the following table:

MENTIONS OF *VARIABLES*

Type of Mention	Example
to *bind* it as an *argument:parameter*	FUNCTION FN(X) ...
to *bind* it as an *internal:parameter*	INTEGER X;
to change it	X← 3;
for its *value*	X + 3;

Every *binding* of a *variable* has associated with it a *scope*. The *scope* is a region of *program* within which that particular *binding* of a *variable* may be referenced either to change the *variable* or to evaluate it. The *scope* must be thought of as something dynamic: it starts to exist when it is activated, and it stops existing when some fixed piece of a *program* is finished.

Rule 1: When a *variable* is *bound* as an *argument:parameter* of a *function*, the *scope* of the *binding* is the body of the *function:definition* (but not including the *scope* of any other *binding* of the same *variable* that is inside the first *binding*). The *scope* exists as soon as the *function* is entered, and ceases to exist when the *value* of the *function* has been computed and control returns to the point from which the *function* was called.

Example:

```
FUNCTION FN(X) 3*X+5;
FN(2);
```

The *variable* X is *bound* as an *argument:parameter*. The *scope* of the *binding* is the body of the *function:definition*, namely 3*X+5. However, merely making a *function:definition* does not activate the *scope*. When the *function* FN is called with the *argument* 2, then the *binding* of X is activated, and throughout its *scope* it has the *value* 2.

Rule 2: When a *variable* is *bound* as an *internal:parameter*, the *scope* of the *binding* is all the *statements* (but not the *declarations*) of the *block* in which the *declaration* is made, but not including the *scope* of any other *binding* of the same *variable* inside the first *binding*. The *scope* of the *binding* exists just prior to the execution of the first *statement* of the *block*, and continues until the *block* is left.

first *binding* as an *argument:parameter*

Example 1: second third *binding* as *internal:parameter*

```
FUNCTION  G(X)  BEGIN H(X); BEGIN REAL X; H(X); BEGIN INTEGER (X);
                |__________1___________| |__________2___________|
     H(X) END END END ;
     |__3___||_2_||_1_|
```

1 , *scope* of first *binding*

2 , *scope* of second *binding*

3 , *scope* of third *binding*

Example 2: *binding* as *argument:parameter*

```
FUNCTION FN(X)  3*X+5;
```

scope as *argument:parameter*

```
BEGIN INTEGER X; X← 6; PRINT (FN(X+1)) END;
```

binding as *internal:parameter* *scope* as *internal:parameter*

Let us repeat that the definition of a *function* defines the *scope* of its *argument::parameter*, but does not activate it. The entity that follows the *function:definition* FN is a *block:expression*. It has an *internal:parameter* X. The execution of the *block* activates the *binding* of X. At first, X has the *value* 0, but this is immediately changed to 6. The *expression* X+1 is then evaluated. This happens before the *function* FN is called. The *value* of this *expression* is 7. The *function* FN is called with the *argument* 7. At this point, the *argument:parameter* X is activated and has the *value* 7. The *value* of FN (which gets printed) is 26, and not 23.

Example 3:

```
BEGIN SYMBOL X; X ← 'A; BEGIN SYMBOL X; X ← 'B; PRINT (X) END;PRINT (X) END;
```

If you concluded that B would be printed first and then A, the conclusion was correct, and your analysis was probably correct.

Each *binding* must be regarded as having an independent existence. When the second *binding* is activated, the first one continues to exist but within the *scope* of the second *binding* it cannot be referenced. When the *scope* of the second *binding* ends, the first one still exists and has not been changed.

A *declaration* such as

```
REAL X;
```

may be made at the top level of a LISP *program*. In this case, the *variable* and its associated *value* exist indefinitely.

10.3 *RETURN:STATEMENTS*

A *return:statement* is of the form RETURN w. The *return:statement* must be used inside a *block:expression*. The effect of the *return:statement* is to terminate a *block:expression* and cause the *block:expression* to take the *value* of w. If two *block:expressions* are nested, then the execution of a *return:statement* that is inside both of them terminates only the innermost one. However, when a *block:statement* or *compound:statement* is nested inside a *block:expression*, control passes outward through these and the *block:expression* that is outside them is terminated. The *reserved:word* RETURN always terminates a *block:expression*.

Example 1:

```
FUNCTION FN(X) BEGIN BEGIN RETURN X END END;
```

The inner *block* is a *compound:statement*. The outer *block* is a *block:expression*. The RETURN terminates the outer *block* and X is the *value* of FN(X). So the *function:definition* defines an *identity:function*.

Example 2:

```
FUNCTION FN(X) BEGIN ATOM BEGIN RETURN X END END;
```

The inner *block* is a *block:expression*, because it is the *argument* of ATOM, and *arguments* are always *expressions*. ATOM BEGIN RETURN X END is a *simple: statement*. Its *value* is true if X is atomic, but this is irrelevant. There are no further *statements* in the outer *block*, and no RETURN from it. So the *value* of FN is always NIL.

10.4 RESTRICTIONS ON *GO:STATEMENTS*

There are certain restrictions on the use of *go:statements*. The rules are:

(1) A *go:statement* may not be used to enter a *block:statement* from a point outside it.

(2) A *go:statement* may not be used either to go into an *expression* from a point outside it or to go out of an *expression* from a point inside it.

These rules have the following consequences for *blocks*.

GO:STATEMENT RESTRICTIONS

Type of *Block*	May Enter?	May Leave?
block:expression	no	no
compound:statement	yes	yes
block:statement	no	yes

If one were to enter a *block:statement* by means of a *go:statement*, this would put the *internal:parameters* of the *block:statement* into an ambiguous condition. Since a *compound:statement* has no *internal:parameters* specific to it, the problem does not arise there.

The body of a *function:definition* is an *expression*; therefore one may not enter or leave the body of a *function:definition* by means of a *go:statement*.

Problem Set 23:

Examine the *statement* GO A in each of the following miniature *programs* and decide whether or not it is legal, and why or why not.

a.
```
FUNCTION FN(X) BEGIN A: RETURN G(X) END;
FUNCTION G(X) BEGIN GO A END;
```

b.
```
BEGIN INTEGER Y;
        BEGIN REAL X; GO A END;
        BEGIN A: Y← 3 END
END
```

c.
```
BEGIN INTEGER Y;
        BEGIN GO A END;
        BEGIN REAL X; A: Y← 3 END
END
```

d.
```
BEGIN INTEGER Y;
        BEGIN GO A END ;
        BEGIN A: Y←3 END
```

e.
```
BEGIN GO A; FN(BEGIN A: ; RETURN X END) END
```

f.
```
BEGIN -BEGIN GO A END; BEGIN A:;END END
```

Answers: See pages 164, 165

10.5 TYPICAL USES FOR *BLOCKS*

(1) A *compound:statement* groups several *statements* together for execution one after another. One use of this technique is as a *consequent* of a *conditional: statement* when several things are to be done if a condition is satisfied.

Example:

```
IF X=0 THEN BEGIN Y←5; GO A END;
```

Without *compound:statements*, one would have to use a circumlocution (or "program around it") such as:

```
IF X/=0 THEN GO B; Y←5; GO A; B:
```

(2) A *conditional:statement* cannot be used as the *consequent* of another *conditional:statement* following the word THEN. This restriction can be overcome by turning the first *conditional:statement* into a *compound:statement* with one *statement* inside it.

Incorrect:

```
IF A THEN IF B THEN GO X ELSE GO Y ELSE GO Z;
```

Correct:

```
IF A THEN BEGIN IF B THEN GO X ELSE GO Y END ELSE GO Z;
```

(3) A *block:expression* is commonly used as the body of a *function:definition* when the *value* of the *function* is computed by means of *statement* programming rather than recursion. For an example of this, study the definition of REV in Chapter 9.

(4) A *block:expression* may be used to avoid several repetitions of the same computation.

Example 1:

```
X←BEGIN REAL Y; Y←A↑2-3*A+B↑2; RETURN LIST (Y, Y-3, SQRT (Y)) END;
```

Alternatively, this could have been written:

```
X←LIST (A↑2-3*A+B↑2, A↑2-3*A+B↑2-3, SQRT (A↑2-3*A+B↑2));
```

the first *program* runs faster.

(5) A *block:statement* may also be used to avoid several repetitions of the same computation.

Example 2:

```
BEGIN SYMBOL Y;
     Y←FN (IF X-3*R<0 THEN CAR (L) ELSE M . CAR (N));
     U←CAR Y;
     V←CADR Y;
     W←CDDR Y
END;
```

CHAPTER 11.

ARRAYS

An *array* in LISP 2 is an indexed collection of data having one or more *dimensions*. We shall explain this further presently. In the meantime, let us note that this is different from an *array* in some other programming languages. In FORTRAN, for example, an *array* is an indexed collection of *variables;* the difference is not trivial.

For an example in LISP 2, let us consider a 3 by 4 by 5 *real:array*. This is a collection of *real:data*, specifically, a collection of exactly 60 *real:numbers*. It is a 3-*dimensional* indexed collection of *real:numbers*. This means that every element of the collection is identified by specifying in sequence three *integers* called the three *coordinates* of the element. If the three *coordinates* are called x, y and z, then the *coordinates* must satisfy $1 \leq x \leq 3$, $1 \leq y \leq 4$, $1 \leq z \leq 5$.

