L I S P

by Eric Norman

This document describes LISP, a high-level language for symbolic computations. The intent is to describe LISP in general. However, specifics are given concerning a particular version, namely the LISP system for the UNIVAC 1108 as implemented at the University of Wisconsin Computing Center.

Contents

1108 LISP

Contents

1108 LISP

Contents

1108 LISP

Chapter 1
INTRODUCTION TO LISP

LISP is yet another programming language that belongs in the category of list-processing languages; that is, it deals with (manipulates, performs computations on) objects that have a certain amount of structure. Although there is essentially only one type of structured object available in LISP, it is a very general one and can serve to represent any type of structured object desired (although sometimes not efficiently). LISP is at its best when dealing with objects that have unpredictable sizes, like representations of theorems in propositional calculus, possible moves to be made in a game such as chess, complex molecules in organic chemistry, or electrical circuits; LISP has been used to solve problems in all these areas.

It has been found that once people get used to LISP, it becomes an extremely useful and powerful tool. This is because LISP assumes most of the "dirty-work" of programming; namely, allocation of memory, saving temporary results and the like. However, it does take a while before one can feel at ease with LISP. This is caused mostly by the syntax which, although very simple and consistent, is somewhat difficult to read (the claim is that LISP is really an acronym for Lots of Insipid Stupid Parentheses).

Furthermore, LISP requires that one think in a manner different from other programming languages; specifically, LISP emphasizes the expression instead of the statement. By an expression we mean something that specifies a value to be computed; a statement indicates something to be done with such a value. The justification for this approach is that the value computed by an expression is that in which we are really interested. For example, when we write an ALGOL assignment statement such as V := A + B, the important thing is the sum of A and B;

the fact that we assign their sum to V is only a means to an end; i.e., we do the assignment because we are going to need the sum of A and B later, either because it is the answer we want or because it is going to be used in another expression. In LISP such behavior is discouraged. Instead of computing the sum of A and B beforehand, we compute it when we want it; that is, we write an expression that computes A + B as a subexpression of some other expression that needs this sum. Or if A + B is the answer that we desire, then we need do nothing more than write it; the LISP system assumes that we are interested in knowing what the sum is and therefore prints it.

The impact of the LISP approach is that in order to understand LISP, we have to forget that we ever knew about such things as assignment statements or sequences of instructions and begin thinking of ways to build up expressions that describe the result we want. Another way of saying all this is that in LISP we describe the answer that we want and not the steps that the machine is to perform in order to compute it.

This programming approach makes LISP a very high-level language and this paper describes it as such. We are going to talk about LISP in more or less abstract concepts and not concern ourselves with the details of what is actually happening inside some computer. Readers who are interested in such details are encouraged to consult other references such as the LISP 1.5 PRIMER (by Weismann; Dickenson Publishing Company, Inc., Belmont, California, 1967) or the LISP 1.5 PROGRAMMER'S MANUAL (by McCarthy, et.al; The M.I.T. Press; Cambridge, Massachusetts, 1962).

Although the primary purpose of this paper is to describe LISP, we are also going to describe a particular LISP system, namely, The University of Wisconsin Implementation for the UNIVAC 1108 computer. It will not always be indicated when LISP in general or 1108 LISP in particular is being described. The assumption being that this will usually be obvious. In those cases where the distinction is not obvious, it is so intentionally. The reason is that 1108 LISP has fundamental differences in design from other LISP systems. For those familiar with other systems, the two most important differences are the manner in which functions are treated and the manner of communicating with the LISP supervisor. It is hoped that the treatment of LISP provided herein is much "cleaner" and less confusing than that of other LISP systems.

We shall be more interested in developing the principles that underlie LISP than we are in describing what LISP actually is. That is, the following chapters are intended to convey only enough information so that we can get a feeling for LISP and how it might be used. Appendix A is included to give all the specifics really available in 1108 LISP.

Chapter 2
THE UNIVERSE OF DISCOURSE

In order to gain an understanding of any programming language, we must eventually be able to answer three questions. First, what objects are available to manipulate; that is, what types of data are allowed? Second, what are the manipulations or computations that we may perform on these objects? And finally, how do we indicate the computations that we desire; i.e., how do we write a program? Although these questions are logically asked in this order, we shall answer them in a different order; namely, we are going to answer the first, then the third, and finally the second. The reason is mostly that in the case of LISP it seems easier to answer them this way, but we shall also find that it is impossible to answer any one of them without giving at least some hint as to the answers of the others.

In this chapter, we shall describe the objects that LISP is capable of manipulating and also give the rules for writing them. That is, we shall indicate which character sequences that we have to submit as input (e.g., via a punched card or teletype) in order to specify a certain LISP object to the system. When the LISP input-routine detects a character sequence in its input that represents a LISP object, it creates the appropriate representation in the memory of the computer. The LISP system will deal exclusively with this internal representation as it goes about its business and will only reconvert it to external form when it is necessary to print a LISP object.

2.1 ATOMIC OBJECTS

Our first concern shall be those objects that are atomic. We say that an object is atomic because we do not consider it to be built out of other objects. The general rule for writing atomic objects is that they must be delimited on both sides by a punctuation mark. The punctuation marks of LISP are space, comma, period, and left- and right-parentheses. In 1108 LISP additional punctuation marks are included, namely: square brackets, broken brackets (greater than and less than signs), the apostrophe, and the question mark punch.

2.1.1 Numbers

As with most other languages, LISP includes numbers in its universe of discourse and, as usual, there are two kinds: integers and reals.

Integers are written as any sequence of decimal digits optionally preceded by a sign and optionally followed by the letter "E" and a decimal scale factor. Examples of integers are: 0, +3, 10, -256, 1E6 (same as 1000000), and -0. Examples of non-integers are: +, A, 3AB, -10EE.

Integers may also be written in octal. This is indicated by a sequence of octal digits with optional sign and followed by the letter "Q" and an octal scale factor. Legal octal numbers are 0Q, 77Q3 (same as 77000Q), and -77Q2. A minus sign indicates that the one's complement is desired and scaling is performed by doing a circular left shift of the signed result; hence, -77Q2 is the same as 777777770077Q.

Real numbers must contain a decimal point, which may not be the first character, and may optionally be followed by the letter "E" and a decimal exponent. Examples of real numbers are 0.0, 3.14159, 4.3E10, and -2.061E-22.

2.1.2 Atomic Symbols

One of the features of LISP that allows symbolic computations is the admission of atomic symbols into the universe of discourse. For the time being, we can think of an atomic symbol as a "character string" although we shall discover later that it is really a little more. We usually write an atomic symbol as any sequence of characters that begins with a letter of the alphabet; however, the 1108 version of LISP will treat any character sequence that cannot be interpreted as a number as an atomic symbol.

Sometimes the need arises to write atomic symbols that contain punctuation marks. This is done by preceding the offending character with an "!". The "!" means that the object being written is an atomic symbol and that the next character is to be included in this atomic symbol no matter what it is.

Examples of atomic symbols are:

ABC, NIL, +10EE, A!(B (prints as "A(B"), A-VERY-LONG-ATOMIC-SYMBOL, 2!3 (note, this is an atomic symbol, not a number; furthermore, it is the same atomic symbol as !23), and !! (prints as "!").

We must notice that "atomic symbol" and "atomic object" do not mean the same thing. The set of atomic objects so far includes as (disjoint) subsets the sets of atomic symbols, integers, octals, and reals. In order to discourage confusion, we shall henceforth use "atom" to refer to atomic objects in general and use "atomic symbol" when we mean exactly that.

2.1.3 Unprintable Objects

There is a class of objects in LISP that cannot be introduced as input; that is, there is no way that an internal representation can be generated for them by the input routine. The only way such objects can be created is by internal computations. The most important member of this class is the function. The impact of allowing a function to be a LISP object is that it may be manipulated as freely as any other object. LISP is somewhat unique in this respect.

Because such objects exist, the system needs a convention to indicate when an attempt is made to print one. Since an unprintable object is almost always a named function, the system will print such a function as [FN], where FN is an atomic symbol that names the function (we shall discover shortly what we mean when we say that a function is "named" by an atomic symbol). In the rare case when an unprintable object is not a named function, the 1108 LISP system will produce a character sequence something like

[4:44006Q]. If one of these funny things appears in our output, it probably indicates that we have made an error; so, for most purposes, such a character sequence should be interpreted as a diagnostic meaning "attempt to print unprintable object".

2.2 COMPOSITE OBJECTS

So far, our universe of discourse is known to contain atoms. In order to expand it even further, we will allow objects to be put together to form larger objects. The LISP operation that allows us to do this is called construction and works as follows: Given any two objects, say A_1 and A_2, we can group them together to form a new object, which we shall write $(A_1 \cdot A_2)$.

Now given only one atom A and the construction operation, we have an infinite class of objects, i.e., A, (A·A), (A·(A·A)), ((A·A)·A), and so forth.

When writing these objects, we should include a space before and after the period in order to avoid confusion with real numbers, although the spaces are not required if no confusion is possible.

2.2.1 Lists

This set of composite objects is fine in theory, but nobody would want to have to write or read a large one; therefore, we shall concentrate on a certain subset of these constructed objects that is much more convenient. We shall call members of the subset lists, i.e., we want the ability to speak of lists of atoms, or lists of lists, etc.

First, we need a conventional representation for the empty list. LISP uses the atomic symbol NIL for this purpose. Now, a list whose elements are the objects $X_1, X_2, X_3, \cdots X_n$ is represented by the composite object

$$(X_1 \cdot (X_2 \cdot (X_3 \cdot \cdots (X_n \cdot \text{NIL}))))$$

We may write such an object as:

$(X_1\ X_2\ X_3\ \cdots\ X_n)$

with spaces or commas separating the elements if they are atomic. Notice, a list is not a new type of object, but only a certain type of composite object that is far more convenient to write.

Looking ahead a bit, we can consider that we will want to perform operations with lists such as getting or removing the first element. These operations are very easy since we have functions that get the left and right parts of composite objects; i.e., the left-hand part of a list is its first element and the right-hand part is the list that remains after the first element is removed. Furthermore, when we remove all the elements, i.e., when we take the right-hand part of a list of one element like $(X_n \cdot \text{NIL})$, we get the empty list just as we would expect.

We can notice the rule that a composite object may be written completely in list notation if and only if every atomic right-hand part of a constructed object contained therein is the atomic symbol NIL.

A few examples of the list notation are:

(A · (B · NIL)) may be written (A B);
((A · NIL) · ((B · NIL) · NIL)) is the same as ((A) (B));
NIL is the same as ( );
((A · B) · ((C · D) · NIL)) may be written ((A·B) (C·D));
while (A · (B · (C · D))) cannot be written as a list because the D is in the wrong place.

As a sidelight, the last example above is so close to being a list that it may be written as (A B C . D) and will in fact be printed that way.

2.3 ADDITIONAL SYNTATIC RULES

The following rules apply when punching objects:

(1) Atoms may not cross card boundaries.

(2) Spaces and commas are equivalent, i.e., (A,B,C) is the same as (A B C).

(3) A punctuation mark preceded by or followed by any number of spaces or commas is equivalent to that punctuation mark.

(4) A question mark punched anywhere on a card causes it and all characters following it to be ignored until the end of the card.

(5) Square and crooked brackets ([,],<, and >) may be used in place of parentheses.

