LEARN LISP
TUTORIAL GUIDE

SUPPORTING THE UO-LISP PROGRAMMING ENVIRONMENT

IMPORTANT NOTE
IMPORTANT NOTE
IMPORTANT NOTE
IMPORTANT NOTE

The tutorial guide included in this package was written for the CP/M based UO-LISP V2. You will find that it refers to a manual you did not receive. It will also refer to functions not included in your system.

The CP/M based version of the tutorial guide is now included with the TRSDOS based version of UO-LISP V1. **WHY INCLUDE IT?** We have found that a high percentage of those purchasing UO-LISP V1 have little or no LISP experience. You will find that about 90% of the tutorial guide is useful in learning LISP. We hope to retrofit the tutorial guide to TRSDOS some time in the future.

CONTENTS

WHAT IS LISP?

Lisp is the language of Artificial Intelligence (AI). The research community has written intelligent Lisp programs for understanding English (and other human languages), programs for solving complex mathematical problems both symbolically and numerically, and programs for controlling robots. Because of its wide use in constructing "intelligent systems", Lisp is becoming an increasingly popular language. In the next few years, knowledge of Lisp will become essential to those wishing to be a part of this exciting field and to those wanting to understand and profit by the uses of the systems implemented in it.

Although Lisp is just now becoming well known, it is actually one of the older programming languages; much older than BASIC, for example. It was invented by John McCarthy in the late 1950's. While most languages of that time (and even now) were oriented towards numerical computation, Lisp was designed to manipulate symbols and structures of symbols. It is a powerful tool for solving problems that cannot be easily handled by number-oriented languages.

A great many important problems are symbolic in nature. For example, how do you develop a plan to run some errands? Probably, you develop the plan by thinking in English, not in numbers. The key point is that humans think symbolically, so to construct programs that think as intelligently as humans, it's best to use languages that permit symbolic computation. That is why Lisp is the main language used by AI researchers to develop intelligent software and even to understand the way that humans think.

A short list of many of the problems being solved by the Lisp programming community includes many (some whimsical) that are not even contemplated by BASIC, C, FORTRAN, and PASCAL programmers:

- Understanding natural language
- Understanding legal documents and interpreting court decisions
- Diagnosing disease and recommending treatment
- Manipulating algebraic equations
- Writing other programs
- Writing fiction
- Developing plans of action based on incomplete and possibly erroneous information (robots)

In the following sections we will slowly introduce the basic concepts of Lisp that will allow you to become an effective programmer and to begin to write your own AI software.

CHAPTER 1
THE LISP INTERPRETER

The first thing you need to know about Lisp is that it is an interpreted language that evaluates S-expressions. To understand what that means you need to understand:

. What an interpreted language is
. What an S-expression is
. What evaluation is

To begin our Lisp tutorial, we will examine each of these topics.

1.1 What is an interpreted language?

An interpreted language is one that you can rapidly interact with. You issue commands to the language (system), and receive immediate responses. Most compiled languages are not at all interactive. For example, if you are a FORTRAN or PASCAL programmer, you are used to first creating your program with an editor, compiling it, then loading and running it, and finally receiving your output. You are at least a couple of steps removed from the language; it is not interactive.

Because it is interpreted, Lisp allows you to construct and debug your programs rapidly and effectively. BASIC is a familiar example of such a language. Lisp is more powerful than BASIC because, in addition to the arithmetic operations provided by BASIC, it also has many primitive symbol and strorage management operations that BASIC does not.

1.2 A UO-LISP Session

To start the **UO-LISP** interpreter you must first place the Lisp Learner floppy disk into the computer and select the disk drive. For example in a two drive system, the standard CP/M system disk is placed in drive A and the Lisp Learner Disk is placed in drive B.

Note In the rest of this book, computer output is in bold type and your input is in normal type. Comments (you need not enter these) are prefixed with a percent sign and run to the end of the line.

Once you have set up the disks as above, the following sequence will initialize the **UO-LISP** interpreter:

HOW TO USE THIS BOOK

This tutorial explains how to program in the language Lisp. The micro-computer owner and user will find operating instructions for the **UO-LISP** system, useful Lisp programming tips and techniques, and a basic knowledge of what the AI revolution is all about. However, this is not a text book. There is no required homework, no complicated busy work, no deadlines. You must only be willing to rethink some of your notions of what computers are used for and how they are programmed.

Lisp is a unique programming language, probably very different from any other language you have ever used. A Lisp learner -- even one who has had some experience with other languages -- will need some help to learn how to use Lisp effectively, and how to exploit the power of the language. This is the purpose of the present tutorial.

No programming language can be learned by just reading about it. To really understand any language, you have to use it. The tutorial contains many examples of interactions with **UO-LISP**, and we encourage you to follow these actions by getting into **UO-LISP**, typing in what you see in the tutorial, and observing how **UO-LISP** responds. More generally we encourage you to try out your ideas about Lisp programming by using **UO-LISP**. If you find yourself thinking "I wonder if...", don't just think about it! Try it out and find the answer.

The tutorial is not a manual. It does not describe all the functions in the **UO-LISP** language; it describes the concepts of Lisp. As we discuss these concepts, however, we will note the sections of your **UO-LISP** Learner's Manual which document the associated Lisp functions. You should consult the manual as you use the tutorial.

```
A> B:
B> UOLISP
```

Note, you don't give it a program to run and leave it alone to execute. The idea is that by calling it up, you are going to engage in a dialogue with Lisp. If you wish, as part of this dialogue, you can tell Lisp to load a file; but you don't have to begin by doing this.

UO-LISP indicates it's ready to start the conversation by displaying its logo and prompt:

```
          U   U   OO     L      III  SSSS  PPP
          U   U  O  O    L       I   S     P  P
          U   U  O  O -- L       I    SS   PPP
          U   U  O  O    L       I      S  P
           UU     OO     LLLL   III  SSSS  P

     Copyright 1984 by Northwest Computer Algorithms.

1:
```

Now you should imagine that Lisp is listening to you, waiting for you to begin the dialogue. To talk to Lisp, or any intelligent being, for that matter, you need to say things it will understand. Lisp understands things called S-expressions, so that's what you have to type. For example, you might say to Lisp:

```
1: (PLUS 3 5)
```

For now, its not important to understand the details of this S-expression. The important point to remember is that once you've typed such an expression, then a carriage return (hereafter "<CR>"), Lisp will try to interpret the S-expression as a command to execute. Put another way, Lisp will treat the S-expression as a program, or piece of Lisp code, and will try to evaluate the S-expression. When Lisp has evaluated the S-expression, it will return and PRINT the value it has found. For the current S-expression, **UO-LISP** performs as follows:

```
+--------------------------------------------------------------------+
|                                                                    |
|     1: (PLUS 3 5)                                                  |
|     8                                                              |
|                                                                    |
|     2:                                                             |
|                                                                    |
+--------------------------------------------------------------------+
```

UO-LISP evaluated the S-expression you typed in as a request to add two numbers, 3 and 5. The value of the piece of Lisp code "(PLUS 3 5)" is 8, which **UO-LISP** prints on the next line. Finally, **UO-LISP** indicates it has finished responding and that it is now your turn in the dialogue by giving you a the prompt "2:". Note that **UO-LISP** increments the prompt number so that you (and it) can keep track of where you are in the conversation.

While talking with **UO-LISP**, you may want to add a comment in English. It is especially important to comment your code so that others can understand it. The percent sign signals the presence of a comment. The **UO-LISP** reader ignores all subsequent characters to the end of the line. For example:

```
+--------------------------------------------------------------------+
|                                                                    |
|     1: (PLUS 1 2)        %I'm adding two numbers                   |
|     3                                                              |
|                                                                    |
+--------------------------------------------------------------------+
```

When you are finished with **UO-LISP**, enter (QUIT), and you will then find yourself talking to the operating system:

```
+--------------------------------------------------------------------+
|                                                                    |
|     2: (QUIT)                                                      |
|     B>                                                             |
|                                                                    |
+--------------------------------------------------------------------+
```

Not really. UOLISP mucks up so many pointers that it's MUCH SAFER to hit RESET.

This, then, is the basic strategy of Lisp:

1. You type an expression that the **UO-LISP** interpreter reads

2. The **UO-LISP** interpreter computes the value of the expression

3. The **UO-LISP** printing function displays that value

4. Go back to step 1

We need to know what S-expressions the **UO-LISP** reading function will accept. Once this syntax is understood we can move on to the evaluation strategy and how values are printed. The following chapter presents the syntax of S-expressions but before you proceed, you should make sure that you can start and stop **UO-LISP** as described above.

CHAPTER 2
S-EXPRESSIONS: THE SYNTAX OF LISP

As with any programming language, to learn Lisp, you need to learn what commands it understands, and which it will not. You need to understand the syntax of the language. Learning the syntax of most languages is a major chore. In BASIC, for example, you have many different kinds of statements -- declarations, assignments, conditionals, branching statements, and so on -- and each type has its own peculiar syntax.

But the syntax of Lisp is much simpler; even trivial. All the commands of the language, all the code that you will write, will be S-expressions, and S-expressions obey a few simple rules. One advantage of Lisp, is that you can spend much more time at the more important task of learning to use it to solve complicated problems.

There are only two rules for forming S-expressions, both very simple:

. Atoms are S-expressions

. Lists are S-expressions

But what are atoms and lists? We will discuss each of these in turn.

2.1 Atoms are S-expressions

Atoms also obey simple rules. There are three basic types:

. Any sequence of alphanumeric characters beginning with an alphabetic character is an atom, an identifier

. Any sequence of numeric characters (possibly preceeded by a "+" or "-" sign) is an atom, a number

. Any sequence of characters enclosed in quotation marks (") is an atom, a string

The first type of atom, the identifier is used as a place holder for function definitions, as a variable name, a place to put properties of things, or as just the characters of its name. Here are a few examples of identifiers:

```
     A                [A single alphabetic character is a sequence]
     ABC              [This starts with an alphabetic character]
     A12              [So does this]
     abcde5           [Lower case is ok too]
     ABCDE5           [The same as abcde5 since case is immaterial]
     aAbBcC           [And you can mix cases if you want]
     aadasddkl4567kaweooOKJASDIKMXKLALDSLLLLadsx14
                      [There is practically no limit tk the length
                       of atoms]
```

Now are a few sequences that are not identifiers:

```
     1ATOM            [If it starts with a number it can't have any
                       alphabetic characters]
     AT-OM            [Generally, you can't use characters that
                       are not numeric or alphabetic in atoms]
     AT OM            [Spaces, or <CR>'s separate atoms, so this
                       is a sequence of two atoms]
```

Numeric atoms are strings of digits optionally prefixed with a + or - sign. They act just like integer numbers in languages like BASIC or FORTRAN. Here are a few numeric atoms:

```
      1234            [All digits in a numeric atom]
     +1234            [A positive integer]
     -1234            [A negative integer]
```

In the Lisp Learner version of **UO-LISP**, integers are restricted to the range -4096 to +4095. This is not as restrictive as it sounds as in Lisp, unlike other languages, numbers are relatively unimportant. More advanced versions of **UO-LISP** permit integers with many thousands of digits and arbitrary precision fixed and floating point numbers.

The final type of of atom is the string, a sequence of characters enclosed in double quotes. To get a quote mark in the string, simply put two in next to each other. Strings are used mostly for information displays and error messages. The following are valid strings:

```
       "Hi there"        [Simple string]
       ""                [Empty string]
       """Hi there"""    [String with quote marks in it]
       "A two line       [Strings can extend across lines]
        string"
```

For more information about identifiers, numbers, and strings, consult chapter 1 of the **UO-LISP** Learner's Manual.

2.2 Lists are S-expressions

Besides atoms, the only other kind of S-expression in Lisp is the <u>list</u>. Again, the rules for forming lists are straightforward:

- An empty pair of parentheses "()" is a list
- Any sequence of atoms (atoms separated by one or more spaces or <CR>s) that is enclosed in parentheses is a list
- Any list of one or more lists is a list

Basically, lists are bounded by parentheses, and have elements inside the parentheses. The three rules above describe different things that can be list elements.

The first rule defines what we call the <u>empty list</u>, a list of no elements. It can appear in a variety of ways:

```
()                [Two parentheses with nothing between
                   them is an empty list]
(  )              [Or you can put in any number of spaces;
                   they are just delimiters in Lisp]
NIL               [Sometimes the empty list is written
                   as the identifier NIL.  This may be a
                   bit confusing, but we'll explain why
                   later.]
```

The second rule defines simple lists. Here are a few simple lists:

```
(A)               [A list with just one atom]
(A b)             [A list can have two or more atoms, too]
(1 B c)           [You can mix numbers and identifiers
                   in lists]
( 1   2 3  )      [You can separate list elements by any
                    number of spaces]
(1 2
   3)             [Or even by <CR>s]
```

And a few non-lists (hence, non S-expressions):

```
(a b c            [A closing parenthesis please!]
(H E L p))        [Parentheses must balance]
```

The most complex of the three rules for lists is the last one. It is called the <u>recursive clause</u> for lists, because it defines what a list can be in terms of itself. (We will be studying recursive Lisp structures in later sections). While the second rule for lists says that atoms can be list elements, the third rule says that lists can also be elements of lists. We refer to these as <u>complex lists</u>. For example:

```
(The rain (in spain) falls (mainly on the plain))
             [A list with 5 elements, the third and
              fifth of which are also lists]
((The ((rain (in)) spain)))
             [Lists can be nested arbitrarily deeply; as
              many parentheses as you like, providing
              they are balanced]
(())         [Since lists can be elements of other lists,
              the empty list "()" can be an element of
              a list]
(NIL)        [This is the same as the last one, since
              () and NIL mean the same thing.  Note,
              (NIL) is not equivalent to NIL.]
```

Since complex lists are so important in Lisp, it is worth taking a moment to really understand their structure. Let's look at:

```
(HERE (IS (A LIST) (OF (IDENTIFIERS))) (FOR YOU))
```

First, check to see that it really is a list. Does each opening parenthesis "(" have a matching closing parenthesis ")"? Find each pair of parentheses that match. How many elements are in this list? The first element is an atom, HERE, that's one. The second element is itself a list, (IS (A LIST) (OF (IDENTIFIERS))), that's two. The third element is a simple list, (FOR YOU). And that's it, so there are three elements in this list, one atom, and two lists (which also have their own elements).

One last example before we continue. How many elements does the list () have? It's the empty list, so it has none. How many elements does this have: (())? The inside parentheses, (), is the empty list, and because it's a list, it can be an element of another list, so (()) is a list with one element, the empty list. Now you can see why () and (()) are not the same: The first is a list with no elements, and the second is a list with one element.

2.3 S-expressions are Both Program and Data in Lisp

Now that you understand what S-expressions are in Lisp, you need to know what they are used for. The answer is: everything! S-expressions are the way you write programs in Lisp, and they are also the data that Lisp programs manipulate. So you now know all about the syntax and datatypes of the Lisp language.

You may find it a bit difficult to get used to the idea that S-expressions in Lisp are both the programs and data of the language. In the languages you might know, like BASIC or FORTRAN, there is a clear distinction between the two. Programs are made up of statements like "GO SUB 400" or "IF (I.GT.1001) I=1", and the data, things represented by variables, are numbers or arrays. In the following sections we will carefully discuss how S-expressions are both program and data. We will come to see that many of the things that make Lisp the simplest of languages -- the fact that the syntax of the language is so easy, the fact that there is no distinction between program and data -- also make it a most powerful language.

CHAPTER 3
THE LISP EVALUATOR: EVAL

If S-expressions are both the programs of Lisp and the data manipulated by the language, how does the Lisp interpreter tell if it should treat something as a piece of program or as a piece of data? The answer is, basically, that what you type to the interpreter gets treated as code. Recall our discussion of the READ-EVAL-PRINT loop in Section 1. When we typed the S-expression:

```
        (PLUS 3 5)
```

to Lisp, it took that S-expression as a piece of code to evaluate, then returned the answer. In this section we will discuss exactly how Lisp evaluates the S-expressions you give it, so that you can learn to construct pieces of Lisp code that do what you want.

3.1 Evaluating Identifiers

How Lisp evaluates an S-expression depends on the kind of expression it is -- atom or list -- so lets begin with identifiers. The values of lisp identifiers are much like the values of variables in other languages. Usually, identifiers don't have values to begin with. For example, if you just called up the **UO-LISP** interpreter and typed X it would give you an error message:

```
1: X
***** (not global X)

2:
```

indicating that the identifier X does not have a value, when Lisp tried to evaluate it. Just as in other languages, you have to give atoms values. Don't worry about how this is done just yet, we'll discuss it in Section 5. For now all you need to know is that the value of an atom can be any S-expression. For example, X could be assigned the value 5, or (A LIST) or (A (COMPLEX (LIST))).