11.1 OPERATIONS

What are the basic operations that may be performed on an *array*? An *array* in LISP 2 is regarded as a single datum and is defined as a type of *atom*. Accordingly, an *array* may be the *argument* or *value* of a *function* and it may be incorporated into a nonatomic *S-expression*. In addition, any specific element of an *array* may be obtained or may be changed.

Since the allocation of storage space in LISP 2 is completely dynamic, *arrays* do not have to be declared in advance. They may be declared at any time and discarded at any time. As soon as an *array* is discarded, the space it occupied in memory is available for other purposes.

11.2 ONE WAY OF DECLARING *ARRAYS*

One of the ways of declaring an *array* is upon entry to a *block*. The following information must be stated:

(1) The type of the *array*. Some of the *array:types* are:

boolean:array

integer:array

real:array

symbol:array

An *integer:array* has only *integers* as its elements, etc. A *symbol:array* may have any type of data for its elements including other *arrays*.

(2) The size of the *array*. The specification must give the number of *dimensions*, and the *bounds* of each *dimension*. The *bound* of a *dimension* is always a positive *integer*.

(3) The data out of which the *array* is initially composed. This is determined as soon as the type and size are declared:

Type of *Array*	Initial Data
boolean:array	all elements are FALSE
integer:array	all elements are 0
real:array	all elements are 0.0
symbol:array	all elements are NIL

Of course, the data in such an *array* are promptly changed during the course of a computation using it.

For example, at the beginning of a *block*, suppose we wish to declare a *real:array:variable* called A containing a 3-*dimensional* *real:array* whose *bounds* are 3, 4, and 5, respectively. We would write:

REAL ARRAY A(3,4,5)

In the place of the *number* 4, for example, we could put an *expression* which would evaluate to the correct *integer bound*.

For another example:

SYMBOL ARRAY A(5), B(X+2), C(FN(W))

This *declaration* declares three one-*dimensional arrays* named A, B, and C of type SYMBOL. The size of the *dimension* of A is 5. The size of the *dimension* of B is equal to X plus 2. The size of the *dimension* of C is equal to FN of W. The second two sizes can only be determined at run time.

We should note that:

1. All the *arrays* specified in any one *declaration* must be of the same type.
2. They may each have any number of *dimensions*.
3. The number of *dimensions* is implied by the number of *expressions* specifying *bounds*.
4. A *bound* does not have to be a predeclared *integer*. Instead, it can be any *expression* that can be evaluated to yield an *integer* at the time that the *array* is activated. This can, for example, be a different *integer* each time the *array* is activated.
5. When an *array:declaration* is placed among the *declarations* of a *block*, the *array:variable* and associated *array* are active just before the first *statement* of the *block* is activated and continue active until the *block* is terminated. The same considerations of *binding* and *scope* apply to *array:variables* as apply to ordinary *variables* (see Chapter 10).
6. An *array:declaration* may be made on the top level of a LISP 2

program rather than inside a *block*. In this case, the *array* remains in existence all the time the LISP 2 *program* is in the computer.

11.3 HOW TO OBTAIN AN *ARRAY:ELEMENT*

Suppose that a 3-*dimensional* *real:array* whose *bounds* are 3, 4 and 5, respectively is associated with the *real:array:variable* A. Then the element whose *coordinates* are I, J and K may be referred to as

A(I,J,K)

I, J, and K are called *subscript:expressions*. They must evaluate to positive *integers*, and must not be greater than their respective *bounds*. Any *expressions* that have these properties may be used as *array:subscript:expressions*.

Example 1: A(2, IF P=0 THEN Q-1 ELSE Q, R)

Example 2: A(3, BEGIN RETURN 4 END, 5)

An *array:variable* followed by its *subscript:expressions* enclosed in parentheses and separated from each other by *commas* is a *form*. In fact it is impossible to tell by examining a *form* whether it begins with an *array:variable* or a *function::name*. *Forms* are *primaries* and consequently they are also *simple:expressions* (see Chapter 5).

When a *form* composed of an *array:variable* and *subscript:expressions* is evaluated, the *subscript:expressions* are evaluated first. If there are the correct number of *subscript:expressions* and if each *subscript:expression* is within *bounds*, then the *value* of the *form* is the specified element of the *array*.

11.4 HOW TO CHANGE AN ELEMENT OF AN *ARRAY*

To change an element of an *array*, we write a *form* with the *array:name* and *subscript:*

expressions and use it as the left side of an *assignment:statement* or *assignment:expression*:

Example 1:

A(2,2,2) ← 3.14159 — This sets the *array* element with *coordinates* 2, 2, and 2 to the *value* 3.14159.

Example 2:

Z(I,K) ← X(I,J)*Y(J,K) — This sets the *array:element* of Z with *coordinate* I and K equal to the product of the *array:element* of X with *coordinates* I and J and the *array:element* of Y with *coordinates* J and K.

11.5 A MATRIX MULTIPLICATION *PROGRAM*

Suppose we wish to define in LISP 2 a *function* MM, a *program* that multiplies two matrices. We shall assume that we have available, two *functions* called VREADIN and VREADOUT that read the data from an external device into or out of an *array*, respectively.

The *arguments* X, Y and Z of MM specify that a matrix of *dimensions* X by Y is to be multiplied by a matrix of *dimensions* Y by Z.

Here is the definition:

```
FUNCTION MM(X,Y,Z) BEGIN REAL ARRAY A(X,Y), B(Y,Z), C(X,Z);
    INTEGER I,J,K;
    VREADIN(A);
    VREADIN(B);
    I←1;
    R: J←1;
    S: K←1;
```

```
        T: C(I,K)← C(I,K)+A(I,J)*B(J,K);
        IF K< Z THEN BEGIN K← K+1; GO T END;
        IF J< Y THEN BEGIN J← J+1; GO S END;
        IF I< X THEN BEGIN I← I+1; GO R END;
        VREADOUT(C)
    END ;
```

11.6 PROCESSING AN *ARRAY* AS A SINGLE DATUM

An *array* in LISP 2 does not necessarily have a name. This is because an *array* is a datum. The situation is quite analogous to any other type of data, say *real:numbers*. If 5.0 is the *value* of the *real:variable* X, then we may refer to X and mean 5. But at some other time, X may not mean 5. In other words, an *array* may be a *constant*, or an *array* may be denoted by a *variable*, and either may be part of an *S-expression*.

The following example consists of a *list*, one of whose elements is an *array*. The *square:brackets* refer either to a row of an *array* or a *sub:array* or the *array* as a whole.

```
(A 3 [INTEGER  [1,0]  ,  [0,1]])
```

The third element of this *list* is a 2 by 2 *integer:array* which is denoted mathematically as the matrix:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

In regard to transmission of *arrays* or *array:elements*, there are some points to be stated. If FN is a *function* of one *argument*, and if A is a *2-dimensional: array*, then FN(A(I,J)) is an *expression* that obtains the i,jth element of A; this *expression* transmits this datum to FN, which then computes the appropriate *value*. Also, an entire *array* may be transmitted as an *argument*, or assigned to an *array:variable* by an *assignment:statement*. In the following example we define a *function* of an *array:variable* X, and then read in an *array* and give it to the *function* as an *argument*.

Example:

```
FUNCTION FN(X) REAL ARRAY X; body;
REAL ARRAY A;
A ← READARRAY ();
FN(A);
```

In the above example we have employed READARRAY to stand for a *program* devised by a user which reads in an *array* from an input file. Since this *function* has no *arguments*, ( ) is used. READARRAY fills in all the elements of the *array* that it creates.

The new techniques appearing in this example are explained in the following statements.

(1) If a *function* is to receive an entire *array* as an *argument* corresponding to a certain *argument:parameter* (X in the above example), this condition should be declared in a *declaration* appearing after the *argument:parameter:list*, and before the body of the *function:definition* ; REAL ARRAY X in the above example. The general form of this *declaration* is:

type ARRAY v_1, ... v_n

where *type* is BOOLEAN, INTEGER, REAL, or SYMBOL; and the v_i are one or more *variables*. The *declaration* is followed by a *semi:colon* to separate it from the next *declaration* or the body of the *function:definition*.

A more complete description of the kinds of *declarations* that may be made after the *argument:parameter:list* in the *function:declaration* is given in Chapter 15.

(2) The *declaration* REAL ARRAY A in the above example specifies that A is a *variable* of type *real:array*. It does not, however, place a *real:array* filled with floating-point zeros (0.0's) in A. To do this, if n is the number of *dimensions*, we write REAL ARRAY A(e_1, ... e_n). Or we make use of READARRAY in the example.

(3) If an *array:type:variable* is used as the left side of an *assignment::statement* (or *assignment:expression*) without *subscript:expressions* (in the line A ← READARRAY ( ) of the example), then the entire *array* (in this case, the current *value* of A) is to be replaced with a new *array* which is the *value* of the right half of the *assignment:statement* (or *assignment:expression*).

In the case of A ← READARRAY ( ), there was no *array* in A to begin with; but an *array* is placed in A by the *function* READARRAY which by the user's definition has an *array* as its *value*. (There is no LISP *system:function* called READARRAY because it would depend too much on the particular machine configuration.)

If one assigns NIL to an *array:variable*, then the *array* that was in it, if any, is discarded, and the storage space occupied by the *array* is released.

