The U.T. LOGO Primer

The Novice's Guide to LOGO

NL-23

by Jonathan Slocum

Dept. of Computer Sciences

INTRODUCTION

This manual is written for those who really do not know what a computer is -- someone who has perhaps never written a program (and might not know what one is), or at least who is "unsophisticated" in the art of speaking to a computer. As such, it may seem outrageously simplistic to an experienced programmer; however, there are other sources of information on LOGO which assume more experience on the part of the reader, and such persons may wish to seek those documents. In particular, although UT LOGO is imbedded in UT LISP, no knowledge of LISP is presumed. UT LOGO appears to the user as a stand-alone system, as intended in LOGO's initial conception.

How to Use this Document

You should read each chapter once to "get the flavor" of the material; after you finish each chapter, you should find a terminal and log in, and start reading the chapter again. Now every time you come to an example line, type it in just as shown. This way, you can see immediately what LOGO does with your line, and what you need to do next; you will also get some practice writing procedures, and in general "interacting" with the computer. If you read the entire manual before trying anything on a terminal, you will probably become more confused than necessary -- so you should endeavor to experiment with each chapter on the terminal, before proceeding to the next. Finally, when a question is asked you should stop immediately and formulate your answer before proceeding; this will give you a chance to see how much you know, and will point-out where you are confused.

A Technical Note

UT LOGO is run under the UT LISP monitor on the CDC 6600/6400, and interfaces to the EXEC monitor on an IMLAC PDS-1 graphics terminal. Non-graphics users may employ any standard "teletype" or crt device to use UT LOGO as a simple programming language. User-defined LOGO procedures are compiled on-the-fly on a line-by-line basis: every line is compiled the first time its execution is attempted; thereafter the LISP form is interpreted directly by the LISP interpreter, with no further overhead from the LOGO system. A primitive editor allows line deletion, insertion and (complete) replacement; in addition, any line within a procedure may be printed-out, or the entire procedure may be printed-out, for inspection.

UT LOGO adheres rather closely to LOGO as defined by the M.I.T. LOGO manual (LOGO MEMO 7); however, some M.I.T. LOGO features are prohibited by UT hardware/software constraints, and some features are inadequately documented in the M.I.T. manual. Usually in these cases the author attempted to substitute a reasonable alternative. There are also some extensions to LOGO: "all of LISP" is available to the LOGO programmer, in the sense that any UT LISP monitor

function (like, MEMBER) may be called by a LOGO procedure, so long as the LISP name is not used by the programmer as the name of one of his own procedures, and with the constraint that they have a fixed number of arguments as defined by the LOGO monitor. Where this constraint is unreasonable -- such as in a PROG or a PROGN or COND -- the LISP form is unavailable. Appendices A and B contain an exhaustive list of the available LOGO and LISP monitor functions, along with their number of arguments.

CHAPTER I
SOME SIMPLE THINGS

LOGO is a simple language for telling the computer what you want it to do. The first step in talking to a computer is getting connected to it -- this is called "logging in." We will assume that you know how to do this, or can find someone to show you, since the precise details may vary slightly from one location on campus to another. What happens when you do this is that the computer writes down in its "log book" the time you log in; when you log out, the computer writes this down, too, so that it knows who to bill (your account) for the computer time you used.

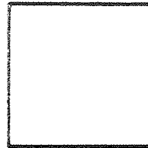
Once you are logged in and the computer types out "CC:" you are ready to do LOGO; you simply type "EXECPF 2331 LOGO" (without the quotes), then hit the "carriage return" key (the one labelled return) which is somewhere on the right side of the keyboard, just like on an electric typewriter. It is always the case that the computer will do nothing with what you type until you hit return; so if you make a bad mistake on a line, you can erase the whole line by hitting the "escape" key (labelled escape, or esc) on the left side of the keyboard. Whether or not this line disappears from view or not depends on what equipment you are using, but in any case the computer will not "see" the line. Once you escape, you can re-type the last line correctly and do a return at the end.

After doing the EXECPF line you are "in" LOGO. First the computer will type out some stuff, then it will type a question mark (?) at the beginning of a line and stop. This means that the computer is waiting for you to tell it what to do. Unfortunately, computers are not so smart yet that you can just type, in English (or Russian, or whatever), what you want -- but we're working on it. Meanwhile, we have invented some simple languages that the computer can understand, so that we may communicate with them. (While you may think these languages difficult, they really are not so, compared to English. Remember, you have had maybe twenty years or so to learn English -- and you are not through yet. Most computer languages -- and especially LOGO -- can be learned in a very short time.) We will start learning LOGO by learning some "direct" commands that will enable us to draw pictures.

The display screen you will be looking at is divided into 1024 X 1024 points; just like the Cartesian plane you learned about in high school, you can "address" any point on the screen by its x- and y-coordinates. The lowest point on the left side of the screen is point (0,0); the lowest on the right is (1023,0); the highest on the left is (0,1023); and the highest on the right is (1023,1023). Thus the middle of the screen, where most drawing will start, is about (512,512). At this start point, there is an imaginary "turtle" which you can direct to move around on the screen: you can tell it to move forward or back any distance (so long as it doesn't "fall off" the edge), or to turn (without moving) to the right or left by a given angle. If its "drawing pen" is down, it will draw a line

whenever it moves; if it is up, the turtle will move but not draw a line as it does. This way, it is possible to draw part of a picture (with the pen down), pick up the pen, move elsewhere, then put the pen back down and draw some more of the picture. Let's draw a square:

```
?RIGHT 90;
?FORWARD 200;
?RIGHT 90;
?FORWARD 200;
?RIGHT 90;
?FORWARD 200;
?RIGHT 90;
?FORWARD 200;
?
```



Since the turtle is initially at the center of the screen pointing up, this will turn the turtle to the right and then move it from the middle (512,512) to the right 200 steps to position (712,512); then it turns right another 90 degrees and moves 200 steps down to position (712,312); then it turns right again and moves to (512,312), and then makes its last right turn and moves up to its initial position (512,512). So at the end, the turtle is back where it started, and still pointing up.