The three types of parentheses also have another useful feature. Whenever a closing bracket is encountered, i.e., ")", "]", or ">", it is forced to match the corresponding opening bracket, i.e., "(", "[", or "<", by generating closing brackets that match opening brackets until the correct match is obtained. For example, if we are going to punch a large composite object we may start it with a "[" and use parentheses within the object. Now, when we get to the end, we do not have to count the number of unmatched left-parentheses that we have written so far; we simply punch a "]" and the system will generate the correct number of right-parentheses. That is [A (B C] is equivalent to (A (B C)) and (A[B C]< (D E) F) is equivalent to (A(B C)((D E) F)).

We should point out that the output routine only uses parentheses. Square brackets are reserved for printing unprintable objects.

Chapter 3
THE LANGUAGE

Now that we have a class of objects at our disposal, we have answered the first question presented at the beginning of Chapter 2. In this chapter we shall answer the third question, namely, how to write LISP programs.

3.1 EVALUATION

As indicated in the introduction, the way that we do things in LISP is by evaluating expressions; that is, assuming that the answer we seek is represented by some LISP object (as it usually is), then all we do is submit an expression that will compute this object. The LISP system will evaluate the expression and print its value.

Intuitively every expression denotes some value by giving a method of obtaining that value in terms of other known or computable values. For instance, we are familiar with (non-LISP) expressions such as 5+1, SIN($\pi \times$ N/4), or $\sum_{x=0}^{10} f(x)$.

In the second example we have expressed a value in terms of the three values N, π, and 4, right? Absolutely not! We have expressed the value in terms of six values, namely, the values that we indicate by the symbols SIN, π, $\times$, N, /, and 4. The point is that in order to evaluate an expression, we have to know the values of all symbols appearing therein, and, furthermore, we must know what we are supposed to do with these values. In the example above, we supposedly know the values of the symbols π, $\times$, and N and we (by convention) agree that the expression $\pi \times$ N indicates that we are supposed to apply the value (function) represented by $\times$ to the values (arguments) represented by π and 4.

The same principles apply in LISP, although there is no restriction that the value of an expression must be a number; it may be any LISP object. What we are going to describe in this chapter are the conventions or rules that allow us to assign values to symbols appearing in expressions and that indicate to us what we are supposed to do with these values.

Application of the rules will henceforth be called "evaluation"; i.e., whenever we say something like "evaluate" from now on, we are referring specifically to the rules about to be described. We shall also reserve the word "value" to use only when we are referring to the object that arises by evaluating a LISP expression. Traditionally we tend to use the word "value" in another context, namely, when we say something like "the value of SIN for $\pi/2$ is 1". Since this is not the way we are using the word "value" here, we shall use the word "result" in this context; i.e., we shall say "the result of SIN" instead of "the value of SIN".

The question now arises, just what is a LISP expression? In LISP, an expression is just another LISP object; that is, LISP programs are also LISP data and the only reason that we would call such an object "program" instead of "data" is that we intend to use it as such. So henceforth, by "expression" we shall mean an object that we intend to evaluate. For convenience, we shall not allow all objects to be expressions, but rather only those objects that can be written in list-notation; i.e., all expressions will be either atoms or lists.

3.2 FUNCTIONAL APPLICATION

When the expression to be evaluated is a list, say (A_1 A_2 ··· A_n), then the rule (with one exception very shortly) is to evaluate each of the expressions A_1 through A_n and then apply the function that arose as the value of A_1 (if the value of A_1 is not a function, we have made a mistake) to the arguments that arose as the values of A_2 through A_n. The result of applying this function is taken as the value of the entire expression. For example, assuming that we already know that the atoms PLUS, 1, and 2 evaluate to the addition function and the numbers 1 and 2, then the expression

(PLUS 1 2) evaluates to 3.

The one exception to this rule is when A_1 is (note: this is not the same as saying A_1 evaluates to) an atomic symbol that indicates a non-standard method of evaluation. Such expressions are called special-forms and will be covered in the next chapter. For the time being, we can mention that the reason we have special-forms is that we desire certain operations that do not quite behave like functions. Specifically, they do not want their "arguments" evaluated.

For example
QUOTE
COND
LAMBDA

3.3 EVALUATION OF ATOMS

When we evaluate an expression that is an atom, we use the following rule:

If the atom is not an atomic symbol, then the value of the expression is the expression itself; i.e., 1 evaluates to 1, 2 to 2, etc. However, if it is an atomic symbol, then it truly must be a symbol; that is, it must stand for some LISP object. This associated object is taken as the value of the atomic symbol. We shall call the mechanism by which an atomic symbol is made to stand for some other object a binding. In LISP there are two mechanisms by which such a binding might be effected; they are called constant and fluid bindings. How such bindings are established and maintained will be covered later. For the time being, we shall describe the intuitive notion of what they mean.

3.3.1 Variables

Fluid bindings are used to associate the dummy variables appearing in the definition of a function with the arguments of the function when it is applied. Therefore, we shall use the word "variable" to refer to an atomic symbol that has a fluid binding. Fluid bindings are temporary, since they only exist during the time that a function is being applied. Exactly how they work will be discussed below when we cover functional abstraction under the special-form LAMBDA.

3.3.2 Constants

Constant bindings are those associations that remain in effect until explicitly changed by the user and are, therefore, more permanent than fluid bindings. They always take precedence over fluid bindings, i.e., if a constant binding has been established for an atomic symbol, any attempt to establish a fluid binding for it is nonsense since the constant binding will always be used.

bullshit
— unnec
essary
constrain

The most important use of the constant binding is to give a name to a function. For example, the system provides a constant binding between the atomic symbol PLUS and the addition function; hence, when we evaluate (PLUS 1 2), the value of PLUS is the addition function that we desire. It is also possible, as we shall see below, to establish such a binding for a function of our own choosing so that we may use the function whenever we desire.

3.4 SUMMARY

Let us apply these rules of evaluation to a simple case so that we can get a better understanding of what is going on. In the beginning of this chapter we used the example SIN(π × N/4); in LISP, it is written (SIN(QUOTIENT(TIMES PI N) 4.0)). Let us evaluate this expression (making appropriate assumptions when we need to).

```
Evaluate (SIN(QUOTIENT(TIMES PI N) 4.0)).  It is a list, so first
evaluate SIN and (QUOTIENT(TIMES PI N) 4.0).
     Evaluate  SIN.
     It is an atomic symbol, so let us assume that it is
     constantly bound to that function that takes the sine
     of its argument.
     Evaluate (QUOTIENT(TIMES PI N) 4.0).
     Again a list, so evaluate QUOTIENT, (TIMES PI N), and 4.0 .
          Evaluate QUOTIENT.
          Assume constantly bound to division function.
          Evaluate (TIMES PI N).
          Another list, so
               Evaluate TIMES.
               Another assumed constant .
               Evaluate PI.
```

Assumed constantly bound to a number.

Evaluate N.

Let us assume that it is fluidly bound to some number, i.e., that N represents the argument of a function that uses this expression to compute its result.

Apply function, i.e., do multiplication.

Evaluate 4.0.

Atom, but not atomic symbol, so value is 4.0.

Apply division function, i.e., take quotient.

Apply function. We are finally computing the sine that we desired.

It seems like an awful lot of work to evaluate an expression whose value is so obvious, but this is so because we chose an expression that had an obvious value. For instance, what if our expression were something like ((FN X) Y)? The value of this expression is not so obvious, but our evaluation scheme still applies. Our rule says that to evaluate the expression, we first evaluate (FN X) and Y and then apply the value of (FN X) to the value of Y. Now here is something that we have not been accustomed to doing, i.e. computing a function to be applied instead of simply using the name of it. According to the LISP evaluation method, we always compute the function to be applied; however, we usually perform the computation by evaluating an atomic symbol that has a constant binding to the desired function, so that, in most cases, we are simply writing the name of the function we desire.

Chapter 4
THE OPERATIONS

Finally, we get to the second question of Chapter 2, namely what manipulations can we do with LISP? We shall begin by learning a few of the functions that are provided by the system, then cover a few special-forms that enable us to build complicated expresstions, and finally see how we may create our own functions. But first, we want to establish a notational convention for functions.

We shall often have occasion to talk about the function that is bound (in the constant fashion) to an atomic symbol. We shall write this function as [X], where X is the atomic symbol whose value is the function being referred to.

The reason for establishing this convention is twofold. First, it is going to save us a lot of writing; and second, the system uses this convention when it prints a function, which remember, is unprintable. That is, when the need arises to print an unprintable object, the LISP system searches through all atomic symbols looking for one to which the object is constantly bound and, if sucessful, prints it out in the form [X].

4.1 SOME PRIMITIVE FUNCTIONS

4.1.1 Construction

We learned above that the fundamental building operation of LISP is construction. There is a function provided by the system that does exactly this and is bound to the atomic symbol CONS.

If, for example, the atomic symbols A and B are currently bound to the number 1 and the list (2 3 4), then the expression (CONS A B) will evaluate to the object (1·(2 3 4)), which we may write as the list (1 2 3 4).

4.1.2 Selection

Since we have the ability to build objects, we shall also desire the ability to take them apart. Specifically, the system provides two functions [CAR] and [CDR], which retrieve the components of a composite object; i.e., if X is currently bound to the composite object $(A_1 \cdot A_2)$, then the value of (CAR X) is A_1 and the value of (CDR X) is A_2. If [CAR] or [CDR] is applied to an atom, the value is unspecified.

Technically, these are the only two selection functions we need; however, it is often useful to be able to get at deeper components of an object. Therefore, the system provides a function for each atomic symbol of the form C···R, where the ellipsis indicates a string of A's and D's whose length, in this implementation, must be between 0 and 35, inclusive. Such functions represent successive applications of [CAR] and [CDR] from right to left, i.e., (CADDR X) evaluates to the same object that (CAR(CDR(CDR X))) does. For the sake of generality, [CR] is the identity function.

It is useful to notice what these operations do to lists. If L is currently bound to a list, then (CONS X L) creates a new list with the value of X as its first element; (CAR L) evaluates to the first element of the list L; (CDR L) evaluates to the tail of the list L; i.e., if L is (1 2 3), (CDR L) is (2 3); (CADR L) evaluates to the second element of the list; and so forth.

4.1.3 Predication

We shall also need the ability to ask questions about objects, such as, is an object atomic, or, are two objects the same? Therefore, we need a convention to represent truth and falsehood since we did not see fit to include truth values in our universe of discourse. In LISP, the atomic symbol NIL is interpreted as falsehood and any other object is interpreted as truth. To facilitate use of truth values, a few constant bindings are provided automatically; namely, both the atomic symbols NIL and F are bound to falsehood (NIL). The atomic symbol T is constantly bound to itself and is conventionally used to represent truth.

LISP has a function [ATOM] whose result is true if its argument is an atom and false if it is composite. There is also a function [EQ] whose result is true if its two arguments are the same atomic symbol and false if they are different atomic symbols. [EQ] is intended to be used on atomic symbols only; if it is not given atomic symbols as arguments, its result is not specified (although it does give one). To compare two objects in general we use [EQUAL], whose result is true if its two arguments are the same (meaning that they would appear identical if printed), whether they be atomic symbols, numbers, or composite objects. However, it does not work for functions (after all, nothing could).

LISP also has the function [NULL], whose result is true if its argument is the empty list and false otherwise. Due to the manner in which lists are represented and the fact that NIL is constantly bound to itself, (NULL X) has the same value as (EQ X NIL).