(4) If an *argument:parameter* of a *function* is of an *array:type*, then the *argument* transmitted to it must be an *array* of such *type*. The *form* FN(A) in the preceding example calls the *function* FN and presents to it the *array* that

is the *value* of A. This *array* as it is being transmitted is no longer referred to as the *value* of the *variable* A, but within the body of FN, is referred to as the *value* of the *argument:parameter* X.

11.7 BASIC *FUNCTIONS* FOR *ARRAYS*

The predicate ARRAYP(X) is true if X is an *array* and false otherwise.

The following *functions* allow one to obtain useful information about *arrays*. In the description below, assume that A is an *argument* which is an *array*, and I is an *integer:argument*.

ARRAYTYPE(A) can be used to find the *type* of an *array*. Its *value* is an *identifier* such as BOOLEAN, INTEGER, REAL or SYMBOL.

ARRAYDIM(A) specifies the number of *dimensions* of its *argument*. Its *value* is an *integer*.

ARRAYSIZE(A,I) specifies the *bound* of a particular *dimension*. The *argument* I specifies the *dimension* about which one is inquiring. The *value* of ARRAYSIZE is an *integer*.

(Note: The following *function* has not been fully specified. A possible implementation is described below because it is useful for the purposes of this primer. It or something similar to it will be implemented.)

The *function* MAKEARRAY can be used to create a new *array*.

MAKEARRAY(d_1, ... , d_n, type)

MAKEARRAY has an indefinite number of *arguments*. The first group of *arguments* are *integers* and specify successively the *bounds* of the new *array* to be created. The number of *bounds* implicitly specifies how many *dimensions* the *array* has. The *type* of the *array* is specified by the last *argument*, which is an *identifier*: BOOLEAN, etc. The *value* of MAKEARRAY will be an *array* of the specified *type* and size. Its initial data will all be FALSE, 0, 0.0, or NIL according to the *type*.

An example: Matrix multiplication

The following is a definition of a *function* that performs matrix multiplication. Unlike the previous example, it is a genuine *function*. It receives two *arrays* as its *arguments* and has their matrix product as its *value*.

```
FUNCTION MXMPLY(A,B) REAL ARRAY A, B; BEGIN
     REAL ARRAY C;
     INTEGER I,J,K,X,Y,Z;
     X ←—ARRAYDIM(A,1);
     Z ←—ARRAYDIM(B,2);
     IF (Y←— ARRAYDIM(A,2)) /= ARRAYDIM(B,1) THEN RETURN
          #ERROR - SECOND DIMENSION OF ARRAY 1 IS NOT THE SAME SIZE AS
          FIRST DIMENSION OF ARRAY 2#;
     C←— MAKEARRAY(X,Z,'REAL);
     I←—1;
R: J←— 1;
S: K←— 1;
T: C(I,K)←— C(I,K)+A(I,J)*B(J,K);
     IF K < Z THEN BEGIN K←— K+1; GO T END;
     IF J < Y THEN BEGIN J←—J+1; GO S END;
     IF I< X THEN BEGIN I←— I+1; GO R END;
     RETURN C
END ;
```

Following are some comments:

REAL ARRAY C specifies an *array:variable* but doesn't put an *array* in it.

Six *variables*, namely I, J, K, L, M, and N, are declared as *integer:parameters*;

A, B are *argument:parameters*; C is an *internal:parameter*;

X, Z are set to the outer *dimensions* of matrix multiplication;

Y is set to the second *dimension* of the first matrix A by an *argument:expression* .

The *value* of the *assignment:expression* is compared to the first *dimension* of B. They must be equal or the *value* of MXMPLY will be a *string* reporting the error. (This is <u>not</u> a recommended way of handling errors.)

The example contains an instance of MAKEARRAY. Its *arguments* are two *integers* and an *identifier* which is *quoted* in this case, because it is constant and always refers to *real:type*.

CHAPTER 12

FOR STATEMENTS

This chapter is a temporary one and will be replaced in the next edition of the Primer. It differs from the rest of the Primer in not being written in a tutorial style, not having any examples, and in its use of intermediate language. It supplants the chapter on FOR statements in the March 1966 preliminary draft of the Reference Manual.

Each type of FOR statement is herein illustrated both in source language and in intermediate language. The semantics of each kind of FOR statement is then completely defined by translating it into a block. This is a complete description of the semantics of the FOR statement because the LISP 2 compiler does in fact replace the FOR statement by the corresponding block via macro expansion.

Some of the FOR statements expand into compound statements, and some expand into block statements. It is correspondingly legal or illegal to transfer into the FOR statement. A FOR statement is never an expression, and it does not have a defined value. It is always possible to transfer out of a FOR statement if other conditions permit.

In the statement schemas that follow, the following symbols are used:

var	to mean any variable
exp	to mean any expression
ae	to mean any arithmetic expression
bool	to mean any Boolean expression
g1, g2 ...	to mean identifiers generated at the time of the macro expansion
st	to mean any statement

12.1 GENERAL CONSIDERATIONS

A FOR statement is a means by which the programmer can specify a program loop controlled in various ways, without explicitly writing out the loop. It is a shorthand notation, and does not permit anything which could not be done without FOR statements but at greater length.

Every FOR statement has a variable associated with it called the <u>control variable</u>. The control variable always appears in the FOR statement immediately after the word FOR and can be recognized accordingly.

```
FOR var ...
```

The FOR variable is never declared or bound by the FOR statement itself. When the control variable is mentioned within the FOR statement, the binding in effect at this time must be the same one as immediately outside of the FOR statement. The value of the variable at the time of entry into the FOR statement may be used inside the FOR statement in certain cases. The last value assigned to the control variable inside the FOR statement is available after the FOR statement has been executed.

The general form of the FOR statement in source language is:

```
FOR var for-element while-exp unless-exp DO st
```

In intermediate lnaguage it is:

```
(FOR var for-element while-exp unless-exp st)
```

In this schema, var stands for the control variable. The different types of FOR elements are explained in the succeeding sections. The statement st is called the <u>object statement</u> of the FOR statement. The object statement and the WHILE and UNLESS expressions are discussed below.

The object statement may be any type of statement including another FOR statement. It is executed repeatedly in a closed loop until the loop is terminated for one of several reasons. One way of terminating a FOR statement is to transfer from within the object statement to a label outside of the FOR statement. A RETURN statement may be used similarly.

The WHILE expression has the form:

```
        WHILE bool                (in source language)
        (WHILE bool)              (in intermediate language)
```

or else it is omitted. The expression bool is evaluated prior to each execution of the object statement. If the value of bool is FALSE, then the FOR statement is terminated immediately.

The UNLESS expression has the form:

```
        UNLESS bool               (in source language)
        (UNLESS bool)             (in intermediate language)
```

or else it is omitted. The expression bool is evaluated prior to the execution of the object statement. If its value is TRUE, then the execution of the object statement is omitted for this one pass through the loop. The FOR statement is not terminated by this action.

Either the WHILE expression or the UNLESS expression or both may be omitted. If they are both present, then the WHILE expression is written first and performed first

12.2 THE EMPTY FOR ELEMENT

```
sl:    FOR var while-exp unless-exp DO st
il:    (FOR var () while-exp unless-exp st)
```

expansion:

```
(BLOCK ()
L LU s_w
     s_u
     st
     (GO L)
LW   )
```

In this expansion (and others) the symbols L, LU, and LW are labels which are genids manufactured at the time that the macro expansion is performed. The statements s_w and s_u are present only if the WHILE expression and the UNLESS expression correspondingly are present in the FOR statement. They have the forms:

s_w: (IF (NOT bool) (GO lw))

s_u: (IF bool (GO lu))

where the boolean expressions from the WHILE expression or the UNLESS expression correspondingly are used.

The FOR statement with an empty FOR element is the one instance in which the control variable has no significance.

Example:

```
sl:     FOR A WHILE B<20 UNLESS C DO B←FN()+1
il:     (FOR A () (WHILE (LS B 20)) (UNLESS C) (SET B (PLUS (FN) 1)))
```

expansion:

```
(BLOCK ()
L LU  (IF (NOT (LS B 20)) (GO LW))
      (IF C (GO LU))
```

```
        (SET B (PLUS (FN) 1))
        (GO L)
    LW)
```

This expansion is literally correct except for the replacement of L, LW and LU by genids. (This example is worthless as a programming example.)

12.3 THE LOOP FOR ELEMENT

sl: FOR var LOOP exp while-exp unless-exp DO st

il: (FOR var (LOOP exp) while-exp unless-exp st)

expansion:

```
(BLOCK ()
L LU  (SET var exp)
      s_w
      s_u
      st
      (GO L)
LW    )
```

The LOOP element resets the control variable for each iteration of the loop. The initial value of var is unimportant unless it is used somewhere in the evaluation of exp.

12.4 THE RESET FOR ELEMENT

sl: FOR var ← expl RESET exp2 while-exp unless-exp DO st

il: (FOR var (RESET expl exp2) while-exp unless-exp st)

expansion:

```
(BLOCK ()
        (SET var expl)
L       s_w
        s_u
        st
LU      (SET var exp2)
        (GO L)
LW      )
```

The RESET element differs from the LOOP element in that the control variable can be set to an initial value via a different computation (expl) than the computation (exp2) that resets it.