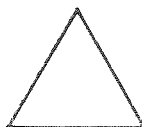
Notice how each command is preceded by a question mark (?). This is not something you type, but rather it is a "prompt character" that the computer types to signal that it is ready for another command. If, after you hit the carriage return, the computer "rings the bell" on your console but doesn't type a prompt character first, it probably means that you forgot to type the semicolon at the end; if this happens, just type the semicolon and hit return again, and you should see the prompt character next time. When the computer types the question mark, it is ready to carry-out a "direct" command. You type in the command, then a semicolon and return, and the computer will carry-out the command and afterwards type another question mark.

The command CLEARSCREEN will erase all pictures from the screen and return the turtle to the center of the screen, pointing up.

```
?CLEARSCREEN;
?
```

Now let's draw a triangle; our first triangle will be an equilateral triangle -- one with all three sides of the same length -- with one side horizontal:

```
?RIGHT 30;
?FORWARD 200;
?RIGHT 120;
?FORWARD 200;
?RIGHT 120;
```



```
?FORWARD 200;
?RIGHT 90;
?
```

Notice that this, too, will leave the turtle just as it was before -- at the center of the screen pointing up. Now if we had previously drawn the square and it was still visible on the screen, the triangle added to it makes a house. Maybe it isn't the most realistic house drawing in the world, but it is definitely a house. Later, we shall do better.

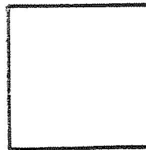
One could get tired of typing in eight commands for every square one wanted to draw. (Consider drawing a house with square windows.) What we would like to do is tell the computer how to draw a square, but only once; thereafter, we could say SQUARE and it would draw one -- as many times as we said SQUARE. So we will learn how to write a procedure. A procedure is a sequence of commands with a name (like, SQUARE). We will write a procedure called SQUARE and this will tell the computer how to draw squares:

```
?TO SQUARE;
>10 RIGHT 90;
>20 FORWARD 200;
>30 RIGHT 90;
>40 FORWARD 200;
>50 RIGHT 90;
>60 FORWARD 200;
>70 RIGHT 90;
>80 FORWARD 200;
>END;
 SQUARE DEFINED
?
```

The direct command TO tells the computer that we are defining a procedure; thereafter, until we type "END;" (which terminates the definition), the computer uses the prompt character ">" and accepts each line we type as part of the definition. The computer does not carry-out these "indirect" commands now. Also, each indirect command is preceded by a number -- called the "statement number" since a command is also called a "statement". The order in which we type the statements of a given procedure is inconsequential: the computer will arrange them in sequence according to their statement number. The numbers do not have to be consecutive, or even multiples of 10 (as shown); but it is preferable to leave "holes", since we may wish to add new lines later, in between old lines. If we don't leave holes, we would have to re-define the whole procedure. The computer does not care how big the numbers are -- within reason -- nor does it care how big the holes are, or whether they are uniform in size. Use whatever numbers suit you.

When we type "END;" then the computer types out SQUARE DEFINED (or whatever the name of the procedure is), then the question mark (whose significance we are already aware of). So now let's tell the

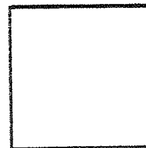
computer to draw us a square:



```
?CLEARSCREEN;
?SQUARE;
?
```

Now that we have taught the computer a new command, it can carry-out the new command as often as we tell it to. The command SQUARE will always draw a square of size 200, starting wherever the turtle is when we say (call) SQUARE, and leaving the turtle in the same place. But ... just how many squares of size 200 will we need in one picture? Perhaps one, and maybe more (2 or 3), but not likely very many. Have we really saved ourselves any work? The trouble is that SQUARE must always draw a square of the same size. Or must it? Wouldn't it be nice if one procedure could draw a square of any size? Luckily, we can do this:

```
?TO SQUARE :SIZE;
>10 RIGHT 90;
>20 FORWARD :SIZE;
>30 RIGHT 90;
>40 FORWARD :SIZE;
>50 RIGHT 90;
>60 FORWARD :SIZE;
>70 RIGHT 90;
>80 FORWARD :SIZE;
>END;
 SQUARE DEFINED
?SQUARE 200;
?
```



Here we define the procedure SQUARE with one argument (or input) -- a number representing the desired size of the square. Now when we want a square drawn, we have to specify some number for the desired size. The computer then uses this number in the procedure, wherever its name (SIZE) appears. Now we can say

```
?SQUARE 400;
```

to get a very large square, or

```
?SQUARE 30;
```

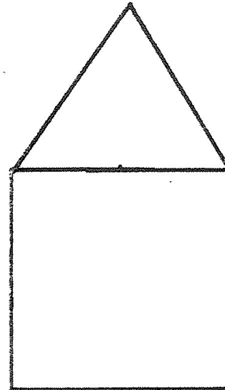
to get a very small square, etc. All we have to remember is that we must specify some size. Now if we define TRIANGLE to be:

```
?TO TRIANGLE :SIZE;
>10 RIGHT 30;
>20 FORWARD :SIZE;
>30 RIGHT 120;
>40 FORWARD :SIZE;
>50 RIGHT 120;
>60 FORWARD :SIZE;
```

```
>70 RIGHT 90;
>END;
 TRIANGLE DEFINED
?
```