3.2 Evaluating Numbers and Strings

Numeric atoms and strings are exceptions to the rule that atoms initially don't have values. While identifiers are initially "unbound", numbers and strings have themselves as their values. If you ask Lisp to evaluate a number, it will just print that number, the same for a string. This is demonstrated in the following **UO-LISP** session:

```
+----------------------------------------------------------------+
|                                                                |
|     1: 9                                                       |
|     9                                                          |
|                                                                |
|     2: -246                                                    |
|     -246                                                       |
|                                                                |
|     3:"On the other hand"                                      |
|     "ON THE OTHER HAND"                                        |
|                                                                |
+----------------------------------------------------------------+
```

3.3 Evaluating Simple Lists

The evaluation of lists is quite different than atoms. Lists, unlike atoms cannot be assigned a value. To understand evaluation of lists, let's begin with a simple example. We will follow the **UO-LISP** interpreter as it evaluates (PLUS 3 5) to produce the value 8.

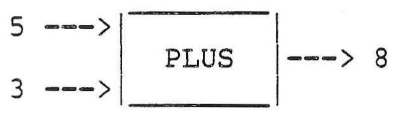
When the interpreter sees a list, it always assumes the first element of the list is the name of a function. A function in Lisp is roughly the same as a procedure, or subroutine, or program, in other languages. Functions are the basic units of computation in Lisp. For example, "ABS" is the name of a built-in absolute value function in FORTRAN, BASIC, and **UO-LISP**.

Names of functions are identifiers, so for this reason, the first element of an S-expression you type to the interpreter should be an identifier. In the case above, **UO-LISP** interprets "PLUS" to be the name of a function, and goes to find the function's definition. Since PLUS is a built-in function, the interpreter has no trouble finding the definition. (In Section 5 we will show you how to define your own functions.) If Lisp can't find a function definition, it will give you an error message. For example:

```
+----------------------------------------------------------------+
|                                                                |
|     1: (FOO 1 2)                                               |
|     ***** (FOO undefined function)                             |
|                                                                |
+----------------------------------------------------------------+
```

When Lisp has found the function definition, using the first element of the S-expression, it interprets the remaining elements as arguments to the function. Then it applies the function definition to those arguments. For example, since PLUS is a function for adding numbers together, and 3 and 5 are numbers, when you apply PLUS to these numbers you get 8. We say this is the value returned from the function PLUS, and it is the value that Lisp will print, if you type "(PLUS 3 5)" to it.

This example illustrates that functions can be thought of as little machines. They take some input (their arguments), manipulate them (apply the definition of the function to the arguments), and produce a result as output (return a value). Try picturing function calls like this:



All calls to Lisp functions follow this simple pattern.

The previous demonstrates one added detail of evaluation. Lisp does not directly apply a function definition to the remaining elements in the list it is evaluating. Rather, it first evaluates the arguments, then applies the function. This was obscured in the previous example, since the value of 5 is 5 and the valua of 3 is 3. In the next example, assume that the value of the atom VAR is 7 and the value of VAR2 is 8. Then we would see

```
+---------------------------------------------------------------------+
|                                                                     |
|    1: (PLUS VAR1 VAR2)                                              |
|    15                                                               |
|                                                                     |
+---------------------------------------------------------------------+
```

In other words, the Lisp interpreter first gets the function definition of "PLUS", then evaluates each of the remaining elements of the expression, obtaining 7 and 8, then applies PLUS's definition to these two numbers, returning and printing 15.

3.4 T and NIL

T and NIL are very special Lisp atoms whose evaluation and role require some discussion. Like numbers, T and NIL are bound to themselves, and these values cannot be changed:

```
+----------------------------------------------------------------------+
|                                                                      |
|     1: T                                                             |
|     T                                                                |
|                                                                      |
|     2: NIL                                                           |
|     NIL                                                              |
|                                                                      |
+----------------------------------------------------------------------+
```

The reason for this is that T and NIL have unique roles in Lisp. In Lisp T means roughly "true" or "yes". Thus if I ask Lisp whether 10 is an atom (by calling the built-in function ATOM), Lisp tells me it is by returning T:

```
+----------------------------------------------------------------------+
|                                                                      |
|     1: (ATOM 10)                                                     |
|     T                                                                |
|                                                                      |
+----------------------------------------------------------------------+
```

One use of NIL is to denote the opposite of T; NIL means "false" or "no". Thus if I ask Lisp whether 10 is greater than 12 (by calling the built-in function "GREATERP"), Lisp tells me it is not by returning NIL:

```
+----------------------------------------------------------------------+
|                                                                      |
|     1: (GREATERP 10 12)                                              |
|     NIL                                                              |
|                                                                      |
+----------------------------------------------------------------------+
```

NIL has a second role in addition to its logical one. It not only means "false", it denotes the empty list or list with no elements, as we discussed in Section 2. NIL's dual role gives it a unique status in Lisp: It is the only S-expression that is both an atom and a list. While this may be a bit confusing at first, it will help to remember that NIL plays only one role at a time. It's an atom when used logically to denote "false", and it's a list when used to represent the empty list.

3.5 Summary of the Rules of Lisp Evaluation

The previous examples show all the basic rules that govern Lisp's evaluation of S-expressions. They are about as simple as the rules governing the structure of the language themselves! Before continuing, let's summarize the rules for evaluation:

```
TO EVALUATE AN S-EXPRESSION, S:
  IF S IS AN ATOM, RETURN ITS VALUE
  IF S IS A LIST THEN
     GET THE DEFINITION, D, OF THE FIRST ELEMENT OF S,
       AND
     EVALUATE EACH OF THE REMAINING ELEMENTS OF S, AND
     APPLY THE DEFINITION OF D TO THE EVALUATED
       ARGUMENTS, AND
     RETURN THE VALUE OF THE FUNCTION APPLICATION
```

The interesting thing about these rules, and what can make them difficult to follow, is their recursive nature. As we said in Section 2, a definition is recursive if the thing being defined is defined (partly) in terms of itself. In this case, we have defined evaluation in terms of itself, because to evaluate a list we have said that you must evaluate the arguments to the list (that is, all elements of the list except the first). To give you a better understanding of the potentially tricky recursive definition of evaluation, we will examine the evaluation of some complex lists.

3.6 Evaluating Complex Lists

How will Lisp evaluate:

```
(PLUS 3 (ADD1 5))
```

It's really not hard, if we just carefully follow the rules set out in the previous section. First, the interpreter assumes "PLUS" is the name of a function, and gets its definition. Next, it evaluates the arguments to PLUS, 3 and (ADD1 5); since 3 is a number, its value is itself, 3. Evaluating (ADD1 5) is a bit more complex because it is a list. We have to get this value before we can apply PLUS to its arguments.

To evaluate (ADD1 5) we have to follow the rules of evaluation all over again, for this S-expression. To evaluate this list, the interpreter will first get the definition of the function ADD1 (since it is the first element of the list). ADD1 is a function that adds 1 to a number. Now Lisp will interpret the rest of the elements in the list (ADD1 5) as arguments to ADD1. That is, 5 is the argument to ADD1. The value of 5 is 5, so now to complete the evaluation of the expression (ADD1 5), the definition of ADD1 is applied to 5, and the value 6 is returned.

Now we are finished evaluating (ADD1 5) and have the values we need for both arguments to PLUS. They are 3 and 6. So finally we can apply PLUS to 3 and 6, getting 9. This is returned and printed by the top level control program.

As you can see from following this example, part of the difficulty in understanding the recursive computation of the Lisp interpreter is that in order to complete evaluating "outside" S-expressions, you have to begin and complete evaluating "inside" expressions, because the interpreter uses the values of the inside expressions to find the value of the outside expression. In this case, for example, (ADD1 5) is inside (PLUS 3 (ADD1 5)) and its evaluation begins after, but completes before, the evaluation of (PLUS 3 (ADD1 5)). One final note, notice that while every S-expression returns a value when evaluated, only the value of the very most outside expression is printed by Lisp. This is called a "top-level" expression.

3.7 Picturing Lisp Evaluation

There is a simple graphical notation that helps you get a better picture of how Lisp's evaluation works. Let's use it to picture the last example:

```
--> (PLUS 3 (ADD1 5))
|     --> 3
|      3 <-- 3
|     --> (ADD1 5)
|     |     --> 5
|     |     5 <-- 5
|     6 <-- (ADD1 5)
9 <-- (PLUS 3 (ADD1 5))
```

Here

"--> [form]" - means Lisp is about to evaluate [form]

"[value] <-- [form]" - means Lisp just finished evaluating [form] and is returning [value] as the result

The indentations and vertical bars also help you see which evaluations are inside which others, and therefore which ones are returning values to be used in other outside evaluations. A notation much like this one is available inside **UO-LISP** to let you trace the evaluation of function calls (see Chapter 2.17 of the **UO-LISP** Learner's Manual under the function TR). You will find it particularly useful in understanding what happens in evaluating recursive functions.

3.8 EVAL Evaluates S-expressions

One fascinating and really very important aspect of Lisp is that the interpreter that performs the Lisp evaluation we have just discussed is not a program written in another language, like machine language, but is in fact just another Lisp function, called EVAL. EVAL is a function that takes one argument, the S-expression to be evaluated, and returns the result of evaluating it. The availability of the Lisp interpreter as a function, EVAL, distinguishes Lisp from almost every other language, and can be exploited to powerful effect. In Section 11, we introduce a Lisp program that uses explicit calls to EVAL.

Reading about how evaluation in Lisp works is fine, but you will acquire a much deeper understanding by getting into **UO-LISP** and letting Lisp evaluate some expressions. Why don't you start up an interactive **UO-LISP** session now?

CHAPTER 4
LIST MANIPULATION FUNCTIONS

Now that you have a good understanding of how Lisp S-expressions are interpreted as programs or function calls, you might well ask: How do you prevent Lisp from treating all S-expressions as code? How can Lisp treat expressions as data too? We answer these questions in this section and discuss the basic, most primitive, Lisp functions that allow you to construct, access and change list data structures.

4.1 Quoting a List

Up until now, every time you typed a list to Lisp, it evaluated it. To get Lisp to treat a list as data you have to have some way of saying: "Don't evaluate this list, I want to be able to manipulate it as a data object". The way to do this is to QUOTE the object. You quote an object by just putting a single quote mark before it. For example:

```
1: '(a b c)
```

Now, instead of evaluating the list, as a call to the function "a" with arguments "b" and "c", Lisp leaves it alone:

```
1: '(a b c)
(a b c)
```

That is, Lisp just returns whatever object you quoted.

4.1.1 Anything can be Quoted

Quoting can be applied to any S-expression to prevent it from being evaluated. For example:

```
1: 'A                         %Quoting literal atoms is ok
A

2: '5                         %You can quote numbers,
5                             %although you don't need to

3: '(a (b (c d (e)) f) (g h)) %You can quote a complex
(a (b (c d (e)) f) (g h))     %list

4: 'NIL                       %You may quote NIL and T
NIL                           %although you don't have to

5: 'T
T
```

Initially, quoting may seem to be a very strange operation. But, in fact it is quite familiar. English uses a similar convention. For example, when I say "John is a good boy", I might be correct, because the appearance of John refers to a particular boy. However, if I say " "John" is a good boy", I'm not making sense, because the appearance of "John" refers to the word, not the referent of the word. I could say ""John" is a four-letter word". Plus, in English, the use of "" protects words against evaluation, just as ' does in Lisp.

4.1.2 The Use of QUOTE to Manipulate Lists

Quoting by itself is not a powerful operation. What makes it powerful is that by protecting lists from being evaluated, we can manipulate them -- construct them, access pieces of them and alter them. Because lists can now be freely manipulated they become full-fledged data objects, just like numbers. In the following sections we will discuss in detail how lists are manipulated.

4.2 CONS Constructs Lists

UO-LISP has many built-in functions that create and manipulate list data objects, just as the built-in function PLUS is used to manipulate numbers. The most basic of these functions is CONS. CONS is used to build lists. It takes two arguments; the first can be any S-expression, and the second is generally a list. CONS makes and returns a new list that adds its first argument to the front of the list that is its second argument. Here are some examples of CONS constructing simple lists:

```
+-----------------------------------------------------------------------+
|                                                                       |
|    1: (CONS 'A '(B))          %CONS puts A on the front of           |
|    (A B)                      %the list (B)                           |
|                                                                       |
|    2: (CONS 'A '())           %the atom A cets added to the           |
|    (A)                        %front of the empty list                |
|                                                                       |
|    3: (CONS 'A NIL)           %Remember NIL is also a name            |
|    (A)                        %for the empty list                     |
|                                                                       |
|    4: (CONS 'A '(NIL))        %Also remember that NIL is not          |
|    (A NIL)                    %same as (NIL)                          |
|                                                                       |
|    5: (CONS '(A) '(B C))      %A list can be CONSed onto a            |
|    ((A) B C)                  %list, not just atoms                   |
|                                                                       |
|    6: (CONS 5 '(6 7))         %You can make lists of numeric          |
|    (5 6 7)                    %atoms as well                          |
|                                                                       |
|    7: (CONS 'A (CONS 'B (CONS 'C NIL)))                               |
|    (A B C)                                                            |
|                                                                       |
+-----------------------------------------------------------------------+
```

Only the last example should require any explanation. Remember, in our discussion of Lisp evaluation, we said that to complete the evaluation of an outside form we had to first evaluate the inside forms that is its arguments. So, to evaluate (CONS 'A (CONS 'B (CONS 'C NIL))), we have to first evaluate (CONS 'B (CONS 'C NIL)), and to evaluate that we need to first do (CONS 'C NIL). So lets begin there.

The value returned from (CONS 'C NIL) is (C), thus (CONS 'B (CONS 'C NIL)) will put B on the front of the list (C), and will return (B C). This is what (CONS 'A (CONS 'B (CONS 'C NIL))) will put 'A onto, thus, finally, it will return (A B C). Here's a diagram of the evaluation:

```
--> (CONS 'A (CONS 'B (CONS 'C NIL)))
|     --> 'A
|     A <-- 'A
|     --> (CONS 'B (CONS 'C NIL))
|     |     --> 'B
|     |     B <-- 'B
|     |     --> (CONS 'C NIL)
|     |     |     --> 'C
|     |     |     C <-- 'C
|     |     |     --> NIL
|     |     |     NIL <-- NIL
|     |     (C) <-- (CONS 'C NIL)
|     (B C) <-- (CONS 'B (CONS 'C NIL))
(A B C) <-- (CONS 'A (CONS 'B (CONS 'C NIL)))
```

UO-LISP provides many different functions for building lists. Another basic and particularly useful function is called LIST. It takes any number of arguments and returns a list made up of them:

```
+----------------------------------------------------------------------+
|                                                                      |
|    1: (LIST 'This 'is 'a 'LIST)                                      |
|    (THIS IS A LIST)                                                  |
|                                                                      |
|    2: (LIST 'This 'is 'a (LIST 'COMPLEX 'list))                      |
|    (THIS IS A (COMPLEX LIST))                                        |
|                                                                      |
+----------------------------------------------------------------------+
```

Note that you can get the same results using CONS:

```
+----------------------------------------------------------------------+
|                                                                      |
|    1: (CONS 'This (CONS 'is (CONS 'a (CONS 'LIST NIL))))             |
|    (THIS IS A LIST)                                                  |
|                                                                      |
|    2: (CONS 'This                                                    |
|             (CONS 'is                                                |
|                   (CONS 'a                                           |
|                         (CONS (CONS 'COMPLEX                         |
|                                     (CONS 'list NIL))                |
|                               NIL))))                                |
|    (THIS IS A (COMPLEX LIST))                                        |
|                                                                      |
+----------------------------------------------------------------------+
```

You can see from these examples that the LIST function often makes it a lot easier to construct the list you want. Each of the different list manipulation functions has its own special purpose, like LIST. Even though you would prefer to use LIST to build the lists mentioned above, it's a good exercise to test your understanding of of Lisp evaluation by tracing through the calls to CONS in the last two examples. Give it a try now. Just remamber -- do the inside evaluations before the outside ones. And if you have any problems, draw a diagram!