If the previous value of var is to be used on the first iteration, then expl should be var. In source language, this may be omitted as follows:

sl: FOR var RESET exp2 while-exp unless-exp DO st

il: (FOR var (RESET var exp2) while-exp unless-exp st)

12.5 THE IN AND ON FOR ELEMENTS

sl: FOR var $\frac{\text{IN}}{\text{ON}}$ exp while-exp unless-exp DO st

il: (FOR var($\frac{\text{IN}}{\text{ON}}$ exp) while-exp unless-exp st)

expansion:

```
(BLOCK ((G1 SYMBOL exp))
L1      (IF (NUL G1) (GO L2))
        (SET var (CAR G1)/G1 )
        s_w
        s_u
        st
```

```
LU    (SET G1 (CDR G1))
      (GO L1)
LW L2  )
```

The IN (ON) FOR element executes the loop as many times as the length of the list which is the value of the expression exp. In successive executions of the loop, the control variable is set to successive elements of (remaining segments of) the list.

12.6 THE STEP FOR ELEMENT

sl: FOR var ← exp1 STEP exp2 UNTIL rel exp3 while-exp unless-exp DO st

il: (FOR var (STEP exp1 exp2 rel exp3) while-exp unless-exp st)

There are six possible relations (rel) in source language which translate into six corresponding relations in intermediate language:

sl	il
<	LS
< =	LEQ
>	GR
> =	GEQ
=	EQ
/=	NQ

The following omissions of parts of the statement are permitted:

1: If "← exp1" is omitted in source language, then exp1 in intermediate language is var.

2: If "UNTIL rel exp3" is omitted in source language, then rel and exp3 are omitted in intermediate language.

expansion:

```
(BLOCK ((G1 ASSIGNED exp2) (G2 ASSIGNED exp3)*)
      (SET var expl)**
L1    (IF (rel var G2) (GO L2))**
      s
       w
      s
       u
      st
LU    (SET var (PLUS var G2))
      (GO L1)
LW L2 )
```

* Omitted if there is no exp3.

** Omitted if this reads (SET var var).

12.7

The variable var may be replaced by any locative with exactly those consequences implied by the macro expansion.

CHAPTER 13.

FLUID:VARIABLES

Every *variable* in LISP 2 has one of three *storage:modes*. The three *storage:modes* are *lexical*, *fluid* and *own*. The *storage:mode* of a *variable* is independent of the *type* of the *variable*. Thus a *variable* may be *real* and *fluid*, or *symbol* and *lexical*, etc. All *variables* that have been considered so far in this Primer are *lexical*, because that is the *storage:mode* that is assumed by the system unless the user specifies otherwise. The *storage:mode* *own* is described in the Reference Manual. Here we shall describe *fluid:variables* and distinguish them from *lexical:variables*.

13.1 EXAMPLES OF *FLUID* AND *LEXICAL:VARIABLES*

The properties of *lexical* and *fluid:variables* are explained in Table 13.1.

Table 13.1

If the *Type* of *Variable* is:	And the *Type* of *Binding* is:	Then the *Scope* of the *Binding* is:	And the Duration of the *Binding* is:	See Example No.
lexical	as an *argument:parameter*	the body of the *function* being defined	while the body is being evaluated	1
fluid	"	the entire *program*	"	2
lexical	an an *internal:parameter*	the sequence of *statement* in the *block* in which the *binding* was made	while these *statements* are being executed	3
fluid	"	the entire *program*	"	4

We should note that any mention of a *variable* lies (or should lie) within the *scope* of exactly one *binding* of that *variable*. In a case where the mention of a *variable* lies within the *scopes* of several *bindings*, it is the innermost *binding* which takes priority, then the next outer *binding*, and so on.

Example 1. This is an example of *lexical:variables* used as *argument:parameters*.

```
FUNCTION MEMBER (X,L);
     NOT NULL X AND
     (X = CAR L OR MEMBER (X, CDR L));
```

In this case both X and L are *lexical:argument:parameters*. The *scope* of their *binding* is from the first *semi:colon* to the second *semi:colon*. The time of their *binding* is while the body (the portion of the *program* between the two *semi:colons*) is being evaluated. Only while the body is being evaluated can X and L be known.

Example 2. This example contains two *fluid:variables* used as *argument:parameters*, and two *lexical:variables* used as *argument:parameters*.

```
FUNCTION SUBST(X,Y,Z) FLUID X,Y; SUBST1(Z);

FUNCTION SUBST1(W) IF ATOM W THEN (IF W=Y THEN X ELSE W)
     ELSE(SUBST1(CAR W) . SUBST1(CDR W));
```

This is the same *function* SUBST that was defined in Chapter 3, but here the definition is a different one, making use of an auxiliary *function* called SUBST1. SUBST is not recursive in this definition; it *binds* the three *variables* X, Y, and Z. SUBST1 is recursive and *binds* W, the *binding* for W changing for each recursion. However, SUBST1 must use the first two *variables* X and Y. SUBST1 is not within the *lexical:scope* of SUBST, but since the *argument:parameters* X and Y of SUBST are declared to be *fluid*, then they may be accessed anywhere in the *program* while the body of SUBST is being evaluated. This includes the time during which SUBST1 is being computed because SUBST still has not been finished.

We say that SUBST1 is within the *fluid:scope* of SUBST when SUBST calls SUBST1. If SUBST1 were called from some other *function*, however, then SUBST1 would not be within the *fluid :scope* of SUBST and it would not be able to take hold of the *bindings* of X and Y. Thus the concept of *fluid:scope* is a highly dynamic one, and depends upon conditions that cannot in general be anticipated before the *program* is run.

When a *variable* is mentioned in a *function:definition* without its being *bound* in that definition either as an *argument:parameter* or as an *internal:parameter*, then it is called a *free:variable*. *Free:variables* are automatically and necessarily *fluid*.

In the definition of SUBST1, X and Y are *free:variables*. They are not *arguments* of SUBST1, but they are referenced for *value*. The only reason they have *values* is that SUBST1 is called by SUBST which *binds* X and Y as *fluid:variables*. If the *declaration* FLUID X, Y were missing in the definition of SUBST, then the *values* of these *bindings* could not be used in SUBST1. If this *declaration* were missing, X and Y would become *lexical:variables* in SUBST, and could be referenced only from within SUBST.

Example 3. This example contains two *lexical:variables* as *internal:parameters*.

```
FUNCTION REVERSE (X) BEGIN SYMBOL Y;
     A:  IF NULL X THEN RETURN Y;
         Y ← (Y . CAR X);
         X ← CDR X;
         GO A;   END ;
```

This *program* of *statements* produces the reverse of a *list*, a *list* in the reverse order.

X and Y are *lexical:variables* used as *internal:parameters*. Their *binding* exists spatially in the sequence of *statements* in the *block* where they are *bound*, and exists in time while those *statements* are being executed.

Example 4. The following artificial example may explain some points about the *scope* of *variables*.

```
FUNCTION P(X) FLUID X; Q();
FUNCTION R(Y); P(Y);
FUNCTION Q(); PRINT(X . 'O);
FUNCTION J(Y); FLUID  Y ; K();
FUNCTION K(); BEGIN FLUID SYMBOL X; X ← Y; Q() END;
P('A);
R('B);
J('C);
```

When this *program* is run, the *S-expressions* (A . D) (B . D) and (C . D) are printed in that order. Here is the description of its operation.

(1) *Function* P *binds* the *fluid:variable* X to the *value* A. The A can then be picked up as the *value* of the *free:variable* X that occurs in *function* Q.

(2) *Function* R transmits its *argument* (which is B) to *function* P. *Function* P then *binds* B to the *fluid:variable* X, where it is picked up by Q.

(3) *Function* J *binds* C to the *fluid:argument:parameter* Y. It then calls K which has no *arguments*. K has a *fluid:internal:parameter* X which is initially *bound* to NIL. It then becomes *bound* to C because of the *assignment:statement* which picks up the

value of the *free:variable* Y and assigns this to X. The C is then picked up by Q.

Example 5.

```
FUNCTION L(X) BEGIN SYMBOL FLUID Y; Y←(X . NIL); M(X);
FUNCTION M(Y) N();
FUNCTION N() PRINT ('B . Y);
D('A); END ;
```

In this case, the *function* N prints (B A) and not (B . A). The *variable* Y occurs *free* in N, and the *value* of Y must be the most recent *fluid:binding* of Y that is still in effect. This is the *internal:parameter* Y declared in L. The Y of *function* M is not *fluid* (because it is not declared to be *fluid* and therefore it is not the *value* of Y that will be used.

A *fluid:variable* may have only one *type* regardless of the area in the *program* where it is used. Thus the following two *declarations*, if made in one *program*, are incompatible even if they may be in different subsections of the same *program*:

Incompatible *declarations*:

```
FLUID INTEGER X;
FLUID REAL ARRAY X;
```

It is a common programming convention in LISP to choose longer, more uncommon names for *fluid:variables* because their *scopes* are so wide, and one may run into collision problems among *fluid:variables*. *Single:letter:identifiers* are commonly used for *lexical:variables*.

13.2 *FLUID:DECLARATIONS*

To declare that an *argument:parameter* or several *argument:parameters* are *fluid*, we put the *declaration*

FLUID $v_1, \ldots, v_n$

after the *argument:parameter:list* and before the body of the *function:definition*. This *function:definition* may be combined with others that may properly be put in this position, such as

FLUID REAL ARRAY X, Y

This *declaration* is always followed by a *semi:colon*.