-- then we can define HOUSE to be:

```
?TO HOUSE :SIZE;
>10 SQUARE :SIZE;
>20 TRIANGLE :SIZE;
>END;
 HOUSE DEFINED
?
```



-- and then we can say

```
?CLEARSCREEN;
?HOUSE 400;
```

to get a large house, or

```
?CLEARSCREEN;
?HOUSE 50;
```

to get a small house. Notice that we have taught the computer three new commands (SQUARE, TRIANGLE, and HOUSE), and that one of them (HOUSE) "calls" each of the others. So you see that you can use a direct command to call any procedure, and then it can call any number of other procedures, etc. (In fact a procedure can call itself, as we shall see later.) Now that one procedure can draw squares of any size, we can use it to draw windows in the house; but in order to do this, we need to learn how to tell the turtle to lift its pen up, so that it doesn't draw when it moves to the place to draw a window. The command PENUP will do this, and the command PENDOWN will put it back down. So to draw windows, lift the pen up, move to the place for one window, put the pen down and draw it, lift the pen up and move to the next window location, etc. These commands might be part of the procedure HOUSE -- you can try this for yourself, just for fun.

```
?TO HOUSE :SIZE;
>10 TRIANGLE :SIZE;
>20 SQUARE :SIZE;
>30 PENUP;
```

and so forth.

If you succeed, you might wish to try for a door; if you fail, you won't yet know how to correct your mistake, except by re-defining a procedure, but we'll learn later. Also you might write a procedure RECT that takes two arguments, LENgth and WIDth --

```
?TO RECT :LEN :WID;
>10 RIGHT 90;
```

```
>20 FORWARD :LEN;
>30 RIGHT 90;
>40 FORWARD :WID;
```

etc. -- and draws a rectangle.

Let's study some more examples of how to put procedures together into larger procedures to draw more complex things:

```
?TO DRAW :DIST;
>10 FORWARD :DIST;
>20 BACK :DIST;
>END;
 DRAW DEFINED
?TO VEE :SIZE;
>10 LEFT 50;
>20 DRAW :SIZE;
>30 RIGHT 100;
>40 DRAW :SIZE;
>50 LEFT 50;
>END;
 VEE DEFINED
?TO ARROW :SIZE;
>10 VEE :SIZE;
>20 FORWARD :SIZE;
>30 FORWARD :SIZE;
>40 VEE :SIZE;
>END;
 ARROW DEFINED
?
```

DRAW will draw one line and then re-trace it, perhaps making it darker -- but the idea is to end up where the turtle started. VEE will use DRAW to draw a "v" of the requested size:

```
?VEE 200;
?
```

ARROW will use VEE once to make the head of the arrow, then will move forward to make the shaft, then use VEE again to make the tail.

```
?ARROW 100;
?
```

The reason why we used FORWARD twice in ARROW is that we wanted to make the shaft longer than the head, but don't know how (to use arithmetic) to do so. But we will learn soon.

Here is a procedure to draw any regular polygon, given the length of a side and the angle to turn:

```
?TO POLY :STEP :ANGLE;
>10 FORWARD :STEP;
>20 LEFT :ANGLE;
```

```
>30 POLY :STEP :ANGLE;
>END;
```

Some examples of what it can do are shown below; did you notice line 30? POLY calls itself. LOGO certainly allows you to do this, but this particular usage has an interesting consequence -- once you call POLY, it never stops drawing. First the computer does statement 10, then 20, then it calls POLY, which does statement 10, then 20, and then calls POLY ... forever. Sooner or later the turtle will begin re-tracing its old path, but it will still continue drawing. Obviously, we shall have to learn how to get around this problem -- we must not tell the computer to keep on going forever. We need to learn arithmetic very soon. But first, here are six examples (do NOT try these) of what POLY could do:

How would you tell the turtle to draw a hexagon and stop when finished? (Hint: rotate the turtle only half as much at the corners as we did for a triangle.) Can you think of a procedure somewhat like POLY that would draw any regular convex geometric figure (one whose sides and angles were equal, but whose lines did not cross) -- to which you passed the number of sides desired, and the length of each side, but not the angle to turn? For instance, if the procedure were called REGULARPOLY, then TRIANGLE might be defined with only one line:

```
?TO TRIANGLE :SIZE;
>10 REGULARPOLY 3 :SIZE;
>END;
```

This procedure will require some arithmetic, so let's learn how to do arithmetic in LOGO.

Suppose you want the computer to form the sum of two numbers. (As everybody knows, computers are good at arithmetic. They never make mistakes, and they are very, very fast.) We use the SUM command:

```
?SUM 29 67;
 96
?
```

In response to this command, LOGO will print-out the result on the next line, and then type the question mark as before. SUM is a command that returns an answer -- computer people call an answer a returned value, or the value of the procedure. This is the first time that we have noticed a procedure return a value; but actually, all procedures always return values. We will discuss this later.

Now suppose you want the difference of two numbers:

```
?DIFFERENCE 96 67;
 29
?
```

Similarly, you can take the product of two numbers, and their quotient:

```
?PRODUCT 5 7;
 35
?QUOTIENT 35 7;
 5
?QUOTIENT 34 7;
 4
?
```

But what is this? 34 divided by 7 is 4? And we thought computers were good at arithmetic. Well, when computers divide integers (sometimes called "whole" numbers), they throw away the remainder.

```
?REMAINDER 34 7;
 6
?
```

If you will remember back to elementary school, you probably first learned to divide by finding the quotient of whole numbers, and the remainder. (How many times did your teacher mark your problems wrong because you forgot to write down the remainder?) Later, you learned what decimal points are, and probably haven't used remainders since. It turns out that computers know about decimal numbers too, but we'll talk about that later.