4.3 Accessing Parts of Lists

Once you've built a list, in order to do anything useful with it, you'll need to be able to isolate the elaments of it. **UO-LISP** has a variety of functions to do this. Just as CONS is the basic Lisp function for building lists, CAR and CDR are the primitive functions for accessing parts of them. CAR takes a single argument, should be a list, and returns the first element of the list. CDR is the complement of CAR: when given a list, it returns all but the first element. Some examples will make this clear:

```
1: (CAR '(4 SCORE AND 7 YEARS))
4

2: (CDR '(4 SCORE AND 7 YEARS))
(SCORE AND 7 YEARS)

3: (CAR '((A B) (C D)))
(A B)

4: (CDR '((A B) (C D)))  %CDR doesn't return the second
((C D))                  % element, but a list of all
                         %the elements but the first
5: (CAR 'A)              %The argument to CAR and CDR
***** (A is not a pair for CAR)    %must be a list

6: (CAR NIL)             %It is an error to try to take
***** (NIL is not a pair for CAR)  %the first element
                         %of an empty list
7: (CDR  NIL)            %And the empty list doesn't
***** (NIL is not a pair for CDR)  % have a remainder
                         %either
```

Multiple applications of CAR and CDR can be used to get at any element of a list, no matter how deeply embedded it is in other lists. For example:

```
+------------------------------------------------------------------+
|                                                                  |
|    1: (CAR (CDR '(A B C)))                                       |
|    B                                                             |
|                                                                  |
|    2: (CAR (CDR (CDR '(A B C))))                                 |
|    C                                                             |
|                                                                  |
|    3: (CDR (CDR (CDR '(A B C))))                                 |
|    NIL                                                           |
|                                                                  |
|    4: (CAR (CDR (CAR '((A B) C D))))                             |
|    B                                                             |
|                                                                  |
+------------------------------------------------------------------+
```

Notice the close relation between CONS, on the one hand, and CAR and CDR on the other. CAR will get back what was the first argument to CONS; CDR will get back the second:

```
+------------------------------------------------------------------+
|                                                                  |
|    1: (CAR (CONS 'A '(B C)))                                     |
|    A                                                             |
|                                                                  |
|    2: (CDR (CONS 'A '(B C)))                                     |
|    (B C)                                                         |
|                                                                  |
+------------------------------------------------------------------+
```

4.4 Testing Properties of Lists

In addition to constructing and accessing parts of lists, another very basic kind of operation is to test lists for various properties. One of the most basic tests is to determine if a list is empty. You can do this using the function NULL. NULL returns T if the list it is given as an argument is empty, otherwise, it returns NIL.

```
+------------------------------------------------------------------+
|                                                                  |
|    1: (NULL '(A))                                                |
|    NIL                                                           |
|                                                                  |
|    2: (NULL NIL)                                                 |
|    T                                                             |
|                                                                  |
|    3: (NULL 'A)     %A is an atom, so can't be the               |
|    NIL              %empty list                                  |
|                                                                  |
+------------------------------------------------------------------+
```

The function NOT is a synonym for NULL and can be used interchangeably:

```
+----------------------------------------------------------------------+
|                                                                      |
|    4: (NOT NIL)                                                      |
|    T                                                                 |
|                                                                      |
|    5: (NOT '(B C D (E F)))                                           |
|    NIL                                                               |
|                                                                      |
+----------------------------------------------------------------------+
```

4.4.1 Testing Lists and Atoms for Equality

Perhaps the most basic test on lists is to determine if two lists, or atoms for that matter, are the same. The primitive Lisp function provided for testing equality is called EQ. For now, you should think of EQ as testing the equality of atoms only. It will return T if the the two arguments you give it are the same atom; otherwise, it returns NIL. Here are some examples of EQ in action:

```
+----------------------------------------------------------------------+
|                                                                      |
|    1: (EQ 'A 'A)                                                     |
|    T                  %These literal atoms are the same              |
|                       %because they have the same name               |
|    2: (EQ 'A 'B)                                                     |
|    NIL                                                               |
|                                                                      |
|    3:  (EQ  54  54)   %The small  integers  that  are                |
|    T                  %numeric  atoms in UOLISP are also             |
|                       %EQ if they have the same name                 |
|    4: (EQ 54 55)                                                     |
|    NIL                                                               |
|                                                                      |
|    5: (EQ NIL NIL)                                                   |
|    T                                                                 |
|                                                                      |
+----------------------------------------------------------------------+
```

The reason to use EQ with only atoms is that it doesn't work as you might expect with lists. For example:

```
+----------------------------------------------------------------------+
|                                                                      |
|    6: (EQ '(A B C) '(A B C))                                         |
|    NIL                                                               |
|                                                                      |
+----------------------------------------------------------------------+
```

The right function to use with lists is called EQUAL. It works as you'd expect:

```
7: (EQUAL '(A B C) '(A B C))
T

8: (EQUAL 'A 'A)
T                    %Note, EQUAL works for atoms too.
                     %In general any two things that
9: (EQUAL  54 54)    %are EQ are EQUAL, but not the
T                    %converse
```

4.4.2 Equality of S-expressions

It may seem odd to you that (EQ '(A B) '(A B)) is not true. This is because the notion of sameness is actually ambiguous. Suppose you have two identical twins. Are they the same? On one view the answer is clearly "No"; after all, they are different people. Those arguing this way interpret "same" as referring to the identity of the object. But another view says they are the same; after all, you can't tell them apart. Those arguing this way are referring to the appearance of the object.

EQ and EQUAL correspond to these different interpretations of "same". EQ refers to the stricter "identity" sense of "same", while EQUAL refers to the looser "appearance" sense of "same". Thus you can clearly see why to (EQUAL '(A B) '(A B)) is true; both lists have the same appearance in print. And the reason that these two lists are not EQ is that, like the twins, while they look the same, they are really two different lists.

This may seem surprising, so let's take some time to understand why it is so. When you ask Lisp to build a list by:

```
1: (LIST 'A 'B)
(A B)
                    %or
2: '(A B)
(A B)
```

Lisp actually takes this as an instruction to create a Lisp entity, a Lisp data object. If you repeat the instruction, you are essentially asking Lisp to create a new object. The new object is a different structure than the old one, even though they look alike, much like two houses from the same blueprint are different structures, even though indistinguishable. Thus, if you say:

```
+------------------------------------------------------------------+
|                                                                  |
|    3: (EQ '(A B) '(A B))                                         |
|    NIL                                                           |
|                                                                  |
+------------------------------------------------------------------+
```

Lisp first evaluates the arguments to EQ, creating two new list objects, then tests them for EQness. Since they are not the identical list, EQ returns NIL.

Maybe you now understand why two same-appearing lists are not EQ, but you might ask: Why are two atoms EQ? Doesn't Lisp create two DIFFERENT versions of A when I say (EQ 'A 'A)? In fact Lisp does not. It only creates new list objects; it never creates two atoms with the same name. And that is why you can use EQ with atoms but should use EQUAL with lists.

4.5 Primitive List Operations

The primitive operations CONS, CAR, CDR, and EQ, provide the basis for constructing most of the complex operations of Lisp Many of these compound operations are so convenient and frequently used, that UO-LISP provides built-in functions that effect them. Chapter 2.3 in the UO-LISP Learner's Manual describes the primitive list manipulation functions, while Chapter 2.14 discusses the built-in composite functions. In Section 5 of this tutorial we describe how you can create your own functions to effect just the compound list operations you wish.

List operations like these are at the heart of symbolic computation in Lisp. You'll need lots of practice to get proficient. The next sections will provide some of the necessary practice.

4.6 Asking for Help

UO-LISP provides much assistance for the beginning user. There are three built-in functions that interface to a data base containing an English description of each built-in function, error message, and editor command. The APROPOS function assists the user in determining the spelling of a function name by providing a list of built-in functions that have a specified sequence of letters in them. The HELP function provides an English description of any built-in function, and ERROR!? a description of an error.

Suppose I would like to substitute the identifier A for every occurrence of the identifier B in a list and do not know if there is a built-in function that does this. If there is such a function it probably has some of the letters of 'substitute' in it. Two distinguishing letters of 'substitute are S and B as normal function name construction uses the first part of the corresponding English word less vowels. The APROPOS function will find the names of all functions that have an S and B in them in that order.

```
1: (APROPOS 'SB)
(SETIOBYTE SUBST SUBLIS SUB1)
```

Of the four functions, only the last three look like they have something to do with list substitution. The HELP function can now be used to discover what these functions do. The first two go something like this:

```
2: (HELP SUBST)

     (SUBST U:any V:any W:any):any [EXPR] {2.14}

         SUBST returns the result of substituting U for
all occurrences of V in W. EQUAL is used for equality
tests.

NIL

3: (HELP SUBLIS)

      (SUBLIS X:alist Y:any):any [EXPR] {2.14}

         SUBLIS returns the result of substituting the
CDR of each element of the alist X for every
occurrence of the CAR part of that element in Y.

NIL
```

Evidently SUBST is the function I want. The textual explanation gives the arguments of the function, the type of the function (see the introduction to Chapter 2 of the UO-LISP Learner's Manual), and the chapter and section number of the function in the UO-LISP Learner's Manual.

UO-LISP provides assistance when a recoverable system error is signaled. Error messages are always prefixed by 5 asterisks (*****) and are sometimes fairly cryptic. The ERROR!? function will provide an English explanation of why the error was signaled and will sometimes suggest alternative courses of action. For example, in the following sequence I tried to open a file, but spelled INPUT wrong.

```
1: (OPEN "XYZ" 'INPT)
***** Cannot OPEN

2: (ERROR!?)

                  ***** Cannot OPEN

       A  file cannot be  opened  for  a  variety  of
reasons. These include: poorly formed file name, disk
not mounted, operating system error,  file  does  not
exist, or the second argument to OPEN is not INPUT or
OUTPUT.

NIL
```

These functions provide a comprehensive interactive manual with immediate access to a large amount of information. Until he has considerable practice with the system the beginning user will find this' access very helpful .

CHAPTER 5
NAMING AND DEFINITION

As you construct progressively larger programs in Lisp you may find yourself repeatedly typing the same expressions. For example, you may be continually retyping a long list like "(TOYOTA NISSAN MAZDA SUBARU HONDA MITSUBISHI)". This is extremely awkward and time-consuming. What you'd like is a way of naming that particular expression, then using the name in your code, not the expression itself. Similarly, you might repeatedly accessing the fifth element of a list, using "(CAR (CDR (CDR (CDR (CDR '(A B C D E F G))))))". Here again, you'd like to associate a name with that particular operation (getting an element of a list at a particular index). Lisp provides powerful ways for naming such things.

5.1 Naming Data Structures

In the first case above, you would like to remember a particular data object. To do this in Lisp, you can set the value of an atom to that data object. This is done using the SET function:

```
1: (GLOBAL '(JAPANESECARS))
NIL

2: (SET 'JAPANESECARS '(TOYOTA NISSAN MAZDA SUBARU
                        HONDA MITSUBISHI))
(TOYOTA NISSAN MAZDA SUBARU HONDA MITSUBISHI).
```

For now, don't worry about the call to the built-in function GLOBAL in the first line; we'll return it that shortly. SET actually does two things. First it will make the value of the atom JAPANESECARS be (TOYOTA NISSAN MAZDA SUBARU HONDA MITSUBISHI), and, second, it returns the value. The value returned, of course, is not the important thing; it happens because in Lisp, every function call returns a value. The important result of SET is its side effect of setting an atom's value.

Now, when we type "JAPANESECARS", it will be evaluated like any atom (see Chapter 3), and its value returned:

```
+-----------------------------------------------------------------+
|                                                                 |
|     3: JAPANESECARS                                             |
|     (TOYOTA NISSAN MAZDA SUBARU HONDA MITSUBISHI)               |
|                                                                 |
|     4: (LIST 'JAPANESECARS JAPANESECARS)                        |
|     (JAPANESECARS (TOYOTA NISSAN MAZDA SUBARU HONDA             |
|     MITSUBISHI))          %remember quoted things aren't        |
|                            %evaluated                           |
|                                                                 |
+-----------------------------------------------------------------+
```

If you want to set the value of an atom using SET, you always have to quote its first argument (i.e., the atom). This can be awkward, so there is a special function, called SETQ, which is just like SET, except it quotes its first argument for you:

```
+-----------------------------------------------------------------+
|                                                                 |
|     1: (GLOBAL '(USCARS GMCARS))                                |
|     NIL                                                         |
|                                                                 |
|     2: (SETQ USCARS '(FORD GM CRYSLER AMERICAN))                |
|     (FORD GM CRYSLER AMERICAN)                                  |
|                                                                 |
|     3: (SETQ GMCARS '(PONTIAC BUICK CHEVROLET CADILLAC))        |
|     (PONTIAC BUICK CHEVROLET CADILLAC)                          |
|                                                                 |
+-----------------------------------------------------------------+
```

Generally, you will find it more convenient to use SETQ than SET.

SETQ will allow you to set any S-expression as the value of an atom:

```
1: (GLOBAL '(ATOMLIST NUMLIST COMLIST REST))
NIL

2: (SETQ ATOMLIST '(A LIST OF LITERAL ATOMS))
(A LIST OF LITERAL ATOMS)

3: (SETQ REST (CDR ATOMLIST))
(LIST OF LITERAL ATOMS)

4: REST
(LIST OF LITERAL ATOMS)

5: (SETQ NUMLIST '(1 2 3 4 5 6 7))
(1 2 3 4 5 6 7)

6: (CAR (CDR NUMLIST))
2

7: (SETQ COMLIST '((THE) (RAIN (IN SPAIN)) FALLS))
((THE) (RAIN (IN SPAIN)) FALLS))

8: (CAR (CDR (CAR (CDR COMLIST))))
(IN SPAIN)
```

The use of SETQ will be familiar to all programmers. It is what Lisp uses to assign values to variables, much as "=" is used in FORTRAN and BASIC, and ":=" in PASCAL.

5.2 Naming Procedures or Functions

In addition to providing names for data objects, it is also useful to provide names for pieces of code or procedures.

Note: There is no difference between a procedure and a function in Lisp and we will use the terms interchangeably.

For example, it is not only briefer, but much clearer to say (CUBE NUMBER)" than to say "(TIMES NUMBER (TIMES NUMBER NUMBER))", when you want to raise a number to the third power. All Lisps have ways to associate a name like "CUBE" with a procedure. This is done by defining your own function, called CUBE. In **UO-LISP** you define new functions by using a built-in function, called DE. The form of DE is as follows:

```
(DE <function name> <parameter list> <body>)
```

where:

<function name> is an atom naming the new function

<parameter list> is a list of atoms that will be bound to the arguments given to the function

<body> is a sequence of S-expressions defining new function

It is best to explain DE through examples. Lets begin by defining our function CUBE which is paraphrased in English as follows:

```
To  cube something multiply it by times itself itself
 |   |    |         |        |       |       |       |
(DE CUBE (NUMBER)  (TIMES NUMBER (TIMES  NUMBER NUMBER)))
```

Now you will be able to use CUBE just as you do the built-in functions of LISP:

```
+-------------------------------------------------------------------+
|                                                                   |
|    1: (DE CUBE (NUMBER)                                           |
|          (TIMES NUMBER (TIMES NUMBER NUMBER)))                     |
|    TIMES                                                          |
|                                                                   |
|    2: (CUBE 5)                                                    |
|    125                                                            |
|                                                                   |
|    3: (CUBE 8)                                                    |
|    512                                                            |
|                                                                   |
|    4: (CUBE 'A)          %CUBE only works for numbers             |
|    ***** Non-numeric argument                                     |
|                                                                   |
|    5: (PLUS 4 (CUBE 4)) %you can combine calls to your own        |
|    68                    %functions and built-in ones             |
|                                                                   |
+-------------------------------------------------------------------+
```

The first thing to notice about the function DE is that, unlike the others we have discussed, you do not have to quote the arguments to DE, because DE doesn't evaluate its arguments. Why not? Because if it did, to define a function called CUBE, you'd have to say:

```
(DE 'CUBE '(NUMBER) ...)
```

But, since the user would always be quoting the arguments to DE, it was designed to do the quoting automatically, sparing the user unnecessary effort. Functions that automatically quote their arguments are called Special Forms (also called FEXPRs in the UO-LISP Learner's Manual). You may have noticed that SETQ is also a special form. There are not very many special forms in Lisp, but they are among the most important so we will be meeting a few more.

Lets take a little closer look at the mechanics of function definition using DE. To define a new function, you have to give it a name (here "CUBE"), and a meaning, in terms of code that will get executed when the function is called (here "(TIMES NUMBER (TIMES NUMBER NUMBER))"). But you also need a way of referring to whatever argument is given to the function, and that's what the parameter list is used for. For example, the parameter list of CUBE is "(NUMBER)". Since the list has only one element, we say CUBE is a function of one variable. Thus when someone calls the function CUBE, with a specific argument, say 5 (i.e., he types "(CUBE 5)", the variable "NUMBER" becomes bound to the value "5", and will have that value when it is referred to in the body of the function definition. In general, "NUMBER" will be bound to whatever argument CUBE is called with.