To declare that an *internal:parameter*is *fluid*, we write

FLUID $v_1, \ldots, v_n$

but it is usual to combine this with another *declaration* such as a *type:declaration*.

Example 6. Suppose we have three *internal:parameters* X, Y and Z with the following *declarations*:

```
FLUID SYMBOL X;
FLUID REAL ARRAY Y;
REAL ARRAY Z;
```

The *statement* that X is of *type SYMBOL* and has *storage:mode fluid* can be stated either by FLUID SYMBOL X or by SYMBOL FLUID X, or by SYMBOL X; FLUID X.

The order of the declaratory words makes no difference so long as all the declaratory words precede all the *variables* to which they apply. The set of *declarations* above could be rewritten as follows with the same effect:

```
FLUID X, Y;
SYMBOL X;
REAL ARRAY Y, Z;
```

CHAPTER 14.

LOCATIVE TRANSMISSION OF *PARAMETERS*

Every *parameter* in LISP has a *type*, and a *storage:mode*, and in addition what is called a *transmission:mode*. There are two *transmission:modes*, called *transmission:by:value* and *transmission:by:location*. This latter is abbreviated to the *reserved:word* LOC. All the *parameters* considered up to this point in this Primer have been *transmitted:by:value*. This is the most common *mode*. For this reason the *transmission:mode*assumed, unless the programmer declares otherwise, is *transmission:by:value*.

We will discuss here *argument:parameters* having *loc:transmission:mode*. The case of *internal:parameters* having *loc:transmission:mode* is rare and outside of the province of the Primer.

14.1 *ARGUMENTS* TRANSMITTED BY *VALUE*

First, let us consider an example which reviews some terms.

Example 1:

```
FUNCTION FN(X) X↑2+3*X;
W←3;
FN(W-7);
```

a. *Argument:Expression*. In this example FN is defined as $x^2 + 3x$; then W is set at 3; and FN of W-7 is called. In this example, W-7 is an *expression* used to compute an *argument* for the *function* FN. W-7 is not itself the *argument*; we call W-7 an *argument:expression*.

b. *Argument.* The *argument* of FN in this example is -4, because -4 is the *value* of the *argument:expression* W-7 when W is set at 3.

c. *Argument:parameter.* The *argument:parameter* of FN is the *variable* X. Its *value,* while the body of FN is being evaluated, is the *argument* -4. The *argument:parameter* X has *value:transmission:mode*; what X means is determined by finding the *value* of X. (This is true because there is no *declaration* specifying that the *transmission:mode* should be *loc*; furthermore, as we shall see, a *declaration* LOC would be illegal in this case.)

Having reviewed this vocabulary, we can now state a rule for transmitting *arguments* by *value.*

Rule: If an *argument:parameter* has *value:transmission:mode*, then at the time the *function* is called, the *argument:expression* corresponding to that *argument:parameter* is evaluated, and the resulting *value* is transmitted to the *function* as the *argument.*

We note that the evaluation of the *argument:expression* to yield an *argument* is performed prior to the call to the *function.*

The term *transmission:by:value* is justified by the fact that it is the *argument* and not the *argument:expression* that is transmitted. Thus, in the preceding example, the *function* FN receives the *argument* -4; there it is immaterial that the *variable* W was in any way related to the method by which -4 was determined.

14.2 *ARGUMENTS* TRANSMITTED BY LOCATION

Let us now try to explain the *loc:transmission:mode.* We shall begin with another example.

Example 2:

```
FUNCTION FN(X) INTEGER LOC X; X←— 5;
BEGIN INTEGER Y;
     Y←— 3;
     FN(Y);
     PRINT(Y)
END;
```

What will be the result of executing this *program*? Let us analyze the steps:

(1) The *function* FN is defined. X is declared *integer* and *locative*. Then X is set equal to 5.

(2) A *block:expression* is entered; the *internal:parameter* Y is declared of *type* INTEGER; it is assigned the *value* 3.

(3) Now FN is called from within the *block:expression*. Corresponding to the *argument:parameter* we have the *argument: expression* Y. But Y is <u>not</u> evaluated to produce 3 as an *argument* for FN. Instead, the *binding* of Y itself is transmitted, i.e., the location of the *value* of Y.

(4) When the *assignment:statement* X←— 5 is executed, X is not *bound* to a *value*; instead it is *bound* indirectly to another *binding*, namely the *binding* of Y. Therefore, it is as if the *statement* Y←— 5 were executed.

(5) This changes the *binding* of Y in the *block:expression*, so that its *value* is now 5. Consequently, 5 is what is printed.

If an *argument:parameter* has *loc:transmission:mode*, severe restrictions are imposed on its *argument:expression*.

One cannot in general use any *expression*. For example, consider

```
FUNCTION FN(X) INTEGER LOC X; X←5;
```

One cannot call FN by FN(3) because this would mean that the *assignment: expression* would then read 3← 5 which is nonsense. Even FN(Y+4) is illegal. This would make the *assignment:statement* read Y+4←5. One could claim that this means Y← 9 but in LISP, it does not, because the *assignment:statement* is not intended as a device for solving implicit equations.

Two possibilities are permitted here (others are discussed in the Reference Manual). First, the *argument* may be a *variable* of the same *type* as the *locative:argument:parameter*. So one may write FN(Y) but only if Y is a *variable* of *type* INTEGER. If Y were of *type* REAL, this would be illegal. Secondly, if the *locative:parameter* is of a simple *type* (such as INTEGER, REAL, etc., but not INTEGER ARRAY etc.) then one may use as an *argument:expression* a *variable* of the corresponding *array:type,* with *subscripts*. Thus if A is a *variable* of *type* INTEGER ARRAY, one may write FN(A(L-3, M*4)). The *array: subscript:expressions* (L-3 and M*4 in this case) are evaluated before FN is called, and a reference to the element A(1,1) is transmitted to FN that makes X correspond with the particular element of the *integer:array* that has been specified.

Example 3.

```
FUNCTION FN(X) INTEGER LOC X; X←5;
     BEGIN INTEGER ARRAY A(6,7);
          INTEGER I,J;
          I← 3;
          J← 4;
```

```
        PRINT (A(2,1));
        FN(A(I-1,J-I));
        PRINT(A(I-1,I-2))
END;
```

The first *number* printed is 0; the second is 5. The *variable* A is *bound* to a 6 by 7 *integer:array* whose data are all 0's. The call to FN *binds* X not to the datum 0 in the 2,1 location, but to the 2,1 location itself, because X is LOC. The *assignment:statement* in the body of FN X←5 has the meaning or interpretation A(2,1)←5.

The following example illustrates one of the peculiar properties of a *parameter* that is both *fluid* and *loc*. Example:

```
FUNCTION FN(Y) FLUID LOC Y; G();
FUNCTION G() Y←'B;
BEGIN SYMBOL X;
    X←'A;
    FN(X);
    PRINT(X)
END;
```

This *program* prints B. When FN is called from within the *block:expression*, X is not evaluated. Instead, the *binding* of X is transmitted as the *binding* of Y because Y is *loc*. When G is called, the *free:variable* Y in G is within the *scope* of the *argument:parameter* Y (of FN) because the *argument:parameter* is *fluid*. Thus the *assignment:statement* in G may be read as X←'B and refers to the *internal:parameter* X of the *block:expression*. But this is only because Y in FN is both *fluid* and *loc*.

The reader may, at this point, be puzzled as to how to treat the rule 2 which states that an *argument:parameter* that is *loc*, and its corresponding *argument: expression* must correspond in *type*. Until now, it has not been stated that every *variable* has a *type*. Yet this is indeed the case. Usually the *type* of a *variable* is determined without the programmer being very much aware of it. But it is important to understand that every *variable* always has a *type*, and that there are rules for determining this. This is treated in the next chapter.

CHAPTER 15.

TYPES AND *DECLARATIONS*

15.1 *TYPES* OF *VARIABLES*

Every datum has a *type*, and every *variable* has a *type*, but the *type* of a datum is a little different than the *type* of a *variable*.

The type of a datum is always deducible by looking at the datum. For example, the *type* of 2.5E3 is *real*, and this is clear from the way in which 2.5E3 is written. Of course, a *symbolic:datum* (*type* SYMBOL) is any datum at all; so 2.5E3 is also a *symbolic:datum*, although 2.5E3 is regularly considered to be a *real:datum* since this is more specific.

The *type* of a *variable* is an intrinsic property of the *variable*. It amounts to a restriction on the *type* of datum that may be assigned to that *variable* as a *value*. Thus, if A is a *real:variable*, its *value* must always be a *real:number*; if B is a *symbolic:variable*, its *values* can be any datum at all.

What is the advantage of having *variables* of different *types*? Why not let all *variables* be of *type* SYMBOL?

The first answer is that when a *variable* is used with *array:subscripts* following it, the *variable* cannot be of *type* SYMBOL. The LISP 2 compiler requires that it be a specific *array:type* of *variable*, (see Chapter 11.) A similar requirement is true of *variables* that designate *functions* as *arguments*, a case which is discussed later.

The second reason is efficiency. If arithmetic *types* of data, such as *integer*, *real*, and *octal*, are specified for *variables*, then the programmer (in return for restricting himself to assign only data of the specified *type* as *values* for that *variable*) thus informs the compiler; and the compiler can generate more efficient code. The *declarations* allow the compiler to assume what kind of datum is in a *variable*; and therefore the test to determine *type* does not have to be made each time the *variable* is referenced as the *program* is executed.