There is a way to get both the quotient and the remainder in one operation:

```
?DIVIDE 34 7;
 4 6
?
```

So you see that DIVIDE and QUOTIENT are very different. One thing you must not do is to confuse them, because the computer isn't smart enough to figure out what you meant; it will do only what you tell it to. Computers are easily confused, and when they are, they do some very strange things, indeed.

Now, anyone can add just two numbers; if computers are to be helpful, they should do things that we find hard -- like add lots of numbers at once. But

```
?SUM 15 17 89;
```

will not do, because the command SUM can take only two numbers. (Try it.) There are two ways to get around this: one is by functional composition, and the other is by writing a procedure. Composition is easy to explain. We note that the sum of 15 and 17 and 89 may be expressed as the sum of 15 and the sum of 17 and 89:

```
?SUM 15 SUM 17 89;
 121
```

Or we could write it another way:

```
?SUM SUM 15 17 89;
 121
```

The first way is more readable, so we prefer it: it sums 17 and 89, and then sums 15 and the result (106) to form 121. Notice that it doesn't print the partial sum (106) -- this is because it isn't finished with the entire command (the entire statement). The second example above works just as well, but differently: it sums 15 and 17 to form the partial sum 32, then sums 32 and 89 to form the total 121, and then prints it. To sum four numbers:

```
?SUM 30 SUM 27 SUM 15 19;
 91
?
```

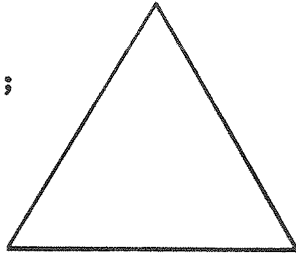
You can add any number of numbers this way, but after a while it may get a little tiresome having to type SUM so many times. What we will do now is write a procedure to add arbitrarily many numbers; in this way we will teach the computer another new command. We will put a pair of parentheses around the numbers to be added -- thus forming a list of numbers: (15 17 34 27 6 10). This list has six numbers in it. We will tell the computer that the ADD of all the numbers in a list is simply the SUM of (1) the first number in the list, and (2) the ADD of the rest of the numbers in the list; computer people (and mathematicians) call this a "recursive" definition. (See how the result of the operation is expressed in terms of the operation itself? This makes a definition recursive.) Now we teach the computer the new command:

```
?TO ADD :LIST;
>10 IF EMPTYP :LIST THEN RETURN 0;
>20 RETURN SUM FIRST :LIST ADD BUTFIRST :LIST;
>END;
 ADD DEFINED
?ADD (15 17 34 27 6 10);
 109
?
```

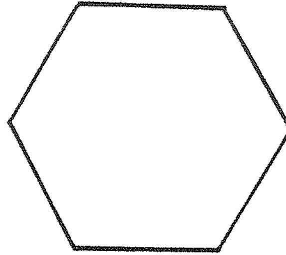
The name of this new command is ADD; its single argument is the list of numbers to be added. (The list counts as one, no matter how many things it contains.) An "empty list" is a list containing nothing; it is written: (). Line 10 says that, if the list is empty (with no numbers in it), then ADD will return 0 (zero) as its value. Otherwise in line 20 ADD will call itself with all the numbers in the list except for the first -- this will subtotal all the numbers but the first -- and then ADD will return as its value the SUM of (1) the first number in the list, and (2) the subtotal just returned (the ADD of the rest of the list).

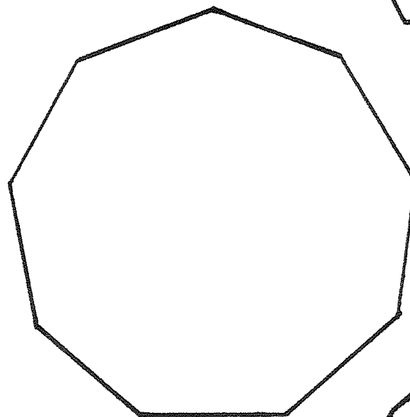
If this seems strange, stop and think a minute: (1) we intend to write a procedure ADD to return as its value the total sum of all the numbers in a list; (2) we have faith that ADD will work properly; (3) we know about composition, and that it works; (4) if ADD can sum-up a list of (say) 6 numbers, then it can sum-up a list of 5 numbers, and then form the SUM of that sub-total and the sixth number. Therefore, ADD must work, as we have defined it. All we need to note is that, although ADD calls itself (like POLY, remember?), it will not in this case continue indefinitely (like POLY). This is because, by calling ADD with the BUTFIRST of the list, we are "shrinking" the list of numbers by one number each time; sooner or later, the list will be down to one element, then none, and when this happens (the list becomes empty), ADD returns a value (0). So ADD doesn't call itself again when the list becomes empty.

?POLY 150 120;

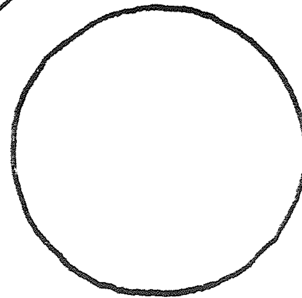


?POLY 75 60;

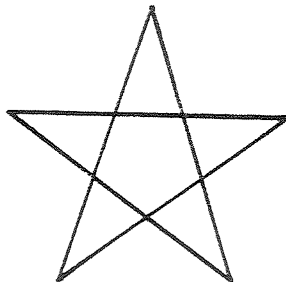




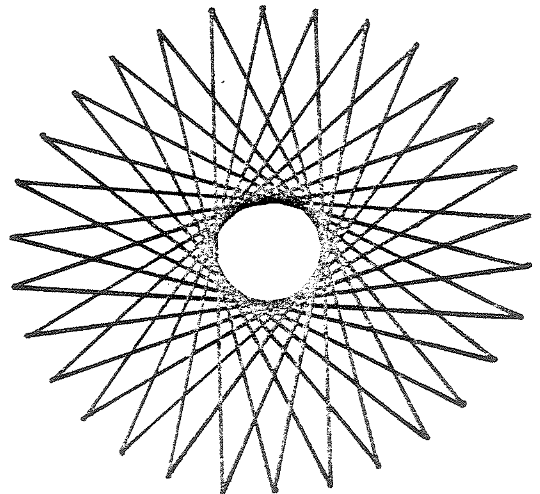
?POLY 75 40;