Note, when you define a function you must provide one formal parameter in the parameter Lisp for each argument you expect the function to be called with. For example, if you want a function to average two numbers to be called like "(AVERAGE 10 6)", then you will have to supply two formal parameters in the parameter list you create:

```
1: (DE AVERAGE (X Y)
       (QUOTIENT (PLUS X Y) 2))
AVERAGE

2: (AVERAGE 10 6)
8

3: (AVERAGE 11 6)
8              %AVERAGE does integer arithmetic.
```

5.2.1 Formal Parameters and Global Variables

The formal parameters of a function are often called local variables. They are variables, because, just like any other atoms that have values, they are "assigned" values when the function is entered, and can be used to access those values. They are local variables, however, because unlike other variables they can only be referenced inside the function for which they are defined. An example will make this clearer:

```
+----------------------------------------------------------------------+
|                                                                      |
|    1: (GLOBAL '(VARIABLE1))                                          |
|    NIL                                                               |
|                                                                      |
|    2: (SETQ VARIABLE1 '(FOO BAR))                                    |
|    (FOO BAR)                                                         |
|                                                                      |
|    3: (DE FUNNYFUNCTION (VARIABLE2)                                  |
|         (LIST 'BANG VARIABLE2 VARIABLE1))                            |
|    FUNNYFUNCTION                                                     |
|                                                                      |
|    4: (FUNNYFUNCTION '(A B)) %VARIABLE2 and VARIABLE both            |
|                              %have values when referenced            |
|    (BANG (A B) (FOO BAR))    %here                                   |
|                                                                      |
|    5: VARIABLE2              %but VARIABLE2 has no value             |
|    ***** (not global VARIABLE2) %outside the scope of                |
|                              %FUNNYFUNCTION.                         |
|    6: VARIABLE1              %however, VARIABLE1 has it              |
|    (FOO BAR)                 %value anywhere                         |
|                                                                      |
+----------------------------------------------------------------------+
```

In contrast with a local variable, like VARIABLE2, whose scope is just the function for which it is defined, variables like VARIABLE1, are defined everywhere; hence they are global in scope, and are often called global variables. Why does Lisp make this distinction between local and global variables? We discuss this in the next section.

5.2.2 Modularity and Function Definitions

There is a good reason formal parameters are local to their functions and that is, the names chosen for the parameters do not affect the meaning of the functions. For example I could have defined CUBE as:

```
+----------------------------------------------------------------------+
|                                                                      |
|    1: (DE CUBE (ANYNAMEIWANT)                                        |
|        (TIMES ANYNAMEIWANT                                           |
|               (TIMES ANYNAMEIWANT ANYNAMEIWANT)))                    |
|    CUBE                                                              |
|                                                                      |
+----------------------------------------------------------------------+
```

Clearly someone using the function CUBE shouldn't have to know the name of the variable that I used as its formal parameter, in order to use the function properly. More generally, he shouldn't have to know anything about the function that doesn't have to do with its meaning. But if formal parameters were global variables he would have to know the names I had used, or else his own functions might not work properly. An example will show why this is so. Suppose that Ralph is writing a Lisp program, and has borrowed some of my functions, including CUBE. Now Ralph writes a new function that uses CUBE:

```
10: (DE SUMCUBE (ANYNAMEIWANT Y)
      (PLUS (CUBE Y) ANYNAMEIWANT))
SUMCUBE
```

Now, assuming ANYNAMEIWANT and the other formal parameters of these functions were global in scope, what would happen when Ralph asked Lisp to evaluate "(SUMCUBE 19 5)"? When SUMCUBE was entered, Y would be set to 5 and ANYNAMEIWANT to 19. Then, the arguments to PLUS would be evaluated. To evaluate the first argument "(CUBE Y)", the function CUBE is entered, and its parametar, ANYNAMEIWANT, is set to 5. CUBE eventually returns 125, and now the second argument to PLUS, the atom "ANYNAMEIWANT" is evaluated. But it no longer has the value it had when SUMCUBE was entered! The variable ANYNAMEIWANT has had its value changed to 5 by CUBE. Thus, the call to SUMCUBE would return 130, not 144.

Needless to say Ralph, would be very confused. His SUMCUBE function is correct, and I promised him that my CUBE function is ok (which it is). The problem is that Ralph unfortunately used a variable name which clashed with one that I used. Thus, if parameter names were global, to write his functikns correctly, Ralph would not only have to know what my functions did, but what names I used. Because the names chosen for parameters don't have anything to do with the meaning of functions, we don't want to require Ralph to know such things. Thus Lisp makes all formal parameters local to their function, not global, and the clash we described above never arises.

The localization of formal parameters is one feature of Lisp that enables the functions you write to be modular "black-boxes". Good functions are black boxes because you should not need to know any of the details about how they are implemented or what goes on inside them when they are operating. You should write your functions so that all any user needs to know to use tham is what the name of the function is, the arguments it needs (their meaning, not their names), what its body does (not how it does it), and what the function returns.

5.2.3 On the Use of Global Variables

Now that we understand why formal parameters are made to be local variables let's discuss the role of global variables a little more carefully. First of all, how do you tell if you are dealing with a global variable? Simple. If you can access the value of the variable anywhere, its global. In particular, only global variables can be referenced at the "top-level" in Lisp:

```
8:  FOOBAR
(A LIST OF 5 THINGS)

9:  BARFOO
***** (not global BARFOO)
```

Here, FOOBAR is global, but BARFOO is not. In **UO-LISP** you can also find out whether a variable is global or not by using the function GLOBALP; if it returns T its argument is global; if it returns NIL, the variable is not global.

```
10: (GLOBALP 'FOOBAR)
T

11: (GLOBALP 'BARFOO)
NIL
```

Second, how do you create a global variable? This is also simple. Just declare it to be global, using the GLOBAL function (see Chapter 2.7 of your UO-LISP Learner's Manual), then you can SETQ it to anything you like. Now you understand all those calls to GLOBAL in the preceding sections! Note that each of the global variables we used in those examples was mentioned in an appropriate GLOBAL declaration before we every tried to reference it.

Finally, when do you use global variables? This is not so simple to answer. You can, of course, use them anywhere you want. But a rule of style in Lisp programming says that you should never use them if you can figure out a way to use local variables instead. In general try to use them to represent pieces of data that must be referenced by many functions. Data that are used by only a few functions can almost always be passed as function parameters. Section 11 will present several examples of the appropriate use of global variables. For now, just be aware that their use should always be carefully considered.

5.2.4 Good Programming Style in Lisp

Writing modular functions and limiting the use of global variables are two aspects of good programming style in Lisp. Other aspects of the use of functions are equally important. Perhaps the most important rule to keep in mind is that well-written large Lisp programs usually comprise a large collection of separate functions. This design of a system is often difficult to appreciate for novice Lisp programmers who are familiar with other languages. In languages like FORTRAN or BASIC, you are encouraged to think of programs as consisting of one large main program, and a few auxiliary subroutines. In fact, in many cases the subroutines are really not separate from the main program, they are just blocks of code you branch into and out of inside "the program". In Lisp, things are completely different. First, there is no such thing as a "main program". There are only modular function definitions, which may call one another. And there need not even be a "top-level" function which calls the others, but is not called by them. Recursion in Lisp enables function "A" to call "B", which, in turn, calls "A" again. Second, rarely are any of the functions in a large Lisp program themselves large. It is almost always a good rule of thumb to never let any Lisp function exceed about 20 lines of code. In the following sections we will have a chance to put these and other rules into practice.

5.3 Saving Functions

We have written a few small functions: nothing that would be difficult to type in every time we started a new session. But sooner or later it will become painful to retype large programs every time we restart the system. **UO-LISP** provides both an editor for modifying the definition of a function and a filing system to save and restore functions from disk.

To save a set of functions and global variables requires four simple steps:

1. Define all the functions using DE.
2. Create a GLOBAL variable by which you want to know this collection of functions. This variable is known as the file control variable. Normally this variable should have the same name as the first characters of the name of the file in which the functions are to be stored.
3. Assign a list to the file control variable using SETQ. The first element of the list should be a string with the name of the file that you want the functions written to. The remaining elements should be the names of functions you want saved or expressions to execute during the loading process (the following example will demonstrate creating and initializing some global variables using this process).
4. Call the SAVE function with the file control variable as its unquoted argument.

In the following example we create a global variable, assign it a value, define a function and save the whole works in the file FIRST.LSP.

```
+----------------------------------------------------------------------+
|                                                                      |
|     1: (GLOBAL '(GGG))                                               |
|     NIL                                                              |
|                                                                      |
|     2: (SETQ GGG 34)                                                 |
|     34                                                               |
|                                                                      |
|     3: (DE FN1 (X Y) (TIMES X (TIMES Y Y)))                          |
|     FN1                                                              |
|                                                                      |
|     4: (GLOBAL '(FIRST))                                             |
|     NIL                                                              |
|                                                                      |
|     5: (SETQ FIRST                                                   |
|          '("FIRST.LSP"                                               |
|            (GLOBAL '(GGG))                                           |
|            (SETQ GGG 34)                                             |
|            FN1))                                                     |
|     ("FIRST.LSP" (GLOBAL (QUOTE (GGG))) (SETQ GGG 34) FN1)           |
|                                                                      |
|     6: (SAVE FIRST)                                                  |
|     FIRST                                                            |
|                                                                      |
+----------------------------------------------------------------------+
```

SAVE should be called periodically during a session so that a hardware or software crash does not cause all your work to be lost. The file system will change the name of the file each time SAVE is entered by changing the extension into Enn and adding one to nn for each call. This will create a string of file names for the above of FIRST.LSP, FIRST.E00, FIRST.E01, FIRST.E02, and so on.

To restore a file created by SAVE is the function of LOAD. LOAD will read a file created by SAVE and create a file control variable for it so that it can be saved again. The name of the variable is returned by LOAD. For example:

```
+----------------------------------------------------------------------+
|                                                                      |
|     1: (LOAD "FIRST.E02")                                            |
|     FIRST                                                            |
|                                                                      |
+----------------------------------------------------------------------+
```

Here, the third incantation of FIRST.LSP is restored and the file control variable FIRST created.

At this point you should try and define a few simple functions, save them on disk, and restore them.

CHAPTER 6
FLOW OF CONTROL

You may have noticed that Lisp, like almost every other computer language, has a default order of evaluating expressions. If you type several expressions to **UO-LISP** at once, like this:

```
1: (LIST 'A 'B) (LIST 'C 'D) (LIST 'E 'F)
```

UO-LISP will evaluate them in the order presented, returning their values in that order:

```
 (A B)

2(C D)

3(E F)

4:
```

Similarly, remember that when Lisp evaluates the arguments to a function it does the first, then the second, then the third, and so on. And when a function is called, the forms in the body of the function are evaluated in the exact order given when the function is defined. These are examples of the sequential flow of control that is standard in Lisp, BASIC, FORTRAN, and probably every other language you know.

However, to develop sophisticated programs you often need to go beyond simple sequential evaluation of expressions. Branching statements, common to many languages, are examples of ways of circumventing sequential flow of control. In this section we discuss a few of the ways that Lisp provides to control the evaluation of S-expressions. In particular we will look at conditional, iterative, and recursive flow of control in Lisp. You will get a glimpse of how in Lisp, unlike other languages which come with a fixed set of ways of controlling execution, it is actually possible to create your own flow of control functions.

6.1 Conditional Flow of Control

It is often the case that you want something to happen only if a certain condition is true. In other languages you might express this using conditional statements such as:

```
        IF (J.EQ.5) N = 1            [FORTRAN]
or
        100 IF J = 5 THEN N = 1      [BASIC]
or
        if j == 5 then n = 1         [C]
```

There are several different ways of expressing these forms in Lisp.

6.1.1 AND and ORD

In Lisp you express conditional evaluation the way you do everything else: with a function. The simplest conditional functions in Lisp are AND and OR. To express the above assignment using AND you would say:

```
+----------------------------------------------------------------------+
|                                                                      |
|    1: (AND (EQ J 5) (SETQ N 1))                                      |
|                                                                      |
+----------------------------------------------------------------------+
```

The general form of AND is:

```
(AND <form1> <form2> <form3> ... <formj> ... <formM>)
```

where <formj> is any S-expression. AND evaluates the <formj>s in order until one returns a NIL value, at which point AND returns NIL and leaves the remaining forms unevaluated. Thus, in the above case, if (EQ J 5) returned NIL (say J had the value 6), AND would not evaluate the next form, and N would not get set to 1. If all AND's arguments evaluate to non-NIL, AND returns the value of the last <formM>.

Note that AND can have any number of arguments, and in the simple case where it is given only two, it has a natural interpretation in terms of familiar IF-THEN constructions:

```
        (AND <form1> <form2>)
```

means:

```
          IF <form1>
             THEN <form2>
```

OR is the converse of AND. It evaluates each of its arguments in order until one returns a NON-NIL value, at which point OR returns that non-NIL value and leaves the remaining forms unevaluated. If all OR's arguments evaluate to NIL, OR returns NIL. In the simple case where OR is given only two arguments, like AND, it also has a natural interpretation in terms of IF-THEN:

```
          (OR <form1> <form2>)
```

means:

```
          IF NOT (<form1>)
              THEN <form2>
```

6.1.2 COND

The Lisp functions AND and OR are sufficient to express simple conditionals, but what if you want to say more complicated things?

```
                { BASIC }

          IF I = 5 THEN LET N = 1
          IF I = 6 THEN LET N = 0
```

or

```
                { PASCAL }

          if i = 5 then n := 1
            else if i = 6 then n := 0;
```

To express complex conditionals, Lisp has a very general function called COND, which combines both the actions of AND and OR. To express the above complex conditional using COND you would say:

```
+----------------------------------------------------------------+
|                                                                |
|    1:  (COND ((EQ I 5) (SETQ N 1))                             |
|              ((EQ I 6) (SETQ N 0)))                           |
|                                                                |
+----------------------------------------------------------------+
```

The general form of COND is:

```
    (COND (<test1>  <sequence1>)
          (<test2>  <sequence2>)
              .
              .
          (<testi>  <sequencei>)
              .
              .
          (<testN>  <sequenceN>))
```

where <testi> is any S-expression, and <sequencei> is any sequence of 0 or more S-expressions. The (<testi> <sequencei>) lists that are the arguments to COND are often referred to COND clauses with the <test> being called the antecedent and the <sequence> the consequent.

COND first evaluates <test1> (the form (EQ J 5) in the previous example). If that returns any non-NIL value, then the remaining forms in that COND clause (<sequencei>) are all evaluated in order. In the previous example there was only one such form, (SETQ N 1), but we could have put any number of S-expressions there. After the <sequence1> forms are evaluated, COND stops, returning the value of the last form in the sequence. No other COND clauses will be evaluated. However, if <test1> had the value NIL, then COND does not evaluate the remaining forms in the COND clause. In this respect it is acting like AND, only evaluating <sequence1> if <test1> is non-NIL. Instead of evaluating <sequence1>, COND repeats the above process with the second COND clause. It evaluates <test2>, and if the value of <test2> is non-NIL, then, analogous to the first clause, the <sequence2> forms are are all evaluated, and the COND stops. Also analogous to the first clause, if <test2> evaluates to NIL, the third COND clause is tried. In general, this evaluation of the <testi> forms continues until one finally returns a non-NIL result, at which time the remaining S-expressions in that clause are evaluated, and the COND returns. In this respect it is acting like OR, stopping evaluation when the first non-NIL <testi> is found. More information about COND can be found in Chapter 2.10 of your **UO-LISP** Learner's Manual.