4.2 SPECIAL-FORMS

We shall now cover a few of the special-forms of LISP, that is, those cases where an expression of the form (A_1 A_2 ··· A_n) does not indicate a functional application, but rather a special method of evaluation that will depend on what A_1 is. One word of warning, the atomic symbol (A_1 above) used to indicate a special-form does have a constant binding to an unprintable object; however, this object is not a function and is not even useful. The only reason that this is mentioned is that it does prohibit us from using such an atomic symbol as a variable.

4.2.1 QUOTE

A very important special-form is signaled by the appearance of the atomic symbol QUOTE. When QUOTE is encountered, it indicates that the object appearing to be the argument is the actual object desired and not an expression that should be evaluated to get the object. For instance, (CONS(QUOTE A)(QUOTE (B C))) evaluates to (A B C); A and (B C) are not evaluated.

This special-form is used so often that 1108 LISP allows an abbreviation for it. Namely, we can write 'A instead of (QUOTE A). This translation is effected at the time the object is read in; i.e., whenever the input routine encounters the punctuation mark apostrophe, it reads the next object and quotes it. Hence, the above example is more conveniently written as (CONS 'A' (B C)).

4.2.2 COND

The if-then-else idea is embodied in the special-form represented by COND. We write a conditional expression like so:

$(\text{COND } (P_1\ E_1)\ (P_2\ E_2) \cdots (P_n\ E_n))$

The rule for evaluating such a form is as follows: Each P is evaluated from left to right until one is found whose value is true; then the associated E is evaluated and its value is the value of the entire conditional expression. It is the case that no P after the first true one and no E except the one selected will be evaluated. If none of the P's is true, then the value of the conditional expression is unspecified. We usually write T as the last P to guarantee that the conditional expression always has a value.

As an example of the use of COND, suppose that M and N are currently bound to numbers and that we wish to obtain the larger of the two; we can do this by writing

```
(COND ((GREATERP M N) M) (T N))
```

where the result of [GREATERP] is true if its first argument is greater than its second.

4.2.3 AND, OR, NOT

The special-forms, AND and OR, and the function [NOT] are usually used in one of the P's of a conditional expression, although there is no restriction that they must be.

The format for AND is:

$(\text{AND } P_1\ P_2 \cdots P_n)$

The rule for evaluating AND is that the P's are evaluated from left to right until one turns out to be false. If one does, the value of the expression is false and no more P's are evaluated;

if no P is false, the value of the expression is true. The value of (AND), by the above rule, is true.

OR is similar to AND, except that a P that is true stops the evaluation with a value of true and no true P causes the value of the expression to be false.

Due to the representations of truth and falsehood in LISP, [NOT] and [NULL] are the same functions.

4.2.4 LAMBDA

We come now to a special-form that gives LISP immense power. The special-form signaled by the appearance of the atomic symbol LAMBDA allows us to create functions. The idea is similar to the use of a declaration, such as PROCEDURE in ALGOL. However, in LISP it is not to be thought of as a declaration; that is, LAMBDA does not allow us to declare a function, it allows us to compute one.

The special-forms and primitive functions that we have already learned allow us to build up complicted expressions. LAMBDA allows us to specify that an expression so built is to be used to compute the result of a function whenever the function is applied. To do so we must also specify which variables appearing in the expression are intended to stand for the arguments of the function being developed.

For instance, under COND we wrote the expression

```
    (COND ((GREATERP M N) M)(T N))
```

which evaluated to the larger of M and N. Now, if we write

```
    (LAMBDA (M N)(COND ((GREATERP M N) M)(T N)))
```

the value of this expression is a function that will select the larger of its two arguments. That is, we have created a function which uses the rule given by the conditional-expression to compute its result and specified that the variables M and N appearing in the expression are to stand for the first and second arguments, respectively, of the function. For example,

```
    ((LAMBDA (X Y)(COND((GREATERP X Y) X)(T Y))) M N)
```

also evaluates to the larger of M and N.

Those who are familiar with the lambda-calculus will recognize this concept immediately; however, we shall find that the treatment of free variables in LISP differs from that of the lambda-calculus.

The format of the special-form LAMBDA is:

$$(\text{LAMBDA } (V_1\ V_2 \cdots V_n)\ E)$$

where each V must be an atomic symbol that will never have a constant binding, and E is any expression (that will probably contain occurrences of the V's). The value of this special-form is a function of n arguments that, whenever applied, will compute its result by evaluating E in an environment in which each V is fluidly bound to the correspond argument.

Now we have to be careful to notice what gets evaluated and when. Specifically, evaluating a LAMBDA-expression only creates a function; the expression E that will be used to compute the result of the function is not evaluated when the function is created, but rather when we apply the function to arguments. Furthermore, immediately before it is evaluated, the variables appearing after the LAMBDA are given fluid bindings to the arguments of the function so that during the evaluation of E, these variables stand for the right things, namely, the arguments to which the function is being applied. After the result is computed, these fluid bindings disappear and any fluid bindings that the variables had before application reappear.

By the way, there is nothing illegal about having functions of no arguments; we create them with expressions like (LAMBDA ( ) E) or (LAMBDA NIL E).

4.3 FUNCTIONAL DEFINITION

Now that we have the ability to create a function via LAMBDA, we want to be able to give the function a name so that we may use it in more than one place. For this purpose we usually use a constant binding.

There is an operator in LISP bound to the atomic symbol CSET which, besides delivering a value when applied to arguments, also has a permanent effect on the system (such operators are called pseudo-functions in LISP). In this case the effect is to establish a constant binding between the first argument of [CSET], which must be an atomic symbol, and its second argument, which may be any object. The value of an expression (CSET (QUOTE X) Y) is the value of Y, but the important thing is the effect; namely, after the evaluation of this expression, the value of X will always be whatever Y evaluated to above unless it is explicitly changed by another CSET.

Ergo, if we evaluate the expression

```
(CSET 'MAX
   (LAMBDA (X Y)
      (COND
      ((GREATERP X Y) X)
      (T Y)
      )))
```

the special-form LAMBDA computes a function that will select the larger of its two arguments, and this function is then bound constantly to the atomic symbol MAX by [CSET]. Now, all we have to do hereafter to get the larger of two numbers, say N_1 and N_2 is evaluate the expression

(MAX N_1 N_2)

4.4 LAMBDA REVISITED

There is one more point about the LISP evaluation method that needs to be mentioned; namely, what happens if an atomic symbol is encountered during the evaluation of an expression that neither has a constant binding nor appears as a variable in a LAMBDA-form that surrounds the expression. For instance suppose that we define a function thus:

```
(CSET 'FN (LAMBDA (X) (PLUS X FV)))
```

If we ever apply this function to an argument, the expression (PLUS X FV) will have to be evaluated. We can evaluate PLUS since it is a constant and we can evaluate X since it will be bound to the argument, but what value do we assign to FV (assuming it is not a constant)? If we go back and look at the mechanism for applying a function created by the special-form LAMBDA, we remember that before the expression defining the result of the application is evaluated, the variables appearing after the LAMBDA are bound to the arguments of the function.

These bindings are added to those bindings for fluid variables that are currently active and when we evaluate a fluid variable, we look through all these bindings and take the most recent one. Therefore, the value of FV above will be that object that appeared as an argument to the most recent function that has an FV as a dummy variable and has not finished computing its result.

Example, suppose we define two or more functions in addition to the one above:

```
(CSET'F1 (LAMBDA (X FV) (FN X)))
(CSET'F2 (LAMBDA (FV X) (FN FV)))
```

It would seem, since we should be able to rename the dummy variables of a function systematically, that these two functions are the same; unfortunately, they are not. The value of (F1 1 2) is 3 since FV is bound (during the evaluation of (PLUS X FV) in [FN]) to 2; while the value of (F2 1 2) is 2 since FV gets bound to 1.

standard FV is used free

If there is no binding available for a variable that is to be evaluated, then the system will object. What happens then depends on the mode of operation (conversational or batch) and will be covered below.

4.4.1 The FUNCTION Function

It often happens that this treatment of fluid variables is not the one desired, i.e., if we use the special-form LAMBDA to create a function in which an atomic symbol appears free (it is neither included in the list of variables following the LAMBDA nor will it ever have a constant binding), the value of this variable will depend on the bindings in effect at the time the function is <ins>applied</ins>. What we often want is the bindings for these free variables that were in effect at the time the function was <ins>created</ins>.

LISP supplies a function [FUNCTION] for such cases. Specifically, [FUNCTION] takes a function as argument and delivers as its result a new function whose free variables will be bound (whenever this function is applied to arguments) to the bindings that they had when [FUNCTION] was applied.

Let us see why this is necessary. Suppose we wish to define a function that will take two functions of one argument and compose them, i.e., we want the expression ((COMPOSE CAR CDR) '(1 2 3)) to evaluate to 2 just as (CADR '(1 2 3)) would. It is tempting to try to define [COMPOSE] thus:

```
    (CSET 'COMPOSE (LAMBDA (FA FB)(LAMBDA (X) (FA (FB X)))))
```

but this does not work since as we leave [COMPOSE], the bindings for FA and FB evaporate so that when we apply the function created by [COMPOSE] (the function created by the second LAMBDA above), FA and FB are not bound to the right things. To get [COMPOSE] to work, we define it thus:

```
    (CSET 'COMPOSE
       (LAMBDA (FA FB)(FUNCTION (LAMBDA (X) (FA (FB X))))))
```

Now after we compute the desired composition, we give this function to [FUNCTION], which captures the bindings of FA and FB (and all other current bindings) and delivers a function that will use the captured bindings whenever it is applied.

The above example is well worth careful study since it not only demonstrates the necessity of [FUNCTION], but also demonstrates one of the exotic things that can be done with the special-form LAMBDA.

Chapter 5
THE LISP SYSTEM

By now we have learned the essentials of LISP. There is a lot more to LISP; however, we shall take time here to study the entire 1108 LISP system so that we can actually run programs.

5.1 THE EVALUATOR

The heart of the LISP system is the evaluator, which will evaluate an expression by the method we have described above. In LISP, the evaluator is available to the user via the function [EVAL]. That is, [EVAL], when supplied an object as an argument, will compute the value of that object. The environment that [EVAL] uses, i.e., bindings for fluid variables, is those bindings that are active at the time [EVAL] is applied. However, it is rarely necessary to use [EVAL] explicitly, since the standard LISP supervisor provides it automatically.

5.2 THE STANDARD SUPERVISOR

The standard supervisor of LISP will operate in the following manner. First, it requests an object to be evaluated by printing the message "EXPRESSION TO EVALUATE:". It then reads an expression that we supply and has the evaluator compute the value of the expression. Finally, it prints the message "VALUE IS:" followed by the value computed for the expression and returns to repeat this cycle.

Therefore, a normal LISP job is a series of expressions that are to be evaluated. The first expressions probably use pseudo-functions (like [CSET]) that establish as constants the functions that we want to use. These will be followed by expressions that use these functions to compute the answers that we are seeking.