?POLY 4 3;



?POLY 150 144;



?POLY 300 156;

Line 10 contains our first exposure to the IF statement. Obviously, the IF statement tests for a certain condition (here, for the empty list) and performs an action (here, returning zero) only provided the test was true. If the test was false, that action (the rest of the statement, after the test) will not be performed, but instead the next line in the procedure is executed. Usually, if the test is true and the rest of the IF statement is executed, then the next line is executed thereafter; however, here the RETURN command says to exit the procedure with the indicated value. With some thinking, you can see that ADD (15 17 34 27 6 10) is the same as SUM 15 SUM 17 SUM 34 SUM 27 SUM 6 SUM 10 0 -- which is how the computer finally performs the operation, given that ADD () returns zero.

We have also learned some other things in this exercise: (1) we have been introduced to lists; (2) there is a procedure EMPTYP that returns true if the list passed to it is empty; (3) there is a procedure FIRST that returns the first thing in a list; (4) there is a procedure BUTFIRST that returns a list composed of all the things in a list except for the first thing; and (5) we have learned that we can write a recursive procedure (one that calls itself) that does not continue executing forever, but (eventually) stops.

Now you write a procedure called MULTIPLY that returns the product of all the numbers in a list.

You see that you can teach a computer how to do a lot of work for you; this is why (and how) we use computers. By themselves, they can do only very, very simple things, but we can teach them more things and have a lot of fun in the process. (The author had to teach our machine LOGO.)

Remember DIVIDE$, It seemed to have two values, didn't it? Well, it really has only one -- but that one is a list (of two numbers). When LOGO prints-out a list, it doesn't print the leftmost (left) and rightmost (right) parentheses. Thus the list (4 6) prints as 4 6. A procedure can only have one OUTPUT (answer). By the way, there is a command PRINT that takes one argument -- whether it be a word, a number, or a list -- and prints it immediately. You might like to define ADD (or MULTIPLY) with a line 5 like this:

```
?TO ADD :LIST;
>5 PRINT :LIST;
>10 IF EMPTYP ...
etc.
```

Go ahead and try it. It doesn't change the answer. (Why?)

What do you think would happen if we defined ADD like this:

```
?TO ADD :LIST;
>10 IF EMPTYP :LIST OUTPUT 0;
>20 SUM FIRST :LIST ADD BUTFIRST :LIST;
```

```
>END;
 ADD DEFINED
?
```

First of all, the word THEN is missing from the IF statement -- but that is ok since it is optional, anyway. Second, the word OUTPUT has replaced the word RETURN -- but that is ok since they are synonyms. Perhaps you should stop here and type it in like it is defined above, and then try it on a couple of examples, like ADD () and ADD (5 6 7).

Surprised? In line 20 we formed the sum correctly, but we forgot (to tell the computer) to RETURN that result. (Go back and look at our former definition.) When you want a procedure to return a value, you must remember to use RETURN or OUTPUT; otherwise, the computer will return as the default value the empty list (). And when it prints a list, it doesn't print the leftmost and rightmost parentheses; in the case of the empty list, this means that there is nothing left to print. This is the only reason why nothing was printed after the picture-drawing procedures we played with earlier: they were returning a value -- the empty list -- which cannot be seen when it is printed, since there is nothing inside it to be printed. The empty list is also the FALSE value: whenever an IF statement's test returns the empty list as its value, that is interpreted as false and the THEN statement is not executed. Forgetting to return a value from a procedure is a frequent source of error -- be warned.

A mistake in a procedure is called a "bug" (because in the old days real bugs could get into a computer and foul it up, causing errors). The process of correcting mistakes (cleaning out the insects) is called "debugging", and we shall learn what to do when we have found a bug, in the next chapter.

Now let's learn how to compare numbers. The procedure GREATER takes two numbers and returns TRUE if the first is greater than the second; otherwise it returns FALSE (the empty list). The function LESS takes two numbers and returns TRUE if the first is less than the second. The function EQUAL takes two numbers and returns TRUE if they are equal. You can use these tests to determine whether (and what) to return from a procedure.

Now you go back and write REGULARPOLY. Does it work for 3, 4, 5, 6, 8, 9, and 10 sides? How about 7? Or 11? If there is an error (and I'll bet there is), what do you think might be the cause? (The clue is provided in the discussion about whole-number arithmetic.) Can you guess how to correct it?