The action of COND may seem confusing at first, because it is such a general conditional statement. You might find it useful to think of COND in terms of IF-THEN conditionals, with which you might be more familiar. Here's what COND looks like in IF-THEN form:

```
IF <test1>
   THEN <form11> <form12> <form13> ...
  ELSEIF <test2>
     THEN <form21> <form22> <form23> ...
    ELSEIF <test3>
       THEN <form31> <form32> <form33>
                     .
                     .
             ELSEIF <testN>
                THEN <formN1> <formN2> <formN3> ...
```

Another example of the use of COND will also give you a better feel for how it is used. Suppose you wanted a function, called ADDRESS, that would remember the addresses of your friends for you. More specifically, ADDRESS should be a function of one argument (a friend's name), and should return the friend's address when called. Here is a simple way to implement ADDRESS using COND:

```
+----------------------------------------------------------------------+
|                                                                      |
|    1: (DE ADDRESS (NAME)                                             |
|         (COND ((EQ NAME 'JED) '(411 LA SALLE))                       |
|               ((EQ NAME 'WILLIAM) '(765 LONG BEACH))                 |
|               ((EQ NAME 'STEPH) '(918 OCEAN))                        |
|               ((EQ NAME 'HANK) '(1010 SANTA MONICA))                 |
|               ((EQ NAME 'DAVE) '(1055 MANNING))))                    |
|    ADDRESS                                                           |
|                                                                      |
|    2: (ADDRESS 'HANK)                                                |
|    (1010 SANTA MONICA)                                               |
|                                                                      |
|    3: (ADDRESS 'STEPH)                                               |
|    (918 OCEAN)                                                       |
|                                                                      |
|    4: (ADDRESS 'ARNOLD)                                              |
|    NIL                                                               |
|                                                                      |
+----------------------------------------------------------------------+
```

Note, as the last example (4:) illustrates, if all of the COND <testi>s return NIL, COND returns NIL. It is common practice to use the last clause of a COND as an "otherwise" clause; a clause whose <sequencei> forms should be evaluated, but only if all the other clauses fail. To accomplish this, you must use a last <test> that is guaranteed to return a non-NIL value. Traditionally T is used for this purpose, since it means the logical opposite of NIL. For example, suppose you wanted to set N to 0 if I was 5, to 1 if I was 6, and otherwise set N to 2. This is easily accomplished by:

```
+----------------------------------------------------------------------+
|                                                                      |
|     1: (COND ((EQ I 5) (SETQ N 1))                                   |
|              ((EQ I 6) (SETQ N 0))                                   |
|              (T (SETQ N 2)))                                         |
|                                                                      |
+----------------------------------------------------------------------+
```

6.1.3 COND, AND and OR are Special Forms

A final thing to note about COND, as well as AND and OR, is that like DE and SETQ they don't evaluate their arguments before operating on them. They are Lisp special forms. If you think about it a bit, this makes sense. COND is a conditional function that shouldn't evaluate its arguments automatically; its only supposed to evaluate the <sequencei> forms, when <testi> evaluates to non-NIL. Automatically evaluating all COND's arguments before entering COND would defeat its conditional purpose!

6.2 Iterative Flow of Control

It is very common to write programs to do things repeatedly. Branching statements like:

```
              { FORTRAN }

          IF (I.LT.10) GO TO 222
or
               { BASIC }

          GO TO 200
```

that allow you to return to a previous labelled statement, are one common way many programming languages provide for iterative flow of control. In this section we will look at one means Lisp provides to do iteration. It is not the only means, or the most elegant, but it is the most basic.

6.2.1 PROG

The basic function that allows you to do iterative, non-sequential, execution is called PROG. It is rather difficult to describe the structure of PROG, and in fact it has several different important features. So we will introduce PROG with a series of examples each demonstrating more of its features, culminating in an example where it is employed to define a useful iterative function.

The first important feature of PROG is that it allows you to declare and use an unlimited number of local variables. The first form in a call to PROG must be a list of atoms, or the null list "()"; for example:

```
   (PROG () ...)               [No variables]
```

or

```
   (PROG (FOOBAR) ...)         [One variable]
```

or

```
   (PROG (A B C D) ...)        [4 variables]
```

The atoms are local variables within that call to PROG. Recalling our discussion of local and global variables in Section 5.2.1, you will remember that this means the variables in the list can only be referenced inside the call to PROG, not outside. Thus, if I say:

```
1: (PROG (FOO) (SETQ FOO '(OH SAY CAN YOU SEE)))
NIL
```

there is no problem, but if I try to access FOO outside the PROG in which it is declared, I get into trouble:

```
1: (PROG (FOO) (SETQ FOO '(OH SAY CAN YOU SEE)))
NIL

2: FOO
***** (not global FOO)
```

We see here a general technique for creating local variables needed to temporarily remember the results of computations inside a function body. If you need to remember a value temporarily, don't assign it to a global variable. Just wrap the expression in a PROG and declare all the locals you need inside the PROG. The value of the PROG is set by executing the special function RETURN, the argument of RETURN becoming the value of the PROG. For example:

```
1: (DE BEERSONG (N)
     (PROG (REFRAIN)
        (SETQ REFRAIN (CONS N '(BOTTLES OF BEER)))
        (RETURN
          (APPEND REFRAIN
            (APPEND '(IN THE WALL)
              (APPEND REFRAIN
                      '(IF ONE OF THE BOTTLES SHOULD
                        HAPPEN TO FALL)))))))
BEERSONG

2: (BEERSONG 47)
(47 BOTTLES OF BEER IN THE WALL 47 BOTTLES OF BEER
IF ONE OF THE BOTTLES SHOULD HAPPEN TO FALL)

3: (BEERSONG 46)
(46 BOTTLES OF BEER IN THE WALL 46 BOTTLES OF BEER
IF ONE OF THE BOTTLES SHOULD HAPPEN TO FALL)
```

The forms after the list of local variables in the PROG function are called the body of the PROG. As the last example illustrates, the PROG body can contain any number of S-expressions, and they are generally evaluated in order. Note also that PROG, like DE, SETQ and COND is a special form; it does not evaluate its arguments before the function evaluation begins.

6.2.2 RETURN

PROG, like COND, and unlike other Lisp functions, has an important feature that lets you stop evaluating forms in the PROG body whenever necessary. If a call to the function RETURN is found anywhere in a PROG, the computation exits from the PROG immediately, evaluating no further forms in the PROG body. For example:

```
1: (DE ODDSQUARE (N)
   % ODDSQUARE returns the square of a number, N, if
   % it is odd, otherwise it returns just N.  A
   % number is odd if, when divided by 2, it yields
   % a remainder of 1. The built-in UOLISP function
   % REMAINDER returns this remainder.
     (PROG (SQ)
         (SETQ SQ (TIMES N N))
         (COND ((EQUAL 1 (REMAINDER SQ 2))
                (RETURN SQ)))
         (RETURN N) ))
ODDSQUARE

2: (ODDSQUARE 5)
125
```

In the previous call to ODDSQUARE, N will be set to 5, and so the SETQ will set the value of the local PROG variable SQ to 125. In the next form, COND first evaluates (EQUAL 1 (REMAINDER SQ 2)) which returns T, since SQ is 125. Then, since the first argument to the COND is non-NIL, COND evaluates its second argument, which is a call to the function RETURN. Note that RETURN has one argument, and its value, 125, will be the value returned from the PROG. Because the RETURN is encountered, the last S-expression of the PROG is not evaluated. If the argument to ODDSQUARE had been even, say 4, then (EQUAL 1 (REMAINDER SQ 2)) would have returned NIL, and COND would not have evaluated its second form, the call to RETURN. As a result, the last form in the PROG body, (RETURN N), would have been evaluated, and its value, 5, returned as the value of the PROG.

One final note: If no RETURN is executed during the body of the PROG, NIL is returned, not the value of the last expression as you might expect.

6.2.3 GO and Labels

The final important feature of PROG enables you to do iteration by looping through a portion of the PROG body code, much like you do in other languages. Within the body of a PROG you can place a label, which can be any Lisp atom. Then, as the code in a PROG body is executed, if a call to the function GO is encountered, with the label as an argument, control will return to S-expression right after the label. Thus, PROG labels in Lisp act much like numeric statement labels in BASIC, FORTRAN, and PASCAL. In this way, a block of S-expressions can be repeatedly evaluated, as shown schematically below:

```
(PROG ()
     .
     .
 LOOP                  % a label called LOOP
     .
<S-expressions>
     .
   (GO LOOP) ...)      % this call to GO results in a
                       % jump to just after the label LOOP
```

Here, the expressions between the label LOOP and the form (GO LOOP) will be evaluated again and again. Of course, you don't want to get into an infinite loop, so you need some way of jumping out of this code. To do this, you can place a call to the function RETURN inside the repeated code. By judiciously using GO, RETURN, and statement labels within a PROG body (and <u>only</u> within a PROG body!), you will find it possible to tailor the flow of evaluation of S-expressions in almost any way you wish. Chapter 2.8 of the <u>UO-LISP Learner's Manual</u> will give you more information on these functions, and how to use them correctly. For now, we will present an example that uses all the important features of PROG to give you an idea of how they are typically used to accomplish iteration.

6.2.4 An Example Using PROG

I want to define a Lisp function, called SUMSEGMENT, that is described as follows:

```
NAME:
  -SUMSEGMENT
INPUT:
  -two integers (M, N)
PLAN
  -if M is greater than N, SUMSEGMENT from M to N is equal to 0
  -if M is equal to N, SUMSEGMENT from M to N is M
  -otherwise, SUMSEGMENT from M to N is the sum of the integers
   M to N
```

Here is how SUMSEGMENT can be implemented using PROG, GO, RETURN, and labels:

```
1: (DE SUMSEGMENT (M N)
      (PROG (SUM)                                    % 1
            (COND ((GREATERP M N) (RETURN 0)))       % 2
           (SETQ SUM M)                              % 3
       LOOP                                          % 4
           (COND ((EQ M N)                           % 5
                  (RETURN SUM))                      % 6
                 (T                                  % 7
                  (SETQ M (PLUS 1 M))                % 8
                  (SETQ SUM (PLUS M SUM))            % 9
                  (GO LOOP)))))                      % 10
SUMSEGMENT

2: (SUMSEGMENT 2 7)
27

3: (SUMSEGMENT 5 5)
5

4: (SUMSEGMENT 5 3)
12
```

Let's go carefully through each line of this definition to fully understand it. The basic idea of this definition is to keep a running sum in the variable SUM, first initializing SUM to M, then incrementing M by 1, adding that new M to SUM, repeating the process until M is equal to N, and finally returning the SUM.

On line 1 (%1) we enter a call to PROG and declare a single PROG variable, SUM, which will be used to hold the accumulating sum. On line 2, we take care of the case where N is greater than M and return 0. If M is less than or equal to N, on line 3 we continue by initializing the variable SUM to M. Line 4 contains the PROG label, LOOP, which announces the start of our repeated block of code in the PROG body. The remainder of the PROG body is a call to COND. On line 5, the COND tests to see if M and N are equal. If so, the call to RETURN on line 6 returns the current value of SUM out of the PROG and out of the function. If M is still less than N, the "otherwise" clause of COND is executed. Line 8 increments M by 1, and line 9 increments the accumulating SUM by M. Then line 10 returns control to the label LOOP, and the COND is repeated. This iterative process will continue until M is incremented to the value of N.

6.3 Recursive Flow of Control in Lisp

The previous example showing how to specify an iterative process in Lisp should be familiar to many of you who already program. It is really not very different from how you might accomplish the same task in FORTRAN or BASIC. However, Lisp provides a way to accomplish this and many other related tasks in a way that is often much more elegant and succinct. In this section we will show how to write the same SUMSEGMENT function as a RECURSIVE computation, not an iterative one; and we will also spend some time giving you a "feel" for recursion -- when to use it, and how to understand recursive functions that others have written.

Let's begin by considering a definition scheme for SUMSEGMENT that is a bit different from the one we used above:

```
NAME:
   -SUMSEGMENT
INPUT:
   -two integers (M, N)
PLAN:
  -if M is greater than N, SUMSEGMENT from M to N is 0
  -if M is equal to N, SUMSEGMENT from M to N is M
  -otherwise, SUMSEGMENT from M to N is M + SUMSEGMENT
   from M + 1 to N
```

The only difference between this scheme and the previous one is the last clauses in their plans. While the first plan described a general, vague property of SUMSEGMENT, "SUMSEGMENT from M to N is the sum the integers M to N" the new plan describes a more precise, but somewhat strange property -- "SUMSEGMENT from M to N is M + SUMSEGMENT from M + 1 to N". While this latter property seems true (think about it!), does it really describe a procedure for computing SUMSEGMENT. In most languages, the answer is "no"; in Lisp, a recursive procedure can be constructed that is quite faithful to this plan. The function ADD1 used here is an abbreviation for (PLUS 1 M).

```
1: (DE SUMSEGMENT (M N)
     (COND ((GREATERP M N) 0)
           ((EQUAL M N) M)
           (T (PLUS M (SUMSEGMENT (ADD1 M) N)))))
SUMSEGMENT

2: (SUMSEGMENT 2 8)
35
```

This a recursive definition of SUMSEGMENT because it is defined in terms of itself (i.e., the definition of SUMSEGMENT embeds a call to SUMSEGMENT). Such a recursive definition is likely to seem strange to you. How can you define a function in terms of itself without getting into some sort of infinite loop? Since recursion is a bit unnatural, and because your previous programming experience may not have taught you about this important concept, we will spend some time looking at exactly how this recursive definition works.

The best way to understand how a function works is to watch it in action. In UO-LISP you can do this by tracing the function (see Chapter 2.17 of your UO-LISP Learner's Manual) with TR and watching as calls to the function are evaluated.

```
3: (TR SUMSEGMENT)
*** Redefined: SUMSEGMENT
SUMSEGMENT

4: (SUMSEGMENT 3 5)
Entering SUMSEGMENT              % 1.
  arg[1] = 3
  arg[2] = 5
Entering SUMSEGMENT              % 2.
  arg[1] = 4
  arg[2] = 5
Entering SUMSEGMENT              % 3.
  arg[1] = 5
  arg[2] = 5
Exiting SUMSEGMENT = 5           % 4.
Exiting SUMSEGMENT = 9           % 5.
Exiting SUMSEGMENT = 12          % 6.
12
```

To evaluate (SUMSEGMENT 3 5) (line % 1.), the COND of SUMSEGMENT is entered, and since none of the other COND <tests> are true, the otherwise (final)clause is evaluated. This is a call to PLUS, and both arguments to PLUS must be evaluated before the call to PLUS can return. The second argument is a call to (SUMSEGMENT 4 5), so SUMSEGMENT is re-entered (line % 2.), this time with the arguments 4 and 5, not 3 and 5. Again only the otherwise clause of the COND in SUMSEGMENT is true, so a call to PLUS is again initiated. As before, to do the PLUS, we must first evaluate its arguments, and this means re-calling SUMSEGMENT one more time, with the arguments 5 and 5 (line % 3.). When SUMSEGMENT is entered this time, however, the second clause of the COND, is evaluated, and the recursive call is never reached.

If you imagine each call to SUMSEGMENT as going down a level (as the above suggests), we are now at the bottom. This particular call to SUMSEGMENT does not result in any further calls. Instead, because M is equal to N, the second clause of the COND will return the value of M, 5, and this will be returned out of the call (SUMSEGMENT 5 5) (line %4.). Since the call (SUMSEGMENT 5 5) has now returned a value, we can complete the call to (PLUS 4 (SUMSEGMENT 5 5)), since both arguments to PLUS are now evaluated. This call to PLUS thus returns 9. We are now beginning the climb back up to our first function call. Recall that (PLUS 4 (SUMSEGMENT 5 5)) was evaluated as the part of the attempt to evaluate (SUMSEGMENT 4 5). Now that the PLUS has returned a value, the call (SUMSEGMENT 4 5) can now return a value, 9 (line % 5.). We have climbed another step back up. Because (SUMSEGMENT 4 5) is finished, the call to (PLUS 3 (SUMSEGMENT 4 5)) has all its arguments evaluated, and itself returns a value of 12. Finally, the last step -- back up to the top -- is to complete the evaluation of (SUMSEGMENT 3 5) by returning the value of its last expression, the call to PLUS. So, finally our top-level call returns 12 (line % 6.).

Following the path of a recursive computation like this is a bit complicated, and it will take a while for you to get a good feel for the flow of evaluation. One good exercise is to get into **UO-LISP**, trace it as above, then call SUMSEGMENT with various pairs of arguments. You can also trace the calls to PLUS, and even EQ, inside SUMSEGMENT, if you want more details of what is happening. I would also suggest beginning with small segments (e.g., 2 to 3, 10 to 12), then proceeding to larger ones.

6.4 Why Use Recursion?

If recursion is tricky to understand, you might well ask: Why use it at all, especially since familiar iterative constructs can do the same thing? The strongest answer is that not everything you might want to do in Lisp can be done iteratively. After you get some experience with Lisp you will see that some recursive operations on trees (lists that have other lists as elements) cannot be captured by iterative functions.