5.3 AN EXAMPLE

The following is an actual printout of a LISP run. This run defines and tests functions that perform basic computations on sets, assuming that a set, say {A1 A2 A3}, is represented by the list (A1 A2 A3).

```
EXPRESSION TO EVALUATE:

?CARTESIAN PRODUCT OF S1 & S2
(CSET 'CARTESIAN(LAMBDA(S1 S2)
    (INDEX S1 NIL(FUNCTION(LAMBDA(I1 J1)
    (INDEX S2 J1 (FUNCTION(LAMBDA(I2 J2)(CONS(CONS I1 I2)J2>

VALUE IS:

[CARTESIAN]

EXPRESSION TO EVALUATE:

?ALL SUBSETS OF S
(CSET'POWER(LAMBDA(S)
    (COND
    ((NULL S)'(NIL))
    (T(INDEX(POWER(CDR S))NIL
                    (FUNCTION(LAMBDA(I J)(CONS(CONS(CAR S)I)(CONS I J>

VALUE IS:

[POWER]

EXPRESSION TO EVALUATE:

?TEST CASES
(UNION '(A1 A2 A3 A5) '(A1 A3 A4 A6))

VALUE IS:

(A2 A5 A1 A3 A4 A6)

EXPRESSION TO EVALUATE:

(INTERSECTION'(A1 A2 A3 A5)'(A1 A3 A4 A6))

VALUE IS:

(A1 A3)

EXPRESSION TO EVALUATE:

(CARTESIAN'(A1 A2 A0) '(B1 B2))

VALUE IS:

((A1 . B1) (A1 . B2) (A2 . B1) (A2 . B2) (A0 . B1) (A0 . B2))

EXPRESSION TO EVALUATE:

(POWER'(A B C D))

VALUE IS:

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

Chapter 6
EMBELLISHMENTS

What we have learned so far is theoretically sufficient; however, there are many additional facilities in LISP that make it much easier to use and more efficient to run.

6.1 PSEUDO-FUNCTIONS

In Chapter 4, Section 3., we introduced the pseudo-function [CSET], whose primary purpose was to affect the system rather than deliver a value. In this section we shall learn of a few more that turn out to be useful.

6.1.1 READ

For input purposes LISP provides the function [READ], which is a function of no arguments whose result is the next object appearing on the input device. For instance, if we submit the following card image to the supervisor

```
     (READ) (A·B)
```

the value is the object (A·B).

6.1.2 PRINT

We can produce our own output via the pseudo-function [PRINT]. [PRINT] will print its argument on the output device and deliver that object as its result.

6.1.3 RPLACA,RPLACD

The pseudo-functions [RPLACA] and [RPLACD] allow us to alter composite objects. For instance, if we evaluate the expression (RPLACA '(A·B) 'X), the object (A·B) is changed to the object (X·B); similarly, the effect of the expression (RPLACD '(A·B) 'X) is to change (A·B) to (A·X). The result in both cases is the first argument, which will be the altered object. We must be sure to notice the essential difference between an expression such as (RPLACA '(A·B) 'X) and (CONS 'X (CDR '(A·B))). The value in either case is an object (X·B); however, the effect is different. In the latter case we have constructed a new object; while in the former, we have altered the structure of an already existing object.

Since these pseudo-functions permanently alter representations within the computer, they must be used with extreme caution. For instance, if we evaluate the expression

```
    ((LAMBDA (X) (RPLACD (CDR X) X)) '(A B))
```

the value and effect is the infinite list

```
    (B A B A B A ···).
```

If an infinite list such as this is given to a function that searches a list (like [EQUAL]), an infinite loop could occur.

6.2 DO

It often happens that we want to use such pseudo-functions for their effect only and ignore their results. In order to accomplish this, we may use the special-form DO. The format of DO is

$$(\mathrm{DO}\ E_1\ E_2\ \cdots\ E_n)$$

and the rule for evaluating it is to evaluate each E_i from left to right and ignore all values except the last, which becomes the value of the DO-expression. For example, to print a request and then read an answer the expression

```
    (DO (PRINT REQUEST) (READ))
```

has the effect and value desired.

6.3 PROG

Also available in LISP is another special-form that allows us to execute statements in a manner similar to other languages, such as ALGOL or FORTRAN. This special mode of operations is signaled by the appearance of the special-form PROG, and is written thus:

$(PROG\ (V_1\ V_2 \cdots V_n)\ S_1\ S_2\ S_3 \cdots S_n)$

Each V_1 is an atomic symbol representing a variable that will be local to the PROG and each S_1 is either an expression that will be evaluated and the value discarded or an atomic symbol that represents a label and can be referenced by the special-form GO.

The rule for evaluating a PROG is as follows: Upon entry to the PROG, each local variable is bound to NIL and these bindings are added to those bindings currently active (i.e., these bindings are maintained in the same fashion as the variables following a LAMBDA). Then the non-atomic statements are evaluated in sequence using these expanded bindings. The value of each such expression will be discarded; hence, it behooves us to use pseudo-functions in the S's so that something useful happens. When we run out of statements, the PROG is exited, NIL becomes the value, and the bindings for the PROG variables disappear.

If PROG were used only as above, its usefulness would be limited; therefore, there are some more special-forms that are used in conjunction with a PROG.

6.3.1 SET,SETQ

The pseudo-function [SET] is used to change the binding of a variable. It is written like [CSET] and operates identically if its first argument already has a constant binding; however, if its first argument is not a constant, then it looks through the fluid bindings currently active and changes the most recent binding of its first argument so that it is bound to the second argument. If this fails (i.e., if it cannot find a fluid binding), [SET] will create a new fluid binding for the variable and put it at such a place that it is local to the most recent PROG, as if it had been written there to begin with.

As with [CSET], the normal use of [SET] will have the first argument quoted, e.g.,

```
(SET (QUOTE V) E)
```

hence, the special-form SETQ is provided that effects this quoting. That is, the above may be more conveniently written

```
(SETQ V E).
```

Comparing LISP with other languages we will recognize SETQ as the LISP version of the assignment statement; namely, the effect of (SETQ V E) is the same as V:=E in ALGOL or V=E in FORTRAN.

6.3.2 GO

The special-form GO is the analogue of the GO TO statement elsewhere. Whenever an expression of the form (GO L) is evaluated, the LISP system immediately returns to the most recent PROG and finds the label L among the statements comprising the body of the PROG. Evaluation will then continue with the statement following the label. If the label cannot be found in the most recent PROG, then the LISP system will complain and terminate the current evalution. We must notice that L is the label desired, not an expression that will evaluate thereto.

6.3.3 RETURN

The special-form RETURN is used to exit a PROG without falling off the end. Whenever (RETURN V) is encountered, V is evaluated, then the most recent PROG is exited with the result being the value of V.

If we wish to have a function that will reverse a list, we can take advantage of the PROG feature and define it:

```
(CSET 'REVERSE (LAMBDA (L)
    (PROG (ANSWER)
    LOOP
    (COND ((NULL L)(RETURN ANSWER)))
    (SETQ ANSWER (CONS (CAR L) ANSWER))
    (SETQ L (CDR L))
    (GO LOOP)
    )))
```

(REVERSE '(A B C)) will evaluate to (C B A) and
(REVERSE '((A B) (C D) (E F))) will evaluate to ((E F) (C D) (A B)).

6.4 PROPERTY LISTS

An often useful feature of LISP is the ability to manipulate property lists. We may think of a property list as a symbol table that is associated with each atomic symbol. Each such symbol table is a list of the form:

$$((I_1 \cdot V_1)\ (I_2 \cdot V_2) \cdots (I_n \cdot V_n))$$

where each I is an atomic symbol and each V is an object. We usually refer to a property list entry, i.e., one of the (I · V)'s, as an attribute-value pair, where I is the attribute and V is the value (not to be confused with the word "value" as used elsewhere). Every atomic symbol initially has an empty property list.

6.4.1 PUT

The pseudo-function [PUT] is used to establish or update an attribute-value pair. [PUT] takes three arguments: the first is the atomic symbol whose property list is desired, the second is an atomic symbol representing the attribute to be entered, and the third is the value to be associated with that attribute. The effect of [PUT] is either to update the entry for the attribute if one already exists or to add a new attribute-value pair if not. The result of [PUT] is its first argument.

6.4.2 GET

The function [GET] allows us to retrieve the values associated with attributes. The two arguments of [GET] are the atomic symbol whose property list is desired and the attribute whose associated value is desired. The result of [GET] is the associated value if one exists or NIL if there is no entry for the attribute.

6.4.3 Flags

There are times when having a value associated with an attribute is not important, but the presence or absence of that attribute is. For such purposes, property lists are also allowed to contain flags; that is, a flag is an atomic symbol that may appear as an element of a property list and indicates something special about the atomic symbol to whose property list it belongs.

6.4.3.1 FLAG

We insert flags on a property list by the pseudo-function [FLAG], whose first argument is the atomic symbol that is to be flagged and whose second argument is an atomic symbol that is the flag to be inserted. The result of [FLAG] is its first argument.

6.4.3.2 IFFLAG

We test for the presence of flags with [IFFLAG]. This function takes two arguments similar to [FLAG] and its result is true if and only if the flag appears on the property list.

6.5 ARITHMETIC

LISP has a supply of arithmetic functions and predicates that may be used to effect whatever number-crunching is necessary. These functions will be detailed in Appendix A; however, a few general principles will be stated here.

First, integers and octals appear to be identical to the arithmetic functions; that is, (PLUS 5 10) and (PLUS 5 12Q) both evaluate to 15.

Second, mixing real and integer arithmetic is allowed. The convention established in LISP is that if any of the arguments of an arithmetic function is real, then the result is real; otherwise, the result is an integer.

Finally, comparisons involving real numbers are subjected to a tolerance of 3.0E-6; that is, two real numbers are compared by taking the absolute value of the quotient of their difference and the first number. If this absolute value is less 3.0E-6 then the two numbers are considered equal.

Chapter 7
THE COMPLETE LISP SYSTEM

In Chapter 5 we learned enough about the 1108 LISP system to enable us to use it. We shall now cover the entire system in all its intimate detail so that we may use it more effectively.

7.1 THE EXPANDED SUPERVISOR

We saw above that the supervisor essentially cycles through a read-evaluate-print sequence. Now that we have the PROG feature, we can write such a supervisor in LISP.