CHAPTER II
EDITING

Assume we had originally defined ADD as we did above, and we have found the bug; what can we do? We can always re-define the procedure, but that is bothersome. Besides, for very large procedures we are liable to make a mistake when typing it -- thus introducing yet another bug while correcting an old one. What we have to help us in these cases is an editor. The command EDIT is available, and its argument is the name of the procedure that we wish to change in some way:

```
?EDIT ADD;
 EDITING ADD
<
```

The message EDITING <name> is typed out by the computer to verify that we are in editing mode, and that the procedure we named in the EDIT command exists. The new prompt character "<" is typed out to indicate that the computer is ready to accept an editing command -- one of the commands that operates on one line of the procedure being edited. To correct the procedure ADD, we need to change line 20, so:

```
<CHANGE 20 RETURN SUM FIRST :LIST ADD BUTFIRST :LIST;
```

While we're at it, let's add a new line that will "trace-print" when ADD is executed:

```
<INSERT 5 PRINT :LIST;
```

Then we will finish editing ADD:

```
<END;
 ADD RE-DEFINED
?ADD (3 4 5);
 3 4 5
 4 5
 5
 12
?
```

-- and now ADD is working properly. This being so, we should now delete the annoying "trace-print" line:

```
?EDIT ADD;
 EDITING ADD
<DELETE 5;
```

Obviously, one is allowed to edit any procedure as often as desired. While editing a procedure, we can request that any of its lines be printed-out so that we may look at it:

```
<SHOW 20
 20 RETURN SUM FIRST : LIST ADD BUTFIRST : LIST ;
<END;
 ADD RE-DEFINED
?
```

Or at any time we can tell the computer to print-out a whole procedure:

```
?SHOW ADD;
 TO ADD : LIST ;
 10 IF EMPTYP : LIST OUTPUT 0 ;
 20 RETURN SUM FIRST : LIST ADD BUTFIRST : LIST ;
 END;
?
```

-- and it will print it out, line by line. These four commands (CHANGE, INSERT, DELETE, and SHOW) are the only special commands one needs to perform any desired editing. Actually, CHANGE isn't strictly necessary since it could be effected using two commands (DELETE and INSERT), but it is included for your convenience. Also, showing of a particular line is unnecessary since one could show the whole procedure, but then the idea of having a computer is to make things (including waiting) easier on you, the user.

CHAPTER III
PRACTICE

Now that we know how to define new procedures and edit old ones, let's get some experience in doing it. We will write a new procedure COUNT which takes one input -- a list -- and returns (OUTPUTs) the number of things in the list.

```
?TO COUNT :LIST;
>10 IF EMPTYP :LIST THEN RETURN 0;
>20 RETURN SUM 1 COUNT BUTFIRST :LIST;
>END;
 COUNT DEFINED
```

An empty list has COUNT zero; a list of one thing -- like (hello) -- has COUNT one, and so on. Essentially, the COUNT of a list is zero if the list is empty, and otherwise it is one plus the COUNT of the BUTFIRST of the list. What do you think is the COUNT of the list: (I (LIKE YOU) TOO) ? Let's try it:

```
?COUNT (I (LIKE YOU) TOO);
 3
```

What we have here is a thing in a list that is itself a list: the list (LIKE YOU) is called a "sublist" of the list (I (LIKE YOU) TOO). No matter how many things a sublist has in it, it counts as only one thing in another list of which it is a sublist. Any list can be a sublist of any list.

Now let's learn how to find out if a certain thing is already in a list. We will write MEMBER; it will return TRUE if its first argument is somewhere in the list that is its second argument, otherwise it will return FALSE:

```
?TO MEMBER :X :L;
>10 IF EMPTYP :L THEN RETURN FALSE;
>20 IF EQUAL :X FIRST :L THEN RETURN TRUE;
>30 RETURN MEMBER :X BUTFIRST :L;
>END;
 MEMBER DEFINED
?MEMBER (B) ((A) (B) (C));
 *T*
?MEMBER (D) ((A) (B) (C));
?
```

Line 10 says that, if the list is empty, nothing can be in it so return false; line 20 says that, if the thing in X is equal to the first thing in the list in L, then we can return true; line 30 says that, otherwise, MEMBER is true or false depending on whether the thing in X is in the rest of the list. Note that EQUAL compares any two things, and not just numbers.

LOGO also allows an ELSE statement within an IF statement; the reader may verify that the entire procedure MEMBER may be written in "one line" as follows:

```
?TO MEMBER :X :L;
>10 IF EMPTYP :L THEN RETURN FALSE
    ELSE IF EQUAL :X FIRST :L THEN RETURN TRUE
    ELSE RETURN MEMBER :X BUTFIRST :L;
>END;
 MEMBER DEFINED
?MEMBER (A) ((A) (B));
 *T*
?MEMBER (C) ((A) (B));
?
```

You see, then, that ELSE is interpreted just as one would expect it to be; it is also the case that ELSE may appear only within an IF statement. That is, the ELSE statement is meaningful only within an IF statement. (The reason why MEMBER results in *T* in the first case(s) is that *T* is what LOGO (LISP) uses to mean TRUE; the reason why nothing gets printed in the second case(s) is that LOGO (LISP) uses the empty list () to mean FALSE, and as we have already seen, LOGO does not print the empty list.)

Notice, too, that line 10 above extends over more than one line on input. This is ok, so long as you type less than 72 columns on one line, and then hit the return key and continue typing. The computer will accept as many lines as you type (until it sees a semicolon) as one statement. (If you type too far over on one line, the computer will do an "automatic" carriage return -- this is not the same as when you do it. You must then hit the return key yourself. But if you are in the middle of a word, you will have to "escape" and re-type the line, hitting return before the last word and starting the next line with that word.)

Now we are ready to learn about handling words and lists. We know what FIRST does when we give it a list; there is also a procedure LAST that returns the last thing in a list:

```
?FIRST (A B C);
 A
?LAST (A B C);
 C
```

We know that BUTFIRST returns a list composed of everything in the list we give it except for the first thing; there is a procedure BUTLAST that returns a list composed of everything in the list we give it except for the last thing:

```
?BUTFIRST (X Y Z);
 Y Z
?BUTLAST (X Y Z);
 X Y
```

We can also put new things into a list, either at the front:

```
?FPUT "X (Y Z);
 X Y Z
```

-- or at the back (last):

```
?LPUT "Z (X Y);
 X Y Z
```

Here we meet with something new: we use the quote mark (") in front of a word (like X, or Z) to mean that we are talking about the word itself, rather than the thing in the word. Always before, we have used the colon; this means to use the thing in the word, rather than the word itself. If a word appears without either a quote mark or a colon in front of it, then it means that we are calling the procedure with that name. (So you see that our use of the word TRUE within LOGO procedures means calling a function (named TRUE) that returns *T*, and that FALSE is the name of a LOGO function that returns the empty list.)