A second reason, which you should be able to appreciate even now, is that the code in a recursive function is often more compact, elegant, and clear than the code in its iterative counterparts. Compare the two definitions of SUMSEGMENT we have given. Notice first, that there is much less code in the recursive version. Also observe that the recursive version doesn't need the local variable SUM, that the iterative version requires, and that the recursive version never has to call the function SETQ to keep a running sum of the segment. Finally, look how much more closely the recursive version reflects its English plan. Recursive versions of functions are usually a much more transparent statement of the properties of the required operation than are iterative versions.

One of the best ways to get the concept of recursion under your belt is to define some useful recursive functions. In the next sections we'll do just that.

CHAPTER 7
COMPOSITE LISP FUNCTIONS

In this section we will define some functions, not only to give you practice with recursion, and using DE, but also to give you some idea of the style of well-written Lisp functions, and to provide you with an overview of some of the composite Lisp functions that are available in **UO-LISP**.

The built-in list manipulation functions that **UO-LISP** provides, such as CAR, CDR and CONS are very general, and ultimately, can be used to do almost any operation you want on a list. However, expressing the operation you need can often be very tedious, especially if you want to repeat that operation many times. By using "DE", you can define highly specific functions that encode any complex list manipulation operation that you want. Some of these functions, while much less general than built-in functions like CAR, are no less useful.

7.1 The Length of Lists

One very important operation on a list is to determine its length. This is such a frequent operation that we should have a function that names the operation, allowing easy use. How should we go about writing this function? When you are just learning Lisp, it is a good idea to plan out the function before writing the code. So for each function we write here, we will first lay out a definition scheme, like the ones we used for SUMSEGMENT in Section 6. The scheme gives us all answers we need to construct the function in an orderly fashion.

Here is a reasonable scheme for a function to compute the length of a list:

```
NAME:
  -LENGTH
INPUT:
  -a list (L)
PLAN:
  -if L is not a list or is the empty list NIL,
   the length of L is 0
  -otherwise, the length of L is 1 + the length of
          (CDR L)
```

As you look at the plan for LENGTH you should recognize its similarity to the last plan for SUMSEGMENT. You can probably guess that the function adhering to the plan will be recursive. Before looking at the definition below, use the plan for LENGTH, and your knowledge of the recursive definition of SUMSEGMENT to see if you can't implement LENGTH on your own. Go and try it in **UO-LISP** now! Write a definition, then try a few calls to your LENGTH function. As you do, notice how quickly you can test out and debug your code in Lisp.

Did your definition of LENGTH work? Compare it to the one below:

```
1: (DE LENGTH (L)
      (COND ((ATOM L) 0)
            (T (ADD1 (LENGTH (CDR L))))))
*** Redefined: LENGTH
LENGTH

2: (LENGTH '(A B C D NIL))
5

3: (LENGTH '(A (B C)))
2

4: (LENGTH NIL)
0
```

If your version of LENGTH didn't work, or you don't understand this one, get back into **UO-LISP**, type in the above definition and trace LENGTH (and possibly ADD1). Than call LENGTH with various lists and non-lists, watching what gets printed out. Soon you'll get the hang of it!

One final note, LENGTH is a built-in function. Once you understand its operation, you should feel free to use the internal version.

7.2 List Membership

Let's continue by writing a couple of other recursive functions for some especially common list operations. One absolutely necessary operation is to determine if an S-expression is in a given list. Here is a scheme for such a function:

```
NAME:
  -MEMBER
INPUT:
  -an S-expression (E)and a list (L)
PLAN:
  -if the L,  has no elements, E is not a member of L
  -if E is equal to the first element of the list,
    E is a member of L
  -otherwise, E is a member of L if E is a member
    of (CDR L)
```

Once again, I encourage you to use this scheme to try to implement a recursive version of MEMBER on your own. As in the previous cases, you will notice the definition of MEMBER follows its recursive plan very closely:

```
1: (DE MEMBER (E L)
     (COND ((ATOM L) NIL)
           ((EQUAL E (CAR L)) L)
           (T (MEMBER E (CDR L)))))
*** Redefined: MEMBER
MEMBER

2: (MEMBER 'B '(A B C))      %Since B is in the list,
(B C)                        %return the sublist,
                             %beginning with "B"
3: (MEMBER '2 '(A B C))
NIL                          %2 is not in the list

4: (MEMBER '(A B) '(1 2 (A B) 3))
((A B) 3)                    %The element you are
                             %looking for does not
                             %have to be an atom
```

The previous examples highlight one peculiarity of our definition of MEMBER. When E is a member of L, MEMBER doesn't just return T, indicating that E is in L; instead it returns the whole sublist of L beginning with E. This is a perfectly adequate value, since it will be non-NIL, and, in fact, the **UO-LISP** built-in function called MEMBER operates in exactly this way.

7.3 Concatenating Two Lists into One

As a final exercise, let's define a function that will take two lists and return one that combines them into one long list. Once again, we will use a recursive plan.

```
NAME:
  -APPEND
INPUT:
  -two (possibly null) lists, L1 and L2
PLAN:
  -if L1 is a null list, the result of appending L2 to
   L1 is L2
  -otherwise, the result of appending L2 to L1 is the
   same as CONSing the first element (CAR) of L1 onto
   the result appending L2 onto the remainder (CDR)
   of L
```

This scheme once again implies a recursive definition. Although this definition is a bit trickier than the previous two, give it a try, and to fully understand the following definition make sure to actually implement it in **UO-LISP** and carefully trace its execution on a variety of inputs.

```
+------------------------------------------------------------------+
|                                                                  |
|     1: (DE APPEND (L1 L2)                                        |
|          (COND ((NULL L1) L2)                                    |
|                (T (CONS (CAR L1)                                 |
|                         (APPEND (CDR L1) L2)))))                 |
|     *** Redefined: APPEND                                        |
|     APPEND                                                       |
|                                                                  |
|     2: (APPEND '(THE RAIN IN SPAIN)                              |
|                '(FALLS MAINLY ON THE PLAIN))                     |
|     (THE RAIN IN SPAIN FALLS MAINLY ON THE PLAIN)                |
|                                                                  |
|     3: (APPEND NIL '(A B C))     %The empty list has no          |
|     (A B C)                      %elements, so doesn't           |
|                                  %contribute anything to         |
|     4: (APPEND '((A) B C) NIL)   %the appended list              |
|     ((A) B C)                                                    |
|                                                                  |
|     5: (APPEND NIL NIL)          %To really test a               |
|     NIL                          %definition it is always        |
|                                  %important to test it on        |
|                                  %special cases and cases        |
|     6:  (APPEND 'A '(B C))       %where it should fail!          |
|     ***** (A not a pair for CAR)                                 |
|                                                                  |
+------------------------------------------------------------------+
```

As you can see, recursive definitions are the rule not the exception. The functions LENGTH, MEMBER, APPEND, as well as many others, are built out of more basic Lisp functions, but are also included in your **UO-LISP** system because they are so useful (see Chapter 2.14 in the **UO-LISP** Learner's Manual).

7.4 Other List Manipulation Functions

The foregoing sections should give you a feel for the wide range of list manipulation functions it is possible to implement in Lisp. All good Lisps come with many such functions built-in, even though the user could implement them, using only the primitive Lisp functions like CAR, CDR and CONS. The functions that come built-in have proved universally useful over the years, so they are supplied to save Lisp programmers the trouble of constructing their own "library" of utility functions. While we cannot go into detail on the semantics of these functions, or even enumerate them, we can classify the the list manipulation functions into several distinct categories, according to what they do to lists.

. List constructor functions make new lists out of other lists and atoms. Examples: CONS, LIST, APPEND.

. List selector functions access parts of existing lists. Examples: CAR, CDR.

. List predicates determine if some property is true of a list. Examples: EQ, EQUAL, MEMBER.

. Other list manipulation functions return various properties of lists. Example: LENGTH.

You will find many of all these kinds of functions described in Chapter 2 of your **UO-LISP** Learner's Manual.

CHAPTER 8
THE STRUCTURE EDITOR

UO-LISP has its own resident structure editor which performs a number of functions.

. Writes functions to disk

. Reads files of functions from disk

. Modifies the definitions of functions already defined

This chapter demonstrates some of the basic features of the structure editor by following a session through creation, testing, and completion of a simple Lisp program: the number guessing game. This program uses a binary search to "guess" at a number thought of by the player.

The creation of a program in the presence of the structure editor can be accomplished in two different ways. The program code can be entered directly into Lisp through the console terminal, or a file of functions may be created by the system editor and read in by the usual method. The first method has the advantage of providing immediate feedback on the matching of parentheses. The file editor has the advantage of permitting retyping of parts of a function without reentering it in its entireity. We suggest the use of the system editor in the initial creation of functions as it permits single character fixes until all the functions can be read in. For the purposes of the this exercise, we recommend that you enter the program using the system editor and make the file correspond exactly to what you see here. Don't fix any bugs or mispellings you find, that will be part of the exercise.

```
(DE NUMBERGUESS () (GUESS 0 127) )

(DE GUESS (LOW HIGH)
  (COND ((EQUAL LOW HIGH) (PRIN2T LOW))
        ((ISITGT (HALF HIGH LOW))
                (GUESS (ADD1 (HALF HIGH LOW)) HIGH))
        (T (GUESS LOW (HALF HIGH LOW))) ))

(DE HALF (A B) (DIVIDE (PLUS A B) 2))

(DE ISITGT (N)
  (PRIN2 "Is it greater than ")
  (PRIN2 N)
  (PRIN2 " (answer Y or N)?")
  (YORN (READ)) )

(DE YORN (C)
  (COND ((EQ C 'Y) (PROGN (TERPRI) T))
        ((EQ C 'N) (TERPRI))
        (T (PRIN2T "Y or N only!")
           (YORN (READ))) ))
STOP
```

Put this into a file called GUESS.LSP using what ever editor you have handy: even the standard CP/M line editor will create files which can be read by Lisp. Verify that what you typed matches the above character for character (don't worry about the spaces though). Then enter the following:

```
+------------------------------------------------------------------+
|                                                                  |
|    1: (LOAD "GUESS.LSP")                                         |
|    GUESS                                                         |
|                                                                  |
+------------------------------------------------------------------+
```

If you have typed things correctly, the editor should respond with GUESS. If not, it will probably stop with the prompt "Please enter STOP" after the LOAD. This usually indicates a missing parentheses or string terminator. If this happens, type ^C to exit from Lisp and fix the file to exactly match that above.

The GUESS returned by LOAD is the name of the file control variable that contains information needed by the structure editor to write the edited file back onto disk. If you look at the value of GUESS, you will see:

```
+------------------------------------------------------------------+
|                                                                  |
|    2: GUESS                                                      |
|    ("GUESS.E00" NUMBERGUESS GUESS HALF ISITGT YORN)              |
|                                                                  |
+------------------------------------------------------------------+
```

The first element of the list is the name of a file which the editor will use to write the functions to after they have been edited. The extension is E followed by a two digit number which is incremented by one each time the file is written out. By this means many versions of the file may be created and changes made can be removed by "rolling back" to some previous version. You will occassionally have to remove some of the old versions of a file to keep from filling up a disk.

The rest of the GUESS list contains the names of the functions in the GUESS.LSP file. Note that if you had entered the functions directly into Lisp, you would also have to create this list and the file name by hand (see section 5.3).

It is time to try our program. The program waits for you to think of a number between 1 and 128. When you type something, it asks you a question which you must answer Y or N. Eventually, if you don't change your mind, it will zero in on the number you were thinking of. Think of the number 10 and after typing (NUMBERGUESS), you should see the following:

```
3: (NUMBERGUESS)
Is it greater than (63 . 1) (answer Y or N)? N

***** Non-numeric argument
```

Evidently there is something wrong with the program as the error and (63 . 1) indicate. Let's address the first issue as the Non-numeric argument error looks like it might be caused by trying to do some arithmetic on (63 . 1). Looking at the source code, we see a DIVIDE function call in HALF which should be a QUOTIENT. The structure editor will allow us to change this function name and reexecute the program without leaving the Lisp environment. To edit a function definition, call EDIT with the unquoted function name to edit as its argument. When EDIT is called, it goes into a loop which reads structure editor commands from the user and executes them. You can find the documentation for these commands in Chapter 3 of the <u>UO-LISP Learner's Manual</u>. The following sequence is one way in which the DIVIDE in HALF can be changed into a QUOTIENT.

```
4: (EDIT HALF)
Edit[1] PP          % Display definition
(EXPR LAMBDA (A B) (DIVIDE (PLUS A B) 2))

Edit[1] F DIVIDE    % Find DIVIDE
Found 1 of 1

Edit[2] R QUOTIENT  % Replace with QUOTIENT
Edit[2] ^           % Exit FIND loop
Edit[1] PP          % Display changed form
(EXPR LAMBDA (A B) (QUOTIENT (PLUS A B) 2))

Edit[1] E           % Save edited form.
HALF
```

When the structure editor command loop is in control, the prompt changes to Edit[n] where n is a number indicating the current editor nesting level. The first thing we want to do is to be sure of what we are editing. The PP command causes the expression currently being edited (the "focus" of the editor) to be prettyprinted. In this case, we see the internal definition of the function HALF. Our strategy is to find and replace each occurrence of the atom DIVIDE by the atom QUOTIENT. The F DIVIDE command causes the structure editor to search the entire expression to all levels for the atom DIVIDE. When a DIVIDE is found, the editor is reinvoked on the expression found and the nesting level is incremented by 1. When an expression is found, a message is printed indicating the number of occurrence of the expression and how many were being looked for. The R QUOTIENT command causes the DIVIDE to be replaced by the atom QUOTIENT. The ^ command causes the focus of the editor to return to the previous command level, Edit[1] which was examining the entire structure. The final PP shows that QUOTIENT has indeed replaced DIVIDE in the HALF function. The E command causes the changes to be saved and the function updated (though only in storage, not in the disk file). If you don't like the changes you've made, the Q command will exit the editor without updating the function definition.

Once this change has been made, the game runs to completion. This time I'm thinking of the number 85.

```
+--------------------------------------------------------------------------+
|                                                                          |
|     5: (NUMBERGUESS)                                                     |
|     Is it greater than 95 (answer Y or N)? N                             |
|                                                                          |
|     Is it greater than 79 (answer Y or N)? Y                             |
|                                                                          |
|     Is it greater than 87 (answer Y or N)? N                             |
|                                                                          |
|     Is it greater than 83 (answer Y or N)? Y                             |
|                                                                          |
|     Is it greater than 85 (answer Y or N)? N                             |
|                                                                          |
|     Is it greater than 84 (answer Y or N)? Y                             |
|                                                                          |
|     85                                                                   |
|                                                                          |
+--------------------------------------------------------------------------+
```

To demonstrate a few more features of the editor, we will add an English explanation of the game at the beginning of the program by editing NUMBERGUESS again. The following sequence adds two print statements to the beginning of the NUMBERGUESS function.

```
+--------------------------------------------------------------------------+
|                                                                          |
|     6: (EDIT NUMBERGUESS)                                                |
|     Edit[1] PP             % Display expression.                         |
|     (EXPR LAMBDA NIL (GUESS 0 127))                                      |
|     Edit[1] 3T             % Move down 3 CDR's.                          |
|     Edit[2] P              % Display expression.                         |
|     ((GUESS 0 127))                                                      |
|     Edit[2] I ((PRIN2T "       Number Game")                             |
|     Edit[2] (PRIN2T                                                      |
|     Edit[2]"Think of a number between 0 and 127 and type T")             |
|     Edit[2] (READ))                                                      |
|     Edit[2] 3^             % Back to top level.                          |
|     Edit[1] PP             % Check correctness.                          |
|     (EXPR LAMBDA                                                         |
|           NIL                                                            |
|           (PRIN2T "       Number Game")                                  |
|           (PRIN2T                                                        |
|            "Think of a number between 0 and 127 and type T")             |
|           (READ)                                                         |
|           (GUESS 0 127))                                                 |
|     Edit[1] E              % Save the new definition.                    |
|     NUMBERGUESS                                                          |
|                                                                          |
+--------------------------------------------------------------------------+
```

The command 3T is the command T repeated 3 times (T stands for Tail, the CDR of the current expression). It simply changes the focus of the editor to CDR of the CDR of the CDR of the expression being edited. Most structure editor commands can be prefixed by repetition counts in this manner. The P command following this assures us that the right expression is being edited. P is like PP but calls the PRINT function instead. The I command inserts a list on the front of a list, that is, the expression being edited is appended to the expression following the I, in this case two PRIN2T and a READ function call. The 3^ instruction "backs out" of the editing to the top level. The final PP verifies that the insertion was done correctly.