```
(CSET 'LISP (LAMBDA (INPUT-GETTER)
    (PROG (VALUE)
    CYCLE
    (CLEARBUFF)
    (PRINT 'EXPRESSION! TO! EVALUATE:)
    (SETQ VALUE (EVAL (INPUT-GETTER)))
    (PRINT 'VALUE! IS:)
    (PRINT VALUE)
    (GO CYCLE)
    )))
```

The only stranger here is the pseudo-function [CLEARBUFF] which sets up the input routine to begin at the front of the next card.

It so happens that this function is actually available in the 1108 LISP system, although the variables INPUT-GETTER and VALUE and the label CYCLE are invisible. But, this does mean that we can call the LISP supervisor ourselves whenever we want to.

For example, when running in the conversational mode, we may attempt to evaluate an atomic symbol that has neither a constant nor a fluid binding. When this happens, the system will request a value from us by having us submit an expression the value of which is to be used as the value of the unbound variable. Now we may need to define a few more functions so that this value can be computed or so that the request will not be made again.

In order to do this we submit (LISP READ) as the expression to be used to compute the value (we can also submit (LISP) since [READ] is assumed if no argument is supplied. When the system evaluates this expression, a new level of supervision is established in which we may do whatever we need to do.

Eventually, we are going to want to leave this new level of supervision with a value to be used in the interrupted evaluation. Since the LISP supervisor is a PROG feature, we get out of it with a RETURN, i.e., we simply submit an expression like (RETURN V) to the current supervisor. The special-form RETURN exits from the most recent PROG, which in this case is the supervisor, and the value computed from V is used to resume the interrupted evaluation.

When LISP starts running,a level of supervison is automatically established for us. If we RETURN from this level, we do exactly what we would expect, we leave LISP and return to the 1108 executive.

The purpose of the variable INPUT-GETTER above is to allow us to use a non-standard form of input if we desire. For instance, if for some reason we prefer the function and list of arguments form of input that is used in other LISP systems, we can get it by starting out with:

```
    (CSET 'AWFUL (LAMBDA NIL (CONS (READ)(QUOTEM (READ)))))
    (CSET 'QUOTEM (LAMBDA (L)
        (COND
        ((NULL L) NIL)
        (T (CONS (LIST 'QUOTE (CAR L))(QUOTEM (CDR L))))
        )))
    (RETURN (LISP AWFUL))
```

After this we submit input as a function name followed by a list of arguments and the function [AWFUL] will be used to transform our input into expressions suitable for evaluation.

7.2 THE CONVERSATIONAL MODE

LISP will operate most effectively when used in the conversational mode. This is not due as much to ad hoc abilities as it is to the overall philosophy of LISP. Specifically, LISP allows us to do our work, so to speak, incrementally. That is, we are allowed to define a function that we know we will need, try it out on a few test cases to be sure that it works, and then forget about it and go on to define other functions. This is not the case with many other languages where all desired functions or subroutines have to be defined before any of them can be used.

However, 1108 LISP does have some abilities that are included only to help us when running conversationally. The fact that the system queries the user when it finds an unbound variable is one of these. The assumption here is that such an unbound variable is either a misspelling of the desired variable or is intended to evaluate to a function that we have not yet established as a constant. In the first case we can temporarily fix things by submitting the correct variable; i.e., this variable can be evaluated and its value (the one we really intended) is used to resume the interrupted evaluation. This does not really cure everything since the message and request for a value will appear every time that the offending variable is evaluated; but we can usually stop the request temporarily by using SETQ to establish a binding for the misspelled variable at the most recent PROG, which is probably the current level of supervision. After that, we will have to patch up the definition of the function so that it is written correctly (there are facilities in LISP that allow us to do this without redefining the entire function).

If it is the case that the unbound variable is supposed to have a constant binding to a function and we have not established the binding yet (whether by accident or design), then all we have to do is supply a suitable function; e.g., we can submit an expression like

```
(CSET 'UBV (LAMBDA (X) ··· ))
```

The value of this expression is exactly the function that we need in order to continue, but the expression also establishes a constant binding so that the value will not be requested again.

We must remember that such requests can only be handled when running conversationally. When running in the batch mode, the same messages will appear indicating unbound variables or the like, but the system has no recourse in this case except to terminate the current evaluation and try again on the next expression submitted for evaluation.

7.2.1 Another Example

Below is a listing from the teletype of an actual conversation with the LISP system. It is included to demonstrate the conversational mode.

```
@LISP

1108 LISP   V 6.2

EXPRESSION TO EVALUATE:
(CSET 'SQRT(LAMBDA(NUM)(NI 1.0)))

VALUE IS:

[SQRT]

EXPRESSION TO EVALUATE:
(SQRT 144)

NO VALUE IS BOUND TO NI

PLEASE SUPPLY ONE
(CSET 'NI(LAMBDA(TRY)(COND
((EQUAL(SQUARE TRY)NUM)TRY)
(T(NI(AVERAGE(QUOTIENT NUM TRY)TRY>

NO VALUE IS BOUND TO SQUARE

PLEASE SUPPLY ONE

(LISP)

EXPRESSION TO EVALUATE:
(CSET 'SQUARE(LAMBDA(X)(TIMES X X>

VALUE IS:

[SQUARE]

EXPRESSION TO EVALUATE:
(CSET 'AVERAGE(LAMBDA(X Y)(QUOTIENT(PLUS X Y)2.0>

VALUE IS:

[AVERAGE]

EXPRESSION TO EVALUATE:
(RETURN SQUARE)

VALUE IS:

1.2E1

EXPRESSION TO EVALUATE:
(SQRT 20)

VALUE IS:

4.47214

EXPRESSION TO EVALUATE:
(RETURN NIL)


END OF LISP
```

7.3 DEBUGGING AIDS

There are a few facilities provided by the LISP system that aid us in debugging our program.

7.3.1 TRACE,UNTRACE

The pseudo-function [TRACE] allows us to monitor the evaluation of expressions. [TRACE] takes one argument, which must be a list of atomic symbols that are constantly bound to functions. The effect of [TRACE] is to cause the arguments of any of these functions to be printed out whenever the function is applied and to cause the result to be printed when the function finishes computing it.

The pseudo-function[UNTRACE] takes one argument just as [TRACE] but removes the tracing from the listed functions. The result of both of these pseudo-functions is NIL.

Note, [TRACE] works equally well whether the function being traced is one that we have defined or one that the system provides for us; but remember, since a special-form is not a function, any attempt to trace it is absurd.

7.3.2 The Back-Trace

Whenever an error occurs that forces termination of an evaluation, the system will provide a printout of the contents of the push-down stack at that time. Most of these entries will seem to be gibberish; however, the entries that are preceded by three periods are probably of interest. These entries are either bindings of variables or expressions that are being evaluated.

Binding lists are kept internally in the same format as property-lists, i.e., variable consed with value. They will always contain occurrences of the character string "[]" when printed (this string is caused by a special gadget inserted to mark places in the environment where PROG's occur so that SETQ can establish bindings if it needs to). For instance when we see something like

```
    ··· ((N · 1)(L · Y)(A 1 2)(L · X)[])
```

in the back-trace, it means that N is bound to 1, L is bound to Y, A is bound to the list (1 2), and the binding of L to X is not currently active (it is shielded by the first L).

When we look at the most recent bindings, the variables that are bound can help identify the last function applied and the values to which they are bound can point out the arguments that this function received.

The expressions being evaluated will show the sequence of evaluations of expressions and their sub-expressions that led up to the error.

7.4 LISP CONTROL CARDS

The control cards of LISP may be used to affect the operation of the system when desired. They are identified as input images that have a colon as the first character followed immediately by a word specifying the action to be taken. They are always recognized and interpreted when encountered by the input routine.

:LIST

appearing anywhere in the input stream instructs the input routine to print cards as they are read. It is transparent to the user; i.e., a :LIST card can be placed anywhere in the input stream and the only noticeable effect will be that cards following the :LIST card are listed.

:UNLIST

instructs the input routine to stop printing cards as they are read. It is also transparent. The system will notice if it is being used from a teletype (conversational mode) and if so, will pretend that it had a :UNLIST card as its first image. Conversely, if the system notices that it is being run in the batch mode, then it automatically starts out listing cards.

:TIME

causes a printout of the time that has been spent evaluating expressions and collecting garbage. The printout format is

```
EVALUATION X.XXX, GARBAGE COLLECTION Y.YYY/N
```

It means that X.XXX seconds have been required to perform all evaluations so far, that Y.YYY of these seconds have been needed to collect garbage, and that N garbage collections have ocurred.

The :TIME card is not transparent; i.e., after this card is processed, the system will return to the latest level of supervision and request the next expression to evaluate.

:STOP

will cause the LISP system to die and return to EXEC just as if a RETURN had been performed on the top level of supervision.

:BACK

causes a back-trace to be printed followed by a return to the latest level of supervision.

:OOPS

causes the system to regress to the beginning of the read routine and attempt to read an object again. It is used when running conversationally and an uncorrectable error was made while submitting the current object.

:FROM

instructs the input routine to begin reading from an alternate file in SDF format that has been assigned to the run. An internal file name must begin in column 7, i.e., there is exactly one space between the :FROM and the file name.

The input routine will continue reading from this file until an end-of-file is reached or until another :FROM card appears. In the latter case it starts reading the second file and will return to the first file when it encounters the end of the second file. Nesting of alternate files is allowed up to a depth of five.

When switching to an alternate file, the current listing mode will stay in effect and will be reset when this file is left. That is, if we are currently listing cards and we begin reading an alternate file in which a :UNLIST card appears, we will begin listing cards again when we finish reading the alternate file.

If we are both reading from an alternate file and not listing cards as they are read, then the messages EXPRESSION TO EVALUATE: and VALUE IS: will not appear.

:EOF

simulates an end-of-file on an alternate file that is being read. If this card is placed in the normal input stream, it is ignored.

:LISP

causes the system to return to the latest level of supervision and request a new expression to evaluate. It also shuts down all alternate files that were being read and resets the listing mode.

:EXEC

is the linkage into EXEC-8 to interpret one of its control cards. An EXEC control card must begin in column 7 (i.e., column 7 contains the master space). For example, the following sequence might be used to read from a input file that has been built previously (say by a @DATA card) and stored on tape:

```
:EXEC @ASG,MT FILE,T,1234
:FROM FILE
:EXEC @FREE FILE.
```

7.5 ALARMS

The messages below will appear when evaluation cannot proceed. When we say that a value is requested from the user after the message is printed, we mean that this only happens in conversational mode. When in batch mode or for other messages, a back-trace is provided followed by a request for the next expression to evaluate unless the error is fatal.

NO VALUE IS BOUND TO X

means that an attempt was made to evaluate an atomic symbol for which no binding exists. It can also be caused by supplying too few arguments to a user-defined function (see below). A value is requested from the user but no binding will be established unless done explicitly.

X IS NOT A FUNCTION

means that the first element of a list neither indicates a special form nor does it evaluate to a function. A function is requested.

CANNOT TAKE CAR OR CDR OF X

means that a car-cdr chain ran across an atomic object. Evaluation is terminated.

WARNING, X CANNOT BE BOUND BECAUSE OF MISSING ARGUMENT

means that too few arguments were supplied to a user-defined function. This is only a warning; evaluation will continue but NO VALUE IS BOUND TO X will appear if the offending variable is ever evaluated.

WARNING, X IS AN ILLEGAL VARIABLE

means that a variable was used after a LAMBDA or PROG that has a constant binding. This also is only a warning. The constant binding of the atomic symbol will not be changed and will continue to be used if the atomic symbol is evaluated.

GO X ILLEGAL

means that the label of a GO does not appear in the most recent PROG. Evaluation terminates.

VALUE OF X IS NOT A FUNCTION

means that an operation (like [TRACE]) needs an atomic symbol that is bound to a function and it is not. A function is requested and bound constantly to the atomic symbol.

STACK OVERFLOW

means what it says and no back-trace is provided. This is probably caused by infinite recursion. If this happens during a garbage collection, the system will die with its last gasp being the message END OF LISP.

MEMORY IS EXHAUSTED

means that the garbage collector could not regain anything. Evaluation terminates.

GUARD MODE and ILLEGAL INSTRUCTION

are error messages that can be caused by real wierd mistakes. The most likely causes are supplying too few arguments to a system-defined function or supplying a bad argument to a system-defined function (like supplying an atom when a list is required).

After any error that causes evaluation to cease, the LISP system will check to make sure that it has not been clobbered. If it has, an appropriate message will appear and processing will halt. Control will return to EXEC.

Chapter 8
EXTENSIONS OF LISP 1.5

In the previous chapters we failed to mention a few facts about the evaluation of special-forms in 1108 LISP. They are not really that necessary and are usually not applicable to other versions of LISP, but they are worth mentioning here.

8.1 IMPLICIT DO

The first extension of LISP 1.5 is involved in the special-forms LAMBDA and COND. In both cases we are allowed to write more than one expression instead of only one as we implied before. When we do so, we mean that each expression is to be evaluated in order and the value of the last one is to be taken as the desired value; just as if we had surrounded the expressions with a DO.

This means that the format of a LAMBDA-expression is now

$$(\text{LAMBDA } (V_1\ V_2 \cdots V_n)\ E_1\ E_2 \cdots E_m)$$

The understanding is that whenever a function created by such an expression is applied, the E's are evaluated while the variables are bound to the arguments and the value of the last E is the result of the function.

Similarily, if we write a conditional expression like:

$$(\text{COND} \cdots (P\ E_1\ E_2 \cdots E_n) \cdots)$$

we mean that if P turns out to be true, then each E is to be evaluated and the last E is to be taken as the value of the conditional expression.

8.2 IMPLICIT AND

A further extension of the conditional expression is that if a P turns out to be true but its corresponding E is undefined (i.e., an unsatisfied conditional expression), then evaluation continues at the next P. For instance, in the expression