Actually, FPUT and LPUT do not change the lists we give them, but instead return altered copies of the lists. Now let's use FPUT to write a procedure APPEND that takes two lists and returns as its value a new list composed of all the things in the first list, followed by all the things in the second list. Here are some examples:

(1) APPEND () (A B) is (A B);
(2) APPEND (A) (A B C) is (A A B C);
(3) APPEND (A B) (C D) is (A B C D).

Now we write it:

```
?TO APPEND :L1 :L2;
>10 IF EMPTYP :L1 THEN RETURN :L2;
>20 FPUT FIRST :L1 APPEND BUTFIRST :L1 :L2;
END;
 APPEND DEFINED
?APPEND () (A B);
 A B
?APPEND (A B) (C D);
?
```

Oops. Something is wrong -- what is it? We forgot to RETURN the result (in line 20). What we should have said was:

```
>20 RETURN FPUT FIRST :L1 APPEND BUTFIRST :L1 :L2;
```

Now you use the editor to make the correction.

What do you think would happen if we had forgotten to type in line 10? Would we have an infinite loop? If so, why? If not, why not? When APPEND is executed, it will do line 20, which causes APPEND to be executed, which causes line 20 to be executed, which

causes APPEND to be executed ... forever? No -- there is another mistake. What is it? Each time line 20 is executed, it calls APPEND with a smaller list in L1; eventually, the list in L1 will become empty. If line 10 existed, then APPEND would then return a value; but it doesn't. So line 20 is executed. Now, what is the FIRST of an empty list? Since there is no first element, FIRST is not defined. That's right -- undefined. We use the term in the mathematical sense, since there is certainly a function called FIRST. What we mean is that the function (procedure) FIRST does not know what to do with the empty list. Therefore the computer "generates an error" and stops working on the latest command. It will also print-out an error message, or else call your error routine. (We will learn about this, later.) More to the point, this will terminate the (otherwise) infinite loop.

Well, we were saved from this infinite loop bug by the fact that we would try to execute an undefined function. But what would happen if this were not the case? Consider this definition of APPEND:

```
?TO APPEND :X :Y;
>10 IF EMPTYP :X RETURN :Y;
>20 RETURN FPUT FIRST :X APPEND :X :Y;
>END;
```

Notice that line 20 neglects to make the list in X smaller. So, unless the first list we pass to APPEND is already empty, we have an infinite loop because it will never become empty, and line 10 will never be able to return a value. This being the case, it is the same as if line 10 didn't exist; the only thing that might have saved us is the error discussed above (FIRST) which will not exist here. So we have an infinite loop.

Infinite loops can never be completed; however, there are ways in which they may be terminated without meaningful result. (We have already discussed one -- the undefined-function error.) LOGO must "remember" where it is when it calls a function (so that it knows where to come back to when the function is completed); LOGO can remember just so many things, before it runs out of memory; to run out of memory is itself an error. Therefore, the "infinite loop" above also turns out to be terminable: eventually, LOGO will run out of the memory it needs to call APPEND again, and this generates an error, which in turn causes the computer to terminate the execution of APPEND. So we are saved. Again. This type of error is called a "recursion limit" error. Recursion limit errors will sooner or later be detected and terminated -- but most importantly, you should be very careful not to construct infinite loops to begin with. After all, you're wasting both your time and computer time.

In this chapter we have had a little practice in writing procedures; we shall get some more practice later. As it happens, the procedures we wrote in this chapter are all so useful that they have already been incorporated into LOGO, so you won't have to

define them for yourself when you want to use them. Also, FIRST, LAST, BUTFIRST, BUTLAST, and COUNT may take a word as their argument, and if so they treat the word somewhat as if it were a list of letters:

```
?FIRST "WHOOPIE;
 W
?LAST "PRINT;
 T
?BUTFIRST "WOW;
 OW
?BUTLAST "WOW;
 WO
?COUNT "WHOOPIE;
 7
?
```

So you see that you can play with words like you can play with lists.

CHAPTER IV
ITERATION IN LOGO

So far, you know a little about how to use recursion in LOGO; now you will learn about iteration. To do iteration you have to know about two things -- how to change the contents of a word (a variable), and how to tell the computer to go to some statement in the procedure other than the next one in sequence. So first we will learn the former: how to put new things into words.

Imagine a word (called a variable) to be like a "pigeon-hole" that can accommodate only one thing at a time. You can tell the computer to put something into the pigeon-hole, which destroys whatever was there before. Or you can tell the computer to look at what is already there. Actually, you know how to do this: the colon in front of a word tells the computer to look inside. And when you call a procedure and pass it some argument(s), the computer puts the argument(s) into the word(s) following the colons on the first line of the procedure (the one starting with TO). But there is a way to tell the computer to put something into a word at any time, whether you are in a procedure or not:

```
?MAKE "X 10;
```