In a case where a *program* does only numerical computation, and all the *variables* are declared to be of *arithmetic:types*, the *program* may run 30 to 100 times faster than a *program* which performs the same computation with all *variables* of *type symbol*.

15.2 *DECLARATIONS* FOR *ARGUMENT:PARAMETERS*

The *declarations* for *argument:parameters* are those that specify *type*, *storage: mode*, and *transmission:mode*. These *declarations* follow directly after the *argument:parameter:list* of the *function* and before its body. Each one is followed by a *semi:colon*. They may be grouped in any convenient way; the order is not significant.

Each *declaration* begins with one or more key words which specify *type* *trans-mission:mode*, and *storage:mode*. They are followed by a list of *parameters*. If there is more than one *parameter*, then they are separated from each other by *commas*. (The key words are not separated from each other or from the first *variable* by any punctuation.) Each *variable* mentioned must be one of the *parameters* in the *parameter:list* preceding the *declarations*.

Some of the key words are:

BOOLEAN	FLUID	LOC
INTEGER	OWN	
REAL		
SYMBOL		
BOOLEAN ARRAY		
INTEGER ARRAY		
REAL ARRAY		
SYMBOL ARRAY		

Example:

```
FUNCTION FN(V,W,X,Y,Z) REAL LOC V, W; SYMBOL X; FLUID V, Z;
     REAL Y, Z; LOC Y; ...body...
```

This example could have been written:

```
FUNCTION FN(V,W,X,Y,Z) REAL LOC FLUID V; REAL LOC W; SYMBOL X;
     REAL LOC Y; REAL FLUID Z; ...body...
```

Incorrect example:

```
FUNCTION G(X,Y) SYMBOL X,Y; REAL LOC Y,Z; ...body...
```

It is inconsistent to assign two *types* to the *parameter* Y. It is also incorrect to mention a *variable* Z which is not an *argument:parameter* of G.

15.3 DECLARATIONS FOR INTERNAL:PARAMETERS

Declarations for *internal:parameters* follow the word BEGIN at the beginning of a *block*. The rules for these are similar to the rules for *declarations* of *argument:parameters*. There are some differences however.

An *internal:parameter* must be mentioned at least once if the *program* is to notice it at all. At a minimum, the *type* of a *variable* can be declared--if there is nothing else that you wish to declare.

Unlike *argument:parameters*, *internal:parameters* are subject to initialization. The initialization may be specified by the user or not. If not, a default initialization will be made by the LISP system. The default initialization depends upon the *type* of the *variable*, and will be NIL, 0, 0.0, etc., accordingly. An explicit assumption is made using what looks like an *assignment:statement*.

Example:

```
REAL A←2.5, B, C←X-Y;
```

This *declaration* specifies three *variables* as being *real:internal:parameters*. A is initialized to 2.5. B is initialized to 0.0. C is initialized to the *value* of the *expression* X-Y.

The initializing *expression* may be any kind of *expression*. All the initializing *expressions* are evaluated <u>before</u> any of the *bindings* of the *internal:parameters* become effective. One consequence of this is illustrated by the following example.

```
FUNCTION FN(X) BEGIN SYMBOL X←'A, Y←X; RETURN Y END; FN('B);
```

The *value* of FN in this case is B. The *argument:parameter* X has as its *value* the *identifier* B. The *internal:parameter* X is initialized to the *identifier* A. When the initialization of Y is computed, the *binding* of the *internal:parameter* X is not in effect yet. (Its *scope* starts with the word RETURN.) Thus, the X referred to in Y←X must be the *argument:parameter* X, and so Y gets initialized to B.

15.4 *DEFAULT:DECLARATIONS*

It would be tedious if the programmer were to specify the *type*, *storage:mode*, and *transmission:mode* of each *parameter*. Fortunately this is not the case. The system is able to deduce these in most cases by a set of rules called the *default:declarations*. If the programmer wishes, he may over-ride these by making specific *declarations*.

Rule 1: If no *declaration* specifies otherwise, then a *parameter* has *value:transmission:mode*.

Rule 2: If no *declaration* specifies otherwise, then a *parameter* has *lexical:storage:mode*.

Rule 3: There is a *type* specified as being the current *section:types*. (See Sections in the LISP 2 Reference Manual.) The *section:type* does not change until the programmer changes it by means of a *section:declaration*. Initially, the *section:type* is SYMBOL. If no *type:declaration* is made for a *parameter*, its *type* is the *section:type*.

15.5 *VALUE:TYPE:DECLARATIONS*

If a *function* always has a *value* which is of some specific *type*, then a *declaration* informing the compiler of this fact increases the efficiency of the *program*. This *declaration* is made just before the word FUNCTION.

Example:

```
REAL FUNCTION SIN(X) ...
```

This *declaration* restricts the *values* of SIN to being *real:numbers*; it makes SIN a more efficient *program* than if its *value* were not so specified.

When no *value:type* is specified, the *section:type* is assumed to be the *value:type*. If the *section:type* is SYMBOL, this creates no problems. But if the *section:type* is, say, INTEGER and if a *function* is to have *S-expressions* as *values*, it is necessary to specify SYMBOL FUNCTION ...

When a *function* is never executed to have a *value*, but only for its effect, then the *declaration* NOVALUE FUNCTION may be used.

15.6 *FREE:DECLARATIONS*

Free:declarations are *declarations* that are not made within *function:definitions* or *blocks*, but on the top level of LISP.

The *declaration*

REAL X, Y;

specifies that the *fluid:variables* X and Y are of *type* REAL. Every *parameter* X or Y which is specified as being *fluid*, and every *free* mention of X and Y must refer to the *real:variables* X and Y which of course are *fluid*.

These *variables* also have a universal *scope* in some sense. If they are referred to *free* in a context in which they are not within the *fluid:scope* of any *binding* of them as *argument:parameters* or *internal:parameters*, then it is this top level that is referred to. This gives the programmer a way of using, non-recursively, *variables* that can retain *values* from one part of a *program* to the next.

We remind the user that all occurrences of any *fluid:variable* must be of the same *type*. Note that there may be a *lexical:parameter* called X which is not of *type* REAL.

The *declaration*

REAL FLUID X,Y;

means something different. It means not only that all *fluid* and *free* references to X and Y refer to a *real:variable*, but that in addition, all references to <u>any</u> X and Y refer to a *fluid:real:variable*. Thus, it over-rides the convention that a *parameter* is of *lexical:storage:mode* unless otherwise stated. Once this *declaration* is made, all *variables* X and Y are *fluid*.

PROBLEM SETS AND ANSWERS

CHAPTER 2

PROBLEM SET 1

Which of the following are *S-expressions*?

a. UVW

b. (A . B . C)

c. (A . BC)

d. ((((A . B) . C) . E) . (F . (G . H)))

e. ((A . B) . (C . D) . (E . F))

f. ((X))))

Answers:

a. Yes

b. No

c. Yes

d. Yes

e. No

f. No

PROBLEM SET 2

Evaluate each of these *expressions*:

a. CONS('WINE, 'CHEESE)

b. CONS('TUOLUMNE, CONS('SANJOAQUIN, 'KINGS))

c. CONS ('(A . B) . '(C . D))

d. CONS (CONS ('(A, 'B) , CONS ('C, 'D))

e. CONS ('(A . B), CONS ('C, 'D))

Answers:

a. (WINE . CHEESE)

b. (TUOLUMNE . (SANJOAQUIN . KINGS))

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

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

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

PROBLEM SET 3

Evaulate each of these *expressions*. (Some of them may be undefined.)

a. CAR('A)

b. CDR('(A . B))

c. CAR(CDR('(STRAVINSKY . (BARTOK . SIBELIUS))))

d. CDR(CAR(CAR('(((HAT . TIE) . SHIRT) . JACKET))))

e. CAR(CDR('((AQUITAINE . GASCONY) . ARAGON)))

f. CAR(CONS('A, 'B))

g. CAR(CDR(CONS('(A . B),'(C . D))))

h. CONS(CAR('(A . B)),CDR('(C . D)))

i. CONS(CAR('(A . B)),CAR('(C . D)))

j. CONS('A,CAR('(C . D)))

k. CADR ('(A . B))

l. CADR('(SHRIMP . (LOBSTER . CRAB)))

m. CAAR(CONS(CONS('A,'B),'C))

n. CDDR(CONS('(A,'(B . C)))

o. CONS(CAAR('((A . B) . C)),CONS('D,CDDR('(E . (F . G)))))

Answers:

a. undefined

b. B

c. BARTOK

d. TIE

e. undefined

f. A

g. C

h. (A . D)

i. (A . C)

j. (A . C)

k. undefined

l. LOBSTER

m. A

n. C

o. (A . (D . G))

PROBLEM SET 4

Rewrite each of these following *S-expressions* using only *dot:notation*.

a. (A)

b. ((A))

c. (HE MADE THE STARS ALSO)

d. (() (A) (A A))

e. (A (A) ((A)))

Rewrite each of the following *S-expressions* using *list:notation* as much as possible:

f. ((A . NIL) . (((B . NIL) . NIL) . NIL)

g. ((A . NIL) . ((B . NIL) . NIL))

h. (A . B)

i. ((((A . NIL) . NIL) . NIL) . NIL)

j. ((X . NIL) . ((NIL . Y) . NIL))