```
(COND (P₁ (COND (P₂ E₂)
                (P₃ E₃)))
      (T E₄))
```

If P_1 is true but P_2 and P_3 are both false, then the value of the entire expression is the value of E_4. Warning, if we ever define a function by a conditional expression that might not be satisfied and use the function as an E in a conditional expression, this same behavior could result and is probably not what we intend.

8.3 INDEFINITE ARGUMENTS

We often desire to be able to define a function that will take an indefinite number of arguments. (An example of this is the system function [PLUS].) To do this we use an atomic symbol other than NIL to terminate the list of variables appearing after the LAMBDA. When such a function is applied, this atomic symbol is bound to a list of the arguments remaining after all others are bound. For instance, a LAMBDA-expression like

```
     (LAMBDA (X Y Z · I) E)
```

ARGUMENTS <u>ARE</u> EVALUATE

means that the function expects at least three arguments. The first three will be bound to X, Y, and Z, while all others will be grouped into a list, which will be bound to I. If it is possible for the function to receive no arguments, then we write it as

```
     (LAMBDA L E)
```

where L will be bound to a list of all arguments supplied.

8.4 INITIALIZING PROG VARIABLES

It often happens that we do not wish a variable appearing after a PROG to be initially bound to NIL. When this is the case, we can (in lieu of doing a SETQ at the beginning to give it the desired binding) write a list of the form (PV E) instead of just the PROG variable. This means that the PROG variable PV is to be bound initially to the value of the expression E instead of NIL. The PROG variables are initialized from left to right. When E is evaluated, all PROG variables preceding (PV E) have their initial values and the variables following it do not.

For example, a function to count the number of elements in a list can be defined by:

```
     (CSET 'LENGTH (LAMBDA (L)
          (PROG ((N O))
          LOOP
          (COND ((NULL L)(RETURN N)))
          (SETQ N (PLUS N 1))
          (SETQ L (CDR L))
          (GO LOOP)
          )))
```

8.5 MACROS

Since LISP expressions are also LISP objects, it seems that it would sometimes be more convenient to compute an expression that will evaluate to the desired value than it would be to compute the value itself. 1108 LISP has a feature that allows us to do exactly this. The pseudo-function [MACRO] takes one argument, which must be a list of atomic symbols to which functions are constantly bound. [MACRO] will set things up so that each of these atomic symbols now represents a special-form with the following method of evaluation. When one of these atomic symbols appears as the first element of an expression, its associated function (the one to which it was bound before [MACRO] was used) will be applied to the expressions appearing after the special-form. Notice, the function receives the actual expressions as arguments, not the values of these expressions, as is normally the case. The result of this functional application will then be evaluated and the value will be used as the value of the original expression.

FOR DEFPROP

For example,we may be writing a program in which we need an analogue of the notation $\sum_{x=m}^{n} f(x)$. To do so we could define function:

```
     (CSET 'SIGMA (LAMBDA (LOWER UPPER FN)
            (COND
            ((GREATERP LOWER UPPER ) 0)
            (T (PLUS (FN LOWER)(SIGMA (PLUS LOWER 1) UPPER FN)))
            )))
```

Now we can get $\sum_{x=0}^{10} x^2$ by evaluating

```
     (SIGMA 0 10 (FUNCTION (LAMBDA (X)(TIMES X X))))
```

If we are really clever, we can use the macro facility to enable us to write:

```
     (SUM X (0 10)(TIMES X X))
```

We do this by first defining a function that will rewrite such an expression into the proper form:

```
     (CSET 'SUM (LAMBDA (VAR LIMITS EXP)
          (LIST 'SIGMA
                (CAR LIMITS)
                (CADR LIMITS)
                (LIST 'FUNCTION
                      (LIST 'LAMBDA
                            (LIST VAR)
                            EXP
                      ))
          )))
```

Then we do (MACRO '(SUM)) and we are ready to go.

Another provision of [MACRO] is the ability to define new special-forms. For instance, the special-form OR cannot be written as a function because all of its "arguments" are not necessarily evaluated. However, we could define it as a macro by:

```
(CSET 'OR (LAMBDA L
     (COND
     ((NULL L) 'F)
     ((EVAL (CAR L)) 'T)
     (T (CONS 'OR (CDR L)))
     )))
(MACRO '(OR))
```

Now when we write (OR P_1 P_2 ··· P_n), the function defined above will return either the expression F if there are no P's, T if P_1 is true, or (OR P_2 ··· P_n) if P_1 is false. The evaluation of this expression gives us exactly what we want.

Appendix A
FACILITIES INDEX

In this appendix we shall list all atomic symbols that are loaded with the 1108 LISP system and have constant bindings, i.e., are either bound to a system-defined function or represent a special-form. We shall also describe the meaning of the associated functions, pseudo-functions, and special-forms. Each entry is headed by a sample form, e.g., the entry for CAR appears under

    (CAR X)

This sample means that [CAR] is a function of one argument, which we shall indicate by the letter X while describing this function. That is, X represents the value of whatever expression is written in its place. This method for representing arguments applies to functions and pseudo-functions; it will be changed when we get to special-forms because the concept of argument is not valid there.

Entries are organized into categories according to their purposes. Under each category or entry some of the labeled paragraphs below may appear. If any appears under a category heading, it applies to every entry in the category.

VALUE:	indicates a description of the value of the sample form, i.e., the result of applying the function or the rule for evaluating a special-form.
EFFECT:	denotes that this entry is a pseudo-function and its effect on the system is described.
RESTRICTIONS:	precedes any restrictions on the arguments, i.e, if they must be numbers, lists, functions, etc.
NOTES:	gives any miscellaneous information.
EXAMPLE:	precedes clarifying examples.
DEFINITION:	precedes a definition of the function in LISP.
LOOPS:	appears if the function could go into an infinite loop or cause a stack overflow if given an argument that is a self-containing object (such objects

	can only be created by [RPLACA], [RPLACD], or some pseudo-function that uses them).
SPECIAL-FORM:	flags certain special-forms that appear along with the entries for functions to contrast them with similar functions.
SEE:	indicates that useful information also appears in previous sections.

BASIC SYMBOL MANIPULATION

(CONS X Y)

VALUE: the composite object (X · Y).

(CAR X)

RESTRICTIONS: X must be composite (created by [CONS]).

VALUE: Z if X is (Z · W)

(CDR X)

RESTRICTIONS: X must be composite.

VALUE: W if X is (Z · W).

(C···R X)

NOTES: ··· indicates any string of A's and D's whose length is between 0 and 35, inclusive.

VALUE: successive applications of [CAR] or [CDR] to X.

EXAMPLE: (CADDR X) is the same as (CAR(CDR(CDR X))).

RESTRICTIONS: X must be so composite that no application of [CAR] or [CDR] acts upon an atom.

(RPLACA X Y)

RESTRICTIONS: X must be composite.

EFFECT: if X is (Z · W) it is changed to (Y · W).

VALUE: X (changed).

SEE: Section 6.1.3.

(RPLACD X Y)

RESTRICTIONS: X must be composite.

EFFECT: if X is (Z · W) it is changed to (Z · Y).

VALUE: X

SEE: Section 6.1.3.

BASIC PREDICATES

(ATOM X)

VALUE: true if X is an atom, false if X is composite.

(EQ X Y)

VALUE: true if X and Y are the same atomic symbol, false if they are different atomic symbols, false if X and Y are different kinds of objects (e.g., X is an integer and Y is composite).

(EQUAL X Y)

VALUE: true if and only if X and Y are the same, whether they be atomic symbols, numbers, or composite. Integers are converted to reals if necessary, i.e., (EQUAL 10 10.0) is true as are (EQUAL 10 12Q) and (EQUAL 10 10.0000001).

NOTES: value not specified if X and Y are functions.

LOOPS:

(NULL X)

VALUE: true if and only if X is the empty list, which is represented by the atomic symbol NIL.

(NOT X)

VALUE: true if and only if X is false.

NOTES: same function as [NULL].

ARITHMETIC FUNCTIONS

RESTRICTIONS: all arguments below must be numbers.

SEE: Section 6.5.

(PLUS N_1 N_2 ··· N_n)

VALUE: $N_1 + N_2 + \cdots + N_n$

NOTES: will accept any number of arguments. (PLUS) evaluates to 0.

(TIMES N_1 N_2 ··· N_n)

VALUE: $N_1 \times N_2 \times \cdots \times N_n$

NOTES: (TIMES) evaluates to 1.

(DIFFERENCE X Y)

VALUE: X - Y

(QUOTIENT X Y)

VALUE: number theoretic quotient of X and Y if neither is real, otherwise real quotient.

(REMAINDER X Y)

VALUE: number theoretic remainder of division of X by Y if neither is real, otherwise floating-point residue of division.

(ADD1 X)

VALUE: X + 1

(SUB1 X)

VALUE: X - 1

(MINUS X)

VALUE: one's complement of X.

(ENTIER X)

VALUE: the largest integer less than or equal to X if X is real, otherwise X.

EXAMPLE: (ENTIER 10.5) = 10, (ENTIER -10.5) = -11.

(LOGOR X_1 X_2 ··· X_n)

VALUE: logical sum of X_1 through X_n regarded as 36 bit words.

NOTES: (LOGOR) = 0Q

(LOGAND X_1 X_2 ··· X_n)

VALUE: logical product of X_1 through X_n.

NOTES: (LOGAND) = -0Q (36 1-bits)

(LOGXOR X_1 X_2 ··· X_n)

VALUE: X_1 through X_n are half-added.

NOTES: (LOGXOR) = 0Q

(LEFTSHIFT X N)

VALUE: X shifted left N bits with 0-bits entering from the right if N is positive. If N is negative, value is X shifted right circularly N bits.

ARITHMETIC PREDICATES

RESTRICTIONS: all arguments must be numbers.

SEE: Section 6.5.

(ZEROP X)

VALUE: true if X is zero, false if not.

NOTES: comparison is exact; that is, real numbers "close" to zero are not considered zero by [ZEROP].

(EQUAL X Y)

NOTES: see BASIC PREDICATES above.

(NUMBERP X)

NOTES: X need not be a number.

VALUE; true if X is an integer, an octal, or a real; false otherwise.

(FIXP X)

VALUE: true if X is not real, false if it is.

(FLOATP X)

VALUE: true if and only if X is real.

(MINUSP X)

VALUE: true if and only if X is negative.

(GREATERP X Y)

VALUE: true if X is greater than Y, false if X is less than or equal to Y.

(LESSP X Y)

VALUE: true if X is less than Y, false if X is greater than or equal to Y.

NOTES: of (EQUAL X Y), (GREATERP X Y), and (LESSP X Y) only one will be true, even if X and Y are real numbers "close" to each other.

BINDING ESTABLISHMENT

(CSET A V)

RESTRICTIONS: A must be an atomic symbol.

EFFECT: A will receive a constant binding to V.

VALUE: V

(CSETQ A V)

SPECIAL-FORM:

NOTES: (CSETQ CON VAL) is equivalent to (CSET (QUOTE CON) VAL).

(DEFINE L)

RESTRICTIONS: L must be a list of the form

$((NAME_1\ E_1)\ (NAME_2\ E_2) \cdots (NAME_n\ E_n))$

EFFECT: each E is evaluated and bound constantly to its corresponding NAME. The NAME's are not evaluated.

VALUE: a list of the NAME's.

NOTES: $(DEFINE'((N_1\ E_1)\ (N_2\ E_2) \cdots (N_n\ E_n)))$
has the same effect as the sequence
$(CSETQ\ N_1\ E_1), (CSETQ\ N_2\ E_2), \cdots, (CSETQ\ N_n\ E_n)$

(SET A V)

RESTRICTIONS: A must be an atomic symbol.

EFFECT: if A has a constant binding, then its binding is changed to V; if A has a fluid binding, its current one is changed to V; otherwise, a binding of A to V is created at the level of the most recent PROG.

VALUE: V

(SETQ A V)

SPECIAL-FORM:

NOTES: (SETQ VAR VALUE) is equivalent to (SET (QUOTE VAR) VALUE).

LIST MANIPULATION

RESTRICTIONS: L must be a list or NIL.

(LIST X_1 X_2 ··· X_n)

VALUE: a list containing the elements X_1 through X_n in that order.

NOTES: is equivalent to (CONS X_1 (CONS X_2 ··· (CONS X_n NIL))). (LIST) evaluates to NIL.

(MEMBER X L)

VALUE: true if and only if X is [EQUAL] to one of the elements of L.

LOOPS:

EXAMPLE: (MEMBER '(A B) '(A B (A (A B)) C)) is false because (A B) is not an element of the list.

(APPEND L_1 L_2)

VALUE: a list consisting of the elements of L_1 followed by the elements of L_2.

EXAMPLE: (APPEND '(A B C) '(D E (F G) H)) evaluates to (A B C D E (F G) H).

LOOPS:

DEFINITION:
```
(CSETQ APPEND (LAMBDA (L1 L2)
        (COND
        ((NULL L1) L2)
        (T (CONS (CAR L1) (APPEND (CDR L1) L2)))
        )))
```

(NCONC L_1 L_2)

VALUE: same as (APPEND L_1 L_2).

EFFECT: L_1 is changed so that L_2 is its final segment.

LOOPS:

NOTES: It is very easy to create a circular list with [NCONC], e.g., by (NCONC X X).

DEFINITION:
```
(CSETQ NCONC (LAMBDA (L1 L2)
        (COND
        ((NULL L1) L2)
        (T (RPLACD L1 (NCONC (CDR L1) L2)))
        )))
```

(LENGTH L)

VALUE: the number of elements in L.

LOOPS:

NOTES: (LENGTH NIL) = 0

(REVERSE L)

VALUE: the list (X_n X_{n-1} ... X_1) if L is the list (X_1 X_2 ... X_n).

LOOPS:

DEFINITION: see example in 6.3.3.

SEQUENCING

RESTRICTIONS: L must be a list, or NIL. FN must be a function of one argument.

(MAP L FN)

EFFECT: FN is applied to every final segment of L.

VALUE: NIL

LOOPS:

DEFINITION:
```
(CSETQ MAP (LAMBDA (L FN)
        (PROG NIL
        LOOP
        (COND ((NULL L) (RETURN NIL)))
        (FN L)
        (SETQ L (CDR L))
        (GO LOOP)
        )))
```

(MAPCAR L FN)

EFFECT: FN is applied to every element of L.

VALUE: NIL

LOOPS:

DEFINITION: same as [MAP] except (FN L) is changed to (FN (CAR L)).

(ONTO L FN)

VALUE: a list whose elements are the results of applying FN to every final segment of L.

LOOPS:

DEFINITION:
```
(CSETQ ONTO (LAMBDA (L FN)
        (COND
        ((NULL L) NIL)
        (T (CONS (FN L) (ONTO (CDR L) FN)))
        )))
```

(MAPLIST L FN)

NOTES: equivalent to (ONTO L FN).

(INTO L FN)

VALUE: a list whose elements are the results of applying FN to each element of L.

LOOPS:

DEFINITION: same as [ONTO] except (FN L) is changed to (FN (CAR L)).

(INDEX L END FNN)

RESTRICTIONS: FNN must be a function of two arguments.

VALUE: if L is the list (X_1 X_2 ··· X_n), the result of [INDEX] is equivalent to the expression
(FNN 'X_1 (FNN 'X_2 ··· (FNN 'X_n END))).

LOOPS:

DEFINITION:
```
(CSETQ INDEX (LAMBDA (L END FNN)
              (COND
              ((NULL L) END)
              (T (FNN (CAR L) (INDEX (CDR L) END FNN)))
              )))
```

EXAMPLE: (APPEND X Y) is equivalent to (INDEX X Y CONS).
(INDEX L 0 PLUS) evaluates to the sum of the elements of L.

(ONDEX (L END FNN)

RESTRICTIONS: same as INDEX

VALUE: the same as the expression
(FNN L (FNN (CDR L) ··· (FNN (CD···DR L) END)))

LOOPS:

DEFINITION: same as [INDEX] except (FNN (CAR L) ··· is changed to (FNN L ···

PROPERTY LIST MANIPULATION

RESTRICTIONS: A and I must be atomic symbols.

SEE: Section 6.4.

(PUT A I V)

EFFECT: the property list of A is altered so that the attribute I becomes associated with the value V.

VALUE: A

(GET A I)

VALUE: the value associated with the attribute I on the property list of A. If the attribute I does not appear on the property list of A, the value is NIL.

(PROP A I FN)

RESTRICTIONS: FN must be a function of no arguments.

VALUE: the entry for I on the property list of A if such an entry exists (i.e., I consed with its associated value); if an entry for I does not exist, the value is FN applied to no arguments.

DEFINITION: assuming that I and FN are bound to the desired attribute and function and that PL gets bound to the property list of A, then this function will have the same result as [PROP].