-- for instance, will put the number 10 into the pigeon-hole called X. Any word could appear where X appears, and that word would have 10 put into it. Also, in place of the number 10 one might put any number, or a list, or a colon or quote followed by another word, or a call to any procedure. For example:

```
?MAKE "TOTAL SUM :TOTAL FIRST :L;
```

will form the sum of the number currently in TOTAL, and the number that is the first thing in the list called L, and will put the result back into TOTAL. The number previously in TOTAL has been displaced, and can no longer be retrieved. Notice the quote mark. Again, it means that we are talking about the word TOTAL, instead of what is in it. Consider carefully the following:

```
?MAKE "RABBIT "HARE;
 HARE
?PRINT "RABBIT;
 RABBIT
?PRINT :RABBIT;
 HARE
?MAKE "HARE "LUPUS;
 LUPUS
?PRINT "HARE;
 HARE
?PRINT :HARE;
 LUPUS
?MAKE :RABBIT "EGG;
 EGG
```

```
?PRINT :RABBIT;
 HARE
?PRINT :HARE;
 EGG
```

Especially note that (MAKE :RABBIT "EGG) puts the word EGG into the pigeon-hole whose name is (not the word RABBIT, but) <u>in</u> the word RABBIT. Generally speaking, this is wrong. (That is, it is not what you intend.) In almost all cases, the procedure MAKE should have as its first argument, a "quoted" word -- not a "colon-ed" word.

Normally, the computer will execute each line within a procedure in numeric sequence. We have already seen where the IF statement controls how much within a single statement is executed. We have also seen that the RETURN (or OUTPUT) statement causes the procedure to be exited, without any further execution. There is another procedure that controls execution of lines: the GO command (which can appear only within a procedure) takes as its argument a line number and tells the computer to go to that line and start executing there. Consider this definition of ADD:

```
?TO ADD :L;
>10 LOCAL :N;
>20 MAKE "N 0;
>30 IF EMPTYP :L RETURN :N;
>40 MAKE "N SUM :N FIRST :L;
>50 MAKE "L BUTFIRST :L;
>60 GO 30;
>END;
```

The LOCAL statement invents a new pigeon-hole called N. Statement 20 puts zero into the word N. Line 30 is our "finished?" test -- if the list in L is empty, then we are done and the contents of N is the answer. Otherwise, line 40 adds the first number in the list to N, and line 50 throws away the first number in the list and puts the rest of the list back into L, and line 60 says to go to line 30 and continue from there.

Performing the same sequence of statements over and over again is called iteration. Earlier, in chapter I, we had a "recursive" definition of ADD; here we have an "iterative" definition. As you can see, the iterative definition has more lines, and thus is more work for you. The computer doesn't care which form you use, although the recursive form might be a little faster -- like, a few micro-seconds. Furthermore, although recursion requires memory -- which can therefore be exhausted by too much demand for it -- iteration does not require memory, and thus an infinite loop error involving iteration may be undetectable. This is one situation that you absolutely, certainly wish to avoid. If you have created an infinite iterative loop, then unless that causes another error to occur (like taking the FIRST of an empty list), you may never know it --- and the computer will grind on and on, wasting time (and

money). If you expect that this is the case, all you can do is abort the entire program -- and lose everything. So if you use iteration, be careful.

```
?TO COUNT :L;
>10 LOCAL :N;
>20 MAKE "N 0;
>30 TEST EMPTYP :L;
>40 IFTRUE RETURN :N;
>50 MAKE "N SUM 1 :N;
>60 MAKE "L BUTFIRST :L;
>70 GO 30;
>END;
```

Here is another definition of COUNT. Lines 30 and 40 contain something new: the procedures TEST and IFTRUE. The procedure TEST takes its argument (here, the result of the procedure EMPTYP) and puts it into a TESTBOX which is automatically associated with the procedure (COUNT). The procedure IFTRUE will investigate this procedure's testbox and, if its contents are TRUE (that is, if it is anything except the empty list, which is the FALSE value), the computer will execute the rest of the statement. If the contents is FALSE (the empty list), then the computer will not execute the rest of the statement. Conversely, there is an IFFALSE statement that operates oppositely; thus lines 30 and 40 could be written:

```
>30 TEST :L;
>40 IFFALSE RETURN :N;
```

Note that the word THEN is not allowed here, nor is there an ELSE portion.

```
?TO MEMBER :X :L;
>10 IF EMPTYP :L RETURN FALSE;
>20 IF EQUAL :X FIRST :L RETURN TRUE;
>30 MAKE "L BF :L;
>40 GO 10;
END;
```

Here we have illustrated the iterative form of MEMBER. Notice the abbreviation BF for BUTFIRST in line 30; several of the LOGO procedures have abbreviations. Appendix A lists all of them.

```
?TO APPEND :L1 :L2;
>10 IF EMPTYP :L1 THEN RETURN :L2;
>20 MAKE "L2 FPUT LAST :L1 :L2;
>30 MAKE "L1 BL :L1;
>40 GO 10;
>END;
```

BL is of course the abbreviation for BUTLAST. Notice that MAKE "L2 is necessary in line 20. Why?

Now we will re-write POLY so that it stops after drawing a certain number of lines:

```
?TO POLY :NSIDES :STEP :ANGLE;
>10 IF LESS :NSIDES 1 THEN STOP;
>20 FD :STEP;
>30 LT :ANGLE;
>40 MAKE "NSIDES DIFFERENCE :NSIDES 1;
>50 GO 10;
>END;
```

--or, better yet:

```
?TO POLY :NSIDES :STEP :ANGLE;
>10 IF LESS :NSIDES 1 STOP;
>20 FD :STEP;
>30 LT :ANGLE;
>40 POLY DIFFERENCE :NSIDES 1 :STEP :ANGLE;
>END;
```

Now you can try the POLY examples from chapter I, adding the appropriate (first) argument. The command STOP is like RETURN (or OUTPUT) except that you may not specify a value to be returned -- false is the value returned.

The only thing you need to know now is how to find bugs, given that you know they exist (because your program is doing funny things that you did not intend, or because it is generating error messages).