The most important thing to remember is saving the functions you have edited in a disk file. This does not happen automatically when you leave Lisp. Use the SAVE function as described previously. SAVE writes the current definitions of the functions onto disk and increments the version number in the control variable by 1.

```
7: (SAVE GUESS)
GUESS
```

The edited file will now be found in GUESS.E00.

This concludes the demonstration of the structure and character editors. There are many more commands explained and a one page synopsis of them in the conclusion of Chapter 3 of the **UO-LISP** Learner's Manual.

CHAPTER 9
PROPERTY LISTS

In Section 6 you will recall we used COND to implement a simple function, ADDRESS, that remembered tha addresses of a set of people. This function hinted at the ability Lisp provides to store and access symbolic data. In this section we discuss a much more general database and data retrieval facility of Lisp.

One problem with the ADDRESS function was that the small database of people and addresses it stored was not extensible. If we wanted to add a new person, we would have to change the definition of ADDRESS. If we wanted to store more information about a person -- say his age as well as his address -- we probably would have to write a totally different function.

There is a very general way Lisp provides to store information about any entity. We can think of entities, like people, as having attributes, such as age, address, height, etc. Each entity is described by specifying specific values for those attributes. For example, Bob's age might be 19, his address 1066 Main St, his height 180 cm, etc. How can we store this kind of information in Lisp? We begin by using an atom to represent the entity we want to describe. We might use BOB to represent our friend Bob. We have already seen that atoms such as BOB can have values, but they also have one other facet. Each Lisp atom also has a property list which is ideal for storing attributes and their values.

The **UO-LISP** function PUT is used for assigning a particular value to an attribute of an entity. It is a function of three arguments with the following general form:

```
(PUT <entity> <attribute> <value>)
```

Here are a few specific examples of its use:

```
1: (PUT 'BOB 'AGE 19)
19                     %Sets the value of BOB's AGE
                       %attribute to 19
3: (PUT 'BOB 'ADDRESS '(1066 MAIN))
(1066 MAIN)            %Sets BOB's ADDRESS attribute.
                       %Note the value is returned
```

Now we need to be able to access any other these attributes and values as needed. The function GET is the main selector function for property lists. It has the form:

```
        (GET <entity> <attribute>)
```

and returns <entity>'s <attribute>. For example:

```
+--------------------------------------------------------------------+
|                                                                    |
|    3: (GET 'BOB 'AGE)                                              |
|    19                                                              |
|                                                                    |
|    4: (GET 'BOB 'ADDRESS)                                          |
|    (1066 MAIN)                                                     |
|                                                                    |
+--------------------------------------------------------------------+
```

You'll probably find property lists a very natural way of thinking about data storage and access. In fact you might already be able to see how to reimplement the ADDRESS function using PUT and GET. Try it! More information about property lists can be found in Chapter 2.5 of your UO-LISP Learner's Manual. We'll also be using property lists in a later example, to give you some more practice.

CHAPTER 10
INPUT AND OUTPUT

We've talked in some detail about the evaluation of S-expressions, but input and output (I/O) refers to the reading of S-expressions, and printing them. Input and output in Lisp is usually pretty simple, and we will only discuss a few I/O functions. They should be all you need to know for a while.

10.1 Input

The basic input function is called READ. READ reads and returns an S-expression from the currently selected input file. For all our purposes, this file will be your terminal, so READ is a way of reading from the terminal. For example:

```
1: (READ) 5
5

2: (READ) (FOO BAR)
(FOO BAR)
```

You may have written a number of Lisp functions into a file and want to read them into Lisp so you can use tham during a session. This could have been done by the SAVE function (see Chapter 8) or by a system editor. To read a whole file you should use the function LOAD:

```
1: (LOAD "FNS.LSP")
FNS
```

Notice that you should specify your file name using a string (characters surrounded by double quotes (")), not using an identifier. The string should contain both the name of the file (FNS), and its extension (.LSP in this case). LOAD will take each of the S-expressions in the file you give it and evaluate them. Thus if you have several calls to DE and SETQ in the file, the LOAD will rasult in a bunch of new variables and functions being defined. Once they are loaded, you will be able to access and use them just as if you had typed them to the Lisp interpreter directly. The value returned by LOAD is the name of the global file control variable (see Chapter 8).

10.2 Output

Just as you may need to read from the terminal or from a file, you might want to print S-expressions to a file to the terminal. The simplest output function is PRIN1. It takes one argument, an S-expression, prints it to the terminal, and returns the value printed:

```
1: (PRIN1 'FIVE)
FIVEFIVE          %The first FIVE is the result of
                  %printing, the second one is the
                  %value returned
2: (PRIN1 5)
55

3: (PRIN1 "FIVE")
"FIVE""FIVE"
```

PRIN2 is very much like PRIN1, except it does not print strings with their double quotes. Compare 3: above with 4: below

```
4: (PRIN2 "FIVE")
FIVE"FIVE"             %Quotes come off the expression
                       %printed, although not off the
                       %value returned
5: (PRIN1 'FIVE)       %Otherwise PRIN2 is the same as
FIVEFIVE               %PRIN1
```

PRINT is also like PRIN1, but it puts out a carriage return after its prints its expression:

```
6: (PRINT "FIVE")
"FIVE"
"FIVE"
```

Finally, PRIN2T is like PRIN2 except it adds a carriage return:

```
7: (PRIN2T "FIVE")
FIVE
"FIVE"
```

There is also a function that just puts out carriage returns. It is called TERPRI. PRIN2T could have been implemented as:

```
8: (DE PRIN2T (E) (PROG2
     (PRIN2 E)
     (TERPRI)))
*** Redefined: PRIN2T
PRIN2T
```

You can learn more about I/O functions in Chapter 2.16 of your UO-LISP Learner's Manual. Chapter 3, on the editor, also discusses the function LOAD.

We must use PROG2 above becase LISP allows only one S-expression to be used as the definition of a function. PROG2 simply makes both the (PRIN2 E) and (TERPRI) into one S-expression: (PROG2 (PRIN2 E) (TERPRI)), which behaves exactly the same as those two expressions evaluated in order.

PROG2 is a function which, after evaluating each S-expression, returns the value of the second S-expression — but it's value isn't really important. PROGN, a similar function, evaluates each S-expression but returns the value of the last expression. It could be used just as well but tends to be a little slower.

CHAPTER 11
AN EXTENDED EXAMPLE

In the previous sections you've learned a lot about Lisp, how it works and how to write Lisp functions. But that is not the same as learning how to use Lisp effectively to solve substantial problems. You know about the "bricks" and "planks" of Lisp; now you need to learn a bit about how to use these materials to build real programs. If you are a BASIC, FORTRAN, or PASCAL programmer, you will find that good programming style in Lisp is quite different than in the languages with which you are familiar. Therefore, in this final section we discuss several techniques for problem solving in Lisp and introduce some of the elements of style that characterize artificial intelligence programming.

11.1 An "Expert System" for Tic-tac-toe

To learn about building programs in Lisp, we'll actually build one. The program we develop below will play the game of Tic-tac-toe (hereafter "TTT"). Although TTT is a very simple game, it will serve to illustrate many important techniques of Lisp programming and problem solving. The approach we take to build our TTT program is very different than you'd take in any other language. We will build our program as a TTT "expert system".

Expert systems are kinds of AI programs that attempt not only to solve problems, but to solve them like humans do. This insistence on "intelligent" or "human-like" systems is a hallmark of much AI research, especially work in expert systems. Intelligent expert systems now exist for aiding in medical diagnosis, configuring computers, and for many other tasks.

How do humans solve problems? Expert systems research has found that people often solve problems by employing "rules of thumb", not necessarily correct algoriphms. For example, people play chess using rules like "If you are beginning a game, then move out your center pawns first, then your bishops". They do not try all moves in their heads, creating massive "lookahead trees" (as do the best computer programs for chess). Thus, expert systems are designed as rule-based programs. They are really very simple in structure. They comprise a (possibly large) set of simple rules of the form "If <condition> then <action>", and, to make a decision, they iterate through their rules until they find one whose <condition> is true. Then they "fire" the rule (do the rule's <action>). Usually, after a rule is fired, an expert system will go back to the start of its ruleset, and iterate through again, to make a new decision. And that's all!

Why is Lisp the obvious language for building expert systems? Take a look at the rules humans use to solve problems. Like the simple chess rule given above, they are typically expressed symbolically, often in English. Humans usually think in symbolic languages, not in numbers or arrays of numbers. Thus a language that supports symbolic computation, like Lisp, is an ideal medium for modeling much of human problem solving.

11.2 Developing the "Expert" Rules for TTT

Expert systems are typically built by getting a human expert, and finding out what rules he uses to solve his problems. For example, if you wanted to built an expert system for car repair, you might talk to a mechanic, or watch him work. This is referred to as "knowledge acquisition". You are acquiring his knowledge for your system. At this stage, the rules are written down in the language the expert finds most natural.

Let's acquire a knowledge base for our TTT expert system. Since we are all experts in TTT, all we have to do is introspect, and try to make explicit the rules we use to play the game. It is often surprisingly difficult to make explicit the knowledge you use to play even a simple game like this -- so much of our expertise is unconscious. Try it!

Here is one set of rules. These aren't the only rules that could be used to play TTT, and they are not the best rules either. Note that they are written down in their "natural" form; I haven't worried at all about how to represent them in Lisp.

Rule 1: If two squares in a line are occupied, and you occupy both, then play the empty square.

Rule 2: If two squares in a line are occupied and your opponent has both then play the empty square in the line.

Rule 3: If the line contains the center square, and its empty, play it.

Rule 4: If the line has an empty corner square, play on it.

Rule 5: If the line has any empty square, play on it.

11.3 Interpretation or Use of the Rules

The rules are written from the point of view of a player who decides upon a move by looking at each of the 8 "lines" of three squares (3 horizontal, 3 vertical and 2 diagonal) of the TTT board. The player stops looking when a rule fires for the particular line he is looking at, giving him a move. The rules themselves are ordered. For example, the player always wants to follow Rule 1, (winning the game), if applicable, rather than Rule 2 (preventing a win for the opponent). Thus the correct way to use this ruleset is to see if the first rule applies to any line; if not, repeat trying the second rule, and so on.

11.4 Implementing the Rules and the Rule Interpreter

We now implement our simple "expert" rules for playing the game. Because each rule naturally divides into an if-part and then-part, we choose to simply represent each rule as a list of two elements: the if-part followed by the then-part. Our implementation of the rules illustrates a top-down approach to problem-solving, which we employ frequently in Lisp programming. In this case, a top-down approach means we simply write the rules the way we wish them to appear; as close to their "natural" representation as possible. Later we will go back to define the required lower-level functions we require to support this natural representation. We do not implement the low-level functions first, then force the rules into a mold they dictate.

```
% TTTRULES as a global variable whose value will be
% the list of all TTT rules
(GLOBAL '(TTTRULES))

(SETQ TTTRULES
                    % Rule 1
 (((AND (EQ 2 (NUMBEROFSQUARES (OCCUPIEDSQUARES LINE)))
        (OCCUPIES SELF (OCCUPIEDSQUARES LINE)))
   (PLAYMOVE SELF (CAR (EMPTYSQUARES LINE))))
                    % Rule 2
  ((AND (EQ 2 (NUMBEROFSQUARES (OCCUPIEDSQUARES LINE)))
        (OCCUPIES OPPONENT (OCCUPIEDSQUARES LINE)))
   (PLAYMOVE SELF (CAR (EMPTYSQUARES LINE))))
                    % Rule 3
  ((AND (INCLUDES LINE CENTERSQUARE)
        (EMPTY CENTERSQUARE))
   (PLAYMOVE SELF CENTERSQUARE))
                    % Rule 4
  ((EMPTYSQUARES (CORNERSQUARES LINE))
   (PLAYMOVE SELF
     (CAR (EMPTYSQUARES (CORNERSQUARES LINE)))))
                    % rule 5
  ((NOT (COMPLETE LINE))
   (PLAYMOVE SELF (CAR (EMPTYSQUARES LINE)))) ) )
```

Once the rules are written, we should specify how they are used in playing TTT and finding a move. The function that manipulates and fires rules should act exactly as outlined in our specification in Section 11.2. The code that controls the execution of rules in expert systems is often called the "rule interpreter". The function FINDMOVE is our TTT rule interpreter. Note that the code is heavily commented to help you understand it. This is a practice we encourage in general.

```
% FINDMOVE dictates how the rules are used to
% determine the next move. We refer to FINDMOVE
% as the rule interpreter. Each rule has an if-part
% which looks at certain features of a TTT line
% (contiguous triplet). If the if-part of the rule
% is correct, the then-part of each rule suggests a
% move to play. Thus, the rule interpreter FINDMOVE
% operates in two loops. First (in OUTERLOOP) it
% iterates thru all of the rules (stored as the value
% of TTTRULES). For each such rule it will look at all
% lines on the board (INNERLOOP), TESTing to see if
% if-part of the rule works for that line. If it does,
% FINDMOVE DOes the then part of the rule, and stops.
% It continues its loops through rules and lines until
% it finds a rule that "fires". If no rule fires
% FINDMOVE returns NIL.

(DE FINDMOVE ()
  (PROG (LINES RULES)
     (SETQ RULES TTTRULES)
OUTERLOOP
     (COND ((NULL RULES) (RETURN NIL)))
     (SETQ LINES TTTLINES)
     (SETQ LINE (CAR LINES))
INNERLOOP
     (COND ((NULL LINES)(PROGN
            (SETQ RULES (CDR RULES))
            (GO OUTERLOOP))))
     (SETQ LINE (CAR LINES))
     (COND ((TEST (IFPART (CAR RULES))) (PROGN
            (DO (THENPART (CAR RULES)))
            (RETURN NIL)))
           (T (PROGN
            (SETQ LINES (CDR LINES))
            (GO INNERLOOP))))))
```

The implementation of the rule interpreter, and choice of representation for the TTT rules, constrains a few lower-level decisions. First, we see that several global variables are referenced in the code for the rules, so let's declare them:

```
% CENTERSQUARE will have the center TTT square as its
% value LINE is a global variable that the rules
% reference and which is set to successive TTT lines
% by the rule-interpreter function, FINDMOVE. SELF
% is a variable set to the mark the TTT expert system
% will use (either "X" or "O"). OPPONENT is set to the
% opponent's mark.
(GLOBAL '(CENTERSQUARE LINA SELF OPPONENT))
```

We can also now write several of the functions used by FINDMOVE:

```
(DE IFPART (RULE) (CAR RULE))

(DE THENPART (RULE) (CAR (CDR RULE)))
```

Because we have completed our representation of rules we can also say what it means to TEST the if-parts and DO the then-parts. The if-part is a single S-expression, and to TEST it we just want to evaluate it as a piece of Lisp code. If the value it returns in non-NIL, we want to do the then-part. The then-part is also a single S-expression, a call to the function PLAYMOVE. So to do the then-part, we also just want to evaluate it too. To force an expression to be evaluated, we just explicitly call the function EVAL:

```
(DE TEST (IFPART) (EVAL IFPART))

(DE DO (THENPART) (EVAL THENPART))
```

11.5 Choosing Data Representations

Now before we write the low-level functions that manipulate lines and squares, in the rules, we need to decide how to represent the TTT board in data structures. We will adopt a simple representation, not the only possible one or necessarily the best. We let the atoms A through I represent each square as follows:

```
A | B | C
---------
D | E | F
---------
G | H | I
```

Further, for each square, let's use the property list of its atom (A - I) to indicate who occupies it. We will use a property called "STATUS", and its value will be NIL when the square is empty, and X or O when occupied. Finally, since each square is represented as a Lisp atom, let's represent the TTT lines as lists of the 3 squares included in the line.

```
(GLOBAL '(TTTSQUARES     %A list of all squares.
          TTTLINES       %A list of all lines.
          CENTERSQUARE   %The center square name.
         ))

(SETQ TTTSQUARES '(A B C D E F G H I))

(SETQ CENTERSQUARE 'E)  %The center square.

(SETQ TTTLINES              %There are 8 lines of 3 squares
      '((A B C) (D E F) (G H I) %Three horizontal lines
        (A D G) (B E H) (C F I) %Three vertical lines
        (A E I) (C E G)))       %And two diagonal lines
```