```
(CSETQ PROPLOOK (LAMBDA (PL I FN)
        (COND    P
        ((NULL L) (FN))
        ((EQ (CAAR PL) I) (CAR PL))
        (T (PROPLOOK (CDR PL) I FN))
        )))
```

(REMPROP A I)

EFFECT: then entry for I is removed from the property list of A. If no such entry exists, nothing happens.

VALUE: A

(FLAG A I)

EFFECT: The flag I is added to the property list of A.

VALUE: A

(IFFLAG A I)

VALUE: True if and only if the flag I appears on the property list of A.

(UNFLAG A I)

EFFECT: The flag I is removed from the property list of A.

VALUE: A

EVALUATION

(EVAL E)

VALUE: the value in the expression E in the current environment; that is, using the current bindings of fluid variables.

LOOPS:

(FUNCTION FN)

RESTRICTIONS: FN must be a function.

VALUE: a new function which, when applied, will have its free variables bound to the values they have now (when we apply [FUNCTION]) instead of the bindings that they would normally have at the time of application.

(LISP IR)

RESTRICTIONS: IR must be a function of no arguments.

EFFECT: a new level of supervision is entered, using IR to fetch expressions to evaluate.

VALUE: whatever appears after the RETURN that causes the level of supervision to be left.

NOTES: IR is optional; if omitted, [READ] is assumed.

DEFINITION: see Section 7.1.

(MACRO L)

RESTRICTIONS: L must be a list of atomic symbols that are bound constantly to functions.

EFFECT: each atomic symbol in L is changed so that it represents a special-form. Whenever such a special-form appears, it is evaluated by applying the function originally bound to this atomic symbol to the expressions (not their values) that appear as "arguments" in the special-form. The result of this application is then evaluated and becomes the value of the special-form.

INPUT

(READ)

VALUE: the next object appearing in the input stream.

EFFECT: the input routine is advanced past this object.

NOTES: There is no such thing as a syntax error. If the input image does not conform to the rules for punching LISP objects, [READ] will make some sense (usually nonsense) out of it.

(READCH)

VALUE: an atomic symbol whose print-name is (i.e., would be printed as) the next character in the input stream.

EFFECT: the input routine advances over this character.

NOTES: the next character is taken unconditionally, even if it is a punctuation mark or an exclamation point. Due to the 1108 executive, there is no guarantee about how many trailing blanks will appear in an input image. LISP control cards are recognized and interpreted.

(TOKEN)

VALUE: the next token in the input stream. This will be either an atomic symbol (then [TOKEN] acts like [READ]) or a <u>non-blank</u> punctuation mark (then it acts like [READCH]).

EFFECT: the input routine is advanced past this token.

(CLEARBUFF)

EFFECT: the input routine is conditioned so that it will begin with the first character of the next input image.

VALUE: NIL

NOTES: if the input routine is already ready to begin on a new card, nothing happens.

(LOAD FILE)

RESTRICTIONS: FILE must be an atomic symbol whose print-name is an internal file name of a FASTRAND file that has been assigned to the run.

VALUE: the first object that has been placed in this file via [DUMP] or NIL if the end-of-file is read.

EFFECT: the constant values and property lists of atomic symbols are altered to reflect their condition when they were dumped.

NOTES: if FILE is omitted, the next object on the last file given to [LOAD] or [DUMP] is loaded.

OUTPUT

(PRIN1 X C)

RESTRICTIONS: C must be an integer between 0 and 128.

EFFECT: the object X is edited into its external representation beginning in column C of the print buffer.

VALUE: NIL

NOTES: C is optional; if omitted, editing begins in the next available column. If the print buffer is filled during editing, it is printed and editing continues in column C of the next buffer.

(PRINT X LIM)

RESTRICTIONS: LIM must be an integer.

EFFECT: X is edited beginning at the next available column and the final (partially filled) print buffer is printed.

VALUE: X

NOTES: LIM is the maximum number of lines that can be printed with the application of [PRINT]. If omitted, 100 is assumed.

(TERPRI)

EFFECT: the print buffer is printed if it contains anything.

VALUE: NIL

NOTES: The print buffer can only contain something if [PRIN1] has been used. If it is empty, nothing happens.

(DUMP X FILE)

RESTRICTIONS: same as LOAD above.

EFFECT: The object X is dumped starting at sector 0 of FILE. This dump includes constant values and property lists of atomic symbols.

VALUE: NIL

NOTES: If FILE is omitted, dumping starts at the next sector of the last file specified in a [DUMP] or [LOAD].

MISCELLANEOUS

(SUBST X Y Z)

VALUE: an object, obtained from Z by changing all occurrences of Y therein to X.

LOOPS:

DEFINITION: this definition is close to what really happens; however, the actual function [SUBST] in 1108 LISP will not create any new objects unless absolutely necessary (because a replacement for Y has been made).