Answers:

a. (A . NIL)

b. ((A . NIL) . NIL)

c. (HE . (MADE . (THE . (STARS . (ALSO . NIL)))))

d. (NIL . ((A . NIL) . ((A . (A . NIL)) . NIL)))

e. ((A . NIL) ((A . (A . NIL)) . ((A . (A . (A . NIL))) . NIL)))

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

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

h. (A . B)

i. ((((A))))

j. ((X) . ((NIL . Y)))

PROBLEM SET 5

Evaluate each of these *expressions*:

a. CAR('(A B C))

b. CADR('(A B C))

c. CADDR('(A B C))

d. CDR('(A B C))

e. CDDR('(A B C))

f. CDDDR('(A B C))

g. CAAR('(A B C))

h. CONS('A, '(B C))

i. CONS('A, CONS('B, '(C)))

j. CONS('A, CONS('B, CONS('C, NIL)))

k. CONS('(A B),'(C D))

l. CONS(CONS('A, NIL), NIL)

m. CDAR('((A B) (C D)))

Answers:

a. A

b. B

c. C

d. (B C)

e. (C)

f. NIL

g. undefined

h. (A . (B C)) = (A B C)

i. (A . (B . (C))) = (A B C)

j. (A B C)

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

l. ((A))

m. (B)

PROBLEM SET 6

Evaluate the following *expressions*:

a. '(HELLO THERE BILL) = '(HELLO THERE JOE)

b. FALSE=( )

c. NIL=( )

d. '(A (B . C)) = '((A . B) . C)

e. CAR('(A B)) = CADR('(B A))

f. CONS(CONS('(A B),'(C D)),'A = 'B)

Answers:

a. FALSE

b. TRUE

c. TRUE

d. FALSE

e. TRUE

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

PROBLEM SET 7

Evaluate the following *expressions*:

a. ATOM('TUVWXYZ)

b. ATOM('A) = ATOM('B)

c. ATOM(CDR('(A B)))

d. ATOM('A = '(B C))

e. ATOM(CAR(CONS(CAR('(A B)), CDR('(C D)))))

Answers:

a. TRUE

b. FALSE

c. FALSE

d. TRUE

e. TRUE

PROBLEM SET 8

Evaluate the following *expressions*:

a. LIST('A, 'B, '(C D))

b. CAR(LIST('A, 'B, 'C))

c. CAR(LIST('(A B C)))

d. ATOM(LIST('A))

e. LIST('A, 'B) = CONS('A, CONS('B, NIL))

Answers:

a. (A B (C D))

b. A

c. (A B C)

d. FALSE

e. TRUE

PROBLEM SET 9

Evaluate the following *expressions*:

a. NULL (CADDR ('(A (B C) D)))

b. CONS ('A, NULL ('A))

c. NULL (LIST ( ))

d. NULL (CDR (LIST 'A)))

Answers:

a. FALSE

b. (A)

c. FALSE

d. TRUE

CHAPTER 4

PROBLEM SET 10

Evaluate each of these *arithmetic:expressions* using the following table to determine the *values* of the *variables* occurring in the *expressions*.

Variable	*Value*
A	2
B	-3.0
C	-5
D	7.5

a. A-1

b. A+B

c. B↑A

d. C-:D

e. C/D

f. A*C

g. D-:1.0

Answers:

a. 1

b. -1.0

c. 9.0

d. 0

e. -.6666666667 if computer provides 10 decimal digits

f. -10

g. 8

PROBLEM SET 11

Examine each *expression*. (1) Insert *parentheses* and produce an equivalent *expression* which if there were no precedence rules would be completely unambiguous. (2) Evaluate this *expression* using the table to determine the *values* of the *variables* occurring within the *expression*.

Variable	*Value*
A	5
B	2.5
C	1
D	-6

a. A-3*C

b. (A-3)*C

c. A-(3*C)

d. D↑C↑A

e. A+B*C+D

f. A*B+C*D

g. -D+A

h. -(D+A)

i. -D-A

j. 6/3/2

k. 6/(3/2)

l. 6/(3*2)

m. 6/3*2

Answers:

	(1)	(2)
a.	A-(3*C)	2
b.	(A-3)*C	2
c.	A-(3*C)	2
d.	D↑(C↑A)	-6
e.	(A+(B*C))+D	1.5
f.	(A*B)+(C*D)	6.5
g.	(-D)+A	11
h.	-(D+A)	1
i.	(-D)-A	1
j.	(6/3)/2	1.0
k.	6/(3/2)	4.0
l.	6/(3*2)	1.0
m.	(6/3)*2	4.0

PROBLEM SET 12

Evaluate the following *expressions* using the table to determine the *values* of the *variables*.

Variable	*Value*
A	2
B	3.0
C	4
D	-0.0E6
E	-1
F	2.5

a. ABS(A)

b. ABS(E)

c. SIGN(-B)

d. SIGN(D)

e. MAX(A,-B)

f. MAX(A,-C)

g. MIN(A,E)

h. ROUND(F)

i. ENTIER(F)

j. ROUND(-F)

k. ENTIER(-F)

l. SQRT(C)

m. SQRT(E)

n. ABS(A)+ABS(B)*ABS(C)

o. -ROUND(E)-ROUND(D)

p. ROUND (-F + .3)

Answers:

a. 2

b. 1

c. -1

d. 0

e. 2.0

f. 2

g. -1

h. 3

i. 2

j. -2

k. -3

l. 2

m. undefined

n. 14.0

o. 1

p. -2.0

CHAPTER 5

PROBLEM SET 13

Evaluate the following *expressions*:

a. CAR('(A B C))

b. CADR('(4 5 6))

c. CDR('(1 2))

d. ATOM(500)

e. REALP(7)

f. REALP(CAR('(3.5 4.5)))

g. CAR('(1.1))

h. CAR('(1 . 1))

i. ATOM('(7))

j. NUMBP (CAR('(7)))

k. CONS('(1 2) ,'(3 4))

Answers:

a. A

b. 5

c. (2)

d. TRUE

e. FALSE

f. TRUE

g. 1.1

h. 1

i. FALSE

j. TRUE

k. ((1 2) 3 4)

PROBLEM SET 14

Evaluate each of the following *expressions*, using the table to determine the *values* of the *variables* occurring in the *expressions*.

Variable	*Value*
A	X
B	NIL
C	3.5
D	(A 4)
E	A

a. CONS(A,B)

b. CONS ('A,B)

c. CONS (E,'B)

d. CDR(D)

e. C + CADR(D)

f. SQRT(CADR(D))

g. CONS(E,C)

h. CONS(C,B)

i. C+2

Answers:

a. (X . NIL) , which equals (X)

b. (A . NIL) , which equals (A)

c. (A . B)

d. (4)

e. 7.5

f. 2.0

g. (A . 3.5)

h. (3.5 . NIL), which equals (3.5)

i. 5.5

PROBLEM SET 15

Rewrite each *expression* adding enough *parentheses* to determine the correct grouping. Then evaluate them using the table to determine the *values* of the *variables*.

Variable	*Value*
W	4
X	(A B)
Y	C
Z	(2)

a. W . NIL

b. Y . X

c. W*3 . CAR Z

d. CAR Z + 2

e. CAR X . CDR Z

f. Y . NIL

g. 'Y . NIL

Answers:

	(1)	(2)
a.	W . NIL	(4 . NIL), which equals (4)
b.	Y . X	(C . (A B)), which equals (C A B)
c.	(W*3) . (CAR Z)	(12 . 2)
d.	(CAR Z) + 2	4
e.	(CAR X) . (CDR Z)	(A . NIL), which equals (A)
f.	Y . NIL	(C . NIL), which equals (C)
g.	'Y . NIL	(Y . NIL), which equals (Y)

PROBLEM SET 16

EValuate these *expressions* using the table to determine the *value* of the *variables*.

Variable	*Value*
A	3
B	2.4
C	3.0
D	A
E	(X Y)

a. A = 3

b. A = C

c. D = A

d. B>= C

e. E = 'X . 'Y . NIL

f. 'A = D

g. CAR E = 'X

h. 0 <B<= 3

i. 2<C + 3<7

j. 2<A<3

Answers:

a. TRUE

b. TRUE

c. FALSE

d. FALSE

e. (NIL . (Y . NIL)), which equals (NIL Y)

f. TRUE

g. TRUE

h. TRUE

i. TRUE

j. FALSE

PROBLEM SET 17

Examine each *simple:expression* below. Then rewrite it adding sufficient *parentheses* to make it unambiguous assuming no rules of precedence.

a. CAR A + B

b. CAR A + CDR B*C

c. A-B/C/D+E

d. A-B/C*D↑ E

e. CAR X = 'A

f. 0<= CAR A = B + SIN(Y)<5

g. A + B ↑ C ↑ CADR D

h. X . 'A . FN(X,Y,CDR Z*W)

i. ATOM X = Y

j. NULL U . NULL CAR X + Y

Answers:

a. (CAR A) + B

b. (CAR A) + ((CDR B)*C)

c. (A-((B/D)/D))+E

d. (A-((B/C)*(D↑E)))

e. (CAR X) = 'A

f. 0 <= (CAR A) = (B + SIN(Y))<5

g. A+(B↑(C↑(CADR D)))

h. X . ('A . FN(X,Y,((CDR Z)*W)))

i. (ATOM X) = Y

j. ((NULL U) . (NULL((CAR X) + Y)))

CHAPTER 6

PROBLEM SET 18

Evaluate the following *expressions* using the list of *values* for *variables*. REALP means "is a *real:number*"; SQRT means "the square root of"; SIGN means "the sign of."