11.6 Functions for the if-parts of TTT rules

These three representation decisions now make it possible to define all the low-level procedures called by the rule interpreter and the TTT rules. The procedures include conventional Lisp functions as well as predicates, which are functions that return only T or NIL, depending on their arguments.

```
% OCCUPIEDSQUARES takes a set of squares as an
% argument and returns the ones occupied by X or O.
(DE OCCUPIEDSQUARES (SQUARES)
  (COND ((NOT SQUARES) NIL)
        ((OCCUPIED (CAR SQUARES))
         (CONS (CAR SQUARES)
               (OCCUPIEDSQUARES (CDR SQUARES))))
        (T (OCCUPIEDSQUARES (CDR SQUARES)))))

% OCCUPIED is a predicate that takes a square, and
% returns T if someone is on it (X or O), otherwise
% it returns NIL.
(DE OCCUPIED (SQUARE)
  (COND ((EQ 'X (GETSTATUS SQUARE)) T)
        ((EQ 'O (GETSTATUS SQUARE)) T)
        (T NIL)))

% The predicate OCCUPIES takes a mark (either "X" or
% "O") and returns if that mark is on all the squares
% give as its second argument.
(DE OCCUPIES (MARK SQUARES)
  (COND ((NOT SQUARES) T)
        ((NOT (EQ MARK (GETSTATUS (CAR SQUARES))))
         NIL)
        (T (OCCUPIES MARK (CDR SQUARES)))))

% To find out if someone is on a square, just look at
% its STATUS property.
(DE GETSTATUS (SQUARE) (GET SQUARE 'STATUS))

% To put some mark on a square, just set its status
% property.
(DE PUTSTATUS (SQUARE MARK) (PUT SQUARE 'STATUS MARK))

% EMPTYSQUARES is the opposite of OCCUPIEDSQUARES. It
% takes a set of squares and returns all those not
% occupied.
(DE EMPTYSQUARES (SQUARES)
  (COND ((NOT SQUARES) NIL)
        ((EMPTY (CAR SQUARES))
         (CONS (CAR SQUARES)
               (EMPTYSQUARES (CDR SQUARES))))
       (T (EMPTYSQUARES (CDR SQUARES)))))
```

```
% EMPTY is the opposite of OCCUPIED. It takes a square,
% and returns T if someone is not on it, otherwise it
% returns NIL.
(DE EMPTY (SQUARE) (NOT (OCCUPIED SQUARE)))

% CORNERSQUARES is a function that takes a set of
% squares and returns all those that are at corners of
% the TTT board (A, C, G and I). Note CORNERSQUARES
% has exactly the same recursive plan for looking
% through the list of squares as both OCCUPIEDSQUARES
% and EMPTYSQUARES.
(DE CORNERSQUARES (SQUARES)
  (COND ((NOT SQUARES) NIL)
        ((CORNER (CAR SQUARES))
         (CONS (CAR SQUARES)
               (CORNERSQUARES (CDR SQUARES))))
        (T (CORNERSQUARES (CDR SQUARES)))))

% The predicate CORNER returns non-NIL if its argument,
% SQUARE, is a corner square, otherwise it returns
% NIL.
(DE CORNER (SQUARE) (MEMQ SQUARE '(A C G I)))

% COMPLETE is a predicate that takes a set of squares,
% and returns T if there are no empty squares in the
% set, otherwise, it returns NIL.
(DE COMPLETE (SQUARES) (NULL (EMPTYSQUARES SQUARES)))

% INCLUDES is a predicate of two arguments, a set of
% squares and a square. It returns NIL only if the set
% does not include the square. Since sets are
% implemented as lists of squares they include, we can
% see if a square is included in the set just by seeing
% if it is a member of the list.
(DE INCLUDES (SQUARES SQUARE) (MEMQ SQUARE SQUARES))

% NUMBEROFSQUARES is a function that returns the
% number of squares in a line or set of squares, given
% as an argument to the function. Since lines are
% implemented as lists, we can determine the number
% of squares by just computing the length of the list.
(DE NUMBEROFSQUARES (SQUARES) (LENGTH SQUARES))
```

```
+------------------------------------------------------------------+
|                                                                  |
| % PLAYMOVE is a function that takes a square and a               |
| % mark (either X or O). If the square is already                 |
| % occupied it just returns NIL. If the square is empty,          |
| % it puts the mark on the square (by modifying the               |
| % status property of the square, and returns the mark.           |
| (DE PLAYMOVE (MARK SQUARE)                                       |
|   (COND ((OCCUPIED SQUARE) NIL)                                  |
|         (T (PUTSTATUS SQUARE MARK))))                            |
|                                                                  |
+------------------------------------------------------------------+
```

11.7 The Importance of Abstraction Barriers

This completes specification of all the functions called by the TTT rules and rule interpreter. Notice how simple most of them were to write. This simplicity is a consequence of the way we chose the represent our data about lines and squares in TTT, and underscores the importance of representation decisions. Taking care in deciding your data representations is really just another part of the top-down approach to programming and problem-solving that we have been advocating. Each time you approach a large programming problem you should take time to write out all such decisions before you write a line of code. Think of making data representation decisions as a phase of programming, just like coding or debugging. You will find that if you make these decisions deliberately and explicitly, most of the functions you need will be very easy to implement, as they were here. On the other hand, if you spend little thought on data representation, you will almost always find it difficult to write your functions. In addition, it will probably take you longer to test and debug your code than it might have, and it will be tougher for other people (even yourself!) to understand how your program is operating.

Since our judicious choice of data representations made any of the above functions very small, one might ask why we bothered to write some of them at all. Why bother with the access and setting functions GETSTATUS and PUTSTATUS? Why not just use GET and PUT directly? And, why bother defining INCLUDES; why not just use call to MEMQ every time you need to see if a square is in a line?

This substitution would actually be a very bad idea. By using MEMQ instead of INCLUDES we would be showing the specific data representation we had chosen for lines and squares. On the other hand, use of INCLUDES abstracts away from the details of the representation we have chosen. There are several reasons this abstraction is very desirable. First, it makes the meaning of the code we write much clearer to others. For example, when someone is reading your TTT rules, it will be much easier for him to understand what you knowledge intend to encode by "(INCLUDES LINE 'E)" than by "(MEMQ 'A '(A B C))". In understanding the meaning of a rule, seeing the details of its representation is not just irrelevant, it is detrimental. The abstraction afforded by INCLUDES creates a barrier between the viewer of the code and the details of imphementation that actually facilitates comprehension.

There is a second even more important reason for creating such abstraction barriers. Assume that had written your TTT program using MEMQ, not INCLUDES. Now suppose someone tells you that thera is a much better way to represent your TTT data, say, using arrays, not lists. You want to change your representation, but you realize that your program has dozens of calls of the form "(MEMQ <square> <list>)". So you have to give up this new, better, representation, because it would take you hours or days to make all the required changes.

By following the policy of using functions like INCLUDES to abstract away from the details of data representation you can avoid this problem, preserving the flexibility and modularity of your code. In this case, for example, to change to an array representation all you sould have to do is change the definition of INCLUDES (and a couple of other functions that manipulate lists of squares directly). All the calls to INCLUDES would remain intact, because the meaning of INCLUDES has not changed, just the specifics of its implementation. Thus abstraction barriers not only make ccde more comprehensible and free of implementation detail, they also make programs much easier to modify and enable you to try out different representations of data.

The important practical lessons to conclude from this discussion of data abstraction are:

. Gather all the code that manipulates (craates, accesses, changes) the low-level data structures that you have selected into a few functions.

. Name these functions to reflect the meaning of the computations they effect, not the low-level data manipulations they do.

. Use these functions in all high-level code, to abstract away from details of your data representation.

11.8 Some Simple Control Functions

Now that all of the functions implementing the TTT rules and FINDMOVE are complete, all we need to make our TTT "expert system" operational are a couple of high-level routines to begin, record, and complete the game. As usual, we proceed in a top-down fashion.

```
% PLAYTTT is the top-level function called to play a
% game. It first initializes all squares to empty,
% then determines who plays X and O, by asking the
% user (the OPPONENT) which he wants to play.  The
% system plays SELF. It then enters a loop. First it
% checks to see if the board is full, in which case it
% announces a tie. If not, a move is played.  If it is
% the opponent's move, the user is asked to pick a
% square, and it is played on. (Note if the user picks
% a square that is already played on, PLAYTTT doesn't
% give him another chaNce; it is as if the user chose
% to make no move on his turn. If it is the system's
% move, it calls FINDMOVE to select a square. After
% each move, PLAYTTT checks to see if there is a
% winner. If so it returns; if not it loops back for
% a new move.
(DE PLAYTTT ()
  (PROG (NEXTMOVE SQUARES)
     (SETQ SQUARES TTTSQUARES)
LOOP1
     (MAKEEMPTY (CAR SQUARES))
     (SETQ SQUARES (CDR SQUARES))
     (AND SQUARES (GO LOOP1))
     (SETQ OPPONENT
      (QUERY "Which do you want to play [X or O]?"))
     (SETQ SELF (OTHERMARK OPPONENT))
     (SETQ NEXTMOVE 'X)
LOOP2
     (COND ((NOT (EMPTYSQUARES TTTSQUARES)) (PROGN
            (PRIN2T "A tie game!")
            (RETURN NIL)))
           ((EQ NEXTMOVE OPPONENT) (PROGN
            (PLAYMOVE OPPONENT
                 (QUERY "Your square [A - I]?"))
            (PRINTBOARD)
            (COND ((WINNER OPPONENT) (PROGN
                   (PRIN2T "Congratulations, you win!")
                   (RETURN NIL)))
              (PROGN (T (SETQ NEXTMOVE SELF)
                  (PROGN (GO LOOP))))))
           (T (PRIN2T "My turn.")
              (FINDMOVE)
              (PRINTBOARD)
              (COND ((WINNER SELF) (PROGN
                     (PRIN2T "I win!")
                     (RETURN NIL)))
                    (T (SETQ NEXTMOVE OPPONENT)
               (PROGN (GO LOOP)))))))))
```

Now the lower level functions required by PLAYTTT. Most are very straightforward.

```
% MAKEEMPTY makes the TTT square it is given have no
% mark on it.
(DE MAKEEMPTY (SQUARE) (PUTSTATUS SQUARE NIL))

% QUERY is a simple utility function for querying the
% user for a response, which it returns. It could be
% used in many contexts.
(DE QUERY (STRING) (PROGN (PRIN2 STRING) (PRIN2 " ") (READ)))

% OTHERMARK returns X, ib given O as an argument, and
% O, if given X. Otherwise it returns NIL.
(DE OTHERMARK (MARK)
  (COND ((EQ 'X MARK) 'O)
        ((EQ 'O MARK) 'X) ))

% PRINTBOARD prints out a simple representation of the
% TTT board.
(DE PRINTBOARD () (PROGN
  (PRINTSQUARE 'A)
  (PRIN2 "|")
  (PRINTSQUARE 'B)
  (PRIN2 "|")
  (PRINTSQUARE 'C)
  (TERPRI)
  (PRIN2T "-----")
  (PRINTSQUARE 'D)
  (PRIN2 "|")
  (PRINTSQUARE 'E)
  (PRIN2 "|")
  (PRINTSQUARE 'F)
  (TERPRI)
  (PRIN2T "-----")
  (PRINTSQUARE 'G)
  (PRIN2 "|")
  (PRINTSQUARE 'H)
  (PRIN2 "|")
  (PRINTSQUARE 'I)
  (TERPRI)))

% PRINTSQUARE prints the mark on a TTT square,
% printing " " if there is no mark.
(DE PRINTSQUARE (SQUARE)
  (COND ((NOT (GETSTATUS SQUARE)) (PRIN2 " "))
        (T (PRIN2 (GETSTATUS SQUARE)))))
```

```
% WINNER returns T if the side it is given as an
% argument has got three in a row in any of the TTT
% lines.
(DE WINNER (MARK)
  (PROG (LINES)
      (SETQ LINES TTPLINES)
LOOP
      (COND ((NOT LINES) (RETURN NIL))
            ((OCCUPIES MARK (CAR LINES))
             (RETURN T))
            (T (SETQ LINES (CDR LINES))
       (PROGN (GO LOOP))))))
```

11.9 Playing with the TTT Expert

The foregoing code represents a complete, although small, expert system for playing TTT. In spite of our extended discussion of the system, you are encouraged to type it into **UO-LISP** and play with it. This play can teach you many things that just reading about Lisp programming cannot.

You might try several things once you have it running. First, play a few games with the system. You'll find that the present system "interface" is pretty simple, and you might want to make it more "user friendly". For example, you should feel free to improve the TTT board display generated by PRINTBOARD; and you might want to change PLAYTTT so that if the user selects a square that is already occupied, the system will re-query the user.

Second, as you play games with the system you will probably notice that the system, while generally intelligent, makes a few dumb moves. Can you characterize the kinds of mistakes it is making? If so, you should consider how to improve its knowledge-base --- its TTTRULES. Try to think of modifications of its present rules, or even new rules to add to TTTRULES. Write down rules you think are better by trying to make explicit the rules that you use to play TTT. Then program them in Lisp, and see if the new TTT expert plays better, or plays more like you do.

More generally, you are encouraged to try all sorts of knowledge experiments. Add rules and take out rules then see how the modified system performs. Can you predict how a set of expert system rules will behave? AI researchers have found it surprisingly difficult to determine the behavior of a set of knowledge-based rules by just looking at them. That is one of the reasons why they build expert systems. Such systems really help us discover and represent the knowledge human experts use to perform difficult tasks.

As you create and examine various different TTT experts, notice how simple this experimentation is. You can create several modified experts in a few minutes. This is because, (unless you are getting pretty sophisticated) you are only modifying the rules in TTTRULES, and the modifications are very simple. For example, you might just take out a rule, switch the order of rules, add a rule, or modify a single rule. This fact points out one of the most important advantages of rule-based expert systems. They represent knowledge in a highly flexible and modular fashion. Contrast this modularity with an implementation of a TTT expert in BASIC or FORTRAN. Those programs might run a little faster than our TTT expert, but could you investigate the behavior of a variety of different experts nearly as rapidly?

Here is a final question for you to ponder as you experiment with the TTT experts. When you modify TTTRULES by adding a rule, or changing one, are you manipulating data, or are you writing new code? The TTT rules are data because they are the value of the variable "TTTRULES". But they are also program because they are EVALuated (see the functions DO and TEST) to generate moves. In fact, the rules are both because they are just S-expressions (like everything else). They become data when treated one way (e.g., assigned as the value of a variable) and become program when treated another way (e.g., evaluated). Expert systems take good advantage of this dual personality of rules to make them easy to modify and to use.

11.10 Summary of Lessons

This has been a particularly long section, so let's summarize the main points that have been made:

. All lisp programs are just a bunch of functions.

. Most lisp functions are small, independent, pieces of program.

. In writing an expert system or any large Lisp program you should program in a top-down fashion.

. Data representation should be regarded as an explicit phase of problem solving with Lisp. Appropriate representations make programs much easier to write.

. Writing programs using the principle of abstraction barriers makes code comprehensible and modifiable.

. Expert systems are AI programs that use sets of symbolic rules, often like those used by humans, to solve complex problems.

. Expert systems have easily modifiable knowledge-bases, facilitating investigation of the performance of a variety of rulesets.

CHAPTER 12
WHERE TO GO NOW

You have come to the end of our introductory tutorial on Lisp and AI. We hope you have enjoyed it enough to want to learn more about these subjects. In the past few years some very useful books have appeared that will hehp you move from a beginning AI programmer to an advanced one. These include:

`Artifical Intelligence', by P. H. Winston. (Addison Wesley, Reading, Mass., 1977).

This book gives the relatively inexperienced user an introduction to both Lisp and AI research, and includes many exercises.

`LISP', by P. H. Winston, and B. Horn. (Addison Wesley, Reading, Mass., 1977).

A book for intermediate-level Lisp programmers, with less emphasis on AI applications.

`Artificial Intelligence Programming', by C. Riesbeck, E. Charniak, and D. McDermott. (Lawrence Erlbaum Associates, Hillsdale, N.J., 1979).

The best advanced book for learning AI programming techniques. Not for beginners.

`Machines Who Think', by P. McCorduck. (W. H. Freeman, San Francisco, CA, 1979).

A somewhat biased introduction to the history and ideas of AI, requiring no background in Lisp.

`The Artificial Intelligence Handbook' (3 Vols.), edited by E. Fiegenbaum.

A voluminous compendium of the ideas and accomplishments of AI. Not for the beginner.

`The Structure and Interpretation of Computer Programs', by H. Abelson, and G. Sussman. (MIT Press, Cambridge, MA 1983).

Not really a book on AI or Lisp, but an outstanding first course in computation, which uses a dialect of Lisp. Highly recommended to those who want a solid beginning in computer science.

Index