```
(CSETQ SUBST (LAMBDA (X Y Z)
        (COND
        ((EQUAL Y Z) X)
        ((ATOM Z) Z)
        (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y (CDR Z))))
        )))
```

(GENSYM H)

VALUE: a brand new atomic symbol guaranteed to be unique; that is, (EQ (GENSYM) X) will always be false, even if X is an atomic symbol whose print-name happens to coincide with the print-name of the atomic symbol generated.

NOTES: When a generated atomic symbol is printed, it will appear as H:N, where N is some integer. If H is omitted, the atomic symbol G is assumed.

EXAMPLE: If [GENSYM] has never been used, the atomic symbol generated by evaluation of (GENSYM (GENSYM 'LABEL)) will be printed as LABEL:1:2.

(OBLIST FN)

RESTRICTIONS: FN must be a function of one argument.

EFFECT: FN is applied to every atomic symbol currently in existence with the exception of those generated by [GENSYM].

VALUE: NIL

NOTES: If FN is omitted, all the atomic symbols will be printed.

(AMB X Y)

VALUE: ambiguous, either X or Y.

NOTES: LISP's version of a random number generator is this random argument picker.

(IFTYPE X N)

RESTRICTION: N must be an integer between 0 and 7.

VALUE: true if and only if the object X is of type N, where N is:

0 means composite object (built by [CONS])

1 means integer

2 means octal

3 means real

4 means address out of bounds (like system defined functions)

5 means compiled code

6 means linkage node (usually, created by the special-form LAMBDA or a car-cdr chain)

7 means atomic symbol

NOTES: Types 4, 5 and 6 are unprintable objects and are the only ones that can be functions.

(ERASE L)

RESTRICTIONS: L must be a list of atomic symbols

EFFECT: Each atomic symbol in L has its property list emptied (set to NIL) and its constant binding (if it has one) removed.

VALUE: NIL

DEBUGGING

(TRACE L N S)

RESTRICTIONS: L must be a list of atomic symbols to which functions are constantly bound. N and S must be integers.

EFFECT: Each function bound to an element of L is trapped so that its arguments will be printed when it is applied and its result is printed after it is computed. Tracing begins as soon as the function has been applied S times and ends after it has been traced N times.

VALUE: NIL

NOTES: If S is omitted, 1 is assumed.
If N is zero, the functions will never stop being traced.
If both N and S are omitted, 0 and 1 are assumed.
Tracing uses [PRINT] with a five line limit.

(UNTRACE L)

RESTRICTIONS: same as L in TRACE above.

EFFECTS: Tracing is removed from the functions indicated in L.

VALUE: NIL

(NEWDEF N FN)

RESTRICTIONS: N must be an atomic symbol to which a function created by evaluation of a LAMBDA-expression is constantly bound. FN is a function of one argument.

EFFECT: The LAMBDA-expression that evaluates to [N] is reconstructed and FN is applied to it. This result is then evaluated and replaces the old binding of N.

VALUE: N

EXAMPLE: This pseudo-function is used to fix faulty definitions when running conversationally. For instance, suppose we have defined a function [TEST] that erroneously contains the expression (CONS X), where we really meant (CONS X L). Instead of redefining [TEST], we can patch it up by evaluating

```
(NEWDEF 'TEST
        (LAMBDA (I) (SUBST '(CONS X L) '(CONS X) I)))
```

<u>COMPILER</u>

(*BEGIN)

(*EMIT I A)

(*ORG A)

(*EPT N)

(*MACRO X)

(*CHAIN X)

(*DEF X)

NOTES: These functions are necessary for the LISP compiler and are of no concern to us. They are only listed here since these atomic symbols do have constant bindings and therefore cannot be used as variables.

(*CAR X)

VALUE: same as (CAR X).

NOTES: There are some wise guys (like the LISP compiler) who want to take CAR's and CDR's of atoms. [*CAR] allows them to transgress in this manner without terminating evaluation.

(*CDR X)

NOTES: Same idea as *CAR.

TRUTH FUNCTIONAL CONSTANTS

NIL

Evaluates to NIL.

T

Evaluates to T.

F

Evaluates to NIL.

SPECIAL-FORMS

A letter used in the sample for a special-form represents whatever object is actually written in its place and not the value of that object, since the concept of that object having a value does not necessarily apply.

(QUOTE X)

VALUE: X

NOTES: [READ] translates 'X into (QUOTE X).

(COND (P_1 E_{11} E_{12} ··· E_{1n}) (P_2 E_{21} ··· E_{2m}) ··· (P_i E_{i1} ··· E_{ij}))

VALUE: Each P is evaluated from left to right. When one is found that is true, each associated E is evaluated. The value of the last such E becomes the value of the conditional expression. If the last E is a conditional expression for which no P is true, then evaluation continues with the next P in the first conditional expression.

(LAMBDA (V_1 V_2 ··· V_n) E_1 E_2 ··· E_m)

RESTRICTIONS: Each V must be an atomic symbol that will never have a constant binding.

VALUE: a function of n arguments whose result is computed by evaluating E_1 through E_m in an environment where each V is bound (fluidly) to the corresponding argument of the function. The value obtained from the last E becomes the result when the function is applied.

(DO E_1 E_2 ··· E_n)

VALUE: the value of the last E, but all E's are evaluated.

EFFECT: whatever effect the E's have.

(AND P_1 P_2 ··· P_n)

VALUE: true if no P evaluates to false; if one does, the value is false and no more P's are evaluated.

(OR P_1 P_2 ... P_n)

VALUE: false if no P evaluates to true; if one does, the value is true and no more P's are evaluated.

(LAMDA(V_1 V_2 ... V_n) E_1 E_2 ... E_m)

VALUE: the same function as the value of (FUNCTION (LAMBDA (V_1 V_2 ... V_n) E_1 E_2 ... E_m))

PROGRAM FEATURE

(PROG (V_1 V_2 ··· V_n) S_1 S_2 ··· S_n)

RESTRICTIONS: Each V must be either of the form PV or (PV E), where PV is an atomic symbol without a constant binding.

EFFECT: For each V of the form PV, PV is bound to NIL; for each V of the form (PV E), PV is bound to the value of E. Then each non-atomic S is evaluated in this expanded environment.

VALUE: NIL, but see RETURN below.

NOTES: The S's that are atomic symbols are labels that may be referenced by GO.

(SETQ PV V)

NOTES: see BINDING ESTABLISHMENT above.

(GO LAB)

RESTRICTIONS: LAB must be a label appearing in the PROG that surrounds this GO.

EFFECT: Evaluation of the statements in the surrounding PROG continues with the statement following the first appearance of the label LAB.

(RETURN V)

EFFECT: V is evaluated and its value becomes the value of the most recent PROG-expression.

Appendix B

RUNNING LISP UNDER EXEC-VIII

Since LISP resides on the 1108 system library, to use the LISP system we submit a run with a standard UWCC 1108 @RUN statement followed by a @LISP control statement followed by the expressions that we wish to evaluate. LISP is not a processor in the 1108 executive sense; that is, there are no automatic facilities in LISP for producing or updating files of source input. To do so we use the DATA processor of EXEC VIII and :FROM control card of LISP.

The format of the @LISP statement is

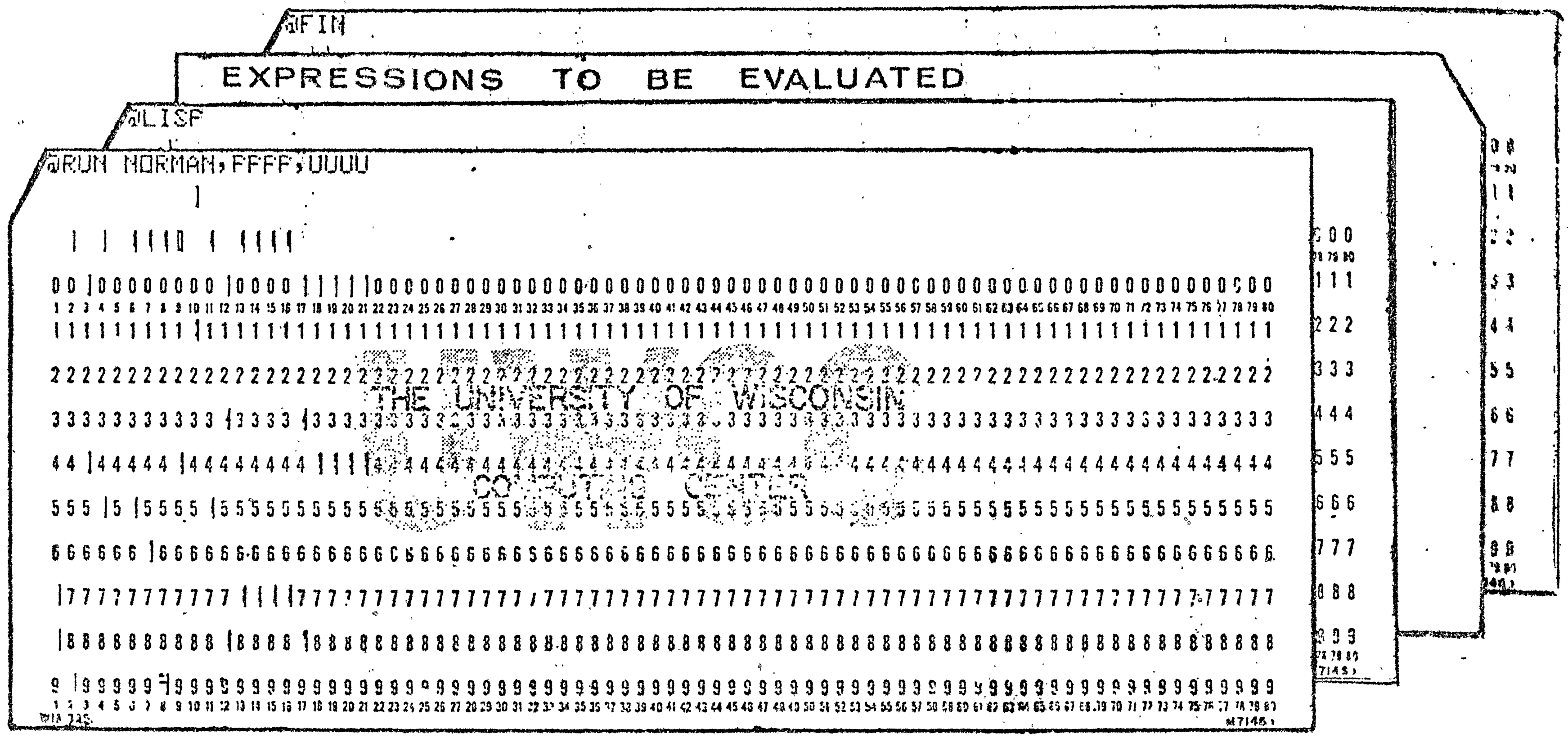
```
@LISP,options
```

where the options can be

B	means that LISP is to be run in batch mode.
C	means that LISP is to be run in conversational mode. Note: neither of these options is necessary, because LISP determines the mode automatically when it is loaded.
Z	inhibits checking to see if [CAR] or [CDR] is being applied to an atom.

If the LISP system cannot recover from a GUARD MODE or ILLEGAL INSTRUCTION error, the 1108 message indicating that such an error occurred will probably appear. The messages are IGDM or IOPR, respectively, followed by a register dump. These errors are fatal.

The deck setup is illustrated on the following page.


@FIN
EXPRESSIONS TO BE EVALUATED
@LISP
@RUN NORMAN,FFFF,UUUU
THE UNIVERSITY OF WISCONSIN
COMPUTING CENTER