Variable	*Value*
A	5
B	2.0
C	(7 14)
X	(3 . 9)
Y	(A B C)
Z	(A C)

a. IF A = 5.0 THEN B

b. IF REALP (Z) THEN C ELSE IF REALP(B) THEN (IF CAR A↑2 = CDR A THEN Y ELSE Z) ELSE X

c. IF IF CAR C = 7 THEN FALSE ELSE TRUE THEN Z

d. IF A = B THEN A = B ELSE A = B

e. IF C THEN A

f. IF SIGN(B) = SIGN(A) THEN (IF SQRT(CDR X) = CAR(X) THEN 'A ELSE A) ELSE 'B

g. IF CAR Y = CAR Z THEN 'ELSE ELSE 'IF

h. IF TRUE THEN 'IF IF 'IF THEN 'THEN

Answers:

a. 2.0

b. (A C)

c. UNDEFINED

d. FALSE

e. 5

f. A

g. ELSE

h. IF

CHAPTER 7

PROBLEM SET 19

In this problem set, several *function:definitions* are given, and a table of *bindings* for *free:variables* is given. The problem is to evaluate the *expressions* that follow using the *function:definitions* and the table of *variable:bindings* where necessary.

When a *variable* occurs within the body of a *function*, and this *variable* is an *argument:parameter* of the *function*, the proper *binding* for the *variable* is the *argument* corresponding to its use as an *argument:parameter*. Only when you cannot obtain a *binding* for a *variable* in this way, make use of the table of *variable: bindings*.

```
FUNCTION POLY(X) ; 2*X↑2+3*X-5;
FUNCTION CHOOSE(X,Y) IF X = 0 THEN Y ELSE Y-X;
FUNCTION TAKE(X,Y) IF ATOM X THEN Y ELSE IF ATOM Y THEN NIL ELSE CAR X . CDR Y
FUNCTION MAKE(X) ; X . Z;
```

Table of *bindings*:

Variable	*Binding*
U	'A
X	3
Z	7

Expressions to be evaluated:

a. POLY(3)

b. POLY(Z)

c. CHOOSE(1,-4)

d. CHOOSE(POLY(Z)-114,X)

e. MAKE(U)

f. TAKE(U,Z)

g. LIST(U, TAKE(X . Z, IF POLY(1)<1 THEN '(D E) ELSE '(F G))

Answers:

a. 22

b. 114

c. -5

d. 3

e. (A . 7)

f. 7

g. (A (3E))

PROBLEM SET 20

a. The following definition of FIBB uses an auxiliary *function* FIBB1. It gives the same answers as the definition in Example 1. Why does this definition lead to more efficient computation of FIBB for large *arguments*?

```
FUNCTION FIBB(N) ; FIBB1(N,1,2);
FUNCTION FIBB1(X,Y,Z) IF X = 1 THEN Y ELSE FIBB1(X-1,Z,Y+Z);
```

b. Is there any set of *arguments*for which SUBST as defined in Example 2 will not converge? Why or why not?

c. Define the recursive *function* COUNT having one *argument*. The *argument* may be any *S-expression*. The *value* of COUNT is the number of *atoms* (not just *identifiers*) in the *argument*.

Answers:

a. This definition is more efficient than the previous one because it avoids computing FIBB of any number more than once.

If the first definition is used to compute FIBB(4), for example, it calls FIBB(3) and FIBB(2), FIBB(3) calls FIBB(2) and FIBB(1). Thus FIBB(2) has been called twice. For large *arguments* of FIBB, this redundancy grows swiftly.

b. No. When Z is atomic, SUBST terminates explicitly with no more recursion. When Z is not atomic, SUBST is defined recursively in terms of SUBST of CAR(Z) and SUBST of CDR(Z).

The process of taking successive CAR's and CDR'd of an *S-expression* and stopping when one reaches *atoms*, always terminates.

c. FUNCTION COUNT (X); IF ATOM (X) THEN 1 ELSE COUNT (CAR X) + COUNT (CDR X);

CHAPTER 8

PROBLEM SET 21

(1) Insert *parentheses* in the following LISP 2 *expressions* in such a way that they are unambiguous assuming no rules of precedence. (2) Evaluate the *expressions* using the table:

Variable	*Value*
A	TRUE
B	( )
C	7.0
X	A
Y	(3 4)
Z	(A B)

a. CAR Y + CADR Y = C AND A

b. B AND 2+2 = 4

c. A OR 2+2 = 5

d. NOT A OR B OR X = Y

e. IF A OR B THEN C

f. IF C THEN C ELSE 'C

g. NOT(A AND B)

h. NOT A AND B

Answers:

	(1)	(2)
a.	(((CAR Y) + (CADR Y)) = C) AND A	TRUE
b.	B AND (2+2 = 4)	NIL
c.	A OR (2+2 = 5)	TRUE

d.	((NOT A) OR B) OR (X = Y)	NIL
e.	IF (A OR B) THEN C	7.0
f.	IF C THEN C ELSE 'C	7.0
g.	NOT (A AND B)	TRUE
h.	(NOT A) AND B	NIL

CHAPTER 9

PROBLEM SET 22

a. Define REV using recursion and without using *block:expressions* and *statements*.

b. Define REV by means of a single non-recursive *function:definition* using *block:expressions* and *statements*.

c. Define the LISP *function* SINE(X,N) that computes an approximation to the sin of X by summing the first N terms of the sequence

$$\sin(x)=x/1!-x^3/3!-x^5/5!-x^7/7!\ldots.$$

(Do not use the LISP system *function* SIN.)

Answers:

a.
```
FUNCTION REV(X); REV1(X,NIL);
FUNCTION REV1(X,Y); IF NULL X THEN Y ELSE REV1(CDR X, REV(CAR X) . Y);
```

```
b.  FUNCTION REV(X); BEGIN SYMBOL Y,U,V;
          A:  IF NULL X THEN (IF NULL U THEN RETURN Y ELSE GO B);
              U←X . U;
              V←Y . V;
              Y←NIL;
              X←CAR X;

          IF NOT ATOM X THEN GO A ELSE Y←X;

          B:  Y←Y . CAR V;
              X←CDAR U;
              U←CDR U;
              V←CDR V;
              GO A
    END ;

c.  FUNCTION SINE(X,N); BEGIN INTEGER I; REAL A;
          I←1;
          A←0;
    L:  IF I>N THEN RETURN A;
        A←A + X↑(2*I-1)/FACTORIAL(2*I-1);
        I←I + 1;
        GO L
    END ;
```

CHAPTER 10

PROBLEM SET 23

Examine the *statement* GO A in each of the following miniature *programs* and decide whether or not it is legal, and why or why not.

a.
```
FUNCTION FN(X) BEGIN A: RETURN G(X) END;
FUNCTION G(X) BEGIN GO A END;
```

b.
```
BEGIN INTEGER Y;
        BEGIN REAL X; GO A END;
        BEGIN A: Y←3 END
END
```

c.
```
BEGIN INTEGER Y;
        BEGIN GO A END;
        BEGIN REAL X; A: Y←3 END
END
```

d.
```
BEGIN INTEGER Y;
        BEGIN GO A END
        BEGIN A: Y←3 END
```

e.
```
BEGIN GO A; FN(BEGIN A: RETURN X END) END
```

f.
```
BEGIN -BEGIN GO A END; BEGIN A: END END
```

Answers:

a. Illegal for two reasons. Each *block* in the example is a *block: expression* because each is the body of a *function:definition*. It is illegal for a *go:statement* (1) to transfer out of an *expression*and (2) to transfer into an *expression*.

b. Legal. A *go:statement* may transfer out of a *block:statement* and into a *compound:statement*.

c. Illegal. A *go:statement* may transfer out of a *compound:statement*, but it may not transfer into a *block:statement*.

d. Legal. A *go:statement* may transfer out of a *compound:statement* and into another *compound:statement*.

e. Illegal. The *argument* of FN is a *block:expression* and a *go: statement* may not transfer into it.

f. Illegal. The *minus:sign* (-) before a *block* determines that the *block* is a *block:expression*; a *go:statement* may not transfer out of it.

DRAFT

DRAFT

Distribution

B. Barancik	2105	
J. Barnett	2025	
R. Berman	4317	
E. Book	2332	
R. Bosak	2013	
J. Burger	9919	
D. Drukey	2105	
Marsha Drapkin	9723	(2)
S. Feingold	9525	
D. Firth	2310	
M. Howard	2042	
H. Howell	9912	
A. Irvine	9627	
E. Jacobs	2344	
B. Jones	2231	
S. Kameny(50)	2009	
R. Long	9913	
E. Myer	2227	
M. Perstein	2334	
D. Perry	2042	
V. Schorre	2330	
J. Schwartz	2123	
R. Simmons	9439	
E. Stefferud	9734	
A. Vorhaus	2213	
C. Weissman(10)	2214	
R. Wolfson	2368	

DRAFT